

VU LEARNING HUB

vulearninghub.xyz

HIGHLIGHTED HANDOUT

CS501

Advance Computer Architecture

CS501 by VU Learning HUB

Prepared for VU students • Free study resources

<https://vulearninghub.xyz>

Lecture Handouts

CS501

Advance Computer Architecture

Table of Contents

Appendix A	4
FALSIM.....	4
Lecture No. 1	17
Introduction.....	17
Lecture No. 2	30
Instruction Set Architecture	30
Lecture No. 3	42
Introduction to SRC Processor	42
Lecture No. 4	48
ISA and Instruction Formats.....	48
Lecture No. 5	62
Description of SRC in RTL	62
Lecture No. 6	71
RTL Using Digital Logic Circuits	71
Lecture No. 7	85
Design Process for ISA of FALCON-A	85
Lecture No. 8	89
ISA of the FALCON-A.....	89
Lecture No. 9	101
Description of FALCON-A and EAGLE using RTL	101
Lecture No. 10	118
The FALCON-E and ISA Comparison.....	118
Lecture No. 11	137
CISC and RISC.....	137
Lecture No. 12	138
CPU Design	138
Lecture No. 13	144
Structural RTL Description of the FALCON-A	144
Lecture No. 14	150
External FALCON-A CPU.....	150
Lecture No. 15	158
Logic Design and Control Signals Generation in SRC.....	158
Lecture No. 16	169
Control Unit Design.....	169
Lecture No. 17	178
Machine Reset and Machine Exceptions	178
Lecture No. 18	183
Pipelining	183
Lecture No. 19	190
Pipelined SRC	190
Lecture No. 20	196
Hazards in Pipelining.....	196
Lecture No. 21	201
Instruction Level Parallelism	201
Lecture No. 22	205
Microprogramming.....	205

Lecture No. 23	214
I/O Subsystems	214
Lecture No. 24	224
Designing Parallel Input and Output Ports	224
Lecture No. 25	232
Input Output Interface	232
Lecture No. 26	241
Programmed I/O	241
Lecture No. 27	251
Interrupt Driven I/O	251
Lecture No. 28	259
Interrupt Hardware and Software	259
Lecture No. 29	270
FALSIM	270
Lecture No. 30	281
Interrupt Priority and Nested Interrupts	281
Lecture No. 31	287
Direct Memory Access (DMA)	287
Lecture No. 32	293
Magnetic Disk Drives	293
Lecture No. 33	297
Error Control	297
Lecture No. 34	301
Number Systems and Radix Conversion	301
Lecture No. 35	307
Multiplication and Division of Integers	307
Lecture No. 36	312
Floating-Point Arithmetic	312
Lecture No. 37	316
Components of Memory Systems	316
Lecture No. 38	320
Memory Modules	320
Lecture No. 39	322
The Cache	322
Lecture No. 40	328
Virtual Memory	328
Lecture No. 41	334
Numerical Examples of DRAM and Cache	334
Lecture No. 42	343
Performance of I/O Subsystems	343
Lecture No. 43	348
Networks	348
Lecture No. 44	353
Communication Medium and Network Topologies	353
Lecture No. 45	359
Review	359

Appendix A

Reading Material

Handouts

Summary

1. Introduction to FALSIM
2. Preparing source files for FALSIM
3. Using FALSIM
4. FALCON-A assembly language techniques

FALSIM

1. Introduction to FALSIM:

FALSIM is the name of the software application which consists of the FALCON-A assembler and the FALCON-A simulator. It runs under Windows XP.

FALCON-A Assembler:

Figure 1 shows a snapshot of the FALCON-A Assembler. This tool loads a FALCON-A assembly file with a (.asmfa) extension and parses it. It shows the parse results in an error log, lets the user view the assembled file's contents in the file listing and also provides the features of printing the machine code, an Instruction Table and a Symbol Table to a FALCON-A listing file. It also allows the user to run the FALCON-A Simulator.

The FALCON-A Assembler has two main modules, the 1st-pass and the 2nd-pass. The 1st-pass module takes an assembly file with a (.asmfa) extension and processes the file contents. It then creates a Symbol Table which corresponds to the storage of all program variables, labels and data values in a data structure at the implementation level. If the 1st-pass completes successfully a Symbol Table is produced as an output, which is used by the 2nd-pass module. Failures of the 1st-pass are handled by the assembler using its exception handling mechanism.

The 2nd-pass module sequentially processes the .asmfa file to interpret the instruction opcodes, register opcodes and constants using the symbol table. It then produces a list file with a .lstfa extension independent of successful or failed pass. If the pass is successful a binary file with a .binfa extension is produced which contains the machine code for the program in the assembly file.

FALCON-A Simulator:

Figure 6 shows a snapshot of the FALCON-A Simulator. This tool loads a FALCON-A binary file with a (.binfa) extension and presents its contents into different areas of the simulator. It allows the user to execute the program to a specific point within a time frame or just executes it, line by line. It also allows the user to view the registers, I/O port values and memory contents as the instructions execute.

FALSIM Features:

The FALCON-A Assembler provides its user with the following features:

Select Assembly File: Labeled as “1” in Figure 1, this feature enables the user to choose a FALCON-A assembly file and open it for processing by the assembler.

Assembler Options: Labeled as “2” in Figure 1.

- *Print Symbol Table*

This feature if selected writes the Symbol Table (produced after the execution of the 1st-pass of the assembler) to a FALCON-A list file with an extension of (.lstfa). The Symbol Table includes data members, data addresses and labels with their respective values.

- *Print Instruction Table*

This feature if selected writes the Instruction Table to a FALCON-A list file with an extension of (.lstfa).

List File: Labeled as “3”, in Figure 1, the List File feature gives a detailed insight of the FALCON-A listing file, which is produced as a result of the execution of the 1st and 2nd-pass. It shows the Program Counter value in hexadecimal and decimal formats along with the machine code generated for every line of assembly code. These values are printed when the 2nd-pass is completed.

Error Log: The Error Log is labeled as “4” in Figure 1. It informs the user about the errors and their respective details, which occurs in any of the passes of the assembler.

Search: Search is labeled as “5” in Figure 1 and helps the user to search for a certain input with the options of searching with “**match whole**” and “**match any**” parts of the string. The search also has the option of checking with/without considering “**case-sensitivity**”. It searches the List File area and highlights the search results using the yellow color. It also indicates the total number of matches found.

Start Simulator: This feature is labeled as “6” in Figure 1. The FALCON-A Simulator is run using the FALCON-A Assembler’s Start Simulator option. The FALCON-A Simulator is invoked by the user from the FALCON-A Assembler. Its features are detailed as follows:

Load Binary File: The button labeled as “11” in Figure 6, allows the user to choose and open a FALCON-A binary file with a (.binfa) extension. When a file is being loaded into the simulator all the register, constants (if any) and memory values are set.

Registers: The area labeled as “12” in Figure 6. enables, the user to see values present in different registers before during and after execution.

Instruction: This area is labeled as “13” in Figure 6 and contains the value of PC, address of an instruction, its representation in Assembly, the Register Transfer Language, the op-code and the instruction type.

I/O Ports: I/O ports are labeled as “14” in Figure 6. These ports are available for the user to enter input operation values and visualize output operation values whenever an I/O operation takes place in the program. The input value for an input operation is given by the user before an instruction executes. The output values are visible in the I/O port area once the instruction has successfully executed.

Memory: The memory is divided into 2 areas and is labeled as “15” in Figure 6, to facilitate the view of data stored at different memory locations before, during and after program execution.

Processor’s State: Labeled as “16” in Figure 6, this area shows the current values of the Instruction register and the Program Counter while the program executes.

Search: The search option for the FALCON-A simulator is labeled as “17” in Figure 6. This feature is similar to the way the search feature of the FALCON-A Assembler works. It offers to highlight the search string which goes as an input, with the “All “ and “ Part “ option. The results of the search are highlighted in the color yellow. It also indicates the total number of matches.

The following is a description of the options available on the button panel labeled as “18” in Figure 6.

Single Step: “Single Step” lets the user execute the program, one instruction at a time. The next instruction is not executed unless the user does a “single step” again. By default, the instruction to be executed will be the one next in the sequence. It changes if the user specifies a different PC value using the Change PC option (explained below).

Change PC: This option lets the user change the value of PC (Program Counter). By changing the PC the user can execute the instruction to which the specified PC points.

Execute: By choosing this button the user is able to execute the instructions with the options of execution with/without breakpoint insertion (refer to Fig. 5). In case of breakpoint insertion, the user has the option to choose from a list of valid breakpoint values. It also has the option to set a limit on the time for execution. This “Max Execution Time” option restricts the program execution to a time frame specified by the user, and helps the simulator in exception handling.

Change Register: Using the Change Register feature, the user can change the value present in a particular register.

Change Memory Word: This feature enables the user to change values present at a particular memory location.

Display Memory: Display Memory shows an updated memory area, after a particular memory location other than the pre-existing ones is specified by the user.

Change I/O: Allows the user to give an I/O port value if the instruction to be executed requires an I/O operation. Giving in the input in any one of the I/O ports areas before instruction execution, indicates that a particular I/O operation will be a part of the program and it will have an input from some source. The value given by the user indicates the input type and source.

Display I/O: Display I/O works in a manner similar to Display Memory. Here the user specifies the starting index of an I/O port. This features displays the I/O ports stating from the index specified.

2. Preparing source files for FALSIM:

In order to use the FALCON-A assembler and simulator, FALSIM, the source file containing assembly language statements and directives should be prepared according to the following guidelines:

- The source file should contain ASCII text only. Each line should be terminated by a carriage return. The extension **.asmfa** should be used with each file name. After assembly, a list file with the original filename and an extension **.lstfa**, and a binary file with an extension **.binfa** will be generated by FALSIM.
- Comments are indicated by a semicolon (;) and can be placed anywhere in the source file. The FALSIM assembler ignores any text after the semicolon.
- Names in the source file can be of one of the following types:
- Variables: These are defined using the **.equ** directive. A value must also be assigned to variables when they are defined.
- Addresses in the “data and pointer area” within the memory: These can be defined using the **.dw** or the **.sw** directive. The difference between these two directives is that when **.dw** is used, it is not possible to store any value in the memory. The integer after **.dw** identifies the number of memory words to be reserved starting at the current address. (The directive **.db** can be used to reserve bytes in memory.) Using the **.sw** directive, it is possible to store a constant or the value of a name in the memory. It is also possible to use pointers with this directive to specify addresses larger than 127. Data tables and jump tables can also be set up in the memory using this directive.
- Labels: An assembly language statement can have a unique label associated with it. Two assembly language statements cannot have the same name. Every label should have a colon (:) after it.
- Use the **.org 0** directive as the first line in the program. Although the use of this line is optional, its use will make sure that FALSIM will start simulation by picking up the first instruction stored at address 0 of the memory. (Address 0 is called the reset address of the processor). A jump [first] instruction can be placed at address 0, so that control is transferred to the first executable statement of the main program. Thus, the label first serves as the identifier of the “entry point” in the source file. The **.org** directive can also be used anywhere in the source file to force code at a particular address in the memory.
- Address 2 in the memory is reserved for the pointer to the Interrupt Service Routine (ISR). The **.sw** directive can be used to store the address of the first instruction in the ISR at this location.

- Address 4 to 125 can be used for addresses of data and pointers¹. However, the main program must start at address 126 or less², otherwise FALSIM will generate an error at the jump [first] instruction.
- The main program should be followed by any subprograms or procedures. Each procedure should be terminated with a ret instruction. The ISR, if any, should be placed after the procedures and should be terminated with the iret instruction.
- The last line in the source file should be the .end directive.
- The .equ directive can be used anywhere in the source file to assign values to variables.
- It is the responsibility of the programmer to make sure that code does not overwrite data when the assembly process is performed, or vice versa. As an example, this can happen if care is not exercised during the use of the .org directive in the source file.

3. Using FALSIM:

- To start FALSIM (the FALCON-A assembler and simulator), double click on the FALSIM icon. This will display the assembler window, as shown in the Figure 1.
- Select one or both assembler options shown on the top right corner of the assembler window labeled as “2”. If no option is selected, the symbol table and the instruction table will not be generated in the list (.lstfa) file.
- Click on the select assembly file button labeled as “1”. This will open the dialog box as shown in the Figure 2.
- Select the path and file containing the source program that is to be assembled.
- Click on the open button. FALSIM will assemble the program and generate two files with the same filename, but with different extensions. A list file will be generated with an extension .lstfa, and a binary (executable) file will be generated with an extension .binfa. FALSIM will also display the list file and any error messages in two separate panes, as shown in Figure 3.

¹ Any address between 4 and 14 can be used in place of the displacement field in load or store instructions. Recall that the displacement field is just 5 bits in the instruction word.

² This restriction is because of the fact that the immediate operand in the movi instruction must fit an 8-bit field in the instruction word.

- Double clicking on any error message highlights and displays the corresponding erroneous line in the program listing window pane for the user. This is shown in Figure 4. The highlight feature can also be used to display any text string, including statements with errors in them. If the assembler reported any errors in the source file, then these errors should be corrected and the program should be assembled again before simulation can be done. Additionally, if the source file had been assembled correctly at an earlier occasion, and a correct binary (.binfa) file exists, the simulator can be started directly without performing the assembly process.
- To start the simulator, click on the start simulation button labeled as “6”. This will open the dialog box shown in Figure 6.
- Select the binary file to be simulated, and click open as shown in Figure 7.
- This will open the simulation window with the executable program loaded in it as shown in Figure 8. The details of the different panes in this window were given in section 1 earlier. Notice that the first instruction at address 0 is ready for execution. All registers are initialized to 0. The memory contains the address of the ISR (i.e., 64h which is 100 decimal) at location 2 and the address of the printer driver at location 4. These two addresses are determined at assembly time in our case. In a real situation, these addresses will be determined at execution time by the operating system, and thus the ISR and the printer driver will be located in the memory by the operating system (called re-locatable code). Subsequent memory locations contain constants defined in the program.
- Click single step button labeled as “19”. FALSIM will execute the jump [main] instruction at address 0 and the PC will change to 20h (32 decimal), which is the address of the first instruction in the main program (i.e., the value of main).
- Although in a real situation, there will be many instructions in the main program, those instructions are not present in the dummy calling program. The first useful instruction is shown next. It loads the address of the printer driver in r6 from the pointer area in the memory. The registers r5 and r7 are also set up for passing the starting address of the print buffer and the number of bytes to be printed. In our dummy program, we bring these values in to these registers from the data area in the memory, and then pass these values to the printer driver using these two registers. Clicking on the single step button twice, executes these two instructions.
- The execution of the call instruction simulates the event of a print request by the user. This transfers control to the printer driver. Thus, when the call r4, r6 instruction is single stepped, the PC changes to 32h (50 decimal) for executing the first instruction in the printer driver.

- Double click on memory location 000A, which is being used for holding the PB (printer busy) flag. Enter a 1 and click the change memory button. This will store a 0001 in this location, indicating that a previous print job is in progress. Now **click single step and note that this value is brought from memory location 000E** into register r1. Clicking single step again will cause the jnz r1, [message] instruction to execute, and control will transfer to the message routine at address 0046h. The nop instruction is used here as a place holder.
- **Click again on the single step button. Note that when the ret r4 instruction executes, the value in r4 (i.e., 28h) is brought into the PC.** The blue highlight bar is placed on the next instruction after the call r4, r6 instruction in the main program. In case of the dummy calling program, this is the halt instruction.
- Double click on the value of the PC labeled as “20”. This will open a dialog box shown below. Enter a value of the PC (i.e., 26h) corresponding to the call r4, r6 instruction, so that it can be executed again. A “list” of possible PC values can also be pulled down using, and 0026h can be selected from there as well.



- Click single step again to enter the printer driver again.
- Change memory location 000A to a 0, and then single step the first instruction in the printer driver. This will bring a 0 in r1, so that when the next jnz r1, [message] instruction is executed, the branch will not be taken and control will transfer to the next instruction after this instruction. This is mivi r1, 1 at address 0036h.
- Continue single stepping.
- Notice that a 1 has been stored in memory location 000A, and r1 contains 11h, which is then transferred to the output port at address 3Ch (60 decimal) when the out r1, controlp instruction executes. This can be verified by double clicking on the top left corner of the I/O port pane, and changing the address to 3Ch. Another way to display the value of an I/O port is to scroll the I/O window pane to the desired position.
- Continue single stepping till the int instruction and note the changes in different panes of the simulation window at each step.

- When the int instruction executes, the PC changes to 64h, which is the address of the first instruction in the ISR. Clicking single step executes this instruction, and loads the address of temp (i.e., 0010h) which is a temporary memory area for storing the environment. The five store instructions in the ISR save the CPU environment (working registers) before the ISR change them.
- Single step through the ISR while noting the effects on various registers, memory locations, and I/O ports till the iredt instruction executes. This will pass control back to the printer driver by changing the PC to the address of the jump [finish] instruction, which is the next instruction after the int instruction.
- Double click on the value of the PC. Change it to point to the int instruction and click single step to execute it again. Continue to single step till the in r1, statusp instruction is ready for execution.
- Change the I/O port at address 3Ah (which represents the status port at address 58) to 80 and then single step the in r1, statusp instruction. The value in r1 should be 0080.
- Single step twice and notice that control is transferred to the movi r7, FFFF³ instruction, which stores an error code of -1 in r1.

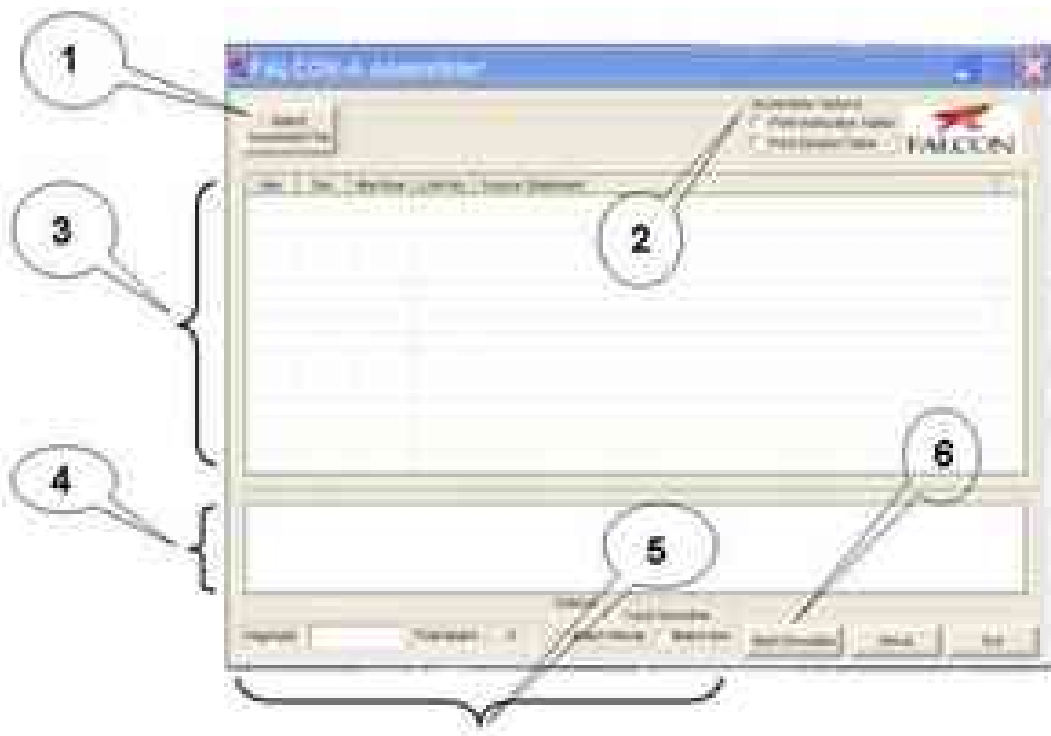


Figure 1

³ The instruction was originally **movi r7, -1**. Since it was converted to machine language by the assembler, and then reverse assembled by the simulator, it became **movi r7, FFFF**. This is

because the machine code stores the number in 16-bits after sign-extension. The result will be the same in both cases.

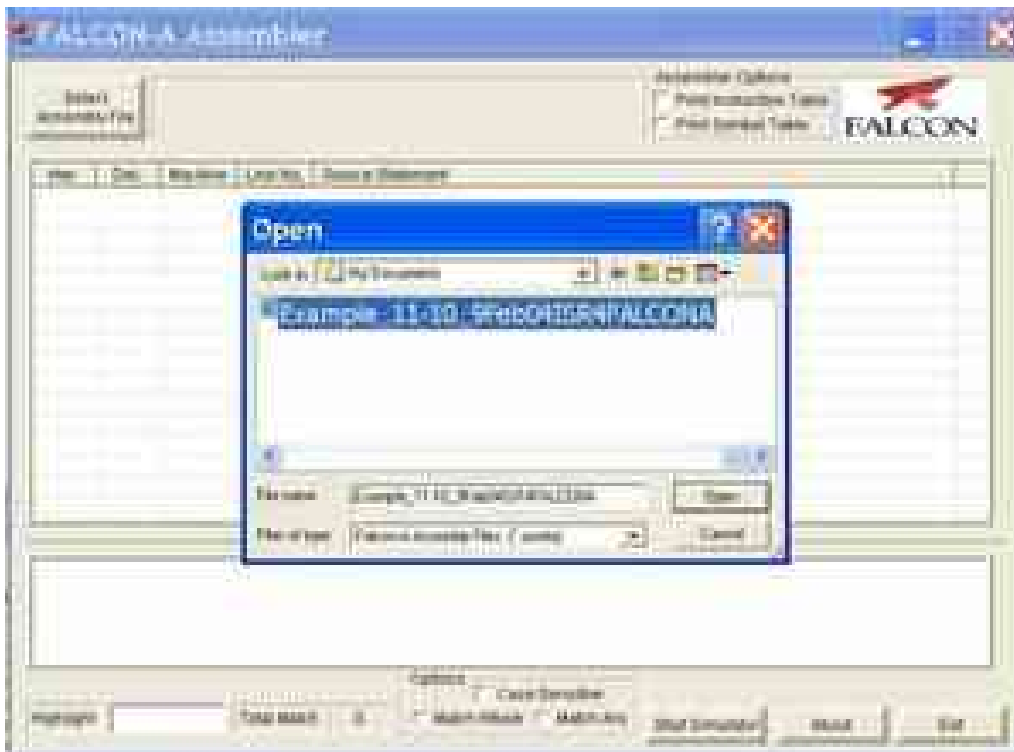


Figure 2

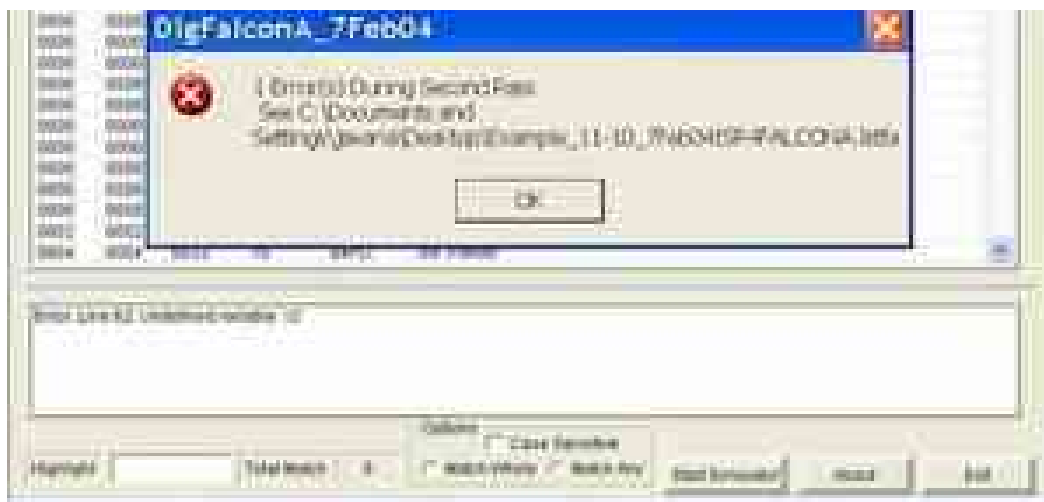


Figure 3

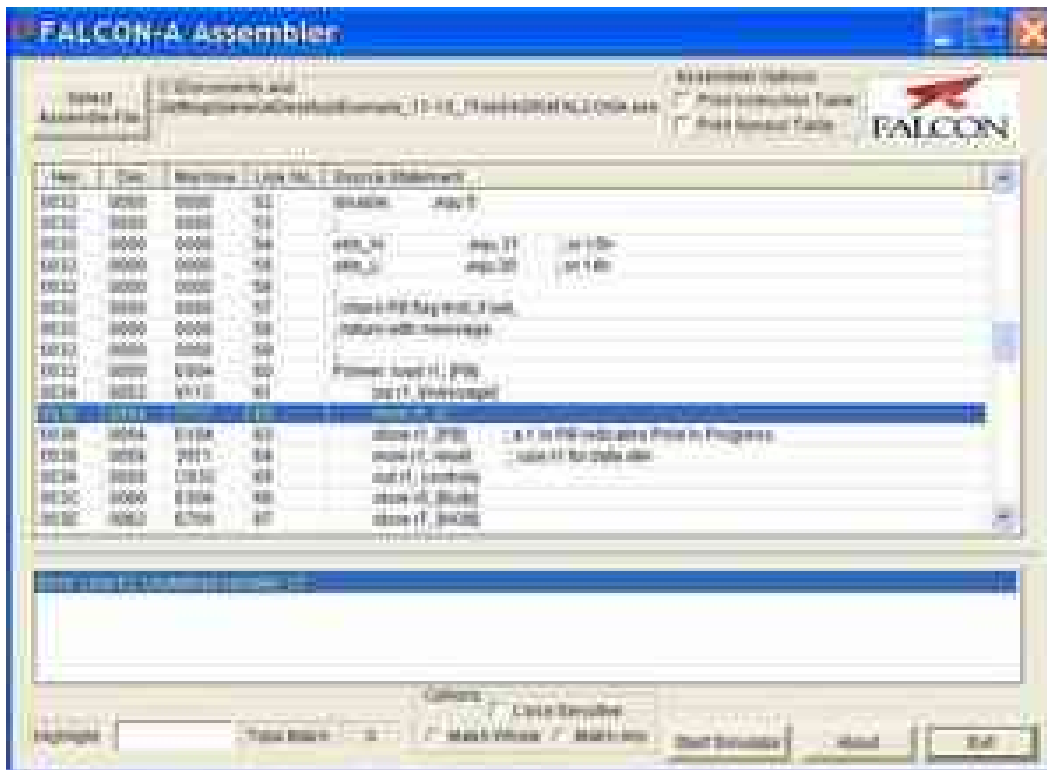


Figure 6

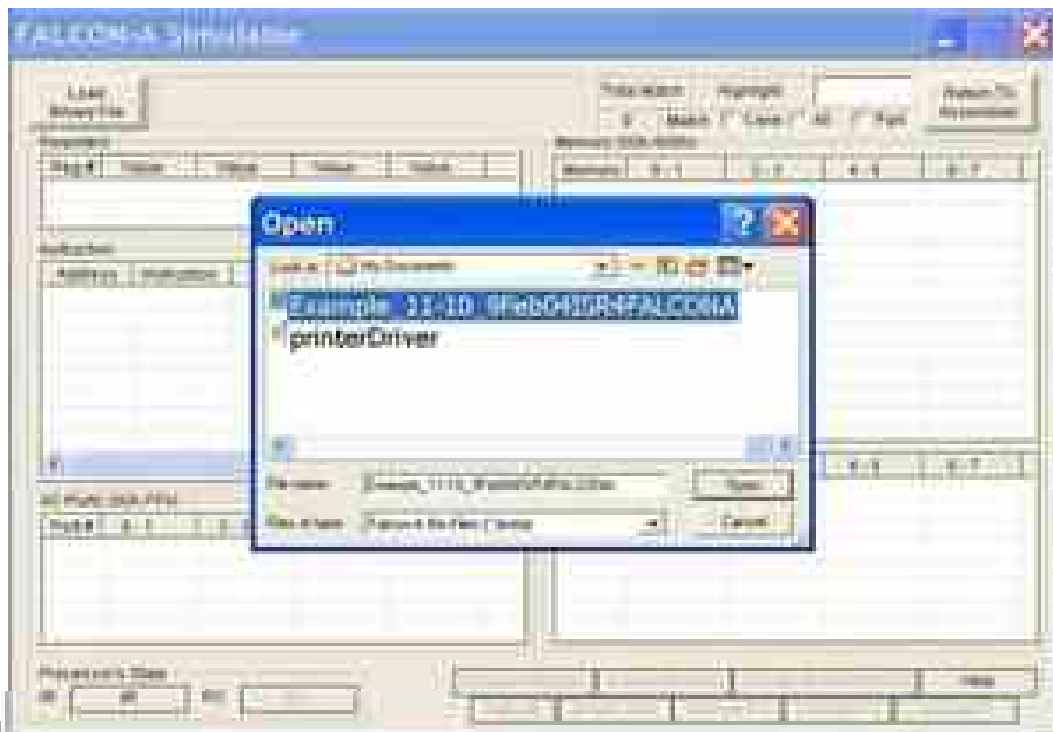


Figure 7

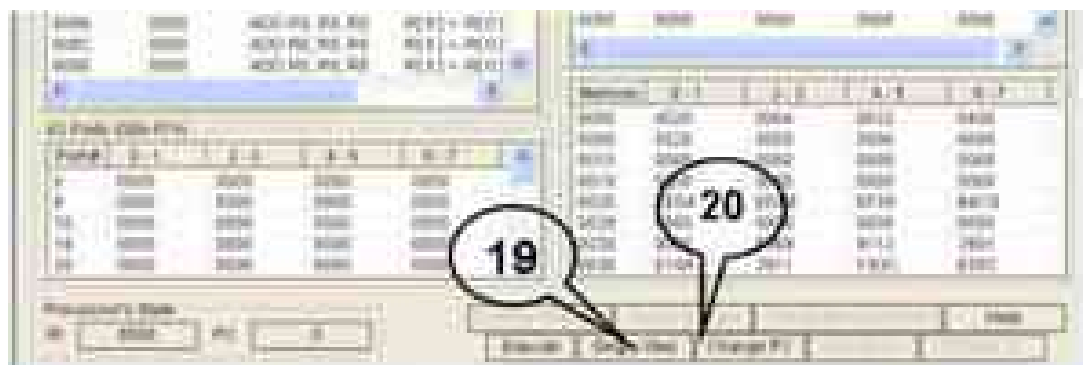


Figure 8

4. FALCON-A assembly language programming techniques:

- If a signed value, x , cannot fit in 5 bits (i.e., it is outside the range -16 to +15), FALSIM will report an error with a load $r1, [x]$ or a store $r1, [x]$ instruction. To overcome this problem, use `movi r2, x` followed by `load r1, [r2]`.
- If a signed value, x , cannot fit in 8 bits (i.e., it is outside the range - 128 to +127), even the previous scheme will not work. FALSIM will report an error with the `movi r2, x` instruction. The following instruction sequence should be used to overcome this limitation of the FALCON-A. First store the 16-bit address in the memory using the `.sw` directive. Then use two load instructions as shown below:

```
.sw x load r2, [a] load r1, [r2]
```

- This is essentially a “memory-register-indirect” addressing. It has been made possible by the `.sw` directive. The value of a should be less than 15.
- A similar technique can be used with immediate ALU instructions for large values of the immediate data, and with the transfer of control (call and jump) instructions for large values of the target address.
- Large values (16-bit values) can also be stored in registers using the **mul** instruction combined with the **addi** instruction. The following instructions bring a 201 in register $r1$.

```
movi r2, 10
movi r3, 20
mul r1, r2, r3      ;    r1 contains 200 after this instruction
addi r1, r1, 1      ;    r1 now contains 201
```

- Moving from one register to another can be done by using the instruction **addi r2, r1, 0**.
- Bit setting and clearing can be done using the logical (and, or, not, etc) instructions.
- Using shift instructions (`shifl`, `asr`, etc.) is faster than **mul** and **div**, if the multiplier or divisor is a power of 2.

Lecture No. 1

Introduction

Reading Material

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

Chapter 1
1.1, 1.2, 1.3, 1.4, 1.5

Summary

- Distinction between computer architecture, organization and design
- Levels of abstraction in digital design
- Introduction to the course topics
- Perspectives of different people about computers
- General operation of a stored program digital computer
- The Fetch-Execute process
- Concept of an ISA (Instruction Set Architecture)

Introduction

This course is about Computer Architecture. We start by explaining a few key terms.

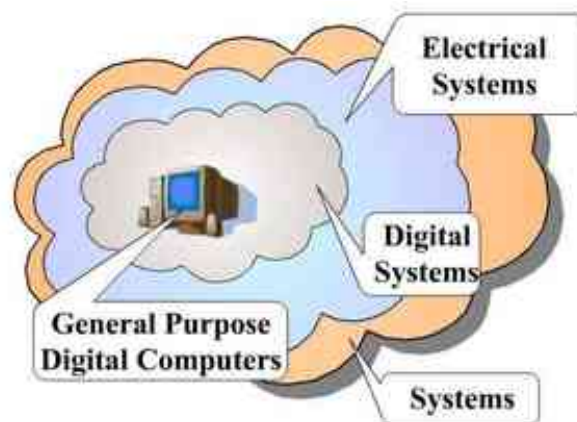
The General Purpose Digital Computer

How can we define a ‘computer’? There are several kinds of devices that can be termed “computers”: from desktop machines to the microcontrollers used in appliances such as a microwave oven, from the Abacus to the cluster of tiny chips used in parallel processors, etc. For the purpose of this course, we will use the following definition of a computer:

“An electronic device, operating under the control of instructions stored in its own memory unit, that can accept data (input), process data arithmetically and logically, produce output from the processing, and store the results for future use.” [1]

Thus, when we use the term computer, we actually mean a digital computer. There are many digital computers, which have dedicated purposes, for example, a computer is used in an automobile that controls the spark timing for the engine.

This means that when we use the term computer, we actually mean a general-purpose digital computer that can perform a variety of arithmetic and logic tasks.



Notion of a System

The Computer as a System

Now we examine the notion of a system, and the place of digital computers in the general universal set of systems. A “system” is a collection of elements, or components, working

together on one or more inputs to produce one or more desired outputs. There are many types of systems in the world. Examples include:

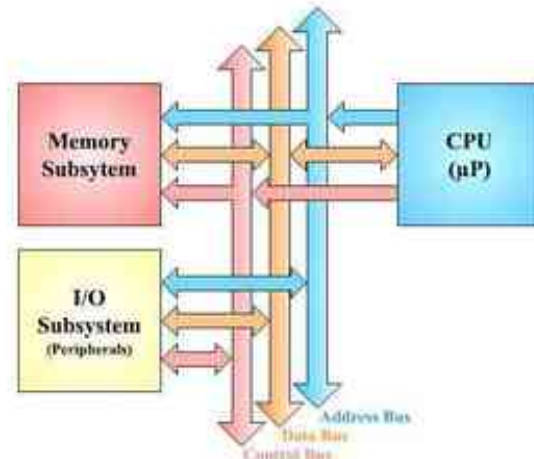
- Chemical systems
- Optical systems
- Biological systems
- Electrical systems
- Mechanical systems, etc.

These are all subsets of the general universal set of “systems”. One particular subset of interest is an “electrical system”. In case of electrical systems, the inputs as well as the outputs are electrical quantities, namely voltage and current. “Digital systems” are a subset of electrical systems. The inputs and outputs are digital quantities in this case. General-purpose digital computers are a subset of digital systems. We will focus on general-purpose digital computers in this course.

Essential Elements of a General Purpose Digital Computer

The figure shows the block diagram of a modern general-purpose digital computer.

We observe from the diagram that a general-purpose computer has three main components: a memory subsystem, an input/output subsystem, and a central processing unit. Programs are stored in the memory, the execution of the program instructions takes place in the CPU, and the communication with the external world is achieved through the I/O subsystem (including the peripherals).



Block Diagram of a Computer System

Architecture

Now that we understand the term “computer” in our context, let us focus on the term architecture. The word architecture, as defined in standard dictionaries, is *“the art or science of building”, or “a method or style of building”*. [2]

Computer Architecture

This term was first used in 1964 by Amdahl, Blaauw, and Brooks at IBM [3]. They defined it as *“The structure of a computer that a machine language programmer must understand to write a correct (time independent) program for that machine.”*

By architecture, they meant the programmer visible portion of the instruction set. Thus, a family of machines of the same architecture should be able to run the same software (instructions). This concept is now so common that it is taken for granted. The x86 architecture is a well-known example.

The study of computer architecture includes

- a study of the structure of a computer
- a study of the instruction set of a computer
- a study of the process of designing a computer

Computer Organization versus Computer Architecture

It is difficult to make a sharp distinction between these two. However, architecture refers to the attributes of a computer that are visible to a programmer, including

- The instruction set
- The number of bits used to represent various data types
- I/O mechanisms
- Memory addressing modes, etc.

On the other hand, organization refers to the operational units of a computer and their interconnections that realize the architectural specifications. These include

- The control signals
- Interfaces between the computer and its peripherals
- Memory technology used, etc.

It is an architectural issue whether a computer will have a specific instruction or not, while it is an organizational issue how that instruction will be implemented.

Computer Architect

We can conclude from the discussion above that a computer architect is a person who designs computers.

Design

Design is defined as

“The process of devising a system, component, or process to meet desired needs.”

Most people think of design as a “sketch”. This is the usage of the term as a noun. However, the standard engineering usage of the term, as is quite evident from the above definition, is as a verb, i.e., “design is a process”. A designer works with a set of stated requirements under a number of constraints to produce the best solution for a given problem. Best may mean a “cost-effective” solution, but not always. Additional or alternate requirements, like efficiency, the client or the designer may impose robustness, etc.. Therefore, design is a decision-making process (often iterative in nature), in which the basic sciences, mathematical concepts and engineering sciences are applied to convert a given set of resources optimally to meet a stated objective.

Knowledge base of a computer architect

There are many people in the world who know how to drive a car; these are the “users” of cars who are familiar with the behavior of a car and how to operate it. In the same way, there are people who can use computers. There are also a number of people in the world who know how to repair a car; these are “automobile technicians”. In the same way, we have computer technicians. However, there are a very few people who know how to design a car; these are “automobile designers”. In the same way, there are only very few experts in the world who can design computers. In this course, you will learn how to design computers!

These computer design experts are familiar with

- The structure of a computer
- The instruction set of a computer
- The process of designing a computer as well as few other related things.

At this point, we need to realize that it is not the job of a single person to design a computer from scratch. There are a number of levels of computer design. Domain experts of that particular level carry out the design activity for each level. These levels of abstraction of a digital computer's design are explained below.

Digital Design: Levels of Abstraction

Processor-Memory-Switch level (PMS level)

The highest is the processor-memory-switch level. This is the level at which an architect views the system. It is simply a description of the system components and their interconnections. The components are specified in the form of a block diagram.

Instruction Set Level

The next level is instruction set level. It defines the function of each instruction. The emphasis is on the behavior of the system rather than the hardware structure of the system.

Register Transfer Level

Next to the ISA (instruction set architecture) level is the register transfer level. Hardware structure is visible at this level. In addition to registers, the basic elements at this level are multiplexers, decoders, buses, buffers etc.

The above three levels relate to “system design”.

Logic Design Level

The logic design level is also called the gate level. The basic elements at this level are gates and flip-flops. The behavior is less visible, while the hardware structure predominates.

The above level relates to “logic design”.

Circuit Level

The key elements at this level are resistors, transistors, capacitors, diodes etc.

Mask Level

The lowest level is mask level dealing with the silicon structures and their layout that implement the system as an integrated circuit.

The above two levels relate to “circuit design”.

The focus of this course will be the register transfer level and the instruction set level, although we will also deal with the PMS level and the Logic Design Level.

Objectives of the course

This course will provide the students with an understanding of the various levels of studying computer architecture, with emphasis on instruction set level and register transfer level. They will be able to use basic combinational and sequential building blocks to design larger structures like ALUs (Arithmetic Logic Units), memory subsystems, I/O subsystems etc. It will help them understand the various approaches used to design computer CPUs (Central Processing Units) of the RISC (Reduced Instruction Set Computers) and the CISC (Complex Instruction Set Computers) type, as well as the principles of cache memories.

Important topics to be covered

- Review of computer organization
- Classification of computers and their instructions
- Machine characteristics and performance
- Design of a Simple RISC Computer: the SRC
- Advanced topics in processor design
- Input-output (I/O) subsystems
- Arithmetic Logic Unit implementation
- Memory subsystems

Course Outline Introduction: • Distinction between Computer Architecture, Organization and design • Levels of abstraction in digital design • Introduction to the course topics
Brief review of computer organization: • Perspectives of different people about computers • General operation of a stored program digital computer • The Fetch – Execute process • Concept of an ISA
Foundations of Computer Architecture: • A taxonomy of computers and their instructions • Instruction set features • Addressing Modes • RISC and CISC architectures • Measures of performance
An example processor: The SRC: • Introduction to the ISA and instruction formats • Coding examples and Hand assembly • Using Behavioral RTL to describe the SRC • Implementing Register Transfers using Digital Logic Circuits
ISA: Design and Development • Outline of the thinking process for ISA design • Introduction to the ISA of the FALCON – A • Solved examples for FALCON-A • Learning Aids for the FALCON-A
Other example processors: • FALCON-E • EAGLE and Modified EAGLE • Comparison of the four ISAs
CPU Design: • The Design Process • A Uni-Bus implementation for the SRC • Structural RTL for the SRC instructions • Logic Design for the 1-Bus SRC • The Control Unit • The 2-and 3-Bus Processor Designs • The Machine Reset • Machine Exceptions
Term Exam – I 
Advanced topics in processor design: • Pipelining • Instruction-Level Parallelism • Microprogramming

Input-output (I/O): • I/O interface design • Programmed I/O • Interrupt driven I/O • Direct memory access (DMA) Term Exam – II 
Arithmetic Logic Shift Unit (ALSU) implementation: • Addition, subtraction, multiplication & division for integer unit • Floating point unit
Memory subsystems: • Memory organization and design • Memory hierarchy • Cache memories • Virtual memory

References

- [1] Shelly G.B., Cashman T.J., Waggoner G.A., Waggoner W.C., Complete Computer Concepts: Microcomputer and Applications. Ferncroft Village Danvers, Massachusetts: Boyd & Fraser, 1992.
- [2] Merriam-Webster Online; The Language Centre, May 12, 2003 (<http://www.m-w.com/home.htm>).
- [3] Patterson, D.A. and Hennessy, J.L., Computer Architecture- A Quantitative Approach, 2nd ed., San Francisco, CA: Morgan Kauffman Publishers Inc., 1996.
- [4] Heuring V.P. and Jordan H.F., Computer Systems Design and Architecture. Melano Park, CA: Addison Wesley, 1997.

A brief review of Computer Organization Perceptions of Different People about Computers

There are various perspectives that a computer can take depending on the person viewing it. For example, the way a child perceives a computer is quite different from how a computer programmer or a designer views it. There are a number of perceptions of the computer, however, for the purpose of understanding the machine, generally the following four views are considered.

The User's View

A user is the person for whom the machine is designed, and who employs it to perform some useful work through application software. This useful work may be composing some reports in word processing software, maintaining credit history in a spreadsheet, or even developing some application software using high-level languages such as C or Java. The list of “useful work” is not all-inclusive. Children playing games on a computer may argue that playing games is also “useful work”, maybe more so than preparing an internal office memo.

At the user's level, one is only concerned with things like speed of the computer, the storage capacity available, and the behavior of the peripheral devices. Besides performance, the user is not involved in the implementation details of the computer, as the internal structure of the machine is made obscure by the operating system interface.

The Programmer's View

By “programmer” we imply machine or assembly language programmer. The machine or the assembly language programmer is responsible for the implementation of software required to execute various commands or sequences of commands (programs) on the computer. Understanding some key terms first will help us better understand this view, the associated tasks, responsibilities and tools of the trade.

Machine Language

Machine language consists of all the primitive instructions that a computer understands and is able to execute. These are strings of 1s and 0s. Machine language is the computer's native language. Commands in the machine language are expressed as strings of 1s and 0s. It is the lowest level language of a computer, and requires no further interpretation.

Instruction Set

A collection of all possible machine language commands that a computer can understand and execute is called its instruction set. Every processor has its own unique instruction set. Therefore, programs written for one processor will generally not run on another processor. This is quite unlike programs written in higher-level languages, which may be portable. Assembly/machine languages are generally unique to the processors on which they are run, because of the differences in computer architecture. Three ways to list instructions in an instruction set of a computer:

- by function categories
- by an alphabetic ordering of mnemonics
- by an ascending order of op-codes

Assembly Language

Since it is extremely tiring as well as error-prone to work with strings of 1s and 0s for writing entire programs, assembly language is used as a substitute symbolic representation using “English like” key words called mnemonics. A pure assembly language is a language in which each statement produces exactly one machine instruction, i.e. there is a one-to-one correspondence between machine instructions and statements in the assembly language. However, there are a few exceptions to this rule, the

Pentium jump instruction shown in the table below serves as an example.

Example

The table provides us with some assembly statement and the machine language equivalents of the Intel x 86 processor families.

Alpha is a label, and its value will be determined by the position of the jmp instruction in the program and the position of the instruction whose address is alpha. So the second byte in the last instruction can be different for different programs.

Hence there is a one-to-many correspondence of the assembly to machine language in this instruction.

Assembly Language	Machine Language (Binary)	Machine Language (Hex)	Instruction type
add cx, dx	0000 0001 1101 0001	01 D1	Arithmetic
mov alx, 34h	1011 1000 0011 0100 0000 0000	B8 34 00	Data transfer
xor ax, bx	0011 0001 1101 1000	31 D8	Logic
jmp alpha	1110 1011 1111 1100	EB FC	Control

Users of Assembly Language

- The machine designer

The designer of a new machine needs to be familiar with the instruction sets of other machines in order to be able to understand the trade-offs implicit in the design of those instruction sets.

- **The compiler writer**

A compiler is a program that converts programs written in high-level languages to machine language. It is quite evident that a compiler writer must be familiar with the machine language of the processor for which the compiler is being designed. This understanding is crucial for the design of a compiler that produces correct and optimized code. 

- **The writer of time or space critical code**

A compiler may not always produce optimal code. Performance goals may force program-specific optimizations in the assembly language.

- **Special purpose or embedded processor programmer**

Higher-level languages may not be appropriate for programming special purpose or embedded processors that are now in common use in various appliances. This is because the functionality required in such applications is highly specialized. In such a case, assembly language programming is required to implement the required functionality.

Useful tools for assembly language programmers

- **The assembler:**

Programs written in assembly language require translation to the machine language, and an assembler performs this translation. This conversion process is termed as the assembly process. The assembly process can be done manually as well, but it is very tedious and error-prone.

An “assembler” that runs on one processor and translates an assembly language program written for another processor into the machine language of the other processor is called a “cross assembler”.

- **The linker:**

When developing large programs, different people working at the same time can develop separate modules of functionality. These modules can then be ‘linked’ to form a single module that can be loaded and executed. The modularity of programs, that the linking step in assembly language makes possible, provides the same convenience as it does in higher-level languages; namely abstraction and separation of concerns. Once the functionality of a module has been verified for correctness, it can be re-used in any number of other modules. The programmer can focus on other parts of the program. This is the so-called “modular” approach, or the “top-down” approach.

- **The debugger or monitor:**

Assembly language programs are very lengthy and non-intuitive, hence quite tedious and error-prone. There is also the disadvantage of the absence of an operating system to handle run-time errors that can often crash a system, as opposed to the higher-level language programming, where control is smoothly returned to the operating system. In addition to run-time errors (such as a divide-by-zero error), there are syntax or logical errors.

A “debugger”, also called a “monitor”, is a computer program used to aid in detecting these errors in a program. Commonly, debuggers provide functionality such as

- The display and altering of the contents of memory, CPU registers and flags
- Disassembly of machine code (translating the machine code back to assembly language)
- Single stepping and breakpoints that allow the examination of the status of the program and registers at desired points during execution.

While syntax errors and many logical errors can be detected by using debuggers, the best debugger in the world can catch not every logical error.

- **The development system**

The development system is a complete set of (hardware and software) tools available to the system developer. It includes

- Assemblers
- Linkers and loaders

- Debuggers
- Compilers
- Emulators
- Hardware-level debuggers
- Logic analyzers, etc.

Difference between Higher-Level Languages and Assembly Language Higher-level languages are generally used to develop application software. These high-level programs are then converted to assembly language programs using compilers. So it is the task of a compiler writer to determine the mapping between the high-level-language constructs and assembly language constructs. Generally, there is a “many-to-many” mapping between high-level languages and assembly language constructs. This means that a given HLL construct can generally be represented by many different equivalent assembly language constructs. Alternately, a given assembly language construct can be represented by many different equivalent HLL constructs.

High-level languages provide various primitive data types, such as integer, Boolean and a string, that a programmer can use. Type checking provides for the verification of proper usage of these data types. It allows the compiler to determine memory requirements for variables and helping in the detection of bad programming practices.

On the other hand, there is generally no provision for type checking at the machine level, and hence, no provision for type checking in assembly language. The machine only sees strings of bits. Instructions interpret the strings as a type, and it is usually limited to signed or unsigned integers and floating point numbers. A given 32-bit word might be an instruction, an integer, a floating-point number, or 4 ASCII characters. It is the task of the compiler writer to determine how high-level language data types will be implemented using the data types available at the machine level, and how type checking will be implemented.

The Stored Program Concept

This concept is fundamental to all the general-purpose computers today. It states that the program is stored with data in computer’s memory, and the computer is able to manipulate it as data. For example, the computer can load the program from disk, move it around in memory, and store it back to the disk.

Even though all computers have unique machine language instruction sets, the ‘stored program’ concept and the existence of a ‘program counter’ is common to all machines. The sequence of instructions to perform some useful task is called a program. All of the digital computers (the general purpose machine defined above) are able to store these sequences of instructions as stored programs. Relevant data is also stored on the computer’s secondary memory. These stored programs are treated as data and the computer is able to manipulate them, for example, these can be loaded into the memory for execution and then saved back onto the storage.

General Operation of a Stored Program Computer

The machine language programs are brought into the memory and then executed instruction by instruction. Unless a branch instruction is encountered, the program is executed in sequence. The instruction that is to be executed is fetched from the memory and temporarily stored in a CPU register, called the instruction register (IR). The instruction register holds the instruction while it is decoded and executed by the central processing unit (CPU) of the computer. However, before loading an instruction into the instruction register for execution, the computer needs to know which instruction to load. The program counter (PC), also called the instruction pointer in some texts, is the register that holds the address of the next instruction in memory that is to be executed.

When the execution of an instruction is completed, the contents of the program counter (which is the address of the next instruction) are placed on the address bus. The memory places the instruction on the corresponding address on the data bus. The CPU puts this instruction onto the IR (**instruction register**) to decode and execute. While this instruction is decoded, its length in

bytes is determined, and the PC (**program counter**) is incremented by the length, so that the PC will point to the next instruction in the memory.

Note that the length of the instruction is not determined in the case of RISC machines, as the instruction length is fixed in these architectures, and so the program counter is always incremented by a fixed number. In case of branch instructions, the contents of the PC are replaced by the address of the next instruction contained in the present branch instruction, and the current status of the processor is stored in a register called the **Processor Status Word** (PSW). Another name for the PSW is the flag register. It contains the status bits, and control bits corresponding to the state of the processor. Examples of status bits include the sign bit, overflow bit, etc. Examples of control bits include interrupt enable flag, etc. When the execution of this instruction is completed, the contents of the program counter are placed on the address bus, and the entire cycle is repeated. This entire process of reading memory, incrementing the PC, and decoding the instruction is known as the **Fetch and Execute** principle of the stored program computer. This is actually an oversimplified situation. In case of the advanced processors of this age, a lot more is going on than just the simple “fetch and execute” operation, such as pipelining etc. The details of some of these more involved techniques will be studied later on during the course.

The Concept of Instruction Set Architecture (ISA)

Now that we have an understanding of some of the relevant key terms, we revert to the assembly language programmer's perception of the computer. The programmer's view is limited to the set of all the assembly instructions or commands that can the particular computer at hand execute understood/, in addition to the resources that these instructions may help manage. These resources include the memory space and the entire programmer accessible registers. Note that we use the term ‘memory space’ instead of memory, because not all the memory space has to be filled with memory chips for a particular implementation, but it is still a resource available to the programmer.

This set of instructions or operations and the resources together form the instruction set architecture (ISA). It is the ISA, which serves as an interface between the program and the functional units of a computer, i.e., through which, the computer's resources, are accessed and controlled.

The Computer Architect's View

The computer architect's view is concerned with the design of the entire system as well as ensuring its optimum performance. The optimality is measured against some quantifiable objectives that are set out before the design process begins. These objectives are set on the basis of the functionality required from the machine to be designed. The computer architect

- Designs the ISA for optimum programming utility as well as for optimum performance of implementation
- Designs the hardware for best implementation of instructions that are made available in the ISA to the programmer
- Uses performance measurement tools, such as benchmark programs, to verify that the performance objectives are met by the machine designed
- Balances performance of building blocks such as CPU, memory, I/O devices, and interconnections
- Strives to meet performance goals at the lowest possible cost
- Software models, simulators and emulators
- Performance benchmark programs
- Specialized measurement programs
- Data flow and bottleneck analysis
- Subsystem balance analysis

- Parts, manufacturing, and testing cost analysis

The Logic Designer's View

The logic designer is responsible for the design of the machine at the logic gate level. It is the design process at this level that determines whether the computer architect meets cost and performance goals. The computer architect and the logic designer have to work in collaboration to meet the cost and performance objectives of a machine. This is the reason why a single person or a single team may be performing the tasks of system's architectural design as well as the logic design.

Useful Tools for the Logic Designer

Some of the tools available that aid the logic designer in the logic design process are

- CAD tools

Logic design and simulation packages Printed circuit layout tools

IC (integrated circuit) design and layout tools

- Logic analyzers and oscilloscopes
- Hardware development systems

The Concept of the Implementation Domain

The collection of hardware devices, with which the logic designer works for the digital logic gate implementation and interconnection of the machine, is termed as the implementation domain. The logic gate implementation domain may be

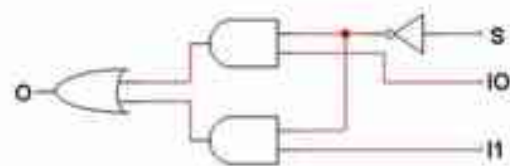
- VLSI (very large scale integration) on silicon
- TTL (transistor-transistor logic) or ECL (emitter-coupled logic) chips
- Gallium arsenide chips
- PLAs (programmable-logic arrays) or sea-of-gates arrays
- Fluidic logic or optical switches

Similarly, the implementation domains used for gate, board and module interconnections are

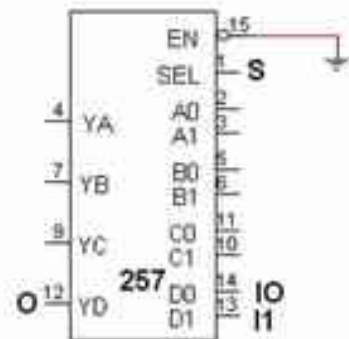
- Poly-silicon lines in ICs
- Conductive traces on a printed circuit board
- Electrical cable
- Optical fiber, etc.

At the lower levels of logic design, the designer is concerned mainly with the functional details represented in a symbolic form. The implementation details are not considered at these lower levels. They only become an issue at higher levels of logic design. An example of a two-to-one multiplexer in various implementation domains will illustrate this point. Figure (a) is the generic logic gate (abstract domain) representation of a 2-to-1 multiplexer.

Figure (b) shows the 2-to-1 multiplexer logic gate implementation in the domain of TTL (VLSI on Silicon) logic using part number '257, with interconnections in the domain of printed circuit board traces.



(a) Abstract view of Boolean logic

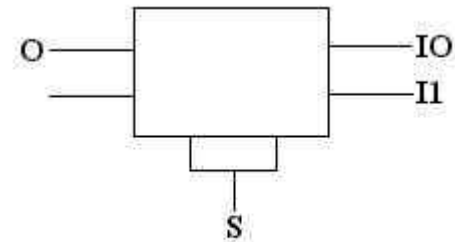


(b) TTL implementation domain

Figure (c) is the implementation of the 2-to-1 multiplexer with a fiber optic directional coupler switch, which has an interconnection domain of optical fiber.

Classical logic design versus computer logic design

We have already studied the sequential circuit design concepts in the course on Digital Logic Design, and thus are familiar with the techniques used. However, these traditional techniques for a finite state machine are not very practical when it comes to the design of a computer, in spite of the fact that a computer is a finite state machine. The reason is that employing these techniques is much too complex as the computer can assume hundreds of states.



(c) Optical switch implementation

Sequential Logic Circuit Design

When designing a sequential logic circuit, the problem is first coded in the form of a state diagram. The redundant states may be eliminated, and then the state diagram is translated into the next state table. The minimum number of flip-flops needed to implement the design is calculated by making “state assignments” in terms of the flip-flop “states”. A “transition table” is made using the state assignments and the next state table. The flip-flop control characteristics are used to complete a set of “excitation tables”. The excitation equations are determined through minimization. The logic circuit can then be drawn to implement the design. A detailed discussion of these steps can be found in most books on Logic Design.

Computer Logic Design

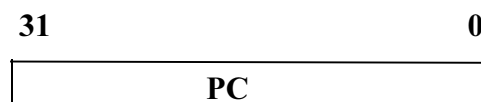
Traditional Finite State Machine (FSM) design techniques are not suitable for the design of computer logic. Since there is a natural separation between the data path and the control path in case of a digital computer, a modular approach can be used in this case.

The data path consists of the storage cells, the arithmetic and logic components and their interconnections. Control path is the circuitry that manages the data path information flow. So considering the behavior first can carry out the design. Then the structure can be considered and dealt with. For this purpose, well-defined logic blocks such as multiplexers, decoders, adders etc. can be used repeatedly.

Two Views of the CPU Program Counter Register

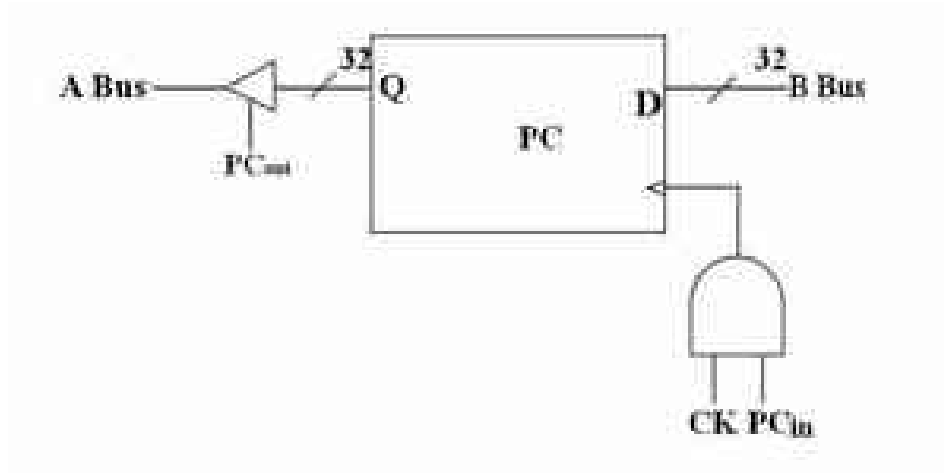
The view of a logic designer is more detailed than that of a programmer. Details of the mechanism used to control the machine are unimportant to the programmer, but of vital importance to the logic designer. This can be illustrated through the following two views of the program counter of a machine.

As shown in figure (a), to a programmer the program counter is just a register, and in this case, of length 32 bits or 4 bytes.



(a) Program Counter: Programmer’s view

Figure (b) illustrates the logic designer's view of a 32-bit program counter, implemented as an array of 32 D flip-flops. It shows the contents of the program counter being gated out on 'A bus' (the address bus) by applying a control signal PC_{out} . The contents of the 'B bus' (also the address bus), can be stored in the program counter by asserting the signal PC_{in} on the leading edge of the clock signal CK, thus storing the address of the next instruction in the program counter.



(b) Program Counter: Logic Designer's View

Lecture No. 2

Instruction Set Architecture

Reading Material

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

Chapter 2, Chapter 3
2.1, 2.2, 3.2

Summary

- A taxonomy of computers and their instructions
- Instruction set features
- Addressing modes
- RISC and CISC architectures

Foundations of Computer Architecture

Taxonomy of computers and their instructions

Processors can be classified on the basis of their instruction set architectures. The instruction set architecture, described in the previous module gives us a ‘programmer’s view’ of the machine. This module discussed a number of topics related to the classifications of computers and their instructions.

CLASSES OF INSTRUCTION SET ARCHITECTURE:

The mechanism used by the CPU to store instructions and data can be used to classify the ISA (Instruction Set Architecture). There are three types of machines based on this classification.

- Accumulator based machines
- Stack based machines
- General purpose register (GPR) machines

ACCUMULATOR BASED MACHINES

Accumulator based machines use special registers called the accumulators to hold one source operand and also the result of the arithmetic or logic operations performed. Thus the accumulator registers collect (or ‘accumulate’) data. Since the accumulator holds one of the operands, one more register may be required to hold the address of another operand. The accumulator is not used to hold an address. So accumulator based machines are also called 1-address machines. Accumulator machines employ a very small number of accumulator registers, generally only one. These machines were useful at the time when memory was quite expensive; as they used one register to hold the source operand as well as the result of the operation. However, now that the memory is relatively inexpensive, these are not considered very useful, and their use is severely limited for the computation of expressions with many operands.

STACK BASED MACHINES

A stack is a group of registers organized as a last-in-first-out (LIFO) structure. In such a structure, the operands stored first, through the push operation, can only be accessed last, through

a pop operation; the order of access to the operands is reverse of the storage operation. An analogy of the stack is a “plate-dispenser” found in several self-service cafeterias. Arithmetic and logic operations successively pick operands from the top-of-the-stack (TOS), and push the results on the TOS at the end of the operation. In stack based machines, operand addresses need not be specified during the arithmetic or logical operations. Therefore, these machines are also called 0-address machines.

GENERAL-PURPOSE-REGISTER MACHINES

In general purpose register machines, a number of registers are available within the CPU. These registers do not have dedicated functions, and can be employed for a variety of purposes. To identify the register within an instruction, a small number of bits are required in an instruction word. For example, to identify one of the 64 registers of the CPU, a 6-bit field is required in the instruction.

CPU registers are faster than cache memory. Registers are also easily and more effectively used by the compiler compared to other forms of internal storage. Registers can also be used to hold variables, thereby reducing memory traffic. This increases the execution speed and reduces code size (fewer bits required to code register names compared to memory). In addition to data, registers can also hold addresses and pointers (i.e., the address of an address). This increases the flexibility available to the programmer.

A number of dedicated, or special purpose registers are also available in general-purpose machines, but many of them are not available to the programmer. Examples of transparent registers include the stack pointer, the program counter, memory address register, memory data register and condition codes (or flags) register, etc.

We should understand that in reality, most machines are a combination of these machine types. Accumulator machines have the advantage of being more efficient as these can store intermediate results of an operation within the CPU.

INSTRUCTION SET

An instruction set is a collection of all possible machine language commands that are understood and can be executed by a processor.

ESSENTIAL ELEMENTS OF COMPUTER INSTRUCTIONS:

There are four essential elements of an instruction; the type of operation to be performed, the place to find the source operand(s), the place to store the result(s) and the source of the next instruction to be executed by the processor.

Type of operation

In module 1, we described three ways to list the instruction set of a machine; one way of enlisting the instruction set is by grouping the instructions in accordance with the functions they perform. The type of operation that is to be performed can be encoded in the op-code (or the operation code) field of the machine language instruction. Examples of operations are mov, jmp, add; these are the assembly mnemonics, and should not be confused with op-codes. Op-codes are simply bit-patterns in the machine language format of an instruction.

Place to find source operands

An instruction needs to specify the place from where the source operands will be retrieved and used. Possible locations of the source operands are CPU registers, memory cells and I/O locations. The source operands can also be part of an instruction itself; such operands are called immediate operands.

Place to store the results

An instruction also specifies the location in which the result of the operation, specified by the instruction, is to be stored. Possible locations are CPU registers, memory cells and I/O locations.

Source of the next instruction

By default, in a program the next instruction in sequence is executed. So in cases where the next-in-sequence instruction execution is desired, the place of next instruction need not be encoded within the instruction, as it is implicit. However, in case of a branch, this information needs to be encoded in the instruction. A branch may be conditional or unconditional, a subroutine call, as well as a call to an interrupt service routine.

Example

The table provides examples of assembly language commands and their machine language equivalents. In the instruction `add cx, dx`, the contents of the location `dx` are added to the contents of the location `cx`, and the result is stored in `cx`. The instruction type is arithmetic, and the op-code for the `add` instruction is 0000, as shown in this example.

Assembly Language	Machine Language (Binary)	Machine Language (Hex)	Instruction type
<code>add cx, dx</code>	0000 0001 1101 0001	01 D1	Arithmetic
<code>mov al, 34h</code>	1011 1000 0011 0100 0000 0000	B8 34 00	Data transfer
<code>xor ax, bx</code>	0011 0001 1101 1000	31 D8	Logic
<code>jmp alpha</code>	1110 1011 1111 1100	EB FC	Control

CLASSIFICATIONS OF INSTRUCTIONS:

We can classify instructions according to the format shown below.

- 4-address instructions
- 3-address instructions
- 2-address instructions
- 1-address instructions
- 0-address instructions

The distinction is based on the fact that some operands are accessed from memory, and therefore require a memory address, while others may be in the registers within the CPU or they are specified implicitly.

4-address instructions

The four address instructions specify the addresses of two source

op code	destination	source 1	source 2	next address
---------	-------------	----------	----------	--------------

operands, the address of the destination operand and the next instruction address.

4-address instructions are not very common because the next instruction to be executed is sequentially stored next to the current instruction in the memory. Therefore, specifying its address is redundant. These instructions are used in the micro-coded control unit, which will be studied later.

op code	destination	source 1	source 2
---------	-------------	----------	----------

3-address instruction

A 3-address instruction specifies the addresses of two operands and the address of the destination operand.

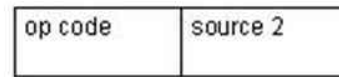
2-address instruction

op code	destination source 1	source 2
---------	-------------------------	----------

A 2-address instruction has three fields; one for the op-code, the second field specifies the address of one of the source operands as well as the destination operand, and the last field is used for holding the address of the second source operand. So one of the fields serves two purposes; specifying a source operand address and a destination operand address.

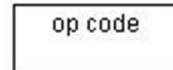
1-address instruction

A 1-address instruction has a dedicated CPU register, called the accumulator, to hold one operand and to store the result. There is no need of encoding the address of the accumulator register to access the operand or to store the result, as its usage is implicit. There are two fields in the instruction, one for specifying a source operand address and a destination operand address.



0-address instruction

A 0-address instruction uses a stack to hold both the operands and the result. Operations are performed on the operands stored on the top of the stack and the second value on the stack. The result is stored on the top of the stack. Just like the use of an accumulator register, the addresses of the stack registers need not be specified, their usage is implicit. Therefore, only one field is required in 0-address instruction; it specifies the op-code.



COMPARISON OF INSTRUCTION FORMATS:

Basis for comparison

Two parameters are used as the basis for comparison of the instruction sets discussed above. These are

- Code size

Code size has an effect on the storage requirements for the instructions; the greater the code size, the larger the memory required.

- Number of memory accesses

The number of memory accesses has an effect on the execution time of instructions; the greater the numbers of memory accesses, the larger the time required for the execution cycle, as memory accesses are generally slow.

Assumptions

We make a few assumptions, which are

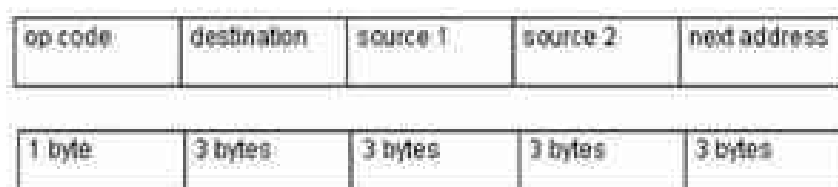
- A single byte is used for the op code, so 256 instructions can be encoded using these 8 bits, as $2^8 = 256$
- The size of the memory address space is 16 Mbytes
- A single addressable memory unit is a byte
- Size of operands is 24 bits. As the memory size is 16Mbytes, with byte-addressable memory, 24 bits are required to encode the address of the operands.
- The size of the address bus is 24 bits
- Data bus size is 8 bits

Discussion

4-address instruction

- The code size is 13 bytes (1+3+3+3+3 = 13 bytes)

- Number of bytes accessed from memory is 22 (13 bytes for instruction fetch + 6 bytes for source operand fetch + 3 bytes for storing destination operand = 22 bytes)



Note that there is no need for an additional memory access for the operand corresponding to the next instruction, as it has already been brought into the CPU during instruction fetch.

3-address instruction

- The code size is 10 bytes

(1+3+3+3 = 10 bytes)

- Number of bytes accessed from memory is 19 (10 bytes for instruction fetch 6 bytes for source operand fetch + 3 bytes for storing destination operand = 19 bytes)

op code	destination	source 1	source 2
1 byte	3 bytes	3 bytes	3 bytes

2-address instruction

- The code size is 7 bytes (1+3+3 = 7 bytes)

- Number of bytes accessed from memory is 16 (7 bytes for instruction fetch + 6 bytes for source operand fetch + 3 bytes for storing destination operand = 16 bytes)

op code	destination source 1	source 2
1 byte	3 bytes	3 bytes

1-address instruction

- The code size is 4 bytes (1+3= 4 bytes)

- Number of bytes accessed from memory is 7 (4 bytes for instruction fetch + 3 bytes for source operand fetch + 0 bytes for storing destination operand = 7 bytes)

op code	source 2
1 byte	3 bytes

0-address instruction

- The code size is 1 byte

- Number of bytes accessed from memory is 10 (1 byte for instruction fetch + 6 bytes for source operand fetch + 3 bytes for storing destination operand = 10 bytes)

op code
1 byte

The following table summarizes this information

Instruction Format	Code size	Number of memory bytes
4-address instruction	10	19
3-address instruction	7	16
2-address instruction	4	7
1-address instruction	1	10

HALF ADDRESSES

In the preceding discussion we have talked about memory addresses. This discussion also applies to CPU registers. However, to specify/ encode a CPU register, less number of bits is required as compared to the memory addresses. Therefore, these addresses are also called “half-addresses”. An instruction that specifies one memory address and one CPU register can be called as a 1½-address instruction

Example

mov al, [34h]

THE PRACTICAL SITUATION

Real machines are not as simple as the classifications presented above. In fact, these machines have a mixture of 3, 2, 1, 0, and 1½-address instructions. For example, the VAX 11 includes instructions from all classes.

CLASSIFICATION OF MACHINES ON THE BASIS OF OPERAND AND RESULT LOCATION:

A distinction between machines can be made on the basis of the ALU instructions; whether these instructions use data from the memory or not. If the ALU instructions use only the CPU registers for the operands and result, the machine type is called “load-store”. Other machines may have a mixture of register-memory, or memory-memory instructions.

The number of memory operands supported by a typical ALU instruction may vary from 0 to 3.

Example

The SPARC, MIPS, Power PC, ALPHA: 0 memory addresses, max operands allowed = 3

X86, 68x series: 1 memory address, max operands allowed = 2

LOAD- STORE MACHINES

These machines are also called the register-to-register machines. They typically use the 1½ address instruction format. Only the load and store instructions can access the memory. The load instruction fetches the required data from the memory and temporarily stores it in the CPU registers. Other instructions may use this data from the CPU registers. Then later, the results can be stored back into the memory by the store instruction. Most RISC computers fall under this category of machines.

Advantages (of register-register instructions)

Register-register instructions use 0 memory operands out of a total of 3 operands. The advantages of such a scheme is:

- The instructions are simple and fixed in length
- The corresponding code generation model is simple
- All instructions take similar number of clock cycles for execution

Disadvantages (register-register instructions)

- The instruction count is higher; the number of instructions required to complete a particular task is more as separate instructions will be required for load and store operations of the memory
- Since the instruction size is fixed, the instructions that do not require all fields waste memory bits

Register-memory machines

In register-memory machines, some operands are in the memory and some are in registers. These machines typically employ 1 or 1½ address instruction format, in which one of the operands is an accumulator or a general-purpose CPU registers.

Advantages

Register-memory operations use one memory operand out of a total of two operands. The advantages of this instruction format are

- Operands in the memory can be accessed without having to load these first through a separate load instruction
- Encoding is easy due to the elimination of the need of loading operands into registers first
- Instruction bit usage is relatively better, as more instructions are provided per fixed number of bits

Disadvantages

- Operands are not equivalent since one operand may have two functions (both source operand and destination operand), and the source operand may be destroyed
- Different size encoding for memory and registers may restrict the number of registers

- The number of clock cycles per instruction execution vary, depending on the operand location operand fetch from memory is slow as compared to operands in CPU registers

Memory-Memory Machines

In memory-memory machines, all three of the operands (2 source operands and a destination operand) are in the memory. If one of the operands is being used both as a source and a destination, then the 2-address format is used. Otherwise, memory-memory machines use 3-address formats of instructions.

Advantages

- The memory-memory instructions are the most compact instruction where encoding wastage is minimal.
- As operands are fetched from and stored in the memory directly, no CPU registers are wasted for temporary storage

Disadvantages

- The instruction size is not fixed; the large variation in instruction sizes makes decoding complex
- The cycles per instruction execution also vary from instruction to instruction
- Memory accesses are generally slow, so too many references cause performance degradation

Example 1

The expression $a = (b+c)*d - e$ is evaluated with the 3, 2, 1, and 0-address machines to provide a comparison of their advantages and disadvantages discussed above.

3-Address	2-Address	1-Address	0-Address
add a, b, c mpy a, a, d sub a, a, e	load a, b add a, c mpy a, d sub a, e	lda b add c mpy d sub e sla a	push b push c add push d mpy push e sub pop a

The instructions shown in the table are the minimal instructions required to evaluate the given expression. Note that these are not machine language instructions, rather the pseudo-code. 

Example 2

The instruction $z = 4(a + b) - 16(c+58)$ is with the 3, 2, 1, and 0-address machines in the table.

Functional classification of instruction sets:

Instructions can be classified into the following four categories based on their functionality.

- Data processing
- Data storage (main memory)
- Data movement (I/O)
- Program flow control

Data processing

Data processing instructions are the ones that perform some mathematical or logical operation on some operands. The Arithmetic Logic

3-Address	2-Address	1-Address	0-Address
add x, a, b mul y, x, 4 add r, c, 58 mul a, r, 16 sub z, y, i	load y, i add y, b mul y, 4 load a, c add a, 58 mul a, 16 lda z sub y, a store z, y	<i>; order changed to reduce code size</i> lda c adda 58 mula 16 sta a lda z adda b mula 4 suba a sta z <i>; add b to acc</i> <i>; subtract acc from a</i>	push a push b add push 4 mul push c push 58 add push 16 mul sub pop z

Unit performs these operations; therefore the data processing instructions can also be called ALU instructions.

- **Data storage (main memory)**

The primary storage for the operands is the main memory. When an operation needs to be performed on these operands, these can be temporarily brought into the CPU registers, and after completion, these can be stored back to the memory. The instructions for data access and storage between the memory and the CPU can be categorized as the data storage instructions.

- **Data movement (I/O)**

The ultimate sources of the data are input devices e.g. keyboard. The destination of the data is an output device, for example, a monitor, etc. The instructions that enable such operations are called data movement instructions.

- **Program flow control**

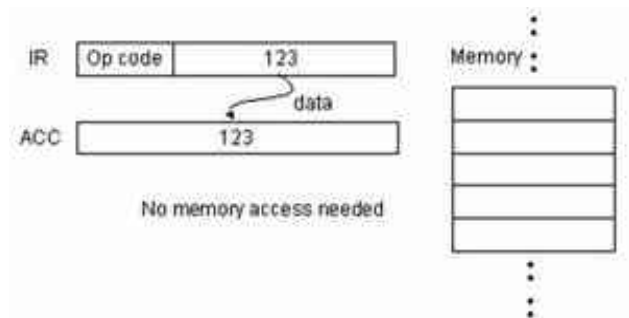
A CPU executes instructions sequentially, unless a program flow-change instruction is encountered. This flow change, also called a branch, may be conditional or unconditional. In case of a conditional branch, if the branch condition is met, the target address is loaded into the program counter.

ADDRESSING MODES:

Addressing modes are the different ways in which the CPU generates the address of operands. In other words, they provide access paths to memory locations and CPU registers.

Effective address

An “effective address” is the address (binary bit pattern) issued by the CPU to the memory. The CPU may use various ways to compute the effective address. The memory may interpret the effective address differently under different situations.



COMMONLY USED

ADDRESSING MODES

Some commonly used addressing modes are explained below.

Immediate addressing mode

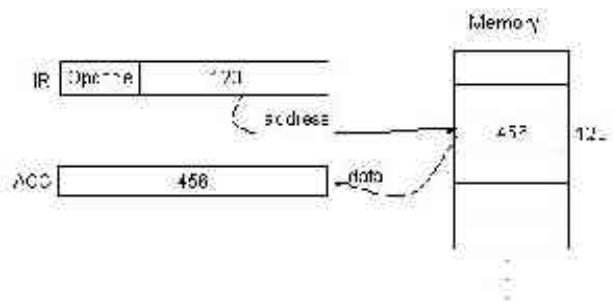
In this addressing mode, data is the part of the instruction itself, and so there is no need of address calculation. However, immediate addressing mode is used to hold source operands only; cannot be used for storing results. The range of the operands is limited by the number of bits available for encoding the operands in the instruction; for n bit fields, the range is $-2^{(n-1)}$ to $+2^{(n-1)}-1$.

Example: lda 123

In this example, the immediate operand, 123, is loaded onto the accumulator. No address calculation is required.

Direct Addressing Mode

The address of the operand is specified as a constant, and this constant is coded as part of the instruction. The address space that can be accessed is limited address space by the operand field size ($2^{\text{operand field size}}$ locations).



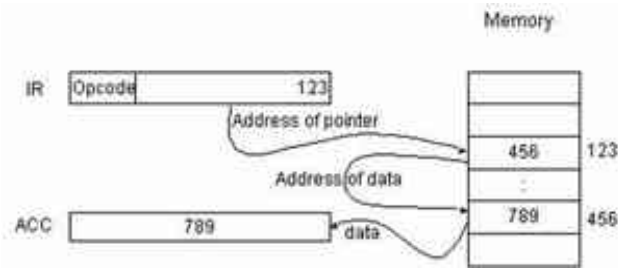
Example: `lda [123]`

As shown in the figure, the address of the operand is stored in the instruction. The operand is then fetched from that memory address.

Indirect Addressing Mode

The address of the location where the address of the data is to be found is stored in the instruction as the operand.

Thus, the operand is the address of a memory location, which holds the address of the operand. Indirect addressing mode can access a large address space ($2^{\text{memory word size}}$ locations).



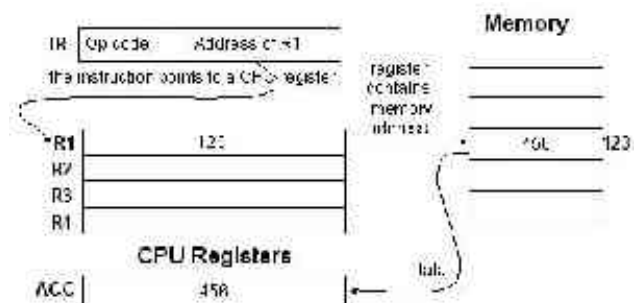
To fetch the operand in this addressing mode, two memory accesses are required. Since memory accesses are slow, this is not efficient for frequent memory accesses. The indirect addressing mode may be used to implement pointers.

Example: `lda [[123]]`

As shown in the figure, the address of the memory location that holds the address of the data in the memory is part of the instruction.

Register (Direct) Addressing Mode

The operand is contained in a CPU register, and the address of this register is encoded in the instruction. As no memory access is needed, operand fetch is efficient. However, there are only a limited number of CPU registers available, and this imposes a limitation on the use of this addressing mode.

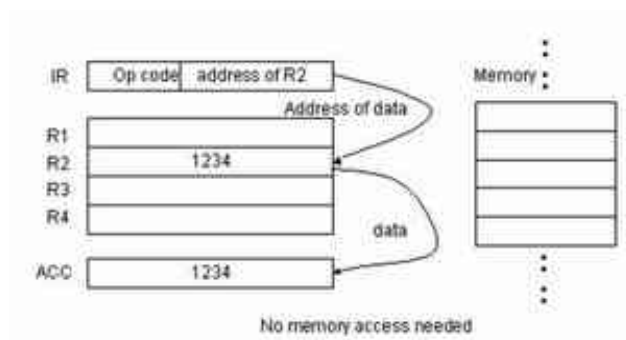


Example: `lda R2`

This load instruction specifies the address of the register and the operand is fetched from this register. As is clear from the diagram, no memory access is involved in this addressing mode.

REGISTER INDIRECT ADDRESSING MODE

In the register indirect mode, the address of memory location that contains the operand is in a CPU register. The address of this CPU register is encoded in the instruction. A large address space can be accessed using this addressing mode ($2^{\text{register size}}$ locations). It involves fewer memory accesses compared to indirect addressing.



Example: `lda [R1]`

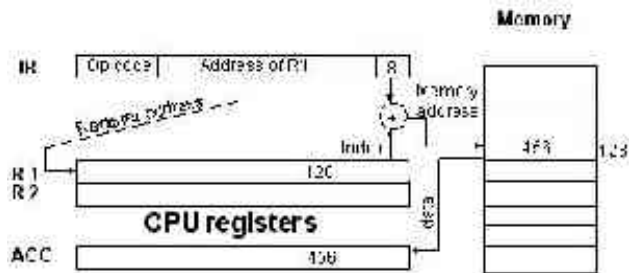
The address of the register that contains the address of memory location holding the operand is encoded in the instruction. There is one memory access involved.

Displacement addressing mode

The displacement-addressing mode is also called based or indexed addressing mode. Effective memory address is calculated by adding a constant (which is usually a part of the instruction) to the value in a CPU register. This addressing mode is useful for accessing arrays. The addressing mode may be called 'indexed' in the situation when the constant refers to the first element of the array (base) and the register contains the 'index'. Similarly, 'based' refers to the situation when the constant refers to the offset (displacement) of an array element with respect to the first element. The address of the first element is stored in a register.

Example: lda [R1 + 8]

In this example, R1 is the address of the register that holds a memory address, which is to be used to calculate the effective address of the operand. The constant (8) is added to this address held by the register and this effective address is used to retrieve the operand.

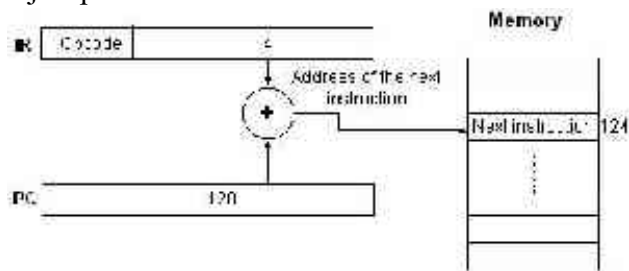


Relative addressing mode

The relative addressing mode is similar to the indexed addressing mode with the exception that the PC holds the base address. This allows the storage of memory operands at a fixed offset from the current instruction and is useful for 'short' jumps.

Example: jump 4

The constant offset (4) is a part of the instruction, and it is added to the address held by the Program Counter.



RISC and CISC architectures:

Generally, computers can be classified as being RISC machines or CISC machines. These concepts are explained in the following discussion.

RISC (Reduced instruction set computers)

RISC is more of a philosophy of computer design than a set of architectural features. The underlying idea is to reduce the number and complexity of instructions. However, new RISC machines have some instructions that may be quite complex and the number of instructions may also be large. The common features of RISC machines are

- **One instruction per clock period**

This is the most important feature of the RISC machines. Since the program execution depends on throughput and not on individual execution time, this feature is achievable by using pipelining and other techniques. In such a case, the goal is issuing an average of one instruction per cycle without increasing the cycle time.

- **Fixed size instructions**

Generally, the size of the instructions is 32 bits.

- **CPU accesses memory only for Load and Store operations**

This means that all the operands are in the CPU registers at the time these are used in an instruction. For this purpose, they are first brought into the CPU registers from the memory and later stored back through the load and store operation respectively.

- **Simple and few addressing modes**

The disadvantage associated with using complex addressing modes is that complex decoding is required to calculate these addresses, which reduces the processor performance as it takes significant time. Therefore, in RISC machines, few simple addressing modes are used.

- **Less work per instruction**

As the instructions are simple, less work is done per instruction, and hence the clock period T can be reduced.

- **Improved usage of delay slots**

A ‘delay slot’ is the waiting time for a load or store operation to access memory or for a branch instruction to access the target instruction. RISC designs allow the execution of the next instruction after these instructions are issued. If the program or compiler places an instruction in the delay slot that does not depend on the result of the previous instruction, the delay slot can be used efficiently. For the implementation of this feature, improved compilers are required that can check the dependencies of instructions before issuing them to utilize the delay slots.

- **Efficient usage of Pre-fetching and Speculative Execution Techniques**

Pre-fetching and speculative execution techniques are used with a pipelined architecture. Instruction pipelining means having multiple instructions in different stages of execution as instructions are issued before the previous instruction has completed its execution; pipelining will be studied in detail later. The RISC machines examine the instructions to check if operand fetches or branch instructions are involved. In such a case, the operands or the branch target instructions can be ‘pre-fetched’. As instructions are issued before the preceding instructions have completed execution, the processor will not know in case of a conditional branch instruction, whether the condition will be met and the branch will be taken or not. But instead of waiting for this information to be available, the branch can be “speculated” as taken or not taken, and the instructions can be issued. Later if the speculation is found to be wrong, the results can be discarded and actual target instructions can be issued. These techniques help improve the performance of processors.

CISC (Complex Instruction Set Computers)

The complex instruction set computers does not have an underlying philosophy. The CISC machines have resulted from the efforts of computer designers to efficiently utilize memory and minimize execution time, yet add in more instruction formats and addressing modes. The common attributes of CISC machines are discussed below.

- **More work per instruction**

This feature was very useful at the time when memory was expensive as well as slow; it allows the execution of compact programs with more functionality per instruction.

- **Wide variety of addressing modes**

CISC machines support a number of addressing modes, which helps reduce the program instruction count. There are 14 addressing modes in MC68000 and 25 in MC68020.

- **Variable instruction lengths and execution times per instruction**

The instruction size is not fixed and so the execution times vary from instruction to instruction.

- **CISC machines attempt to reduce the “semantic gap”**

‘Semantic gap’ is the gap between machine level instruction sets and high-level language constructs. CISC designers believed that narrowing this gap by providing complicated instructions and complex-addressing modes would improve performance. The concept did not work because compiler writers did not find these “improvements” useful. The following are some of the disadvantages of CISC machines.

- **Clock period T , cannot be reduced beyond a certain limit**

When more capabilities are added to an instruction the CPU circuits required for the execution of these instructions become complex. This results in more stages of logic circuitry and adds propagation delays in signal paths.

This in turn places a limit on the smallest possible value of T and hence, the maximum value of clock frequency.

Advance Computer Architecture – CS501

- **Complex addressing modes delay operand fetch from memory**

The operand fetch is delayed because more time is required to decode complex instructions.

- **Difficult to make efficient use of speedup techniques**

These speedup techniques include

- Pipelining
- Pre-fetching (Intel 8086 has a 6 byte queue)
- Super scalar operation
- Speculative execution

Lecture No. 3

Introduction to SRC Processor

Reading Material

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

Chapter2, Chapter 3
2.3, 2.4, 3.1

Summary

- Measures of performance
- Introduction to an example processor SRC
- SRC Notation
- SRC features and instruction formats

Measures of performance:

Performance testing

To test or compare the performance of machines, programs can be run and their execution times can be measured. However, the execution speed may depend on the particular program being run, and matching it exactly to the actual needs of the customer can be quite complex. To overcome this problem, standard programs called “benchmark programs” have been devised. These programs are intended to approximate the real workload that the user will want to run on the machine. Actual execution time can be measured by running the program on the machines.

Commonly used measures of performance

The basic measure of performance of a machine is time. Some commonly used measures of this time, used for comparison of the performance of various machines, are

- Execution time
- MIPS
- MFLOPS
- Whetstones
- Dhrystones
- SPEC

Execution time

Execution time is simply the time it takes a processor to execute a given program. The time it takes for a particular program depends on a number of factors other than the performance of the CPU, most of which are ignored in this measure. These factors include waits for I/O, instruction fetch times, pipeline delays, etc.

The execution time of a program with respect to the processor, is defined as

$$\text{Execution Time} = \text{IC} \times \text{CPI} \times \text{T}$$

Where,

IC = instruction count

CPI = average number of system clock periods to execute an instruction T = clock period

Strictly speaking, (IC×CPI) should be the sum of the clock periods needed to execute each instruction. The manufacturers for each instruction in the instruction set usually provide such information. Using the average is a simplification.

MIPS (Millions of Instructions per Second)

Another measure of performance is the millions of instructions that are executed by the processor per second. It is defined as

$$\text{MIPS} = \text{IC} / (\text{ET} \times 10^6)$$

This measure is not a very accurate basis for comparison of different processors. This is because of the architectural differences of the machines; some machines will require more instructions to perform the same job as compared to other machines. For example, RISC machines have simpler instructions, so the same job will require more instructions. This measure of performance was popular in the late 70s and early 80s when the VAX 11/780 was treated as a reference.

MFLOPS (Millions of Floating Point Instructions per Second)

For computation intensive applications, the floating-point instruction execution is a better measure than the simple instructions. The measure MFLOPS was devised with this in mind. This measure has two advantages over MIPS:

- Floating point operations are complex, and therefore, provide a better picture of the hardware capabilities on which they are run
- Overheads (operand fetch from memory, result storage to the memory, etc.) are effectively lumped with the floating point operations they support

Whetstones

Whetstone is the first benchmark program developed specifically as a benchmark program for performance measurement. Named after the Whetstone Algol compiler, this benchmark program was developed by using the statistics collected during the compiler development. It was originally an Algol program, but it has been ported to FORTRAN, Pascal and C. This benchmark has been specifically designed to test floating point instructions. The performance is stated in MWIPS (millions of Whetstone instructions per second).

Dhrystones

Developed in 1984, this is a small benchmark program to measure the integer instruction performance of processors, as opposed to the Whetstone's emphasis on floating point instructions. It is a very small program, about a hundred high-level-language statements, and compiles to about 1~ 1½ kilobytes of code.

Disadvantages of using Whetstones and Dhrystones

Both Whetstones and Dhrystones are now considered obsolete because of the following reasons.

- Small, fit in cache
- Obsolete instruction mix
- Prone to compiler tricks
- Difficult to reproduce results
- Uncontrolled source code

We should note that both the Whetstone and Dhrystone benchmarks are small programs, which encourage 'over-optimization', and can be used with optimizing compilers to distort results.

SPEC

SPEC, System Performance Evaluation Cooperative, is an association of a number of computer companies to define standard benchmarks for fair evaluation and comparison of different processors. The standard SPEC benchmark suite includes:

- A compiler
- A Boolean minimization program
- A spreadsheet program

- A number of other programs that stress arithmetic processing speed the latest version of these benchmarks is SPEC CPU2000.

Advantages

- It provides for ease of publication.
- Each benchmark carries the same weight.
- SPEC ratio is dimensionless.
- It is not unduly influenced by long running programs.
- It is relatively immune to performance variation on individual benchmarks.
- It provides a consistent and fair metric.

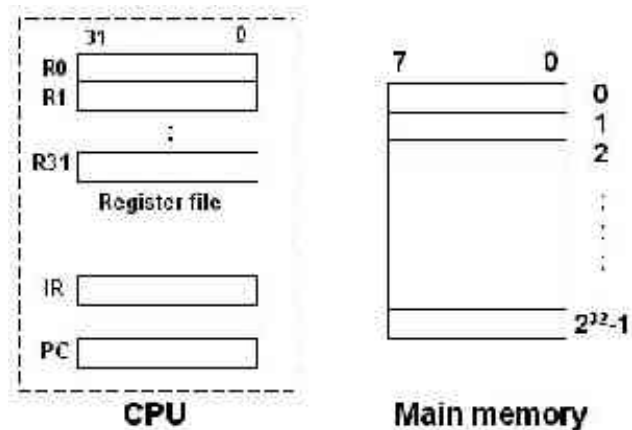
An example computer: the SRC: “simple RISC computer”

An example machine is introduced here to facilitate our understanding of various design steps and concepts in computer architecture. This example machine is quite simple, and leaves out a lot of details of a real machine, yet it is complex enough to illustrate the fundamentals.

SRC Introduction

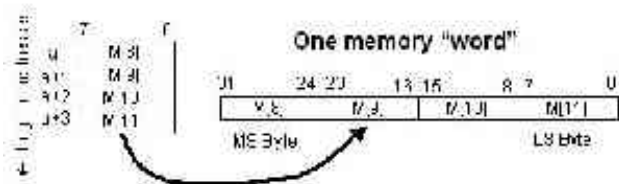
Attributes of the SRC

- The SRC contains 32 General Purpose Registers: R0, R1, ..., R31; each register is of size 32-bits.
- Two special purpose registers are included: Program Counter (PC) and Instruction Register (IR)
- Memory word size is 32 bits
- Memory space size is 2^{32} bytes
- Memory organization is $2^{32} \times 8$ bits, this means that the memory is byte aligned
- Memory is accessed in 32 bit words (i.e., 4 byte chunks)
- Big-endian byte storage is used



Programmer's View of the SRC

The figure shows the attributes of the SRC; the 32, 32-bit registers that are a part of the CPU, the two additional CPU registers (PC & IR), and the main memory which is 2^{32} 1-byte cells.



SRC Notation

We examine the notation used for the SRC with the help of some examples.

- R[3] means contents of register
- 3 (R for register)
- M[8] means contents of memory location 8 (M for memory)
- A memory word at address 8 is defined as the 32 bits at address 8,9,10 and 11 in the memory. This is shown in the figure.
- A special notation for 32-bit memory words is $M[8]<31...0>:=M[8] M[9] M[10] M[11]$ is used for concatenation.

Some more SRC Attributes

- All instructions are 32 bits long (i.e., instruction size is 1 word)
- All ALU instructions have three operands
- The only way to access memory is through load and store operations
- Only a few addressing modes are supported

SRC: Instruction Formats

Four types of instructions are supported by the SRC. Their representation is given in the figure shown.

Before discussing these instruction types in detail, we take a look at the encoding of general purpose registers (the ra, rb and rc fields).



Encoding of the General Purpose Registers

The encoding for the general purpose registers is shown in the table; it will be used in place of ra, rb and rc in the instruction formats shown above. Note that this is a simple 5 bit encoding. ra, rb and rc are names of fields used as “place-holders”, and can represent any one of these 32 registers. An exception is rb = 0; it does not mean the register R0, rather it means no operand. This will be explained in the following discussion.

Register	Code	Register	Code	Register	Code	Register	Code
R0	00000	R8	01000	R16	10000	R24	11000
R1	00001	R9	01001	R17	10001	R25	11001
R2	00010	R10	01010	R18	10010	R26	11010
R3	00011	R11	01011	R19	10011	R27	11011
R4	00100	R12	01100	R20	10100	R28	11100
R5	00101	R13	01101	R21	10101	R29	11101
R6	00110	R14	01110	R22	10110	R30	11110
R7	00111	R15	01111	R23	10111	R31	11111

Type A

Type A is used for only two instructions:

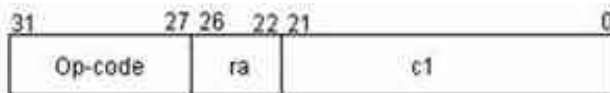
- No operation or nop, for which the op-code = 0. This is useful in pipelining
- Stop operation stop, the op-code is 31 for this instruction.

Both of these instructions do not need an operand (are 0-operand instructions).



Type B

Type B format includes three instructions; all three use relative addressing mode. These are

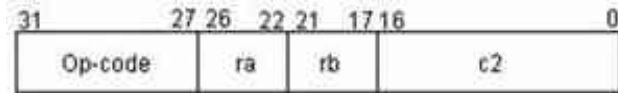


- The ldr instruction, used to load register from memory using a relative address. (op-code = 2).
 - Example:
 - ldr R3, 56
 - This instruction will load the register R3 with the contents of the memory location M [PC+56]
- The lar instruction, for loading a register with relative address (op-code = 6)

- Example: `lar R3, 56`
This instruction will load the register R3 with the relative address itself (PC+56).

- The `str` is used to store register to memory using relative address (op-code = 4)
 - Example: `str R8, 34`
This instruction will store the register R8 contents to the memory location $M[PC+34]$

The effective address is computed at run-time by adding a constant to the PC. This makes the instructions 're-locatable'.



Type C

Type C format has three load/store instructions, plus three ALU instructions. These load/store instructions are

- `ld`, the load register from memory instruction (op-code = 1)
 - Example 1:
`ld R3, 56`
This instruction will load the register R3 with the contents of the memory location $M[56]$; the `rb` field is 0 in this instruction, i.e., it is not used. This is an example of direct addressing mode.
 - Example 2:
`ld R3, 56(R5)`
The contents of the memory location $M[56+R[5]]$ are loaded to the register R3; the `rb` field $\neq 0$. This is an instance of indexed addressing mode.
- `la` is the instruction to load a register with an immediate data value (which can be an address) (op-code = 5)
 - Example 1:
`la R3, 56`
The register R3 is loaded with the immediate value 56. This is an instance of immediate addressing mode.
 - Example 2:
`la R3, 56(R5)`
The register R3 is loaded with the indexed address $56+R[5]$. This is an example of indexed addressing mode.
- The `st` instruction is used to store register contents to memory (op-code = 3)
 - Example 1: `st R8, 34`
This is the direct addressing mode; the contents of register R8 ($R[8]$) are stored to the memory location $M[34]$
 - Example 2: `st R8, 34(R6)`
An instance of indexed addressing mode, $M[34+R[6]]$ stores the contents of $R8(R[8])$

The ALU instructions are

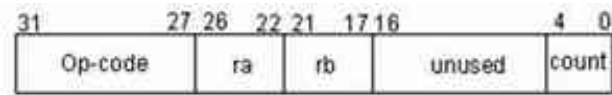
- `addi`, immediate 2's complement addition (op-code = 13)
 - Example:
`addi R3, R4, 56`
 $R[3] \leftarrow R[4] + 56$ (*rb field = R4*)

- **andi**, the instruction to obtain immediate logical AND, (op-code = 42)
 - Example: `andi R3, R4, 56`
- R3 is loaded with the immediate logical AND of the contents of register R4 and 56 (constant value)
- **ori**, the instruction to obtain immediate logical OR (op-code = 23)
 - Example: `ori R3, R4, 56`
- R3 is loaded with the immediate logical OR of the contents of register R4 and 56(constant value)

Note:

1. Since the constant c2 field is 17 bits,
 - For direct addressing mode, only the first 2^{16} bytes of memory can be accessed (or the last 2^{16} bytes if c2 is negative)
 - In case of the la instruction, only constants with magnitudes less than $\pm 2^{16}$ can be loaded
 - During address calculation using c2, sign extension to 32 bits must be performed before the addition

2. **Type C instructions**, with some modifications, may also be used for shift instructions. Note the modification in the following figure.



The four shift instructions are

- **shr** is the instruction used to shift the bits right by using value in (5-bit) c3 field(shift count) (op-code = 26)
 - Example: `shr R3, R4, 7`
shift R4 right 7 times in to R3. Immediate addressing mode is used.
- **shra**, arithmetic shift right by using value in c3 field (op-code = 27)
 - Example: `shra R3, R4, 7`
This instruction has the effect of shift R4 right 7 times in to R3. Immediate addressing mode is used.
- The **shl** instruction is for shift left by using value in (5-bit) c3 field (op-code = 28)
 - Example: `shl R8, R5, 6`
shift R5 left 6 times in to R8. Immediate addressing mode is used.
- **shc**, shift left circular by using value in c3 field (op-code = 29)
 - Example: `shc R3, R4, 3`
shift R4 circular 3 times in to R3. Immediate addressing mode is used.

Lecture No. 4

ISA and Instruction Formats

Reading Material

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

Chapter 2
2.3, 2.4, slides

Summary

- Introduction to ISA and instruction formats
- Coding examples and Hand assembly

An example computer: the SRC: “simple RISC computer”

An example machine is introduced here to facilitate our understanding of various design steps and concepts in computer architecture. This example machine is quite simple, and leaves out a lot of details of a real machine, yet it is complex enough to illustrate the fundamentals.

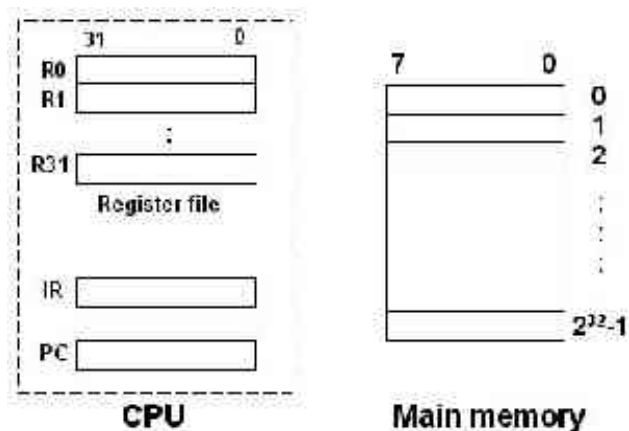
SRC Introduction

Attributes of the SRC

- The SRC contains 32 General Purpose Registers: R0, R1... R31; each register is of size 32-bits.
- Two special purpose registers are included: Program Counter (PC) and Instruction Register (IR)
- Memory word size is 32 bits
- Memory space size is 2^{32} bytes
- Memory organization is $2^{32} \times 8$ bits, this means that the memory is byte aligned
- Memory is accessed in 32 bit words (i.e., 4 byte chunks)
- Big-endian byte storage is used

Programmer's View of the SRC

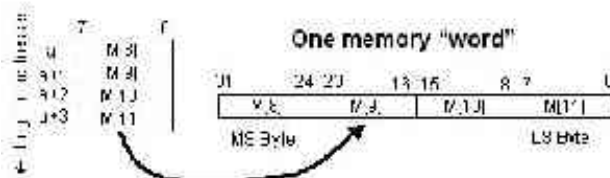
The figure below shows the attributes of the SRC; the 32, 32-bit registers that are a part of the CPU, the two additional CPU registers (PC & IR), and the main memory which is 2^{32} 1-byte cells.



SRC Notation

We examine the notation used for the SRC with the help of some examples.

- R[3] means contents of register 3 (R for register)
- M[8] means contents of memory location 8 (M for memory)



- A memory word at address 8 is defined as the 32 bits at address 8,9,10 and 11 in the memory. This is shown in the figure below.
- A special notation for 32-bit memory words is $M[8]<31...0>:=M[8] M[9] M[10] M[11]$ is used for concatenation.

Some more SRC Attributes

- All instructions are 32 bits long (i.e., instruction size is 1 word)
- All ALU instructions have three operands
- The only way to access memory is through load and store operations
- Only a few addressing modes are supported

SRC: Instruction Formats

Four types of instructions are supported by the SRC. Their representation is given in the following figure. Before discussing these instruction types in detail, we take a look at the encoding of general-purpose registers (the ra, rb and rc fields).



Encoding of the General Purpose Registers

The encoding for the general purpose registers is shown in the following table; it will be used in place of ra, rb and rc in the instruction formats shown above. Note that this is a simple 5 bit encoding. ra, rb and rc are names of fields used as “place-holders”, and can represent any one of these 32 registers. An exception is $rb = 0$; it does not mean the register R0, rather it means no operand. This will be explained in the following discussion.

Register	Code	Register	Code	Register	Code	Register	Code
R0	00000	R16	01000	R32	10000	R48	11000
R1	00001	R17	01001	R33	10001	R49	11001
R2	00010	R18	01010	R34	10010	R50	11010
R3	00011	R19	01011	R35	10011	R51	11011
R4	00100	R20	01100	R36	10100	R52	11100
R5	00101	R21	01101	R37	10101	R53	11101
R6	00110	R22	01110	R38	10110	R54	11110
R7	00111	R23	01111	R39	10111	R55	11111

Type A

Type A is used for only two instructions:

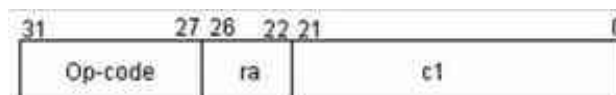
- No operation or nop, for which the op-code = 0. This is useful in pipelining
- Stop operation stop, the op-code is 31 for this instruction.



Both of these instructions do not need an operand (are 0-operand instructions).

Type B

Type B format includes three instructions; all three use relative addressing mode.

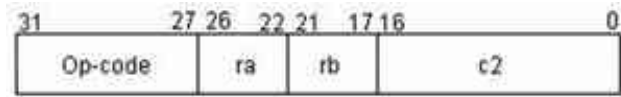


These are

- The **ldr** instruction, used to load register from memory using a relative address. (op-code = 2).
 - Example: **ldr R3, 56**
This instruction will load the register R3 with the contents of the memory location **M [PC+56]**
- The **lar** instruction, for loading a register with relative address (op-code = 6)
 - Example: **lar R3, 56**
This instruction will load the register R3 with the relative address itself (PC+56).
- The **str** is used to store register to memory using relative address (op-code = 4)
 - Example: **str R8, 34**
This instruction will store the register R8 contents to the memory location **M [PC+34]**
- The effective address is computed at run-time by adding a constant to the PC. This makes the instructions 're-locatable'.

Type C

Type C format has three load/store instructions, plus three ALU instructions. These load/ store instructions are



- **ld**, the load register from memory instruction (op-code = 1)
 - Example 1: **ld R3, 56**
This instruction will load the register R3 with the contents of the memory location **M [56]**; the **rb** field is 0 in this instruction, i.e., it is not used. This is an example of direct addressing mode.
 - Example 2: **ld R3, 56(R5)**
The contents of the memory location **M [56+R [5]]** are loaded to the register R3; the **rb** field $\neq 0$. This is an instance of indexed addressing mode.
- **la** is the instruction to load a register with an immediate data value (which can be an address) (op-code = 5)
 - Example 1: **la R3, 56**
The register R3 is loaded with the immediate value 56. This is an instance of immediate addressing mode.
 - Example 2: **la R3, 56(R5)**
The register R3 is loaded with the indexed address **56+R [5]**. This is an example of indexed addressing mode.
- The **st** instruction is used to store register contents to memory (op-code = 3)
 - Example 1: **st R8, 34**
This is the direct addressing mode; the contents of register R8 (**R [8]**) are stored to the memory location **M [34]**
 - Example 2: **st R8, 34(R6)**
An instance of indexed addressing mode, **M [34+R [6]]** stores the contents of R8 (**R [8]**)

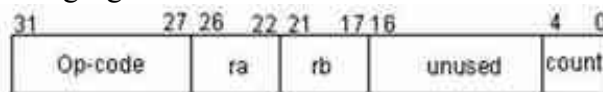
The ALU instructions are

- **addi**, immediate 2's complement addition (op-code = 13)
 - Example: **addi R3, R4, 56**
 $R[3] \leftarrow R[4] + 56$ (*rb field = R4*)

- **andi, the instruction to obtain immediate logical AND, (op-code = 21)**
 - Example: andi R3, R4, 56
 - R3 is loaded with the immediate logical AND of the contents of register R4 and 56(constant value)
- **ori, the instruction to obtain immediate logical OR (op-code = 23)**
 - Example: ori R3, R4, 56
 - R3 is loaded with the immediate logical OR of the contents of register R4 and 56(constant value)

Note:

1. Since the constant c2 field is 17 bits,
 - For direct addressing mode, only the first 2^{16} bytes of memory can be accessed (or the last 2^{16} bytes if c2 is negative)
 - In case of the la instruction, only constants with magnitudes less than $\pm 2^{16}$ can be loaded
 - During address calculation using c2, sign extension to 32 bits must be performed before the addition
2. Type C instructions, with some modifications, may also be used for shift instructions. Note the modification in the following figure.

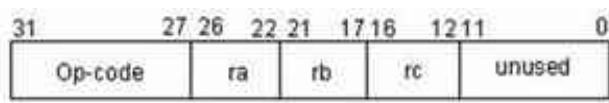


The four shift instructions are

- **shr is the instruction used to shift the bits right by using value in (5-bit) c3 field(shift count) (op-code = 26)**
 - Example: shr R3, R4, 7
 - Shift R4 right 7 times in to R3 and shifts zeros in from the left as the value is shifted right. Immediate addressing mode is used.
- **shra, arithmetic shift right by using value in c3 field (op-code = 27)**
 - Example: shra R3, R4, 7
 - This instruction has the effect of shift R4 right 7 times in to R3 and copies the msb into the word on left as contents are shifted right. Immediate addressing mode is used.
- **The shl instruction is for shift left by using value in (5-bit) c3 field (op-code = 28)**
 - Example: shl R8, R5, 6
 - Shift R5 left 6 times in to R8 and shifts zeros in from the right as the value is shifted left. Immediate addressing mode is used.
- **shc, shift left circular by using value in c3 field (op-code = 29)**
 - Example: shc R3, R4, 3
 - Shift R4 circular 3 times in to R3 and copies the value shifted out of the register on the left is placed back into the register on the right. Immediate addressing mode is used.

Type D

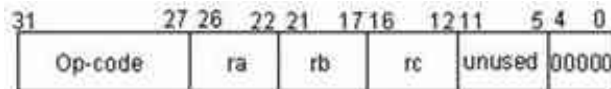
Type D includes four ALU instructions, four register based shift instructions, two logical instructions and two branch instructions.



The four ALU instructions are given below

- **add, the instruction for 2's complement register addition (op-code = 12)**
 - Example:
add R3, R5, R6
Result of 2's complement addition $R[5] + R[6]$ is stored in R3. Register addressing mode is used.
- **sub, the instruction for 2's complement register subtraction (op-code = 14)**
 - Example:
sub R3, R5, R6
R3 will store the 2's complement subtraction, $R[5] - R[6]$. Register addressing mode is used.
- **and, the instruction for logical AND operation between registers (op-code = 20)**
 - Example:
and R8, R3, R4
R8 will store the logical AND of registers R3 and R4. Register addressing mode is used.
- **or, the instruction for logical OR operation between registers (op-code = 22)**
 - Example:
or R8, R3, R4
R8 is loaded with the value $R[3] \vee R[4]$, the logical OR of registers R3 and R4. Register addressing mode is used.

The four register based shift instructions use register addressing mode. These use a modified form of type D, as shown in figure



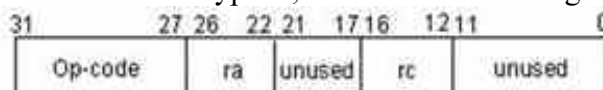
- **shr, shift right by using value in register rc (op-code = 26)**
 - Example: shr R3, R4, R5
This instruction will shift R4 right in to R3 using number in R5
- **shra, the arithmetic shift right by using register rc (op-code = 27)**
 - Example:
shra R3, R4, R5
A shift of R4 right using R5, and the result is stored in R3
- **shl is shift left by using register rc (op-code = 28)**
 - Example:
shl R8, R5, R6

The instruction shifts R5 left in to R8 using number in R6

- **shc, shifts left circular by using register rc (op-code = 29)**
 - Example:
shc R3, R4, R6
This instruction will shift R4 circular in to R3 using value in R6

The two logical instructions also use a modified form of the Type D, and are the following.

- **neg stores the 2's complement of register rc in ra (op-code = 15)**
 - Example: neg R3, R4
Negates (obtains 2's complement) of R4 and stores in R3. 2-address format and register addressing mode is used.

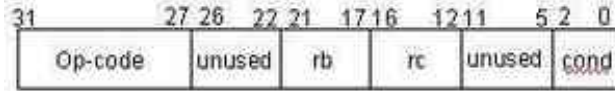


- not stores the 1's complement of register rc in ra (op-code = 24)

- Example:
not R3, R4

Logically inverts R4 and stores in R3. 2-address format with register addressing mode is used.

Type D has two-branch instruction, modified forms of type D.



- br, the instruction to branch to address in rb depending on the condition in rc. There are five possible conditions, explained through examples. (op-code = 8). All branch instructions use register-addressing mode.

- Example 1:
brzr R3, R4
Branch to address in R3 (if R4 == 0)
- Example 2:
brnz R3, R4
Branch to address in R3 (if R4 ≠ 0)
- Example 3:
brpl R3, R4
Branch to address in R3 (if R4 ≥ 0)
- Example 4:
brmi R3, R4
Branch to address in R3 (if R4 < 0)
- Example 5:
br R3, R4
Branch to address in R3 (unconditional)

- Brl the instruction to branch to address in rb depending on condition in rc. Additionally, it copies the PC in to ra before branching (op-code = 9)

- Example 1:
brlzl R1, R3, R4
R1 will store the contents of PC, then branch to address in R3 (if R4 == 0)
- Example 2:
brlnz R1, R3, R4
R1 stores the contents of PC, then a branch is taken, to address in R3 (if R4 ≠ 0)
- Example 3:
brlpl R1, R3, R4
R1 will store PC, then branch to address in R3 (if R4 ≥ 0)
- Example 4:
brlmi R1, R3, R4
R1 will store PC and then branch to address in R3 (if R4 < 0)

Mnemonic	op-code	Branch Condition
brlnv	000	Link but never branch*
br, brl	001	Unconditional branch
brzr, brlzl	010	Branch if rc is zero
brnz, brlnz	011	Branch if rc is not zero
brpl, brlpl	100	Branch if rc is positive
brmi, brlmi	101	Branch if rc is negative

- Example 5:
brl R1,R3, R4
R1 will store PC, then it will ALWAYS branch to address in R3
- Example 6:
brlnv R1,R3, R4
R1 just stores the contents of PC but a branch is not taken (NEVER BRANCH)

In the modified type D instructions for branch, the bits <2...0> are used for specifying the condition; these condition codes are shown in the table.

The SRC Instruction Summary

The instructions implemented by the SRC are listed, grouped on functionality basis.

Functional Groups of Instructions

Group	Op Code	Condition
Shift right by count	srw	1 1 1 1 1 1
Shift right by count in a register	srrw	1 1 1 1 1 1
Shift right by count	srlw	1 1 1 1 1 1
Shift right by count in a register	srrlw	1 1 1 1 1 1
Shift left by count	slw	1 1 1 1 1 1
Shift left by count in a register	srrlw	1 1 1 1 1 1
Shift zero by count	slw	1 1 1 1 1 1
Shift zero by count in a register	srrlw	1 1 1 1 1 1

Condition	Op Code	Condition
Branch when ...	Op Code	Condition
never	brnv	1 1 1 1 1 1
always	br	1 1 1 1 1 1
for zero	brnz	1 1 1 1 1 1
positive (including zero)	brpl	1 1 1 1 1 1
negative	brni	1 1 1 1 1 1
Branch and link when ...	Op Code	Condition
never	brlnv	1 1 1 1 1 1
always	brl	1 1 1 1 1 1
for zero	brlnz	1 1 1 1 1 1
positive (including zero)	brlpl	1 1 1 1 1 1
negative	brlni	1 1 1 1 1 1

Alphabetical Listing based on SRC Mnemonics

Notice that the op code field for all br instructions is the same. The difference is in the condition code field, which is in effect, an op code extension.

Mnemonic	31	30	29	28	27
add	0	1	1	0	0
addi	0	1	1	0	1
and	1	0	1	0	0
andi	1	0	1	0	1
br	0	1	0	0	0
brl	0	1	0	0	1
brlni	0	1	0	0	1
brlnv	0	1	0	0	1
brlnz	0	1	0	0	1
brlpl	0	1	0	0	1
brlnr	0	1	0	0	1
brmi	0	1	0	0	0
brnv	0	1	0	0	0
brnz	0	1	0	0	0
brpl	0	1	0	0	0
brnr	0	1	0	0	0

Op Code	31	30	29	28	27
la	0	0	1	0	1
lar	0	0	1	1	0
ld	0	0	0	0	1
ldr	0	0	0	1	0
neg	0	1	1	1	1
nop	0	0	0	0	0
not	1	1	0	0	0
or	1	0	1	1	0
ori	1	0	1	1	1
shc	1	1	1	0	1
shc	1	1	1	0	1
shl	1	1	1	0	0
shl	1	1	1	0	0
shr	1	1	0	1	0
shr	1	1	0	1	0
shra	1	1	0	1	1
shra	1	1	0	1	1
st	0	0	0	1	1
stop	1	1	1	1	1
str	0	0	1	0	0
sub	0	1	1	1	0

Examples

Some examples are studied in this section to enhance the student's understanding of the SRC.

Branching Instructions	Op Code	Condition
Branch when ...	Op Code	Condition
never	brnv	1 1 1 1 1 1
always	br	1 1 1 1 1 1
for zero	brnz	1 1 1 1 1 1
positive (including zero)	brpl	1 1 1 1 1 1
negative	brni	1 1 1 1 1 1
Branch and link when ...	Op Code	Condition
never	brlnv	1 1 1 1 1 1
always	brl	1 1 1 1 1 1
for zero	brlnz	1 1 1 1 1 1
positive (including zero)	brlpl	1 1 1 1 1 1
negative	brlni	1 1 1 1 1 1

Example 1: Expression Evaluation

Write an SRC assembly language program to evaluate the expression:

$$z = 4(a + b) - 16(c + 58)$$

Your code should not change the source operands.

Solution A: Notice that the SRC does not have a multiply instruction. We will make use of the fact that multiplication with powers of 2 can be achieved by repeated shift left operations. A possible solution is give below:

```
ld R1, c           ; c is a label used for a memory location
addi R3, R1, 58     ; R3 contains (c+58)
shl R7, R3, 4       ; R7 contains 16(c+58)
ld R4, a
ld R5, b
add R6, R4, R5      ; R6 contains (a+b)
shl R8, R6, 2       ; R8 contains 4(a+b)
sub R9, R8, R7      ; the result is in R9
st R9, z            ; store the result in memory location z
```

Note:

The memory labels a, b, c and z can be defined by using assembler directives like .dw or .db, etc. in the source file.

A semicolon ';' is used for comments in assembly language.

Solution B:

We may solve the problem by assuming that a multiply instruction, similar to the add instruction, exists in the instruction set of the SRC. The shl instruction will be replaced by the mul instruction as given below.

```
ld R1, c           ; c is a label used for a memory location
addi R3, R1, 58     ; R3 contains (c+58)
mul R7, R3, 4       ; R7 contains 16(c+58)
ld R4, a
ld R5, b
add R6, R4, R5      ; R6 contains (a+b)
mul R8, R6, 2       ; R8 contains 4(a+b)
sub R9, R8, R7      ; the result is in R9
st R9, z
                   ; store the result in memory location z
```

Note:

The memory labels a, b, c and z can be defined by using assembler directives like .dw or .db, etc. in the source file.

Solution C:

We can perform multiplication with a multiplier that is not a power of 2 by doing addition in a loop. The number of times the loop will execute will be equal to the multiplier.

Example 2: Hand Assembly

Convert the given SRC assembly language program in to an equivalent SRC machine language program.

```
ld R1, c           ; c is a label used for a memory location
```

```
addi R3, R1, 58      ; R3 contains (c+58)
shl R7, R3, 4        ; R7 contains 16(c+58)
ld R4, a
ld R5, b
add R6, R4, R5        ; R6 contains (a+b)
shl R8, R6, 2        ; R8 contains 4(a+b)
sub R9, R8, R7        ; The result is in R9
st R9, z             ; store the result in memory location z
```

Note:

This program uses memory labels a,b,c and z. We need to define them for the assembler by using assembler directives like .dw or .equ etc. in the source file.

Assembler Directives

Assembler directives, also called pseudo op-codes, are commands to the assembler to direct the assembly process. The directives may be slightly different for different assemblers. All the necessary directives are available with most assemblers. We explain the directives as we encounter them. More information on assemblers can be looked up in the assembler user manuals.

Source program with directives

```
.ORG 200      ; start the next line at address 200
a:           .DW 1      ; reserve one word for the label a in the memory
b:           .DW 1      ; reserve a word for b, this will be at address 204
c:           .DW 1      ; reserve a word for c, will be at address 208
z:           .DW 1      ; reserve one word for the result
```

```
.ORG 400 ; start the code at address 400 ; all numbers are in decimal unless otherwise stated
ld R1, c      ; c is a label used for a memory location
addi R3, R1, 58 ; R3 contains (c+58)
shl R7, R3, 4  ; R7 contains 16(c+58)
ld R4, a
ld R5, b
add R6, R4, R5 ; R6 contains (a+b)
shl R8, R6, 2  ; R8 contains 4(a+b)
sub R9, R8, R7 ; the result is in R9
st R9, z       ; store the result in memory location z
```

This is the way an assembly program will appear in the source file. Most assemblers require that the file be saved with an .asm extension.

Solution:

Observe the first line of the program

```
.ORG 200 ; start the next line at address 200
```

This is a directive to let the following code/ variables ‘originate’ at the specified address of the memory, 200 in this case.

Variable statements and another .ORG directive follow the .ORG directive.

```
a:           .DW 1      ; reserve one word for the label a in the memory
b:           .DW 1      ; reserve a word for b, this will be at address 204
c:           .DW 1      ; reserve a word for c, will be at address 208
z:           .DW 1      ; reserve one word for the result
.ORG 400      ; start the code at address 400
```

We conclude the following from the above statements: The code starts at address 400 and each instruction takes 32 bits in the memory. The memory map for the program is shown in given table.

Memory Map for the SRC example program

Memory Address	Memory Contents
200	unknown
204	unknown
208	unknown
212	unknown
...	...
400	ld R1, c
404	addi R3, R1, 58
408	sll R7, R3, 4
412	ld R4, a
416	ld R5, b
420	add R6, R4, R5
424	sll R8, R6, 2
428	sub R9, R7, R8
432	st R9, z

Label	Address	Value
a	200	unknown
b	204	unknown
c	208	unknown
z	212	unknown

Memory Address	Memory Contents	Hexadecimal Memory Contents
200	unknown	
204	unknown	
208	unknown	
212	unknown	
...	...	...
400	ld R1, c	004000D0h
404	addi R3, R1, 58	
408	sll R7, R3, 4	
412	ld R4, a	
416	ld R5, b	
420	add R6, R4, R5	
424	sll R8, R6, 2	
428	sub R9, R7, R8	
432	st R9, z	

We have to convert these instructions to machine language. Let us start with the first instruction:

ld R1, c

Notice that this is a type C instruction with the rb field missing.

1. We pick the op-code for this load instruction from the SRC instruction tables given in the SRC instruction summary section. The op-code for the load register 'ld' instruction is 00001.
2. Next we pick the register code corresponding to register R1 from the register table (given in the section 'encoding of general purpose registers'). The register code for R1 is 00001.
3. The rb field is missing, so we place zeros in the field: 00000
4. The value of c is provided by the assembler, and should be converted to 17 bits. As c has been assigned the memory address 208, the binary value to be encoded is 00000 0000 1101 0000.
5. So the instruction ld R1, c is 00001 00001 00000 00000 0000 1101 0000 in the machine language.
6. The hexadecimal representation of this instruction is 0 8 4 0 0 0 D 0 h.

We can update the memory map with these values.

We consider the next instruction,

addi R3, R1, 58.

Memory Address	Memory Contents	Hexadecimal Memory Contents
200	unknown	
204	unknown	
208	unknown	
212	unknown	
...	...	...
400	ld R1, c	004000D0h
404	addi R3, R1, 58	68C2005Ah
408	sll R7, R3, 4	
412	ld R4, a	
416	ld R5, b	
420	add R6, R4, R5	
424	sll R8, R6, 2	
428	sub R9, R7, R8	
432	st R9, z	

Notice that this is a type C instruction.

1. We pick the op-code for the instruction **addi** from the SRC instruction table. It is 01101
2. We pick the register codes for the registers R3 and R1, these codes are 00011 and 00001 respectively
3. For the immediate data, 58, we use the binary value, 00000 0000 0011 1010
4. So the complete instruction becomes: 01101 00011 00001 00000 0000 0011 1010
5. The hexadecimal representation of the instruction is 6 8 C 2 0 0 3 A h

We update the memory map, as shown in table.

The next instruction is **shl R7,R3, 4**, at address 408.

Again, this is a type C instruction.

1. The op-code for the instruction **shl** is picked from the SRC instruction table. It is 11100

2. The register codes for the registers R7 and R3 from the register table are 00111 and 00011 respectively

3. For the immediate data, 4, the corresponding binary value 00000 0000 0000 0100 is used.

4. We can place these codes in accordance with the type C instruction format to obtain the complete instruction: 11100 00111 00011 00000 0000 0000 0100

5. The hexadecimal representation of the instruction is **E1C60004**

The memory map is updated, as shown in table.

The next instruction, **ld R4, a**, is also a type C instruction. Rb field is missing in this instruction. To obtain the machine equivalent, we follow the steps given below.

1. The op-code of the load instruction 'ld' is 00001
2. The register code corresponding to the register R4 is obtained from the register table, and it is 00100

3. As the 5 bit rb field is missing, we can encode zeros in its place: 00000

4. The value of a is provided by the assembler, and is converted to 17 bits. It has been assigned the memory address 200, the binary equivalent of which is: 00000 0000 1100 1000

5. The complete instruction becomes: 00001 00100 00000 00000 0000 1100 1000

6. The hexadecimal equivalent of the instruction is 0 9 0 0 0 0 C 8 h

Memory map can be updated with this value.

The next instruction is also a load type C instruction, with the rb field missing.

ld R5, b

The machine language conversion steps are

1. The op-code of the load instruction is obtained from the SRC instruction table; it is 00001
2. The register code for R5, obtained from the register table, is 00101
3. Again, the 5 bit rb field is missing. We encode zeros in its place: 00000

Memory Address	Memory Contents	Hexadecimal Memory Contents
200	unknown	
204	unknown	
208	unknown	
212	unknown	
...	...	...
400	ld R1, c	084000D0h
404	addi R3, R1, 58	68C2003Ah
408	shl R7, R3, 4	E1C60004h
412	ld R4, a	
416	ld R5, b	
420	add R6, R4, R5	
424	shl R8, R6, 2	
428	sub R9, R7, R8	
432	st R9, z	

Memory Address	Memory Contents	Hexadecimal Memory Contents
200	unknown	
204	unknown	
208	unknown	
212	unknown	
...	...	...
400	ld R1, c	084000D0h
404	addi R3, R1, 58	68C2003Ah
408	shl R7, R3, 4	E1C60004h
412	ld R4, a	090000C8h
416	ld R5, b	
420	add R6, R4, R5	
424	shl R8, R6, 2	
428	sub R9, R7, R8	
432	st R9, z	

Memory Address	Memory Contents	Hexadecimal Memory Contents
200	unknown	
204	unknown	
208	unknown	
212	unknown	
...	...	...
400	ld R1, c	084000D0h
404	addi R3, R1, 58	68C2003Ah
408	shl R7, R3, 4	E1C60004h
412	ld R4, a	090000C8h
416	ld R5, b	094000CCh
420	add R6, R4, R5	
424	shl R8, R6, 2	
428	sub R9, R7, R8	
432	st R9, z	

4. The value of label b is provided by the assembler, and should be converted to 17 bits. It has been assigned the memory address 204, so the binary value is: 00000 0000 1100 1100

5. The complete instruction is: 00001 00101 00000 00000 0000 1100 1100

6. The hexadecimal value of this instruction is 0 9 4 0 0 0 C C h

Memory map is then updated with this value.

The next instruction is a type D-add instruction, with the constant field missing:

add R6,R4,R5

The steps followed to obtain the assembly code for this instruction are

1. The op-code of the instruction is obtained from the SRC instruction table; it is 01100

2. The register codes for the registers R6, R4 and R5 are obtained from the register table; these are 00110, 00100 and 00101 respectively.

3. The 12 bit constant field is unused in this instruction, therefore we encode zeros in its place: 0000 0000 0000

4. The complete instruction becomes: 01100 00110 00100 00101 0000 0000 0000

5. The hexadecimal value of the instruction is 6 1 8 8 5 0 0 0 h

Memory map is then updated with this value.

The instruction **shl R8,R6, 2** is a type C instruction with the rc field missing. The steps taken to obtain the machine code of the instruction are

1. The op-code of the shift left instruction 'shl', obtained from the SRC instruction table, is 11100

2. The register codes of R8 and R6 are 01000 and 00110 respectively

3. Binary code is used for the immediate data 2: 00000 0000 0000 0010

4. The complete instruction becomes: 11100 01000 00110 00000 0000 0000 0010

5. The hexadecimal equivalent of the instruction is E 2 0 C 0 0 0 2

Memory map is then updated with this value.

The instruction at the memory address 428 is **sub R9, R7, R8**.

This is a type D instruction.

We decode it into the machine language, as follows:

1. The op-code of the subtract instruction 'sub' is 01110

2. The register codes of R9, R7 and R8, obtained from the register table, are 01001, 00111 and 01000 respectively

3. The 12 bit immediate data field is not used, zeros are encoded in its place: 0000 0000 0000

4. The complete instruction becomes: 01110 01001 00111 01000 0000 0000 0000

5. The hexadecimal equivalent is 7 2 4 E 8 0 0 0 h We again update the memory map

The last instruction is a type C instruction with the rb field missing:

Memory Address	Memory Contents	Hexadecimal Memory Contents
200	unknown	
204	unknown	
208	unknown	
212	unknown	
...	...	...
400	ld R1,c	084000D0h
404	addi R3,R1,38	68C2003Ah
408	shl R7,R3,4	E1C60004h
412	ld R4,a	090000C8h
416	ld R5,b	094000CCh
420	add R6,R4,R5	61885000h
424	shl R8,R6,2	
428	sub R9,R7,R8	
432	st R9,x	

Memory Address	Memory Contents	Hexadecimal Memory Contents
200	unknown	
204	unknown	
208	unknown	
212	unknown	
...	...	...
400	ld R1,c	084000D0h
404	addi R3,R1,38	68C2003Ah
408	shl R7,R3,4	E1C60004h
412	ld R4,a	090000C8h
416	ld R5,b	094000CCh
420	add R6,R4,R5	61885000h
424	shl R8,R6,2	E20C0002h
428	sub R9,R7,R8	
432	st R9,x	

Memory Address	Memory Contents	Hexadecimal Memory Contents
200	unknown	
204	unknown	
208	unknown	
212	unknown	
...	...	...
400	ld R1,c	084000D0h
404	addi R3,R1,38	68C2003Ah
408	shl R7,R3,4	E1C60004h
412	ld R4,a	090000C8h
416	ld R5,b	094000CCh
420	add R6,R4,R5	61885000h
424	shl R8,R6,2	E20C0002h
428	sub R9,R7,R8	724E8000h
432	st R9,x	

st R9, z

The machine equivalent of this instruction is obtained through the following steps:

1. The op-code of the store instruction 'st', obtained from the SRC instruction table, is 00011
2. The register code of R9 is 01001
3. Notice that there is no register coded in the 5 bit rb field, therefore, we encode zeros: 00000. The value of the label z is provided by the assembler, and should be converted to 17 bits. Notice that the memory address assigned to z is 212. The 17 bit binary equivalent is: 00000 0000 1101 0100
4. The complete instruction becomes: 00011 01001 00000 00000 0000 1101 0100
5. The hexadecimal form of this instruction is 1 A 4 0 0 0 D 4 h

The memory map, after the conversion of all the instructions, is shown below. We have shown the memory map as an array of 4 byte cells in the above solution. However, since the memory of the SRC is arranged in 8 bit cells (i.e. memory is byte aligned), the real representation of the memory map is:

Memory Address	Memory Contents	Hexadecimal Memory Contents
200	unknown	
204	unknown	
208	unknown	
212	unknown	
...	...	...
400	ld R1, c	68400D0h
404	addi R3, R1, 38	68C2003Ah
408	shl R7, R3, 4	E1C60004h
412	ld R4, a	09000C8h
416	ld R5, b	09400CC h
420	add R6, R4, R5	61865000h
424	shl R8, R6, 2	E20C0002h
428	sub R9, R7, R8	734E9900h
432	st R9, z	1A400D4h

Memory Address	Memory Contents
...	...
400	6840
401	0D0h
402	68C2
403	003Ah
404	E1C6
405	0004h
406	0900
407	0C8h
408	0940
409	0CC h
410	6186
411	5000h
412	E20C
413	0002h
414	0900
415	0C8h
416	0940
417	0CC h
...	...

Example 3: SRC instruction analysis

Identify the formats of following SRC instructions and specify the values in the fields

Instruction	format	ra	rb	rc	c1	c2	c3
neg r1, r2							
add r0, r2, r3							
nop							
ld r2, 6							
shl r0, r1, 3							

Solution:

Instruction	format	ra	rb	rc	c1	c2	c3
neg r1, r2	D	r1	-	r2	-	-	-
add r0, r2, r3	D	r0	r2	r3	-	-	-
nop	A	-	-	-	-	-	-
ld r2, 6	C	r2	-	-	6	-	-
shl r0, r1, 3	C	r0	r1	-	-	-	3

Lecture No. 5

Description of SRC in RTL

Reading Material

Handouts

Slides

Summary

- Reverse Assembly
- Description of SRC in the form of RTL
- Behavioral and Structural description in terms of RTL

Reverse Assembly

Typical Problem:

Given a machine language instruction for the SRC, it may be required to find the equivalent SRC assembly language instruction

Example:

Reverse assemble the following SRC machine language instructions:

68C2003A h

E1C60004 h

61885000 h

724E8000 h

1A4000D4 h

084000D0 h

Solution:

1. Write the given hexadecimal instruction in binary form 68C2003A h → 0110 1000 1100 0010 0000 0000 0011 1010 b
2. Examine the first five bits of the instruction, and pick the corresponding mnemonic from the SRC instruction set listing arranged according to ascending order of op-codes 01101 b → 13 d → addi → add immediate
3. Now we know that this instruction uses the type C format, the two 5-bit fields after the op-code field represent the destination and the source registers respectively, and that the remaining 17-bits in the instruction represent a constant

01101	00011	00001	00000 0000 0011 1010 b
op-code	ra field	rb field	17-bit cl field
↓	↓	↓	↓
addi	R3	R1	3A h=58 d

4. Therefore, the assembly language instruction is
addi R3, R1, 58

Summary

Given machine language instruction	Equivalent assembly language instruction
68C2003A h E1C60004 h 61885000 h	addi R3, R1, 58
724E8000 h 1A4000D4 h 084000D0 h	

We can do it a bit faster now! **Step1:** Here is step1 for all instructions

Given instruction in hexadecimal	Equivalent instruction in binary
E1C60004 h	11 0 000 1100 0...0 0000 0000 0000 0000 b
61885000 h	01 0 000 1000 1000 0101 0000 0000 0000 0000 b
724E8000 h	01 1 0 100 0011 1001 1001 1000 0000 0000 b
1A4000D4 h	0001 1000 0000 0000 0000 0000 1101 0000 b
084000D0 h	0000 1000 0000 0000 0000 0000 1101 0000 b

Step 2: Pick up the op code for each instruction

Given instruction in hexadecimal	Op-code field	mnemonic	
E1C60004 h	1110 0 b	shl	
61885000 h	0110 0 b	add	
724E8000 h	0111 0 b	sub	
1A4000D4 h	0001 1 b	st	
084000D0 h	0000 1 b	ld	

Step 3: Determine the instruction type for each instruction

Given instruction in hexadecimal	mnemonic	Instruction type
E1C60004 h	shl	
61885000 h	add	
724E8000 h	sub	
1A4000D4 h	st	
084000D0 h	ld	

The meaning of the remaining fields will depend on the instruction type (i.e., the instruction format)

Summary

Given machine language instruction	Equivalent assembly language instruction
68C2003A h	addi R3, R1, 58
F1C60004 h	
61885000 h	
724E8000 h	
1A400074 h	
084000D0 h	

Note: rest of the fields of above given tables are left as an exercise for students. **Using RTL to describe the SRC**

RTL stands for Register Transfer Language. The Register Transfer Language provides a formal way for the description of the behavior and structure of a computer. The RTL facilitates the design process of the computer as it provides a precise, mathematical representation of its functionality. In this section, a Register Transfer Language is presented and introduced, for the SRC (Simple 'RISC' Computer), described in the previous discussion.

Behavioral RTL

Behavioral RTL is used to describe the 'functionality' of the machine only, i.e. what the machine does.

Structural RTL

Structural RTL describes the 'hardware implementation' of the machine, i.e. how the functionality made available by the machine is implemented.

Behavioral versus Structural RTL:

In computer design, a top-down approach is adopted. The computer design process typically starts with defining the behavior of the overall system. This is then broken down into the behavior of the different modules. The process continues, till we are able to define, design and implement the structure of the individual modules. Behavioral RTL is used for describing the behavior of machine whereas structural RTL is used to define the structure of machine, which brings us to the some more hardware features.

Using RTL to describe the static properties of the SRC

In this section we introduce the RTL by using it to describe the various static properties of the SRC.

Specifying Registers

The format used to specify registers is

Register Name<register bits>

For example, IR<31..0> means bits numbered 31 to 0 of a 32-bit register named "IR" (Instruction Register).

"Naming" using the := naming operator:

The := operator is used to 'name' registers, or part of registers, in the Register Transfer Language. It does not create a new register; it just generates another name, or "alias" for an already existing register or part of a register. For example,

Op<4..0> := IR<31..27> means that the five most significant bits of the register IR will be called op, with bits 4..0.

Fields in the SRC instruction

In this section, we examine the various fields of an SRC instruction, using the RTL.

op<4..0>: = IR<31..27>; operation code field

The five most significant bits of an SRC instruction, (stored in the instruction register in this example), are named op, and this field is used for specifying the operation. ra<4..0>: = IR<26..22>; target register field

The next five bits of the SRC instruction, bits 26 through 22, are used to hold the address of the target register field, i.e., the result of the operation performed by the instruction is stored in the register specified by this field.

rb<4..0>: = IR<21..17>; operand, address index, or branch target register

The bits 21 through 17 of the instruction are used for the rb field. rb field is used to hold an operand, an address index, or a branch target register.

rc<4..0>: = IR<16..12>; second operand, conditional test, or shift count register

The bits 16 through 12, are the rc field. This field may hold the second operand, conditional test, or a shift count.

c1<21..0>: = IR<21..0>; long displacement field

In some instructions, the bits 21 through 0 may be used as long displacement field. Notice that there is an overlap of fields. The fields are distinguished in a particular instruction depending on the operation.

c2<16..0>: = IR<16..0>; short displacement or immediate field

The bits 16 through 0 may be used as short displacement or to specify an immediate operand.

c3<11..0>: = IR<11..0>; count or modifier field

The bits 11 through 0 of the SRC instruction may be used for count or modifier field.

Describing the processor state using RTL

The Register Transfer Language can be used to describe the processor state. The following registers and bits together form the processor state set.

PC<31..0>; program counter (it holds the memory address of next instruction to be executed)

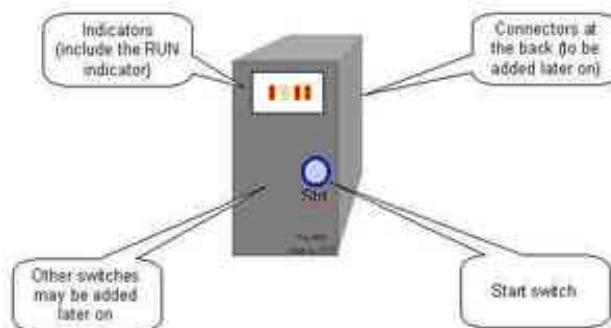
IR<31..0>; instruction register, used to hold the current instruction

Run; one bit run/halt indicator

Strt; start signal

R [0..31]<31..0>; 32, 32 bit general purpose registers

SRC in a Black Box



Difference between our notation and notation used by the text (H&J)

Key Symbols	Meaning	Symbols in use	The Symbol or Language	Meaning	Symbol used by RAL
→	Unconditional Transfer	→	RFL	Register Transfer Language	RTN
←	Unconditional Statement	←	Behavioral RFL		Abstract RTN
→	Transfer Statement	→	Internal RFL		Concrete RFL
→	Flowing operation	→	Register status		State assignments
←	Assignment	←	RAA	Memory Address Register	RAA
RA	Logical RAA	RA	RAA	Memory Buffer Register	RAA
→	Logical RTN	→			
←	Logical RFL	←			
→	Transfer Statement	→			
←	Register	←			
RA	Register of the Register (RAA)	RAA			

Difference between “,” and “;” in RTL

Statements separated by a “,” take place during the same clock pulse. In other words, the order of execution of statements separated by “;” does not matter.

On the other hand, statements separated by a “;” take place on successive clock pulses. In other words, if statements are separated by “;” the one on the left must complete before the one on the right starts. However, some things written with one RTL statement can take several clocks to complete.

So in the instruction interpretation, fetch-execute cycle, we can see that the first statement, ! Run & Strt : $\text{Run} \leftarrow 1$, executes first. After this statement has executed and set run to 1, the statements $\text{IR} \leftarrow \text{M}[\text{PC}]$ and $\text{PC} \leftarrow \text{PC} + 4$ are executed concurrently.

Note that in statements separated by “;”, all right hand sides of Register Transfers are evaluated before any left hand side is modified (generally though assignment).

Using RTL to describe the dynamic properties of the SRC The RTL can be used to describe the dynamic properties.

Conditional expressions can be specified through the use of RTL. The following example will illustrate this

```
(op=14) : R [ra] ← R [rb] - R[rc];
```

The $\leftarrow$ operator is the RTL assignment operator. ‘;’ is the termination operator. This conditional expression implies that “IF the op field is equal to 14, THEN calculate the difference of the value in the register specified by the rb field and the value in the register specified by the rc field, and store the result in the register specified by the ra field.”

Effective address calculations in RTL (performed at runtime)

In some instructions, the address of an operand or the destination register may not be specified directly. Instead, the effective address may have to be calculated at runtime.

These effective address calculations can be represented in RTL, as illustrated through the examples below.

Displacement address

$$\text{disp}\langle 31..0 \rangle := ((\text{rb}=0) : \text{c2}\langle 16..0 \rangle \{\text{sign extend}\},$$
$$(\text{rb} \neq 0) : \text{R} [\text{rb}] + \text{c2} \langle 16..0 \rangle \{\text{sign extend}\},$$

The displacement (or the direct) address is being calculated in this example. The “,” operator separates statements in a single instruction, and indicates that these statements are to be executed

simultaneously. However, since in this example these are two disjoint conditions, therefore, only one action will be performed at one time.

Note that register R0 cannot be added to displacement. $rb = 0$ just implies we do not need to use the R [rb] field.

Relative address

$rel<31..0> := PC<31..0> + c1<21..0> \{sign\ extend\}$,

In the above example, a relative address is being calculated by adding the displacement after sign extension to the contents of the program counter register (that holds the next instruction to be executed in a program execution sequence).

Range of memory addresses

The range of memory addresses that can be accessed using the displacement (or the direct) addressing and the relative addressing is given.

- Direct addressing (displacement with $rb=0$)
 - If $c2<16>=0$ (positive displacement) absolute addresses range from 00000000h to 0000FFFFh
 - If $c2<16>=1$ (negative displacement) absolute addresses range from FFFF0000h to FFFFFFFFh
- Relative addressing
 - The largest positive value of $C1<21..0>$ is $2^{21}-1$ and its most negative value is -2^{21} , so addresses up to $2^{21}-1$ forward and 2^{21} backward from the current PC value can be specified

Instruction Interpretation

(Describing the Fetch operation using RTL)

The action performed for all the instructions before they are decoded is called 'instruction interpretation'. Here, an example is that of starting the machine. If the machine is not already running ($\neg Run$, or 'not' running), AND (&) if the condition start (Strt) becomes true, then Run bit (of the processor state) is set to 1 (i.e. true).

instruction_Fetch := (
 ! Run & Strt: Run $\leftarrow$ 1 ; instruction_Fetch
 Run : (IR $\leftarrow$ M [PC], PC $\leftarrow$ PC + 4; instruction_Execution));

The $:=$ is the naming operator. The ; operator is used to add comments in RTL. The , operator, specifies that the statements are to be executed simultaneously, (i.e. in a single clock pulse). The ; operator is used to separate sequential statements. $\leftarrow$ is an assignment operator. & is a logical AND, $\sim$ is a logical OR, and ! is the logical NOT. In the instruction interpretation phase of the fetch-execute cycle, if the machine is running (Run is true), the instruction register is loaded with the instruction at the location M [PC] (the program counter specifies the address of the memory at which the instruction to be executed is located). Simultaneously, the program counter is incremented by 4, so as to point to the next instruction, as shown in the example above. This completes the instruction interpretation.

Instruction Execution

(Describing the Execute operation using RTL)

Once the instruction is fetched and the PC is incremented, execution of the instruction starts. In the following, we denote instruction Fetch by "iF" and instruction execution by "iE".

iE:= (
 (op<4..0>= 1) : R [ra] $\leftarrow$ M [disp],
 (op<4..0>= 2) : R [ra] $\leftarrow$ M [rel],
 ...
 ...
 (op<4..0>=31) : Run $\leftarrow$ 0; iF);

As shown above, Instruction Execution can be described by using a long list of conditional operations, which are inherently “disjoint”.

One of these statements is executed, depending on the condition met, and then the instruction fetch statement (iF) is invoked again at the end of the list of concurrent statements. Thus, instruction fetch (iF) and instruction execution statements invoke each other in a loop. This is the fetch-execute cycle of the SRC.

Concurrent Statements

The long list of concurrent, disjoint instructions of the instruction execution (iE) is basically the complete instruction set of the processor. A brief overview of these instructions is given below.

Load-Store Instructions

(op<4..0>= 1) : R [ra] ← M [disp], load register (ld)

This instruction is to load a register using a displacement address specified by the instruction, i.e. the contents of the memory at the address ‘disp’ are placed in the register R [ra].

(op<4..0>= 2) : R [ra] ← M [rel], load register relative (ldr)

If the operation field ‘op’ of the instruction decoded is 2, the instruction that is executed is loading a register (target address of this register is specified by the field ra) with memory contents at a relative address, ‘rel’. The relative address calculation has been explained in this section earlier.

(op<4..0>= 3) : M [disp] ← R [ra], store register (st)

If the op-code is 3, the contents of the register specified by address ra, are stored back to the memory, at a displacement location 'disp'.

(op<4..0>= 4) : M[rel] $\leftarrow$ R[ra], store register relative (str)

If the op-code is 4, the contents of the register specified by the target register address *ra*, are stored back to the memory, at a relative address location ‘rel’.

(op<4..0>= 5) : R [ra] ← disp, load displacement address (la)

For op-code 5, the displacement address `disp` is loaded to the register `R` (specified by the target register address `ra`).

(op<4..0>= 6) : R [ra] ← rel, load relative address (lar)

For op-code 6, the relative address rel is loaded to the register R (specified by the target register address ra).

Branch Instructions

(op<4..0>= 8) : (cond : PC ← R [rb]), conditional branch (br)

If the op-code is 8, a conditional branch is taken, that is, the program counter is set to the target instruction address specified by rb, if the condition ‘cond’ is true.

(op<4..0>= 9) : (R [ra] ← PC,

cond : (PC $\leftarrow$ R [rb]) , branch and link (brl)

If the op field is 9, branch and link instruction is executed, i.e. the contents of the program counter are stored in a register specified by ra field, (so control can be returned to it later), and then the conditional branch is taken to a branch target address specified by rb. The branch and link instruction is useful for returning control to the calling program after a procedure call returns.

The conditions that these ‘conditional’ branches depend on are specified by the field c3 that has 3 bits. This simply means that when $c3 < 2..0$ is equal to one of these six values. We substitute the expression on the right hand side of the : in place of cond. These conditions are explained here briefly.

```
cond := (
    c3<2..0>=0 : 0,           never
    If the c3 field is 0, the branch is never taken.
    c3<2..0>=1 : 1,           always
    If the field is 1, branch is taken
```

c3<2..0>=2 : R [rc]=0, if register is zero

If c3 = 2, a branch is taken if the register rc = 0.

c3<2..0>=3 : R [rc] ≠ 0, if register is nonzero

If c3 = 3, a branch is taken if the register rc is not equal to 0.

c3<2..0>=4 : R [rc]<31>=0 if positive or zero

If c3 is 4, a branch is taken if the register value in the register specified by rc is greater than or equal to 0.

c3<2..0>=5 : R [rc]<31>=1), if negative

If c3 = 5, a branch is taken if the value stored in the register specified by rc is negative.

Arithmetic and Logical instructions

(op<4..0>=12) : R [ra] ← R [rb] + R [rc],

If the op-code is 12, the contents of the registers rb and rc are added and the result is stored in the register ra.

(op<4..0>=13) : R [ra] ← R [rb] + c2<16..0> {sign extend},

If the op-code is 13, the content of the register rb is added with the immediate data in the field c2, and the result is stored in the register ra.

(op<4..0>=14) : R [ra] ← R [rb] – R [rc],

If the op-code is 14, the content of the register rc is subtracted from that of rb, and the result is stored in ra.

(op<4..0>=15) : R [ra] ← -R [rc],

If the op-code is 15, the content of the register rc is negated, and the result is stored in ra.

(op<4..0>=20) : R [ra] ← R [rb] & R [rc],

If the op field equals 20, logical AND of the contents of the registers rb and rc is obtained and the result is stored in register ra.

(op<4..0>=21) : R [ra] ← R [rb] & c2<16..0> {sign extend},

If the op field equals 21, logical AND of the content of the registers rb and the immediate data in the field c2 is obtained and the result is stored in register ra.

(op<4..0>=22) : R [ra] ← R [rb] ~ R [rc],

If the op field equals 22, logical OR of the contents of the registers rb and rc is obtained and the result is stored in register ra.

(op<4..0>=23) : R [ra] ← R [rb] ~ c2<16..0> {sign extend},

If the op field equals 23, logical OR of the content of the registers rb and the immediate data in the field c2 is obtained and the result is stored in register ra.

(op<4..0>=24) : R [ra] ← ¬R [rc],

If the op-code equals 24, the content of the logical NOT of the register rc is obtained, and the result is stored in ra.

Shift instructions

(op<4..0>=26): R [ra]<31..0 > ← (n α 0) © R [rb] <31..n>,

If the op-code is 26, the contents of the register rb are shifted right n bits times. The bits that are shifted out of the register are discarded. 0s are added in their place, i.e. n number of 0s is added (or concatenated) with the register contents. The result is copied to the register ra.

(op<4..0>=27) : R [ra]<31..0 > ← (n α R [rb] <31>) © R [rb] <31..n>,

For op-code 27, shift arithmetic operation is carried out. In this operation, the contents of the register rb are shifted right n times, with the most significant bit, bit 31, of the register rb added in their place. The result is copied to the register ra.

(op<4..0>=28) : R [ra]<31..0 > ← R [rb] <31..n..0> © (n α 0),

For op-code 28, the contents of the register rb are shifted left n bits times, similar to the shift right instruction. The result is copied to the register ra.

(op<4..0>=29) : R [ra]<31..0> $\leftarrow$ R [rb] <31-n..0> © R [rb]<31..32-n> ,

The instruction corresponding to op-code 29 is the shift circular instruction. The contents of the register rb are shifted left n times, however, the bits that move out of the register in the shift process are not discarded; instead, these are shifted in from the other end (a circular shifting). The result is stored in register ra.

where

n := (

**(c3<4..0>=0) : R [rc],
(c3<4..0>!=0) : c3 <4..0>),**

Notation:

α means replication

© Means concatenation

Miscellaneous instructions

(op<4..0>= 0) , No operation (nop)

If the op-code is 0, no operation is carried out for that clock period. This instruction is used as a stall in pipelining.

(op<4..0>= 31) : Run $\leftarrow$ 0, Halt the processor (Stop)

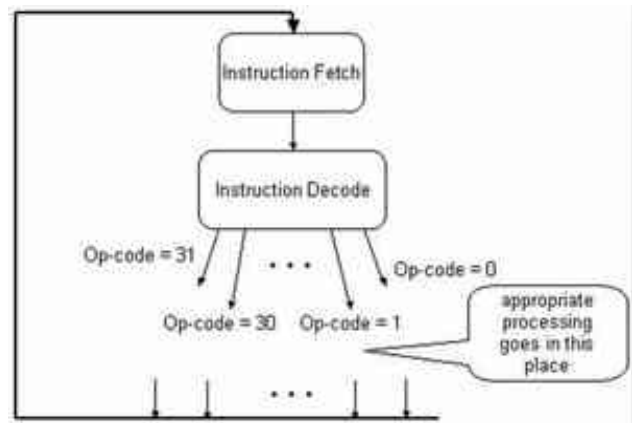
); iF);

If the op-code is 31, run is set to 0, that is, the processor is halted.

After one of these disjoint instructions is executed, iF, i.e. instruction Fetch is carried out once again, and so the fetch-execute cycle continues.

Flow diagram

Flow diagram is the symbolic representation of Fetch-Execute cycle. Its top block indicates instruction fetch and then next block shows the instruction decode by looking at the first 5-bits of the fetched instruction which would represent op-code which may be from 0 to 31. Depending upon the contents of this op-code the appropriate processing would take place. After the appropriate processing, we would move back to top block, next instruction is fetched and the



same process is repeated until the instruction with op-code 31 would reach and halt the system.

Note: For SRC Assembler and Simulator consult Appendix.



Lecture No. 6

RTL Using Digital Logic Circuits

Reading Material

Handouts

Slides

Summary

- Using Behavioral RTL to Describe the SRC (continued)
- Implementing Register Transfer using Digital Logic Circuits

Using behavioral RTL to Describe the SRC (continued)

Once the instruction is fetched and the PC is incremented, execution of the instruction starts. In the following discussion, we denote instruction fetch by “iF” and instruction execution by “iE”.

```
iE:= (
(op<4..0>= 1) : R [ra] ← M [disp],
(op<4..0>= 2) : R [ra] ← M [rel],
...
...
(op<4..0>=31) : Run ← 0,); iF);
```

As shown above, instruction execution can be described by using a long list of conditional operations, which are inherently “disjoint”. Only one of these statements is executed, depending on the condition met, and then the instruction fetch statement (iF) is invoked again at the end of the list of concurrent statements. Thus, instruction fetch (iF) and instruction execution statements invoke each other in a loop. This is the fetch-execute cycle of the SRC.

Concurrent Statements

The long list of concurrent, disjoint instructions of the instruction execution (iE) is basically the complete instruction set of the processor. A brief overview of these instructions is given below:

Load-Store Instructions

(op<4..0>= 1) : R [ra] ← M [disp], load register (ld)

This instruction is to load a register using a displacement address specified by the instruction, i.e., the contents of the memory at the address ‘disp’ are placed in the register R [ra].

(op<4..0>= 2) : R [ra] ← M [rel], load register relative (ldr)

If the operation field ‘op’ of the instruction decoded is 2, the instruction that is executed is loading a register (target address of this register is specified by the field ra) with memory contents at a relative address, ‘rel’. The relative address calculation has been explained in this section earlier.

(op<4..0>= 3) : M [disp] ← R [ra], store register (st)

If the op-code is 3, the contents of the register specified by address ra, are stored back to the memory, at a displacement location 'disp'.

(op<4..0>= 4) : M[rel] ← R[ra], store register relative (str)

If the op-code is 4, the contents of the register specified by the target register address ra, are stored back to the memory, at a relative address location 'rel'.

(op<4..0>= 5) : R [ra] ← disp, load displacement address (la)

For op-code 5, the displacement address disp is loaded to the register R (specified by the target register address ra).

(op<4..0>= 6) : R [ra] ← rel, load relative address (lar)

For op-code 6, the relative address rel is loaded to the register R (specified by the target register address ra).

Branch Instructions

(op<4..0>= 8) : (cond : PC ← R [rb]), conditional branch (br)

If the op-code is 8, a conditional branch is taken, that is, the program counter is set to the target instruction address specified by rb, if the condition 'cond' is true.

(op<4..0>= 9) : (R [ra] ← PC,

cond : (PC ← R [rb])), branch and link (bri)

If the op field is 9, branch and link instruction is executed, i.e. the contents of the program counter are stored in a register specified by ra field, (so control can be returned to it later), and then the conditional branch is taken to a branch target address specified by rb. The branch and link instruction is useful for returning control to the calling program after a procedure call returns.

The conditions that these 'conditional' branches depend on, are specified by the field c3 that has 3 bits. This simply means that when c3<2..0> is equal to one of these six values, we substitute the expression on the right hand side of the : in place of cond. These conditions are explained here briefly.

cond := (

c3<2..0>=0 : 0, never

If the c3 field is 0, the branch is never taken.

c3<2..0>=1 : 1, always

If the field is 1, branch is taken

c3<2..0>=2 : R [rc]=0, if register is zero

If c3 = 2, a branch is taken if the register rc = 0.

c3<2..0>=3 : R [rc] ≠ 0, if register is nonzero

If c3 = 3, a branch is taken if the register rc is not equal to 0.

c3<2..0>=4 : R [rc]<31>=0 if positive or zero

If c3 is 4, a branch is taken if the register value in the register specified by rc is greater than or equal to 0.

c3<2..0>=5 : R [rc]<31>=1), if negative

If c3 = 5, a branch is taken if the value stored in the register specified by rc is negative.

Arithmetic and Logical instructions

(op<4..0>=12) : R [ra] ← R [rb] + R [rc],

If the op-code is 12, the contents of the registers rb and rc are added and the result is stored in the register ra.

(op<4..0>=13) : R [ra] ← R [rb] + c2<16..0> {sign extended},

If the op-code is 13, the content of the register rb is added with the immediate data in the field c2, and the result is stored in the register ra.

(op<4..0>=14) : R [ra] $\leftarrow$ R [rb] + R [rc],

If the op-code is 14, the content of the register rc is subtracted from that of rb, and the result is stored in ra.

(op<4..0>=15) : R [ra] $\leftarrow$ -R [rc],

If the op-code is 15, the content of the register rc is negated, and the result is stored in ra.

(op<4..0>=20) : R [ra] $\leftarrow$ R [rb] & R [rc],

If the op field equals 20, logical AND of the contents of the registers rb and rc is obtained and the result is stored in register ra.

(op<4..0>=21) : R [ra] $\leftarrow$ R [rb] & c2<16..0> {sign extended},

If the op field equals 21, logical AND of the content of the registers rb and the immediate data in the field c2 is obtained and the result is stored in register ra.

(op<4..0>=22) : R [ra] $\leftarrow$ R [rb] ~ R [rc],

If the op field equals 22, logical OR of the contents of the registers rb and rc is obtained and the result is stored in register ra.

(op<4..0>=23) : R [ra] $\leftarrow$ R [rb] ~ c2<16..0> {sign extended},

If the op field equals 23, logical OR of the content of the registers rb and the immediate data in the field c2 is obtained and the result is stored in register ra.

(op<4..0>=24) : R [ra] $\leftarrow$!R [rc],

If the op-code equals 24, the content of the logical NOT of the register rc is obtained, and the result is stored in ra.

Shift instructions

(op<4..0>=26) : R [ra]<31..0> $\leftarrow$ (n α 0) © R [rb] <31..n>,

If the op-code is 26, the contents of the register rb are shifted right n bits times. The bits that are shifted out of the register are discarded. 0s are added in their place, i.e. n number of 0s is added (or concatenated) with the register contents. The result is copied to the register ra.

(op<4..0>=27) : R [ra]<31..0> $\leftarrow$ (n α R [rb] <31>) © R [rb] <31..n>,

For op-code 27, shift arithmetic operation is carried out. In this operation, the contents of the register rb are shifted right n times, with the most significant bit, i.e., bit 31, of the register rb added in their place. The result is copied to the register ra.

(op<4..0>=28) : R [ra]<31..0> $\leftarrow$ R [rb] <31-n..0> © (n α 0),

For op-code 28, the contents of the register rb are shifted left n bits times, similar to the shift right instruction. The result is copied to the register ra.

(op<4..0>=29) : R [ra]<31..0> $\leftarrow$ R [rb] <31-n..0> © R [rb]<31..32-n> ,

The instruction corresponding to op-code 29 is the shift circular instruction. The contents of the register rb are shifted left n times, however, the bits that move out of the register in the shift process are not discarded; instead, these are shifted in from the other end (a circular shifting). The result is stored in register ra.

where

n := (

(c3<4..0>=0) : R [rc],

(c3<4..0>!=0) : c3 <4..0>),

Notation:

α means replication

© means concatenation

Miscellaneous instructions

(op<4..0>= 0) , No operation (nop)

If the op-code is 0, no operation is carried out for that clock period. This instruction is used as a stall in pipelining.

(op<4..0>= 31) : Run $\leftarrow$ 0, Halt the processor (Stop)

); iF);

If the op-code is 31, run is set to 0, that is, the processor stops execution.

After one of these disjoint instructions is executed, iF, i.e. instruction Fetch is carried out once again, and so the fetch-execute cycle continues.

Implementing Register Transfers using Digital Logic Circuits

We have studied the register transfers in the previous sections, and how they help in implementing assembly language. In this section we will review how the basic digital logic circuits are used to implement instructions register transfers. The topics we will cover in this section include:

1. A brief (and necessary) review of logic circuits
2. Implementing simple register transfers
3. Register file implementation using a bus
4. Implementing register transfers with mathematical operations
5. The Barrel Shifter
6. Implementing shift operations

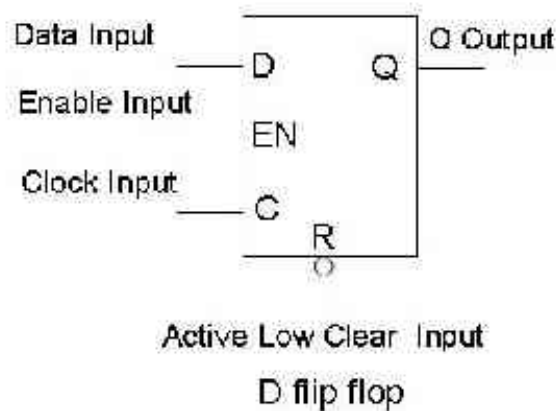
Review of logic circuits

Before we study the implementation of register transfers using logic circuits, a brief overview of some of the important logic circuits will prove helpful. The topics we review in this section include

1. The basic D flip flop
2. The n-bit register
3. The n-to-1 multiplexer
4. Tri-state buffers

The basic D flip flop

A flip-flop is a bi-stable device, capable of storing one bit of Information. Therefore, flip-flops are used as the building blocks of a computer's memory as well as CPU registers.



There are various types of flip-flops; most common type, the D flip-flop is shown in the figure given. The given truth table for this positive-edge triggered D flip-flop shows that the flip-flop is set (i.e. stores a 1) when the data input is high on the leading (also called the positive) edge of the

clock; it is reset (i.e., the flip-flop stores a 0) when the data input is 0 on the leading edge of the clock. The clear input will reset the flip-flop on a low input.

EN	D	Q
0	X	X
1	0	0
1	1	1

Truth table: D-Flip Flop

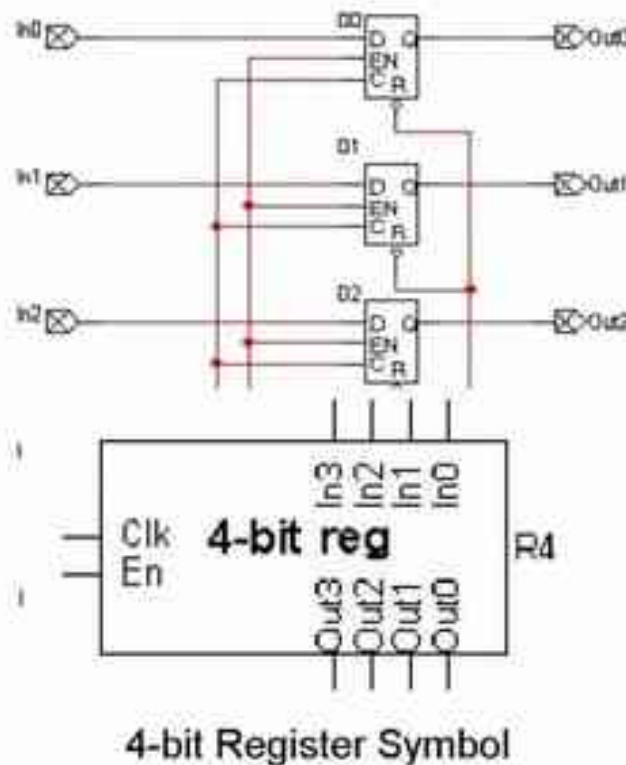
The n-bit register

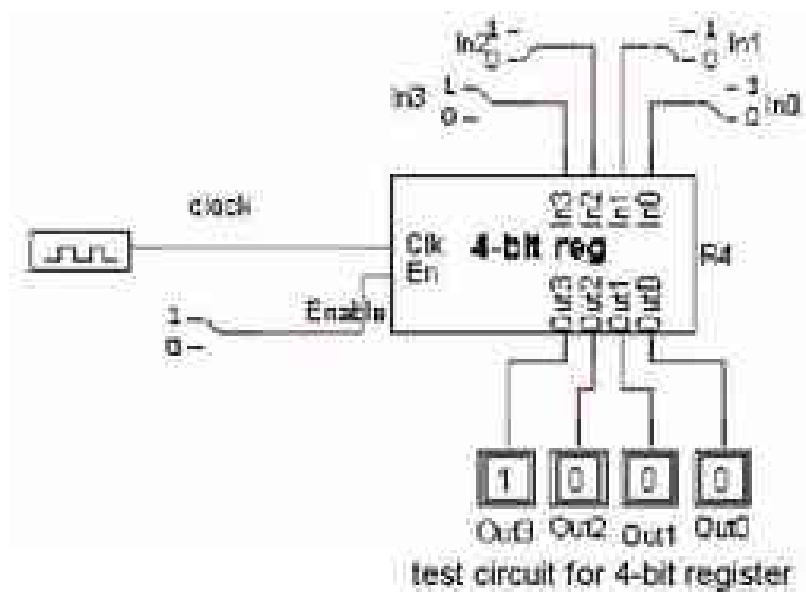
An n-bit register can be formed by grouping n flip-flops together. So a register is a device in which a group of flip-flops operate synchronously.

A register is useful for storing binary data, as each flip-flop can store one bit. The clock input of the flip-flops is grouped together, as is the enable input. As shown in the figure, using the input lines a binary number can be stored in the register by applying the corresponding logic level to each of the flip-flops simultaneously at the positive edge of the clock.

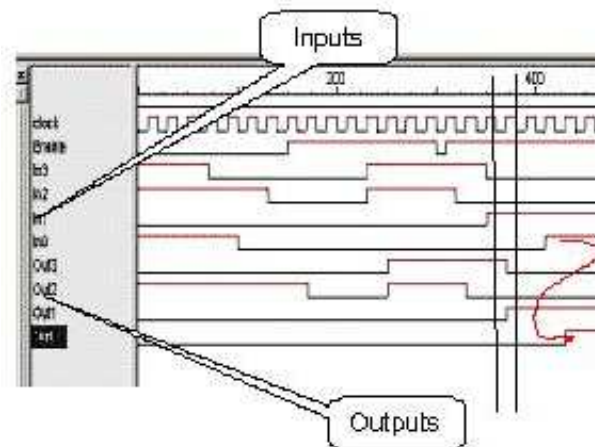
The next figure shows the symbol of a 4-bit register used for an integrated circuit. In0 through In3 are the four input lines, Out0 through Out3 are the four output lines, Clk is the clock input, and En is the enable line. To get a better understanding of this register, consider the situation where we want to store the binary number 1000 in the register. We will apply the number to the input lines, as shown in the figure given.

On the leading edge of the clock, the number will be stored in the register. The enable input has to be high if the number is to be stored into the register.





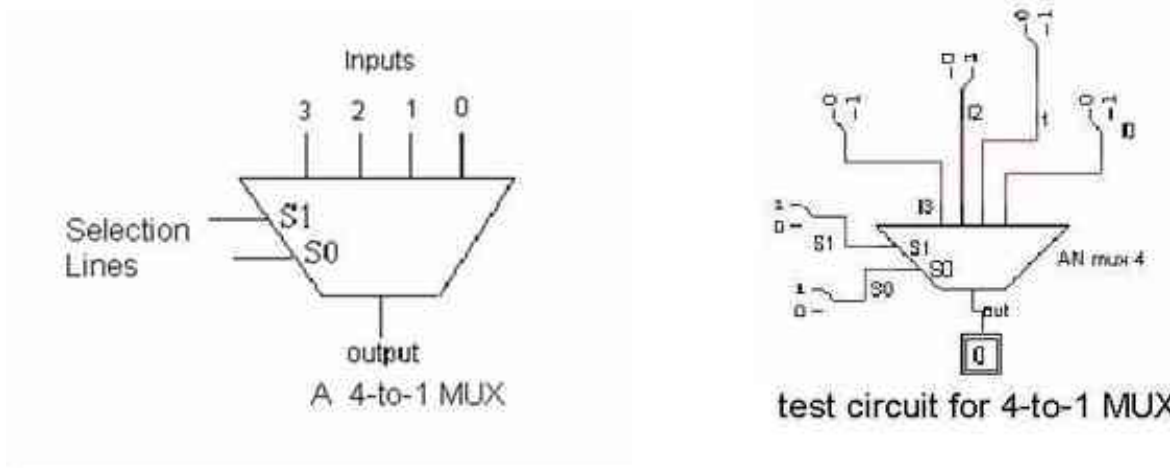
Waveform/Timing diagram



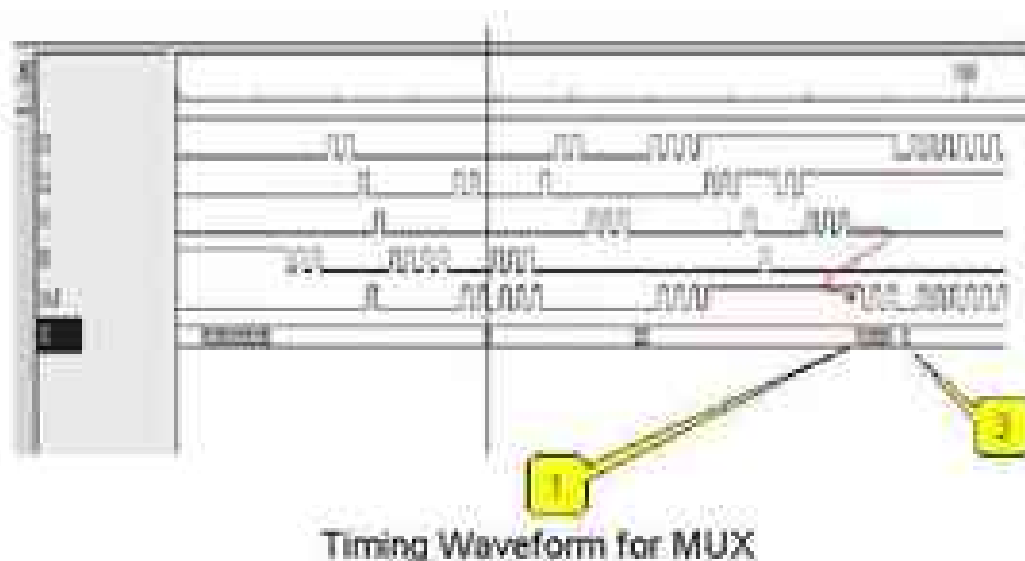
The n-to-1 multiplexer

A multiplexer is a device, constructed through combinational logic, which takes n inputs and transfers one of them as the output at a time. The input that is selected as the output depends on the selection lines, also called the control input lines. For an n -to-1 multiplexer, there are n input lines, $\log_2 n$ control lines, and 1 output line. The given figure shows a 4-to-1 multiplexer. There are 4 input lines; we number these lines as line 0 through line 3. Subsequently, there are 2 select lines (as $\log_2 4 = 2$).

For a better understanding, let us consider a case where we want to transfer the input of line 3 to the output of the multiplexer. We will need to apply the binary number 11 on the select lines (as the binary number 11 represents the decimal number 3). By doing so, the output of the multiplexer will be the input on line 3, as shown in the test circuit given.

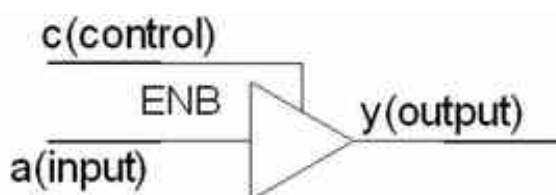


Timing waveform



Tri-state buffers

The tri-state buffer, also called the three-state buffer, is another important component in the digital logic domain. It has a single input, a single output, and an enable line. The input is concatenated to the output only if it is enabled through the enable line, otherwise it gives a high impedance output, i.e. it is tri-stated, or electrically disconnected from the input. These buffers are available both



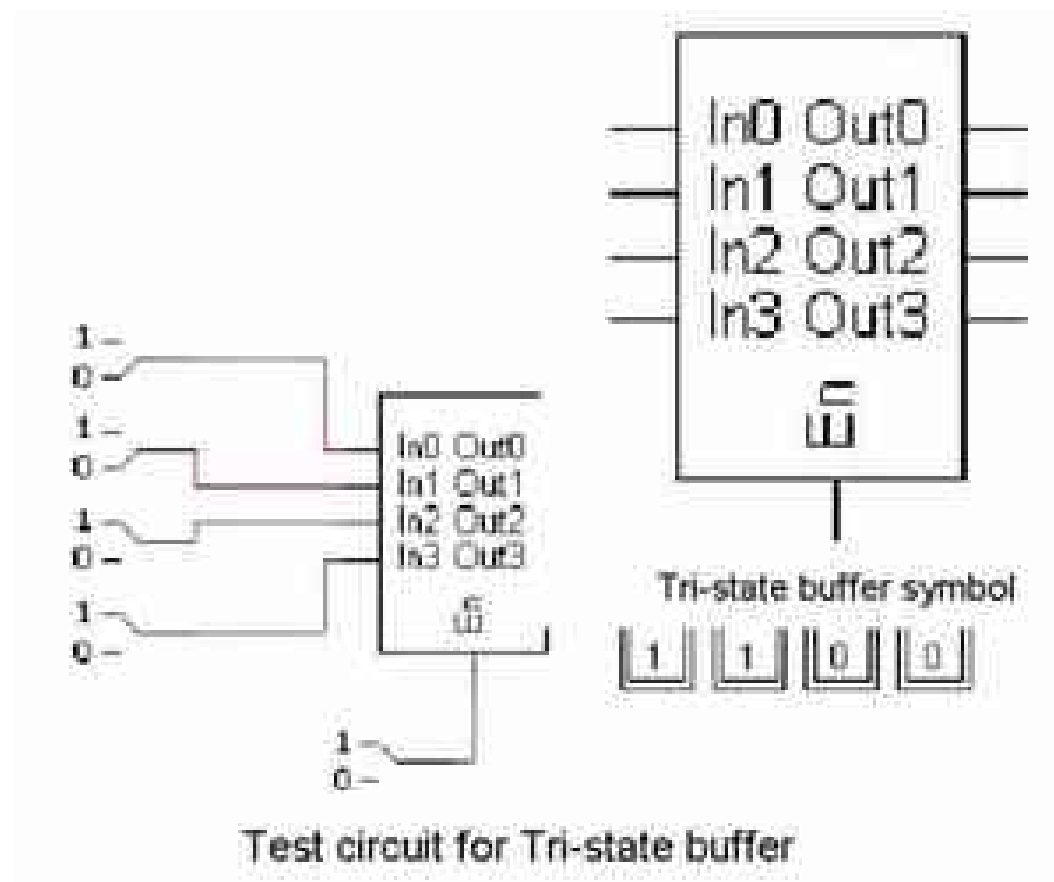
c	a	y
0	0	Z
0	1	Z
1	0	0
1	1	1

Truth table :Tri-state buffer

in the inverting and the non-inverting form. The inverting tri-state buffers output the 'inverted' input when they are enabled, as opposed to their non-inverting counterparts that simply output the input when enabled. The circuit symbol of the tri-state buffers is shown.

The truth table further clarifies the working of a non-inverting tri-state buffer.

We can see that when the enable input (or the control input) c is low (0), the output is high impedance Z . The symbol of a 4-bit tri-state buffer unit is shown in the figure. There are four input lines, an equal number of output lines, and an enable line in this unit. If we apply a high on the input 3 and 2, and a low on input 1 and 0, we get the output 1100, only when the enable input is high, as shown in the given figure.



Implementing simple register transfers

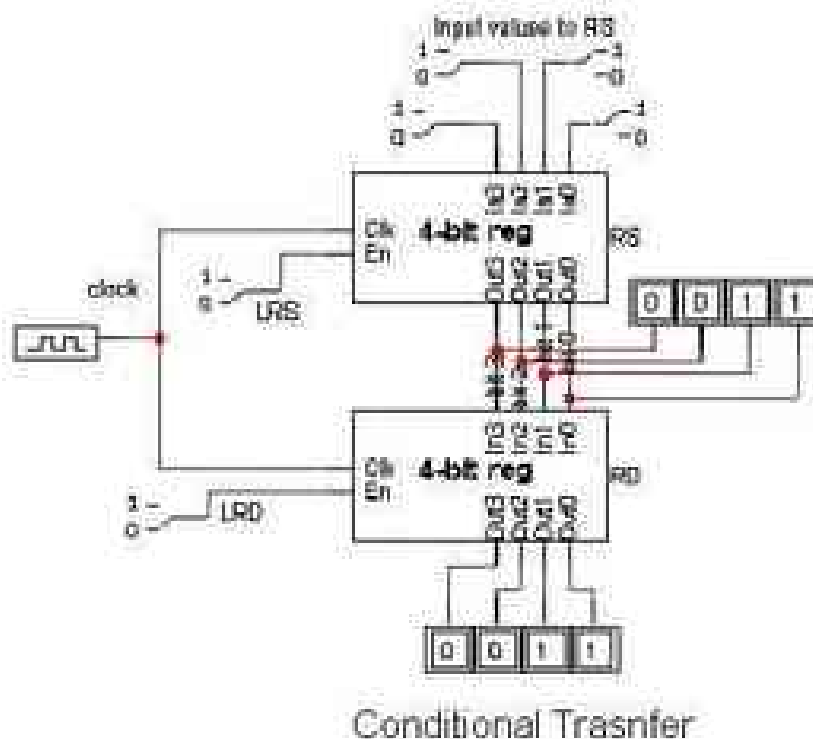
We now build on our knowledge of the primitive logic circuits to understand how register transfers are implemented. In this section we will study the implementation of the following

- Simple conditional transfer
- Concept of control signals
- Two-way transfers
- Connecting multiple registers
- Buses
- Bus implementations

Simple conditional transfer

In a simple conditional transfer, a condition is checked, and if it is true, the register transfer takes place. Formally, a conditional transfer is represented as

Cond: $RD \leftarrow RS$



This means that if the condition ‘Cond’ is true, the contents of the register named RS (the source register) are copied to the register RD (the destination register). The following figure shows how the registers may be interconnected to achieve a conditional transfer. In this circuit, the output of the source register RS is connected to the input of the destination registers RD. However, notice that the transfer will not take place unless the enable input of the destination register is activated. We may say that the ‘transfer’ is being controlled by the enable line (or the control signal). Now, we are able to control the transfer by selectively enabling the control signal, through the use of other combinational logic that may be the equivalent of our condition. The condition is, in general, a Boolean expression, and in this example, the condition is equivalent to $LRD = 1$.

Two-way transfers

In the above example, only one-way transfer was possible, i.e., we could only copy the contents of RS to RD if the condition was met. In order to be able to achieve two-way transfers, we must also provide a path from the output of the register RD to input of register RS. This will enable us to implement

Cond1: $RD \leftarrow RS$

Cond2: $RS \leftarrow RD$

Connecting multiple registers

We have seen how two registers can be connected. However, in a computer we need to connect more than just two registers. In order to connect these registers, one may argue that a connection between the input and output of each be provided. This solution is shown for a scenario where there are 5 registers that need to be interconnected.

We can see that in this solution, an m-bit register requires two connections of m-wires each. Hence five m-bit registers in a “point-to-point” scheme require 20 connections; each with m wires. In general, n registers in a point to point scheme require $n(n-1)$ connections. It is quite obvious that this solution is not going to scale well for a large number of registers, as is the case in real machines. The solution to this problem is the use of a bus architecture, which is explained in the following sections.

Buses

A bus is a device that provides a shared data path to a number of devices that are connected to it, via a 'set of wires' or a 'set of conductors'. The modern computer systems extensively employ the bus architecture. Control signals are needed to decide which two entities communicate using the shared medium, i.e. the bus, at any given time. This control signals can be open collector gate based, tri-state buffer based, or they can be implemented using multiplexers.



Multiple register connections

Register file implementation using the bus architecture

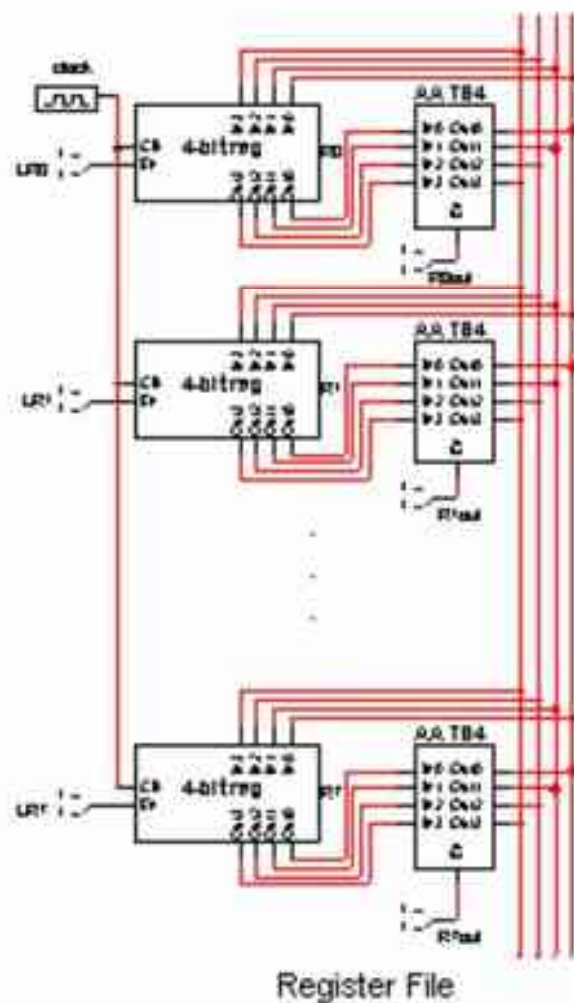
A number of registers can be inter-connected to form a register file, through the use of a bus. The given diagram shows eight 4-bit registers (R0, R1, ..., R7) interconnected through a 4-bit bus using 4-bit tri-state buffer units (labeled AA_TS4). The contents of a particular register can be transferred onto the bus by applying a logical high input on the enable of the corresponding tri-state buffer. For instance, R1out can be used to enable the tri-state buffers of the register R1, and in turn transfer the contents of the register on the bus.

Once the contents of a particular register are on the bus, the contents may be transferred, or read into any other register. More than one register may be written in this manner; however, only one register can write its value on the bus at a given time.

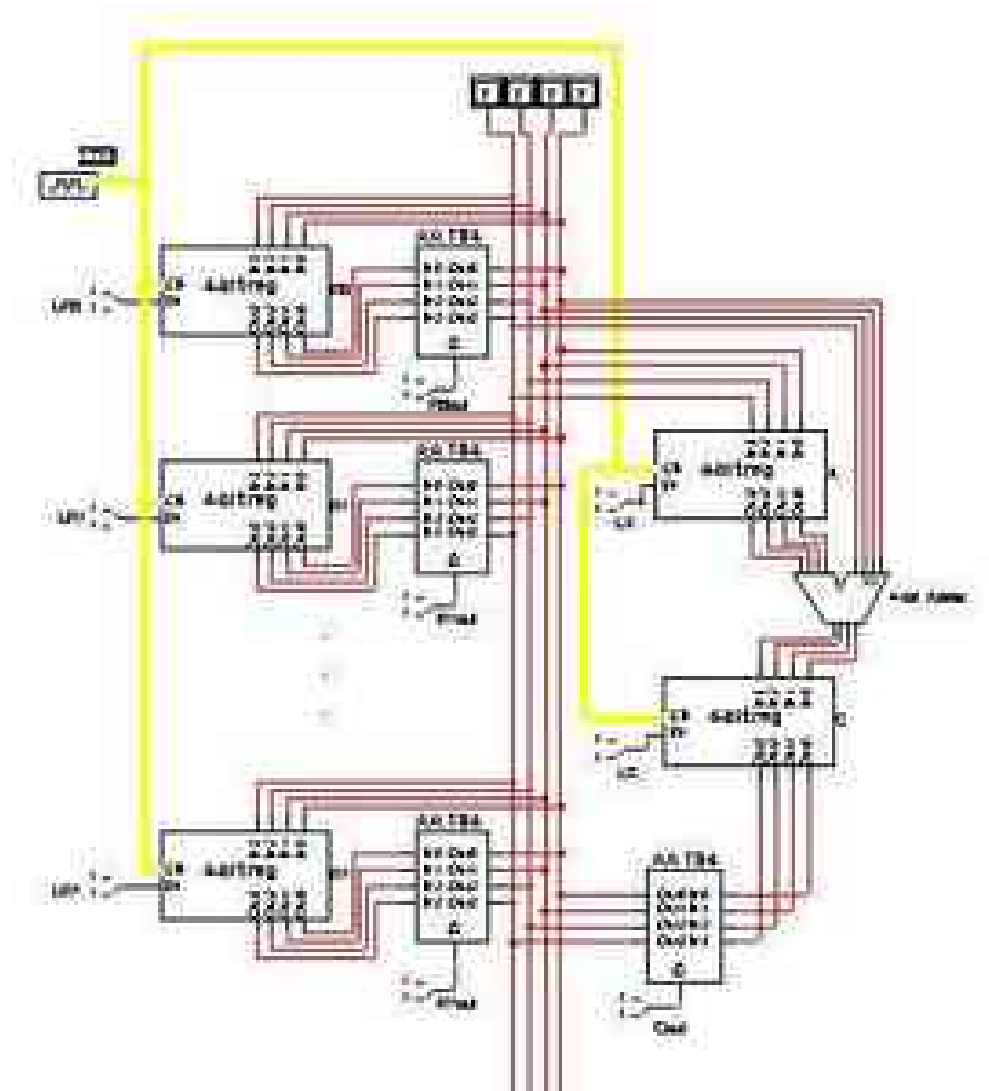
Implementing register transfers with mathematical operations

We have studied the implementation of simple register transfers; however, we frequently encounter register transfers with mathematical operations. An example is (opc=1): $R4 \leftarrow R3 + R2$;

These mathematical operations may be achieved by introducing appropriate combinational logic; the above operation can be implemented in hardware by including a 4-bit adder with the register files connected through the bus. There are two more registers in this configuration, one for holding one of the operands, and the other for holding the result before it is transferred to the destination register. This is shown in the figure below.



Register File



We now take a look at the steps taken for the (conditional, mathematical) transfer ($opc=1$): $R4 \leftarrow R3 + R2$. First of all, if the condition $opc = 1$ is met, the contents of the first operand register, R3, are transferred to the temporary register A through the bus. This is done by activating R3out.

Time step	Operation to be performed (structural RTL)	Control signals to be activated
1	$A \leftarrow R3$	LA, R3out
2	$C \leftarrow A + R2$	LC, R2out
3	$R4 \leftarrow C$	LR4, Cout

Structural RTL: add operation

It lets the contents of the register R3 to be loaded on the bus. At the same time, applying a logical high input to LA enables the load for the register A. This lets the binary number on the bus (the contents of register R3) to be loaded into the register A. The next step is to enable R2out to load the contents of the register R2 onto the bus. As can be observed from the figure, the output of the register A is one of the inputs to the 4-bit adder; the other input to the adder is the bus itself. Therefore, as the contents of register R2 are loaded onto the bus, both the operands are available

to the adder. The output can then be stored to the register RC by enabling its write. So a high input is applied to LC to store the result in register RC.

The third and final step is to store (transfer) the resultant number in the destination register R4. This is done by enabling Cout, which writes the number onto the bus, and then enabling the read of the register R4 by activating the control signal to LR4. These steps are summarized in the given table.

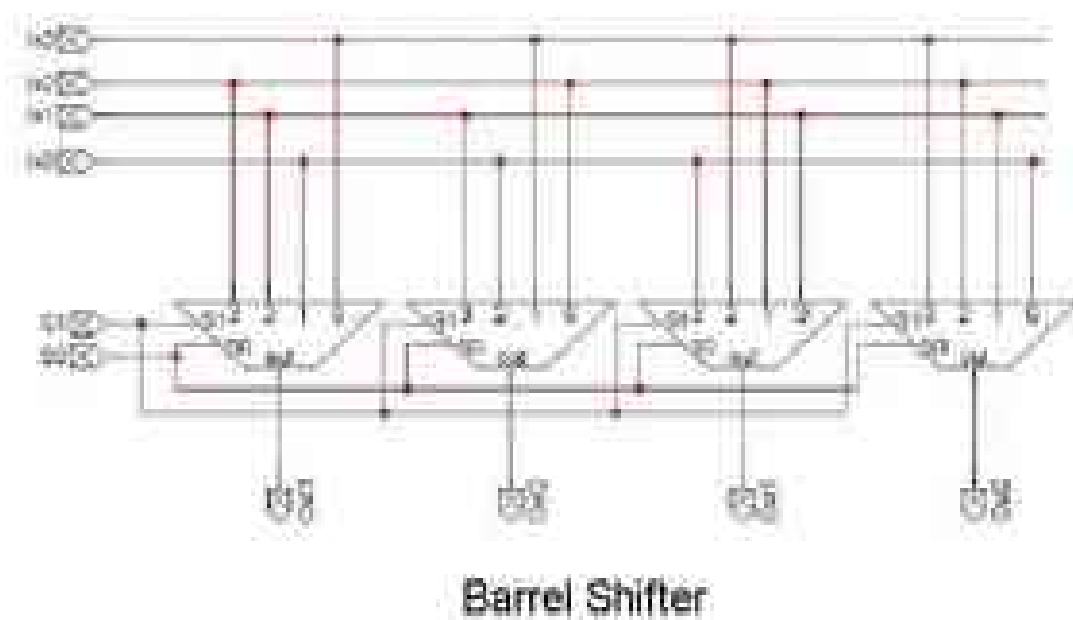
The barrel shifter

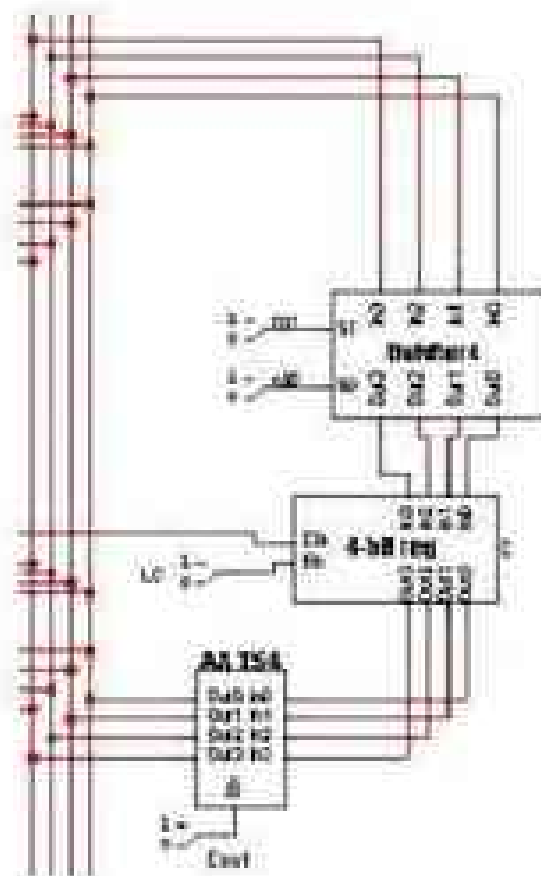
Shift operations are frequently used operations, as shifts can be used for the implementation of multiplication and division etc. A bi-directional shift register with a parallel load capability can be used to perform shift operations. However, the delays in such structures are dependent on the number of shifts that are to be performed, e.g., a 9 bit shift requires nine clock periods, as one shift is performed per clock cycle. This is not an optimal solution. The barrel shifter is an alternative, with any number of shifts accomplished during a single clock period. Barrel shifters are constructed by using multiplexers. An n-bit barrel shifter is a combinational circuit implemented using n multiplexers. The barrel provides a shifted copy of the input data at its output. Control inputs are provided to specify the number of times the input data is to be shifted. The shift process can be a simple one with 0s used as fillers, or it can be a rotation of the input data. The corresponding figure shows a barrel shifter that shifts right the input data; the number of shifts depends on the bit pattern applied on the control inputs S0, S1.

The function table for the barrel shifter is given. We see from the table that in order to apply single shift to the input number, the control signal is 01 on (S1, S0), which is the binary equivalent of the decimal number 1. Similarly, to apply 2 shifts, control signal 10 (on S1, S0) is applied; 10 is the binary equivalent of the decimal number 2. A control input of 11 shifts the number 3 places to the right.

Now we take a look at an example of the shift operation being implemented through the use of the barrel shifter: $R4 \leftarrow \text{ror } R3 \text{ (2 times)}$;

The shift functionality can be incorporated into the register file circuit with the bus architecture we have been building, by introducing the barrel shifter, as shown in the given figure.





Shift operation using Barrel Shifter



Barrel Shifter Symbol

S1	S0	Output in terms of the inputs
0	0	In3 In2 In1 In0
0	1	In0 In3 In2 In1
1	0	In1 In0 In3 In2
1	1	In2 In1 In0 In3

Function table: Barrel shifter

To perform the operation, $R4 \leftarrow \text{ror } R3$ (2 times),

The first step is to activate R3out, nb1 and LC. Activating R3out will load the contents of the register R3 onto the bus. Since the bus is directly connected to the input of the barrel shifter, this number is applied to the input side. nb1 and nb0 are the barrel shifter's control lines for specifying the number of shifts to be applied. Applying a high input to nb1 and a low input to nb0 will shift the number two places to the right. Activating LC will load the shifted output of the barrel shifter into the register C. The second step is to transfer the contents of the register C to the register R4. This is done by activating the control Cout, which will load the contents of register C onto the data bus, and by activating the control LR4, which will let the contents of the bus be written to the register R4. This will complete the conditional shift-and-store operation. These steps are summarized in the table shown below.

Time step	Operation to be performed (structural RTL)	Control signals to be activated
1	$C \leftarrow R3$ (after rotating right twice)	R3out, nb1, LC
2	$R4 \leftarrow C$	LR4, Cout

Structural RTL: Shift operation

Lecture No. 7

Design Process for ISA of FALCON-A

Reading Material

Handouts

Slides

Summary

- Outline of the thinking process for ISA Design
- Introduction to the ISA of FALCON-A

Instruction Set Architecture (ISA) Design: Outline of the thinking process

In this module we will learn to appreciate, understand and apply the approach adopted in designing an instruction set architecture. We do this by designing an ISA for a new processor. We have named our processor FALCON-A, which is an acronym for First Architecture for Learning Computer Organization and Networks (version A). The term Organization is intended to include Architecture and Design in this acronym.

Elements of the ISA

Before we go onto designing the instruction set architecture for our processor FALCON-A, we need to take a closer look at the defining components of an ISA. The following three key components define any instruction set architecture.

1. The operations the processor can execute
2. Data access mode for use as operands in the operations defined
3. Representation of the operations in memory

We take a look at all three of the components in more detail, and wherever appropriate, apply these steps to the design of our sample processor, the FALCON-A. This will help us better understand the approach to be adopted for the ISA design of a processor. A more detailed introduction to the FALCON-A will be presented later.

The operations the processor can execute

All processors need to support at least three categories (or functional groups) of instructions

- Arithmetic, Logic, Shift
- Data Transfer
- Control

ISA Design Steps – Step 1

We need to think of all the instructions of each type that ought to be supported by our processor, the FALCON-A. The following are the instructions that we will include in the ISA for our processor.

Arithmetic:

add, addi (and with an immediate operand), subtract, subtract-immediate, multiply, divide

Logic:

and, and-immediate, or, or-immediate, not

Shift:

shift left, shift right, arithmetic shift right

Data Transfer:

Data transfer between registers, moving constants to registers, load operands from memory to registers, store from registers to memory and the movement of data between registers and input/output devices

Control:

Jump instructions with various conditions, call and return from subroutines, instructions for handling interrupts

Miscellaneous instructions:

Instructions to clear all registers, the capability to stop the processor, ability to “do nothing”, etc.

ISA Design Steps – Step 2

Once we have decided on the instructions that we want to add support for in our processor, the second step of the ISA design process is to select suitable mnemonics for these instructions. The following mnemonics have been selected to represent these operations.

Arithmetic:

add, addi, sub, subi, mul, div

Logic:

and, andi, or, ori, not

Shift:

shifl, shiftr, asr

Data Transfer:

load, store, in, out, mov, movi

Control:

jpl, jmi, jnz, jz, jump, call, ret, int.iret

Miscellaneous instructions:

nop, reset, halt

ISA Design Steps – Step 3

The next step of the ISA design is to decide upon the number of bits to be reserved for the op-code part of the instructions. Since we have 32 instructions in the instruction set, 5 bits will suffice (as $2^5 = 32$) to encode these op-codes.

ISA Design Steps – Step 4

The fourth step is to assign op-codes to these instructions. The assigned op-codes are shown below.

Arithmetic:

add (0), addi (1), sub (2), subi (3), mul (4), div (5)

Logic:

and (8), andi (9), or (10), ori (11), not (14)

Shift:

shifl (12), shiftr (13), asr (15)

Data Transfer:

load (29), store (28), in (24), out (25), mov (6), movi (7)

Control:

jpl (16), jmi (17), jnz (18), jz (19), jump (20), call (22), ret (23), int (26), irect (27)

Miscellaneous instructions:

nop (21), reset (30), halt (31)

Now we list these instructions with their op-codes in the binary form, as they would appear in the machine instructions of the

FALCON-A.

00000	add	01000	addi	10000	sub	11000	subi
00001	addi	01001	andi	10001	mul	11001	div
00010	sub	01010	ori	10010	jpl	11010	jmi
00011	subi	01011	not	10011	jump	11011	jnz
00100	mul	01100	asr	10100	call	11100	int
00101	div	01101	shifl	10101	ret	11101	irect
00110	andi	01110	shiftr	10110	call	11110	reset
00111	ori	01111	asr	10111	ret	11111	halt

Data access mode for operations

As mentioned earlier, the instruction set architecture of a processor defines a number of things besides the instructions implemented; the resources each instruction can access,

the number of registers available to the processor, the number of registers each instruction can access, the instructions that are allowed to access memory, any special registers, constants and any alternatives to the general-purpose registers. With this in mind, we go on to the next steps of our ISA design.

ISA Design Steps – Step 5

We now need to select the number and types of operands for various instructions that we have selected for the FALCON-A ISA.

ALU instructions may have 2 to 3 registers as operands. In case of 2 operands, a constant (an immediate operand) may be included in the instruction.

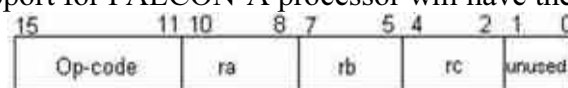
For the load/store type instructions, we require a register to hold the data that is to be loaded from the memory, or stored back to the memory. Another register is required to hold the base address for the memory access. In addition to these two registers, a field is required in the instruction to specify the constant that is the displacement to the base address.

In jump instructions; we require a field for specifying the register that holds the value that is to be compared as the condition for the branch, as well as a destination address, which is specified as a constant.

Once we have decided on the number and types of operands that will be required in each of the instruction types, we need to address the issue of assigning specific bit-fields in the instruction for each of these operands. The number of bits required to represent each of these operands will eventually determine the instruction word size. In our example processor, the FALCON-A, we reserve eight general-purpose registers. To encode a register in the instructions, 3 bits are required (as $2^3=8$). The registers are encoded in the binary as shown in the given table.

Registers	Encoding
R0	000
R1	001
R2	010
R3	011
R4	100
R5	101
R6	110
R7	111

Therefore, the instructions that we will add support for FALCON-A processor will have the given general format. The instructions in the FALCON-A processor are going to be variations of this format, with four different formats in all. The exact format is dependent on the actual number of operands in a particular instruction.



ISA Design Steps – Step 6

The next step towards completely defining the instruction set architecture of our processor is the design of memory and its organization. The number of the memory cells that we may have in the organization depends on the size of the Program Counter register (PC), and the size of the address bus. This is because the size of the program counter and the size of the address bus put a limitation on the number of memory cells that can be referred to for loading an instruction for execution. Additionally, the size of the data bus puts a limitation on the size of the memory word that can be referred to in a single clock cycle.

ISA Design Steps – Step 7

Now we need to specify which instructions will be allowed to access the memory. Since the FALCON-A is intended to be a RISC-like machine; only the load/ store instructions will be allowed to access the memory.

ISA Design Steps – Step 8

Next we need to select the memory-addressing modes. The given table lists the types of addressing modes that will be supported for the load/store instructions

Addressing Mode	Format	Example
Register	[register offset]	[R1, #4]
Displacement	[register + constant offset]	[R1, #5] or [R1+5]
Register indirect	[register]	[R3]

FALCON-A: Introduction

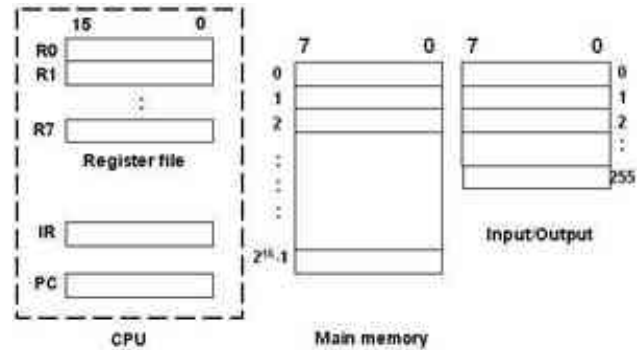
FALCON stands for First Architecture for Learning Computer Organization and Networks. It is a 'RISC-like' general-purpose processor that will be used as a teaching aid for this course. Although the FALCON-A is a simple machine, it is powerful enough to explain a variety of fundamental concepts in the field of Computer Architecture.

Programmer's view of the FALCON-A

FALCON-A, an example of a GPR (General Purpose Register) computer, is the first version of the FALCON processor. The programmer's view of the FALCON-A is given in the figure

shown. As it is clear from the figure, the CPU contains a register file of 8 registers, named R0 through R7. Each of these registers is 16 bits in length. Aside from these registers, there are two special-purpose registers, the Program Counter (PC), and the Instruction Register (IR). The main memory is organized as $2^{16} \times 8$ bits, i.e. 2^{16} cells of 1 byte each. The memory word size is 2 bytes (or 16 bits).

The input/output space is 256 bytes (8 bit I/O ports). The storage in these registers and memory is in the big-endian format.



Lecture No. 8

ISA of the FALCON-A

Reading Material

Handouts

Slides

Summary

- Introduction to the ISA of the FALCON-A
- Examples for the FALCON-A

Introduction to the ISA of the FALCON-A

We take a look at the notation that we are going to employ when studying the FALCON-A. We will refer to the contents of a register by enclosing in square brackets the name of the register, for instance, R [3] refers to the contents of the register 3. Memory contents are to be referred to in a similar fashion; for instance, M [8] refers to the contents of memory at location 8 (or the 8th memory cell).

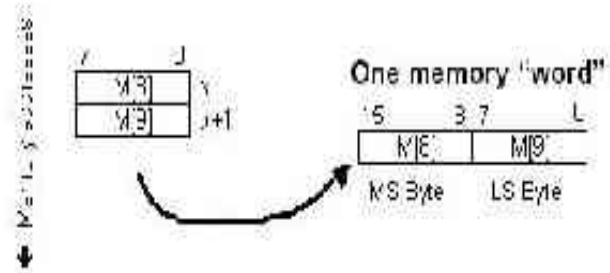


Fig. Big- Endian Notation

Since memory is organized into cells of 1 byte, whereas the memory word size is 2 bytes, two adjacent memory cells together make up a memory word. So, memory word at the memory address 8 would be defined as 1 byte at address 8 and 1 byte at address 9. To refer to 16-bit memory words, we make use of a special notation, the concatenation of two memory locations. Therefore, to refer to the 16-bit memory word at location 8, we would write M[8]©M[9]. As we employ the big-endian format,

$M[8] \langle 15 \dots 0 \rangle := M[8] \text{©} M[9]$

So in our notation © is used to represent concatenation.

Little endian puts the smallest numbered byte at the least-significant position in a word, whereas in big endian, we place the largest numbered byte at the most significant position. Note that in our case, we use the big-endian convention of ordering bytes. However, within each byte itself, the ordering of the bits is little endian.

FALCON-A Features

The FALCON-A processor has fixed-length instructions, each 16 bits (2 bytes) long. Addressing modes supported are limited, and memory is accessed through the load/store instructions only.

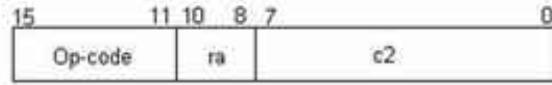
FALCON-A Instruction Formats

Three categories of instructions are going to be supported by the FALCON-A processor; arithmetic, control, and data transfer instructions. Arithmetic instructions enable mathematical computations. Control instructions help change the flow of the program as and when required. Data transfer operations move data between the processor and memory. The arithmetic category also includes the logical instructions. Four different types of instruction formats are used to specify these instructions. A brief overview of the various fields in these instructions formats follows.

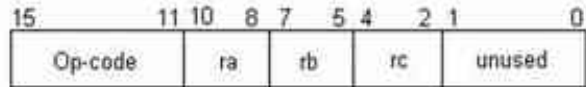
Type I instruction format is shown in the given figure. In it, 5 bits are reserved for the op-code (bits 11 through 15). The rest of the bits are unused in this instruction type, which means they are not considered.



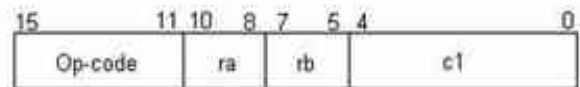
Type II instruction shown in the given figure, has a 5-bit op-code field, a 3-bit register field, and an 8-bit constant (or immediate operand) field.



Type III instructions contain the 5-bit op-code field, two 3-bit register fields for source and destination registers, and an immediate operand field of length 5 bits.



Type IV instructions contain the op-code field, two 3-bit register fields, a constant filed on length 3 bits as well as two unused bits. This format is shown in the given figure.



Encoding of registers

We have a register file comprising of eight general-purpose registers in the CPU. To encode these registers in the binary, so they can be referred to in various instructions, we require $\log_2(8) = 3$ bits. Therefore, we have already allocated three bits per register in the instructions, as seen in the various instruction formats. The encoding of registers in the binary format is shown in the given table.

It is important to note here that the register R0 has special usage in some cases. For instance, in load/ store operations, if register R0 is used as a second operand, its value is considered to be zero. R0 has special usage in the multiply and divide (mul & div) instructions as well.

Registers	Encoding
R0	000
R1	001
R2	010
R3	011
R4	100
R5	101
R6	110
R7	111

Fig. Register Encodings

Instructions and instruction formats

We return to our discussion of instruction formats in this section. We will now classify which instructions belong to what instruction format types.

Type I

Five of the instructions included in the instruction set of FALCON-A belong to type I instruction format. These are

1.nop (op-code = 21)

This instruction is to instruct the processor to ‘do nothing’, or, in other words, do ‘no operation’. This instruction is generally useful in pipelining. We will study pipelining later in the course.

2.reset (op-code = 30)

3.halt (op-code=31)

4.int (opcode= 26)

5.iret (op-code= 27)

All of these instructions take no operands, therefore, besides the 5 bits used for the op-code, the rest of the bits are unused.

Type II

There are nine FALCON-A instructions that belong to this type. These are listed below.

1. movi (op-code = 7)

The movi instruction loads a register with the constant (or the immediate value) specified as the second operand. An example is

movi R3, 56 $R[3] \leftarrow 56$

This means that the register R3 will have the value 56 stored in it as this instruction is executed.

2. in (op-code = 24)

This instruction is to load the specified register from input device. An example and its interpretation in register transfer language are

in R3, 57 $R[3] \leftarrow IO[57]$

3. out (op-code = 25)

The 'out' instruction will move data from the register to the output device specified in the instruction, as the example demonstrates:

out R7, 34 $IO[34] \leftarrow R[7]$

4. ret (op-code=23)

This instruction is to return control from a subroutine. This is done using a register, where the return address is stored. As shown in the example, to return control, the program counter is assigned the contents of the register.

ret R3 $PC \leftarrow R[3]$

5. jz (op-code= 19)

When this instruction is executed, the value of the register specified in the field ra is checked, and if it is equal to zero, the Program Counter is advanced by the jump(value) specified in the instruction.

jz r3, [4] $(R[3]=0): PC \leftarrow PC + 4;$

In this example, register r3's value is checked, and if found to be zero, PC is advanced by 4.

6. jnz (op-code= 18) This instruction is the reverse of the jz instruction, i.e., the jump (or the branch) is taken, if the contents of the register specified are not equal to zero. jnz r4, [variable] $(R[4] \neq 0): PC \leftarrow PC + \text{variable};$

7. jpl (op-code= 16) In this instruction, the value contained in the register specified in the field ra is checked, and if it is positive, the jump is taken. jpl r3, [label] $(R[3] \geq 0): PC \leftarrow PC + (\text{label} - PC);$

8. jmi (op-code= 17) In this case, PC is advanced (jump/branch is taken) if the register value is negative jmi r7, [address] $(R[7] < 0): PC \leftarrow PC + \text{address};$

Note that, in all the instructions for jump, the jump can be specified by a constant, a variable, a label or an address (that holds the value by which the PC is to be advanced). A variable can be defined through the use of the '.equ' directive. An address (of data) can be specified using the directive '.db' or '.dw'. A label can be specified with any instruction. In its usage, we follow the label by a colon ':' before the instruction itself. For example, the following is an instruction that has a label 'alfa' attached to it alfa: movi r3 r4

Labels implement relative jumps, 128 locations backwards or 127 locations forward (relative to the current position of program control, i.e. the value in the program counter). The compiler handles the interpretation of the field c2 as a constant/ variable/ label/ address. The machine code just contains an 8-bit constant that is added to the program counter at run-time.

9. jump (op-code= 20)

This instruction instructs the processor to advance the program counter by the displacement specified, unconditionally (an unconditional jump). The assembler allows the displacement (or the jump) to be specified in any of the following ways

```
jump [ra + constant]
jump [ra + variable]
jump [ra + address]
jump [ra + label]
```

The types of unconditional jumps that are possible are

- Direct
- Indirect
- PC relative (a ‘near’ jump)
- Register relative (a ‘far’ jump)

The c2 field may be a constant, variable, an address or a label.

A direct jump is specified by a PC-label.

An indirect jump is implemented by using the C2 field as a variable.

In all of the above instructions, if the value of the register ra is zero, then the Program Counter is incremented (or decremented) by the sign-extended value of the constant specified in the instruction. This is called the PC-relative jump, or the ‘near’ jump. It is denoted in RTL as:

$(ra = 0): PC \leftarrow PC + (8\alpha C2<7>) \odot C2<7..0>;$

If the register ra field is non-zero, then the Program Counter is assigned the sum of the sign-extended constant and the value of register specified in the field ra. This is known as the register-relative, or the ‘far’ jump. In RTL, this is denoted as: $(ra \neq 0): PC \leftarrow R[ra] + (8\alpha C2<7>) \odot C2<7..0>;$

Note that C2 is computed by sign extending the constant, variable, address, or (label – PC). Since we have 8 bits available for the C2 field (which can be a constant, variable, address or a PC-label), the range for the field is -128 to + 127. Also note that the compiler does not allow an instruction with a negative sign before the register name, such as ‘jump [-r2]’. If the C2 field is being used as an address, it should always be an even value for the jump instruction. This is because our instruction word size is 16 bits, whereas in instruction memory, the instruction memory cells are of 8 bits each. Two consecutive cells together make an instruction.

Type III

There are nine instructions of the FALCON-A that belong to Type III. These are:

1. andi (op-code = 9)

The andi instruction bit-wise ‘ands’ the constant specified in the instruction with the value stored in the register specified in the second operand register and stores the result in the destination register. An example is:

```
andi r4, r3, 5
```

This instruction will bit-wise and the constant 5 and R[3], and assign the value thus obtained to the register R[4], as given .

$R[4] \leftarrow R[3] \& 5$

2. addi (op-code = 1)

This instruction is to add a constant value to a register; the result is stored in a

destination register. An example:

`addi r4, r3, 4` $R[4] \leftarrow R[3] + 4$

3. **subi** (op-code = 3)

The subi instruction will subtract the specified constant from the value stored in a source register, and store to the destination register. An example follows.

`subi r5, r7, 9` $R[5] \leftarrow R[7] - 9$

4. **ori** (op-code= 11)

Similar to the andi instruction, the ori instruction bit-wise ‘ors’ a constant with a value stored in the source register, and assigns it to the destination register. The following instruction is an example.

`ori r4, r7, 3` $R[4] \leftarrow R[7] \sim 3$

5. **shl** (op-code = 12)

This instruction shifts the value stored in the source register (which is the second operand), and shifts the bits left as many times as is specified by the third operand, the constant value. For instance, in the instruction
`shl r4, r3, 7`

The contents of the register are shifted left 7 times, and the resulting number is assigned to the register r4.

6. **shfr** (op-code = 13)

This instruction shifts to the right the value stored in a register. An example is:
`shfr r4, r3, 9`

7. **asr** (op-code = 15)

An arithmetic shift right is an operation that shifts a signed binary number stored in the source register (which is specified by the second operand), to the right, while leaving the sign-bit unchanged. A single shift has the effect of dividing the number by 2. As the number is shifted as many times as is specified in the instruction through the constant value, the binary number of the source register gets divided by the constant value times 2.

An example is `asr r1, r2, 5`

This instruction, when executed, will divide the value stored in r2 by 10, and assign the result to the register r1.

8. **load** (op-code= 29)

This instruction is to load a register from the memory. For instance, the Instruction

`load r1, [r4 +15]`

will add the constant 15 to the value stored in the register r4, access the memory location that corresponds to the number thus resulting, and assign the memory contents of this location to the register r1; this is denoted in RTL by:

$R[1] \leftarrow M[R[4]+15]$

9. **store** (op-code= 28)

This instruction is to store a value in the register to a particular memory location.

In the example:

`store r6, [r7+13]`

The contents of the register r6 are being stored to the memory location that corresponds to the sum of the constant 13 and the value stored in the register r7.

$M[R[7]+13] \leftarrow R[6]$

Type III Modified

There are 3 instructions in the modified form of the Type III instructions. In the modified Type III instructions, the field c1 is unused.

1.mov (op-code = 6)

This instruction will move (copy) data of a source register to a destination register. For instance, in the following example, the contents of the register r3 are copied to the register r4.

mov r4, r3

In RTL, this can be represented as

$$R[4] \leftarrow R[3]$$

2.not (op-code = 14)

This instruction inverts the contents of the source register, and assigns the value thus obtained to the destination register. In the following example, the contents of register r2 are inverted and assigned to register r4.

not r4, r2

In RTL:

$$R[4] \leftarrow !R[2]$$

3.call (op-code = 22)

Procedure calls are often encountered in programming languages. To add support for procedure (or subroutine) calls, the instruction call is used. This instruction first stores the return address in a register and then assigns the Program Counter a new value (that specifies the address of the subroutine). Following is an example of the call instruction

call r4, r3

This instruction saves the current contents (the return address) of the Program Counter into the register r4 and assigns the new value to the PC from register r3.

$$R[4] \leftarrow PC, PC \leftarrow R[3]$$

Type IV

Six instructions belong to the instruction format Type IV. These are

1.add (op-code = 0)

This instruction adds contents of a register to those of another register, and assigns to the destination register. An example:

add r4, r3, r5

$$R[4] \leftarrow R[3] + R[5]$$

2.sub (op-code = 2)

This instruction subtracts value of a register from another the value stored in another register, and assigns to the destination register. For example,

sub r4, r3, r5

In RTL, this is denoted by

$$R[4] \leftarrow R[3] - R[5]$$

3.mul (op-code = 4)

The multiply instruction will store the product of two register values, and stores in the destination register. An example is

mul r5, r7, r1

The RTL notation for this instruction will be

$$R[5] \leftarrow R[7] * R[1]$$

4.div (op-code= 5)

This instruction will divide the value of the register that is the second operand, by the number in the register specified by the third operand, and assign the result to the destination register.

div r4, r7, r2 $R[4] \leftarrow R[7] / R[2], R[0] \leftarrow R[7] \% R[2]$

5. and (op-code= 8)

This ‘and’ instruction will obtain a bit-wise ‘and’ of the values of two registers and assigns it to a destination register. For instance, in the following example, contents of register r4 and r5 are bit-wise ‘anded’ and the result is assigned to the register r1.

and r1, r4, r5

In RTL we may write this as

$R[1] \leftarrow R[4] \& R[5]$

6. or (op-code= 10)

To bit-wise ‘or’ the contents of two registers, this instruction is used. For instance, or r6, r7, r2

In RTL this is denoted as

$R[6] \leftarrow R[7] \sim R[2]$

FALCON-A: Instruction Set Summary

We have looked at the various types of instruction formats for the FALCON-A, as well as the instructions that belong to each of these instruction format types. In this section, we have simply listed the instructions on the basis of their functional groups; this means that the instructions that perform similar class of operations have been listed together.

Data Transfer Instructions	Mnemonic	opcode
move	mov	00110 (6)
Move immediate	movi	00111 (7)
Input to register	in	11000 (24)
Output from register	out	11001 (25)
Load from memory	load	11101 (29)
Store into memory	store	11100 (28)

Fig. Data Transfer Instructions

Jump Instruction	Mnemonic	opcode
jump if positive	jp	10000 (16)
jump if negative	jn	10001 (17)
jump if not zero	jnz	10010 (18)
jump if zero	je	10011 (19)
jump	jump	10100 (20)

Fig. Jump Instructions

Control Instruction	Mnemonic	opcode
No operation	nop	10101 (21)
call	call	10110 (22)
return	ret	10111 (23)
interrupt	int	11010 (26)
interrupt return	int	11011 (27)
movl	movl	11110 (30)
hll	hll	11111 (31)

Fig. Control Instructions

Examples for FALCON-A

In this section we take up a few sample problems related to the FALCON-A processor. This will enhance our understanding of the FALCON-A processor, as well as of the general concepts related to general processors and their instruction set architectures. The problems we will look at include

1. Identification of the instruction types and operands
2. Addressing modes and RTL description
3. Branch condition and status of the PC
4. Binary encoding for instructions

Example 1:

Identify the types of given FALCON-A instructions and specify the values in the fields

Instruction	Type	ra	rb	rc	c1	c2
movl r1, 2						
add r1, r2, r3						
nop						
load r2, [r5 + 4]						
je r0, [3]						

Fig. Example 1

Solution

The solution to this problem is quite straightforward. The types of these instructions, as well as the fields, have already been discussed in the preceding sections.

Instruction	Type	ra	rb	rc	c1	c2
movi r1, 2	II	r1	-	-	-	2
add r1, r2, r3	IV	r1	r2	r3	-	-
nop	I	-	-	-	-	-
load r2, [r3 + 6]	III	r2	r3	-	6	-
je r0, [3]	II	r0	-	-	-	3

Fig. Solution 1

We can also find the machine code for these instructions. The machine code (in the hexadecimal representation) is given for these instructions in the given table.

Instruction	Machine Code	ra	rb	rc	c1	c2
movi r1, 2	3902h	r1	-	-	-	2
add r1, r2, r3	014Ch	r1	r2	r3	-	-
nop	A800h	-	-	-	-	-
load r2, [r3 + 6]	EAAGh	r2	r3	-	6	-
je r0, [3]	9803h	r0	-	-	-	3

Fig. Machine Code

Example 2:

Identify the addressing modes and Register Transfer Language (RTL) description (meaning) for the given FALCON-A instructions

Instruction	Addressing mode	RTL description (meaning)
load r2, [r4 + 8]		
jmp r1, [54]		
shiftl r1, r2, 4		
addl r1, r6, 2		
sub r1, r7, r2		

Fig. Example 2

Solution

Addressing modes relate to the way architectures specify the address of the objects they access. These objects may be constants and registers, in addition to memory locations.

Instruction	Addressing mode	RTL description (meaning)
load r2, [r4 + 8]	Displacement	$R[2] \leftarrow M[R[4] + 8]$
jmp r1, [54]	Relative	$(R[1] \neq 0):$ $PC \leftarrow PC + 54$
shiftl r1, r2, 4	Immediate	Shift r2 left 4 times and store in r1
addl r1, r6, 2	Immediate	$R[3] \leftarrow R[6] + 2$
sub r1, r7, r2	Register	$R[1] \leftarrow R[7] - R[2]$

Fig. Solution 2

Example 3: Specify the condition for the branch instruction and the status of the PC after the branch instruction executes with a true branch condition

Instruction	Condition	PC status
je r2, [35]		
jump [12]		
jnz r6, [3]		
jpl r1, [45]		
jmi r2, [20]		

Fig. Example 3

Solution

We have looked at the various jump instructions in our study of the FALCON-A. Using that knowledge, this problem can be solved easily.

Instruction	Condition	PC status
je r2, [35]	if R[2] == 0	PC $\leftarrow$ PC+35
jump [12]	always	PC $\leftarrow$ PC+12
jnz r6, [3]	if R[6] $\neq$ 0	PC $\leftarrow$ PC+3
jpl r1, [45]	if R[1] $\geq$ 0	PC $\leftarrow$ PC+45
jmi r2, [20]	if R[2] < 0	PC $\leftarrow$ PC+20

Fig. Solution 3

Example 4: Specify the binary encoding of the different fields in the given FALCON-A instructions.

Instruction	TYPE	op=4b	ra	rb	rc	C1@4bit OR C2@4bit
store R ₁ , [R ₂ +R ₃]						
add R ₁ , R ₂ , R ₃						
addR ₁ , R ₂ , R ₃						
jump [R ₂]						
halt						

Fig. Example 4

Solution

We can solve this problem by referring back to our discussion of the instruction format types. The op-codes for each of the instructions can also be looked up from the tables. ra, rb and rc (where applicable) registers' values are obtained from the register encoding table we looked at. The constants C1 and C2 are there in instruction type III and II respectively. The immediate constant specified in the instruction can also be simply converted to binary, as shown.

Instruction	TYPE	op=4b	ra	rb	rc	C1@4bit OR C2@4bit
store R ₁ , [R ₂ +R ₃]	II	1110	300	001	-	01000
add R ₁ , R ₂ , R ₃	IV	0000	011	110	101	-
addR ₁ , R ₂ , R ₃	III	0110	100	110	-	01001
jump [R ₂]	I	1010	-	-	-	0000 1010
halt	I	1111	-	-	-	-

Fig. Solution 4

Lecture No. 9

Description of FALCON-A and EAGLE using RTL

Reading Material

Handouts

Slides

Summary

- Use of Behavioral Register Transfer Language (RTL) to describe the FALCON-A
- The EAGLE
- The Modified EAGLE

Use of Behavioral Register Transfer Language (RTL) to describe the FALCON-A

The use of RTL (an acronym for the Register Transfer Language) to describe the FALCON-A is discussed in this section. FALCON-A is the sample machine we are building in order to enhance our understanding of processors and their architecture.

Behavior vs. Structure

Computer design involves various levels of abstraction. The behavioral description of a machine is a higher level of abstraction, as compared with the structural description. Top-down approach is adopted in computer design. Designing a computer typically starts with defining the behavior of the overall system. This is then broken down into the behavior of the different modules. The process continues, till we are able to define, design and implement the structure of the individual modules.

As mentioned earlier, we are interested in the behavioral description of our machine, the FALCON-A, in this section.

Register Transfer Language

The RTL is a formal way of expressing the behavior and structure of a computer.

Behavioral RTL

Behavioral Register Transfer Language is used to describe what a machine does, i.e. it is used to define the functionality the machine provides. Basically, the behavioral architecture describes the algorithms used in a machine, written as a set of process statements. These statements may be sequential statements or concurrent statements, including signal assignment statements and wait statements.

Structural RTL

Structural RTL is used to describe the hardware implementation of the machine. The structural architecture of a machine is the logic circuit implementation (components and their interconnections), that facilitates a certain behavior (and hence functionality) for that machine.

Using RTL to describe the static properties of the FALCON-A

We can employ the RTL for the description of various properties of the FALCON-A that we have already discussed.

Specifying Registers

In RTL, we will refer to a register by its abbreviated, alphanumeric name, followed by the number of bits in the register enclosed in angle brackets '<>'. For instance, the instruction register (IR), of 16 bits (numbered 0 to 15), will be referred to as, IR<15..0>

Naming of the Fields in a Register

We can name the different fields of a register using the $:=$ notation. For example, to name the most significant bits of the instruction register as the operation code (or simply op), we may write:

$op<4..0> := IR<15..11>$

Note that using this notation to name registers or register fields will not create a new copy of the data or the register fields; it is simply an alias for an already existing register, or part of a register.

Fields in the FALCON-A Instructions

We now use the RTL naming operator to name the various fields of the RTL instructions. Naming the fields appropriately helps us make the study of the behavior of a processor more readable.

$op<4..0> := IR<15..11>:$	operation code field
$ra<2..0> := IR<10..8>:$	target register field
$rb<2..0> := IR<7..5>:$	operand or address index
$rc<2..0> := IR<4..2>:$	second operand
$c1<4..0> := IR<4..0>:$	short displacement field
$c2<7..0> := IR<7..0>:$	long displacement or the immediate field

We are already familiar with these fields, and their usage in the various instruction formats of the RTL.

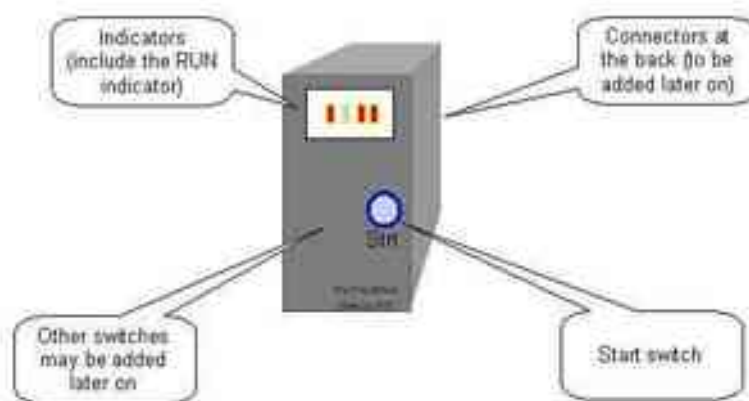
Describing the Processor State using RTL

The processor state defines the contents of all the register internal to the CPU at a given time. Maintaining or restoring the machine or processor state is important to many operations, especially procedure calls and interrupts; the processor state needs to be restored after a procedure call or an interrupt so normal operation can continue. Our processor state consists of the following:

PC<15..0>:	program counter (the PC holds the memory address of the next instruction)
IR<15..0>:	instruction register (used to hold the current instruction)
Run:	one bit run/halt indicator
Strt:	start signal
R	8 general purpose registers, each consisting of 16 bits
[0..7]<15..0>:	

FALCON-A in a black box

The given figure shows what a processor appears as to a user. We see a start button that is basically used to start up the processor, and a run indicator that turns on when the processor is in the running state.



There may be several other indicators as well. The start button as well as the run indicator can be observed on many machines.

Using RTL to describe the dynamic properties of the FALCON-A

We have just described some of the static properties of the FALCON-A. The RTL can also be employed to describe the dynamic behavior of the processor in terms of instruction interpretation and execution.

Conditional expressions can be specified using the RTL. For instance, we may specify a conditional subtraction operation employing RTL as

$$(op=2) : R[ra] \leftarrow R[rb] - R[rc];$$

This instruction means that “if” the operation code of the instruction equals 2 (00010 in binary), then subtract the value stored in register rc from that of register rb, and store the resulting value in register ra.

Effective address calculations in RTL (performed at runtime)

The operand or the destination address may not be specified directly in an instruction, and it may be required to compute the effective address at run-time. Displacement and relative addressing modes are instances of such situations. RTL can be used to describe these effective address calculations.

Displacement address

A displacement address is calculated, as shown:

$$\text{disp} \langle 15..0 \rangle := (R[rb] + (11a \ c1 \langle 4 \rangle) \odot c1 \langle 4..0 \rangle);$$

This means that the address is being calculated by adding the constant value specified by the field c1 (which is first sign extended), to the value specified by the register rb.

Relative address

A relative address is calculated by adding the displacement to the contents of the program counter register (that holds the instruction to be executed next in a program flow). The constant is first sign-extended. In RTL this is represented as, $\text{rel} \langle 15..0 \rangle := PC + (8ac2 \langle 7 \rangle) \odot c2 \langle 7..0 \rangle;$

Range of memory addresses

Using the displacement or the relative addressing modes, there is a specific range of memory addresses that can be accessed.

- Range of addresses when using direct addressing mode (displacement with rb=0)
 - If $c1 \langle 4 \rangle = 0$ (positive displacement) absolute addresses range: 00000b to 01111b (0 to +15)
 - If $c1 \langle 4 \rangle = 1$ (negative displacement) absolute addresses range: 11111b to 10000b (-1 to -16)
- Address range in case of relative addressing
 - The largest positive value that can be specified using 8 bits (since we have only 8 bits available in $c2 \langle 7..0 \rangle$), is 27-1, and the most negative value that can be represented using the same is 27. Therefore, the range of addresses or locations that can be referred to using this addressing mode is 127 locations forward or 128 locations backward from the Program Counter (PC).

Instruction Fetch Operation (using RTL)

We will now employ the notation that we have learnt to understand the fetch-execute cycle of the FALCON-A processor.

The RTL notation for the instruction fetch process is

```
instruction_Fetch := (  
    !Run&Strt : Run ← 1,  
    Run : (IR ← M[PC], PC ← PC + 2;  
        instruction_Execution) );
```

This is how the instruction-fetch phase of the fetch-execute cycle for FALCON-A can be represented using RTL. Recall that “:=” is the naming operator, “!” implies a logical NOT, “&” implies a logical AND, “←” represents a transfer operation, “;” is used to separate sequential statements, and concurrent statements are separated by “,”. We can observe that in the instruction_Fetch phase, if the machine is not in the running state and the start bit has been set, then the run bit is also set to true. Concurrently, an instruction is fetched from the instruction memory; the program counter (PC) holds the next instruction address, so it is used to refer to the memory location from where the instruction is to be fetched. Simultaneously, the PC is incremented by 2 so it will point to the next instruction. (Recall that our instruction word is 2 bytes long, and the instruction memory is organized into 1-byte cells). The next step is the instruction execution phase. Difference between “;” and “,” in RTL

We again highlight the difference between the “,” and “;”. Statements separated by a “,” take place during the same clock pulse. In other words, the order of execution of statements separated by “,” does not matter.

On the other hand, statements separated by a “;” take place on successive clock pulses. In other words, if statements are separated by “;” the one on the left must complete before the one on the right starts. However, some things written with one RTL statement can take several clocks to complete.

We return to our discussion of the instruction-fetch phase. The statement

!Run&Strt : Run ← 1

is executed when ‘Run’ is 0, and ‘Strt’ is 1, that is, Strt has been set. It is used to set the Run bit. No action takes place when both ‘Run’ and ‘Strt’ are 0.

The following two concurrent register transfers are performed when ‘Run’ is set to 1, (as ‘:’ is a conditional operator; if the condition is met, the specified action is taken).

**IR ← M[PC]
PC ← PC + 2**

Since these instructions appear concurrent, and one of the instructions is using the value of PC that the other instruction is updating, a question arises; which of the two values of the PC is used in the memory access? As a rule, all right hand sides of the register transfers are evaluated before the left hand side is evaluated/updated. In case of simultaneous register transfers (separated by a “,”), all the right hand side expressions are evaluated in the same clock-cycle, before they are assigned. Therefore, the old, un-incremented value of the PC is used in the memory access, and the incremented value is assigned to the PC afterwards. This corresponds to “master-slave” flip-flop operation in logic circuits.

This makes the PC point to the next instruction in the instruction memory. Once the instruction has been fetched, the instruction execution starts. We can also use i.F for

instruction_Fetch and i.E for instruction_Execution. This will make the Fetch operation easy to write.

iF := (!Run&Strt : Run ← 1, Run : (IR ← M[PC], PC ← PC + 2; iE));

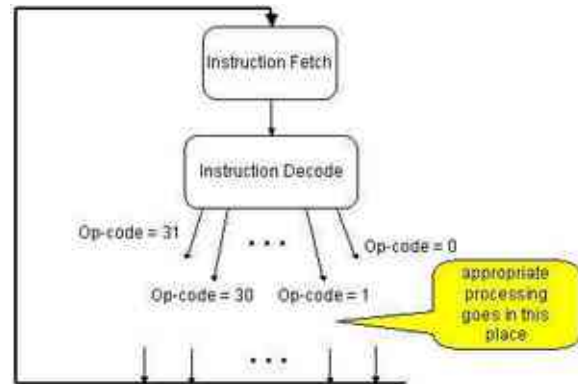
Instruction Execution (Describing the Execute operation using RTL)

Once an instruction has been fetched from the instruction memory, and the program counter has been incremented to point to the next instruction in the memory, instruction execution commences. In the instruction fetch-execute cycle we showed in the preceding discussion, the

entire instruction execution code was aliased iE (or instruction_Execution), through the assignment operator “:=”. Now we look at the instruction execution in detail.

```
iE := (
    (op<4..0>= 1) : R[ra] ← R[rb]+ (11α c1<4>)© c1<4..0>,
    (op<4..0>= 2) : R[ra] ← R[rb]-R[rc],
    ...
    ...
    (op<4..0>=31) : Run ← 0,); iF );
```

As we can see, the instruction execution can be described in RTL by using a long list of concurrent, conditional operators that are inherently ‘disjoint’. Being inherently disjointed implies that at any instance, only one of the conditions can be met; hence one of the statements is executed. The long list of statements is basically all of the instructions that are a part of the FALCON-A instruction set, and the condition for their execution is related to the operation code of the instruction fetched. We will take a closer look at the entire list in our subsequent discussion. Notice that in the instruction execute phase, besides the long list of concurrent, disjoint instructions, there is also the instruction fetch or iF sequenced at the end. This implies that once **one of the instructions from the list is executed, the instruction fetch is called to fetch the next instruction**. As shown before, the instruction fetch will call the instruction execute after fetching a certain instruction, hence the instruction fetch-execute cycle continues.



The instruction fetch-execute cycle is shown schematically in the above given figure. We now see how the various instructions in the execute code of the fetch-execute cycle of FALCON-A, are represented using the RTL. These instructions form the instruction set of the FALCON-A.

Jump instructions

Some of the instructions listed for the instruction execution phase are jump instruction, as shown. (Note ‘. . .’ implies that more instructions may precede or follow, depending on whether it is placed before the instructions shown, or after).

```
iE := (
    ...
    ...
```

If op-code is 20, the branch is taken unconditionally (the jump instruction).

```
(op<4..0>=20) : (cond : (PC ← R[ra]+C2(sign extended)),
```

If the op-code is 16, the condition for branching is checked, and if the condition is being met, the branch is taken; otherwise it remains untaken, and normal program flow will continue.

```
(op<4..0>= 16) : (cond : (PC ← PC+C2 (sign extended ))
```

```
...
...)
```

Arithmetic and Logical Instructions

Several instructions provide arithmetic and logical operations functionality. Amongst the list of concurrent instructions of the iE phase, the instructions belonging to this category are highlighted:

```
iE := (
    ...
```

...

If op-code is 0, the instruction is 'add'. The values in register rb and rc are added and the result is stored in register ra

(op<4..0>=0) : $R[ra] \leftarrow R[rb] + R[rc]$,

Similarly, if op-code is 1, the instruction is addi; the immediate constant specified by the constant field C1 is sign extended and added to the value in register rb. The result is stored in the register ra.

(op<4..0>=1) : $R[ra] \leftarrow R[rb] + (11\alpha C1<4>) \odot C1<4..0>$,

For op-code 2, value stored in register rc is subtracted from the value stored in register rb, and the result is stored in register ra.

(op<4..0>=2) : $R[ra] \leftarrow R[rb] - R[rc]$,

If op-code is 3, the immediate constant C1 is sign-extended, and subtracted from the value stored in rb. Result is stored in ra.

(op<4..0>=3) : $R[ra] \leftarrow R[rb] - (11\alpha C1<4>) \odot C1<4..0>$,

For op-code 4, values of rb and rc register are multiplied and result is stored in the destination register.

(op<4..0>=4) : $R[ra] \leftarrow R[rb] * R[rc]$,

If the op-code is 5, contents of register rb are divided by the value stored in rc, result is concatenated with 0s, and stored in ra. The remainder is stored in R0.

**(op<4..0>=5) : $R[ra] \leftarrow R[0] \odot R[rb]/R[rc]$,
 $R[0] \leftarrow R[0] \odot R[rb] \% R[rc]$,**

If op-code equals 8, bit-wise logical AND of rb and rc register contents is assigned to ra.

(op<4..0>=8) : $R[ra] \leftarrow R[rb] \& R[rc]$,

If op-code equals 8, bit-wise logical OR of rb and rc register contents is assigned to ra.

(op<4..0>=10) : $R[ra] \leftarrow R[rb] \sim R[c]$,

For op-code 14, the contents of register specified by field rc are inverted (logical NOT is taken), and the resulting value is stored in register ra.

(op<4..0>=14) : $R[ra] \leftarrow ! R[rc]$,

...

...

Shift Instructions

The shift instructions are also a part of the instruction set for FALCON-A, and these are listed in the instruction execute phase in the RTL as shown.

iE := (

...

...

If the op-code is 12, the contents of the register rb are shifted right N bits. N is the number specified in the constant field. The space that has been created due to the shift out of bits is filled with 0s through concatenation. In RTL, this is shown as:

(op<4..0>=12) : $R[ra]<15..0> \leftarrow R[rb]<(15-N)..0> \odot (N\alpha 0)$,

If op-code is 13, rb value is shifted left, and 0s are inserted in place of shifted out contents at the right side of the value. The result is stored in ra.

(op<4..0>=13) : $R[ra]<15..0> \leftarrow (N\alpha 0) \odot R[rb]<(15)..N>$,

For op-code 15, arithmetic shift right operation is carried out on the value stored in rb. The arithmetic shift right shifts a signed binary number stored in the source register to the right, while **leaving the sign-bit unchanged. Note that α means replication, and $\odot$ means concatenation.** 

(op<4..0>=15) : $R[ra]<15..0> \leftarrow N\alpha(R[rb]<15>) \odot (R[rb]<15..N>)$,

...

...

Data transfer instructions

Several of the instructions belong to the data transfer category.

iE := (
 ...
 ...
)

Op-code 29 specifies the load instruction, i.e. a memory location is referenced and the value stored in the memory location is copied to the destination register. The effective address of the memory location to be referenced is calculated by sign extending the immediate field, and adding it to the value specified by register rb.

(op<4..0>=29) : R[ra] ← M[R[rb] + (11α C1<4>)@ C1<4..0>],

A value is stored back to memory from a register using the op-code 28. The effective address in memory where the value is to be stored is calculated in a similar fashion as the load instruction.

(op<4..0>=28) : M[R[rb] + (11α C1<4>)@ C1<4..0>] ← R [ra],

The move instruction has the op-code 6. The contents of one register are copied to another register through this instruction.

(op<4..0>=6) : R[ra] ← R[rb],

To store an immediate value (specified by the field C2 of the instruction) in a register, the op-code 7 is employed. The constant is first sign-extended.

(op<4..0>=7) : R[ra] ← (8α C2<7>)@ C2<7..0>,

If the op-code is 24, an input is obtained from a certain input device, and the input word is stored into register ra. The input device is selected by specifying its address through the constant C2.

(op<4..0>=24) : R[ra] ← IO[C2],

Unconditional branch (jump) If the op-code is 25, an output (the register ra value) is sent to an output device (where the address of the output device is specified by the constant C2).

(op<4..0>=25) : IO[C2] ← R[ra],

...
 ...

Miscellaneous instructions

Some more instruction included in the FALCON-A are

iE := (
 ...
 ...
)

The no-operation (nop) instruction, if the op-code is 21. This instructs the processor to do nothing.

(op<4..0>= 21) : ,

If the op-code is 31, setting the run bit to 0 halts the processor.

(op<4..0>= 31) : Run ← 0, Halt the processor (halt)

At the end of this concurrent list of instructions, there is an instruction i.F (the instruction fetch). Hence when an instruction is executed, the next instruction is fetched, and the cycle continues, unless the processor is halted.

); iF);

Note: For Assembler and Simulator Consult Appendix. 

The EAGLE

(Original version)

Another processor that we are going to study is the EAGLE. We have developed two versions of it, an original version, and a modified version that takes care of the limitations in the original

version. The study of multiple processors is going to help us get thoroughly familiar with the processor design, and the various possible designs for the processor. However, note that these machines are simplified versions of what a real machine might look like.

Introduction

The EAGLE is an accumulator-based machine. It is a simple processor that will help us in our understanding of the processor design process. EAGLE is characterized by the following:

- Eight General Purpose Registers of the CPU. These are named R0, R1...R7. Each register is 16-bits in length.
- Two 16-bit system registers transparent to the programmer are the Program Counter (PC) and the Instruction Register (IR). (Being transparent to the programmer implies the programmer may not directly manipulate the values to these registers. Their usage is the same as in any other processor)
- Memory word size is 16 bits
- The available memory space size is 2^{16} bytes
- Memory organization is $2^{16} \times 8$ bits. This means that there are 2^{16} memory cells, each one byte long.
- Memory is accessed in 16 bit words (i.e., 2 byte chunks)
- Little-endian byte storage is employed.

Programmer's View of the EAGLE

The programmer's view of the EAGLE processor is shown by means of the given figure.

EAGLE: Notation

Let us take a look at the notation that will be employed for the study of the EAGLE.

Enclosing the register name in square

brackets refers to register contents; for instance, R[3] means contents of register R3.

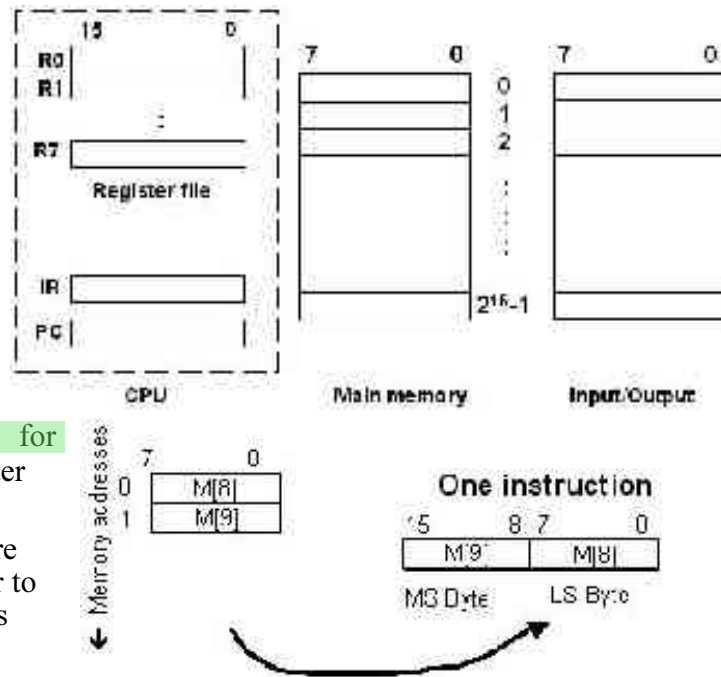
Enclosing the location address in square brackets, preceded by 'M', lets us refer to **memory contents**. Hence M[8] means contents of memory location 8.

As little endian storage is employed, a

memory word at address x is defined as the 16 bits at address x + 1 and x. For instance, the bits at memory location 9,8 define the memory word at location 8. So employing the special notation for 16-bit memory words, we have

$M[8] \langle 15 \dots 0 \rangle := M[9] \odot M[8]$

Where $\odot$ is used to represent concatenation



EAGLE Features

The following features characterize the EAGLE.

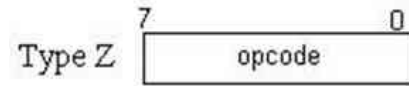
- Instruction length is variable. Instructions are either 8 bits or 16 long, i.e., instruction size is either 8-bits or 16-bits.
- The instructions may have either one or two operands.
- The only way to access memory is through load and store instructions.
- Limited addressing modes are supported

EAGLE: Instruction Formats

There are five instruction formats for the EAGLE. These are

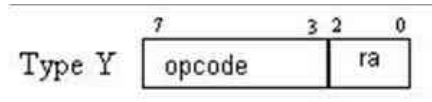
Type Z Instruction Format

The Z format instructions are half-word (1 byte) instructions, containing just the op-code field of 8 bits, as shown



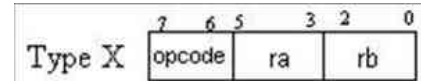
Type Y Instruction Format

The type Y instructions are also half-word. There is an op-code field of 5 bits, and a register operand field ra.



Type X Instruction Format

Type X instructions are also half-word instructions, with a 2-bit op-code field, and two 3-bit operand register fields, as shown. **Type W instruction format**

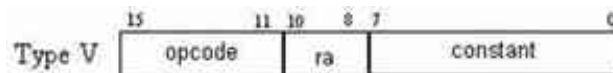


The instructions in this type are 1-word (16-bit) in length. 8 bits are reserved for the op-code, while the remaining 8 bits form the constant (immediate value) field.



Type V instruction format

Type V instructions are also 1-word



instructions, containing an op-code field of 5 bits, an operand register field of 3 bits, and 8bits for a specifying a constant.

Encoding of the General Purpose Registers

The encoding for the eight GPRs is shown in the table. These binary codes are to be used in place of the 'placeholders' ra, rb in the actual instructions of the processor **EAGLE**.

Register	Code	Register	Code
R1	111	R4	100
R5	101	R7	111
R2	110	R6	100
R3	111	R7	111

Listing of EAGLE instructions with respect to instruction formats

The following is a brief introduction to the various instructions of the processor EAGLE, categorized with respect to the instruction formats.

Type Z

There are four type Z instructions,

- **halt(op-code=250)**
This instruction halts the processor
- **nop(op-code=249)**
nop, or the no-operation instruction stalls the processor for the time of execution of a single instruction. It is useful in pipelining.
- **init(op-code=251)**
This instruction is used to initialize all the registers, by setting them to 0
- **reset(op-code=248)**

This instruction is used to initialize the processor to a known state. In this instruction the control step counter is set to zero so that the operation begins at the start of the instruction fetch and besides this PC is also set to a known value so that machine operation begins at a known instruction.

Type Y

Seven instructions of the processor are of type Y. These are

- **add(op-code=11)**

The type Y add instruction adds register ra's contents to register R0. For example, add r1
In the behavioral RTL, we show this as
 $R[0] \leftarrow R[1] + R[0]$

- **and(op-code=19)**

This instruction obtains the logical AND of the value stored in register specified by field ra and the register R0, and assigns the result to R0, as shown in the example:
and r5
which is represented in RTL as $R[0] \leftarrow R[1] \& R[0]$

- **div(op-code=16)**

This instruction divides the contents of register R0 by the value stored in the register ra, and assigns result to R0. The remainder is stored in the divisor register, as shown in example,
div r6
In RTL, this is
 $R[0] \leftarrow R[0] / R[6]$
 $R[6] \leftarrow R[0] \% R[6]$

- **mul (op-code = 15)**

This instruction multiplies the values stored in register R0 and the operand register, and assigns the result to R0. For example,
mul r4
In RTL, we specify this as $R[0] \leftarrow R[0] * R[4]$

- **not (op-code = 23)**

The not instruction inverts the operand register's value and assigns it back to the same register, as shown in the example
not r6
 $R[6] \leftarrow ! R[6]$

- **or (op-code=21)**

The or instruction obtains the bit-wise OR of the operand register's and R0's value, and assigns it back to R0. An example,
or r5
 $R[0] \leftarrow R[0] \sim R[5]$

- **sub (op-code=12)**

The sub instruction subtracts the value of the operand register from R0 value, assigning it back to register R0. Example:
sub r7
In RTL: $R[0] \leftarrow R[0] - R[7]$

Type X

Only one instruction falls under this type. It is the ‘mov’ instruction that is useful for register transfers

- **mov (op-code = 0)**

The contents of one register are copied to the destination register ra.

Example: mov r5, r1

RTL Notation: $R[5] \leftarrow R[1]$

Type W

Again, only one instruction belongs to this type. It is the branch instruction

- **br (op-code = 252)**

This is the unconditional branch instruction, and the branch target is specified by the 8-bit immediate field. The branch is taken by incrementing the PC with the new value. Hence it is a ‘near’ jump. For instance,

br 14

$PC \leftarrow PC+14$

Most of the instructions of the processor EAGLE are of the format type V. These are

- **addi (op-code = 13)**

The addi instruction adds the immediate value to the register ra, by first sign-extending the immediate value. The result is also stored in the register ra. For example,

addi r4, 31

In behavioral RTL, this is

$R[4] \leftarrow R[4] + (8ac<7>)@c<7...0>;$

- **andi (op-code = 20)**

Logical ‘AND’ of the immediate value and register ra value is obtained when this instruction is executed, and the result is assigned back to register ra. An example, andi r6, 1

$R[6] \leftarrow R[6] \&1$

- **in (op-code=29)**

This instruction is to read in a word from an IO device at the address specified by the immediate field, and store it in the register ra. For instance,

in r1, 45

In RTL this is

$R[1] \leftarrow IO[45]$

- **load (op-code=8)**

The load instruction is to load the memory word into the register ra. The immediate field specifies the location of the memory word to be read. For instance,

load r3, 6

$R[3] \leftarrow M[6]$

- **brn (op-code = 28)**

Upon the brn instruction execution, the value stored in register ra is checked, and if it is negative, branch is taken by incrementing the PC by the immediate field value. An example is

brn r4, 3

In RTL, this may be written as if $R[4]<0$, $PC \leftarrow PC+3$

- **brnz (op-code = 25)**

For a brnz instruction, the value of register ra is checked, and if found non-zero, the PC-relative branch is taken, as shown in the example,

brnz r6, 12 Which, in RTL is

if $R[6] \neq 0$, $PC \leftarrow PC+12$

- **brp (op-code=27)**

brp is the 'branch if positive'. Again, ra value is checked and if found positive, the PC-relative near jump is taken, as shown in the example:

brp r1, 45

In RTL this is

if $R[1] > 0$, $PC \leftarrow PC + 45$

- **brz (op-code=8)**

In this instruction, the value of register ra is checked, and if it equals zero, PC-relative branch is taken, as shown,

brz r5, 8

In RTL:

if $R[5] = 0$, $PC \leftarrow PC + 8$

- **loadi (op-code=9)**

The loadi instruction loads the immediate constant into the register ra, for instance,

loadi r5, 54 $R[5] \leftarrow 54$

- **ori (op-code=22)**

The ori instruction obtains the logical 'OR' of the immediate value with the ra register value, and assigns it back to the register ra, as shown,

ori r7, 11 In RTL,

$R[7] \leftarrow R[7] \sim 11$

- **out (op-code=30)**

The out instruction is used to write a register word to an IO device, the address of which is specified by the immediate constant. For instance,

out 32, r5

In RTL, this is represented by $IO[32] \leftarrow R[5]$

- **shlrl (op-code=17)**

This instruction shifts left the contents of the register ra, as many times as is specified through the immediate constant of the instruction. For example: shlrl r1, 6

- **shlrr (op-code=18)**

This instruction shifts right the contents of the register ra, as many times as is specified through the immediate constant of the instruction. For example: shlrr r2, 5

- **store (op-code=10)**

The store instruction stores the value of the ra register to a memory location specified by the immediate constant. An example is,

store r4, 34

RTL description of this instruction is $M[34] \leftarrow R[4]$

- **subi (op-code=14)**

The subi instruction subtracts the immediate constant from the value of register ra, assigning back the result to the register ra. For instance,

subi r3, 13

RTL description of the instruction

$R[3] \leftarrow R[3] - 13$

(ORIGINAL) ISA for the EAGLE

(16-bit registers, 16-bit PC and 32-bit memory)

Instruction	opcode	operand and register and constant			Format	Behavioral RTL
		1 bits	8 bits	8 bits		
add	00011	ra	-	-	N	$R[ra] \leftarrow R[ra] + R[ra]$
addi	01101	ra	-	a	N	$R[ra] \leftarrow R[ra] + \text{imm} \ll 7 \ll ra$
and	00011	ra	-	-	N	$R[ra] \leftarrow R[ra] \& R[ra]$
andi	00100	ra	-	a	N	$R[ra] \leftarrow R[ra] \& (\text{imm} \ll 7 \ll ra)$
ori	01111100	-	-	a	N	$PC \leftarrow PC + \text{imm} \ll 7 \ll ra$
orori	011100	ra	-	a	N	$(R[ra] \ll 0), PC \leftarrow PC + \text{imm} \ll 7 \ll ra$
setr	01101	ra	-	a	N	$(R[ra] \ll 0), PC \leftarrow PC + \text{imm} \ll 7 \ll ra$
setd	00011	ra	-	a	N	$(R[ra] \ll 0), PC \leftarrow PC + \text{imm} \ll 7 \ll ra$
seti	00010	ra	-	a	N	$(R[ra] \ll 0), PC \leftarrow PC + \text{imm} \ll 7 \ll ra$
div	00000	ra	-	-	N	$R[ra] \leftarrow R[ra] / R[ra], R[ra] \leftarrow R[ra] \% R[ra]$
shift	01111000	-	-	-	Z	$XL7 \leftarrow 0$
shl	01101	ra	-	-	N	$R[ra] \leftarrow R[ra] \ll 7$
shr	01111011	-	-	-	Z	$R[ra] \leftarrow R[ra] \gg 7$
load	00000	ra	-	-	N	$R[ra] \leftarrow R[ra]$
loadh	00001	ra	-	-	N	$R[ra] \leftarrow R[ra] \gg 7$
mov	00	ra	ra	-	X	$R[ra] \leftarrow R[ra]$
mul	00111	ra	-	-	N	$R[ra] \leftarrow R[ra] \times R[ra]$
neg	01111001	-	-	-	Z	$-$
not	00111	ra	-	-	N	$R[ra] \leftarrow \neg R[ra]$
or	00101	ra	-	-	N	$R[ra] \leftarrow R[ra] R[ra]$
rd	00110	ra	-	a	N	$R[ra] \leftarrow R[ra] \gg (\text{imm} \ll 7 \ll ra)$
rdi	01110	ra	-	a	N	$R[ra] \leftarrow R[ra]$
regid	01111000	-	-	-	Z	$TL7$
stath	00001	ra	-	a	N	$R[ra] \leftarrow R[ra] \ll (\text{imm} \ll 7 \ll ra)$
stath	00010	ra	-	a	N	$R[ra] \leftarrow (\text{imm} \ll 7 \ll ra) \ll ra$
store	00110	ra	-	a	N	$R[ra] \leftarrow R[ra]$
sub	01100	ra	-	-	N	$R[ra] \leftarrow R[ra] - R[ra]$
subi	01110	ra	-	a	N	$R[ra] \leftarrow R[ra] - (\text{imm} \ll 7 \ll ra)$

Symbol	Meaning	Symbol	Meaning
a	Replication	%	Remainder after integer division
C	Concatenation	&	Logical AND
-	Constant constants (0-255)		Logical OR
~	Sequential constants	!	Logical NOT or complement
.	Constant constants	←	LOAD or assignment operator

Limitations of the ORIGINAL EAGLE ISA

The original 16-bit ISA of EAGLE has severe limitations, as outlined below.

1. Use of R0 as accumulator

In most cases, the register R0 is being used as one of the source operands as well as the destination operand. Thus, R0 has essentially become the accumulator. However, this will require some additional instructions for use with the accumulator. That should not be a problem since there are some unused op-codes available in the ISA. Unequal and inefficient op-code assignment

The designer has apparently tried to extend the number of operations in the ISA by op-code extension. Op-code 11111 combine three additional bits of the instruction for five instructions: unconditional branch, nop, halt, reset and init. while there is a possibility of including three more instructions in this scheme, notice that op-code 00 for register to register mov is causing a “loss” of eight “slots” in the original 5-bit op-code assignment. (The mov instruction is, in effect, using eight op-codes). A better way would be to assign

a 5-bit op-code to mov and use the remaining op-codes for other instructions. Number of the operands

Looking at the mov instruction again, it can be noted that this is the only instruction that uses two operands, and thus requires a separate format (Format#1) for instruction encoding. If the job of this instruction is given to two instructions (copy register to accumulator, and copy accumulator to register), the number of instruction formats can be reduced thereby, simplifying the assembler and the compiler needed for this ISA.

2. Use of registers for branch conditions

Note that one of the GPRs is being used to hold the branch condition. This would require that the result from the accumulator be copied to the particular GPR before the branch instruction. Including flags with the ALSU can eliminate this restriction

The Modified EAGLE

The modified EAGLE is an improved version of the processor EAGLE. As we have already discussed, there were several limitations in EAGLE, and these have been remedied in the modified EAGLE processor.

Introduction

The modified EAGLE is also an accumulator-based processor. It is a simple, yet complex enough to illustrate the various concepts of a processor design. The modified EAGLE is characterized by

- A special purpose register, the 16-bit accumulator: ACC
- 8 General Purpose Registers of the CPU: R0, R1, ..., R7; 16-bits each
- Two 16-bit system registers transparent to the programmer are the Program Counter (PC) and the Instruction Register (IR).
- Memory word size: 16 bits
- Memory space size: 2^{16} bytes
- Memory organization: $2^{16} \times 8$ bits
- Memory is accessed in 16 bit words (i.e., 2 byte chunks)
- Little-endian byte storage is employed

Programmer's View of the Modified EAGLE

The given figure is the programmer's view of the modified EAGLE processor.

Notation

The notation that is employed for the study of the modified EAGLE is the same as the original EAGLE processor. Recall that we know that:

Enclosing the register name in square brackets refers to register contents; for instance, R [3] means contents of register R3.

Enclosing the location address in square brackets, preceded by 'M', lets us refer to **memory contents**. Hence M [8] means contents of memory location 8.

As little endian storage is employed, a **memory word** at address x is defined as the 16 bits at address x+1 and x. For instance, the bits at memory location 9,8 define the memory word at location 8. So employing the special notation for 16-bit memory words, we have

$$M[8] \langle 15 \dots 0 \rangle := M[9] \odot M[8]$$

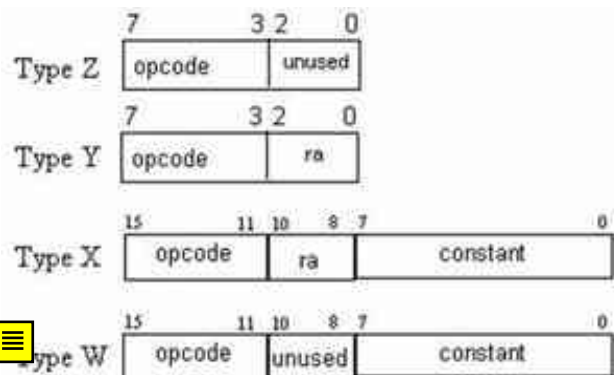
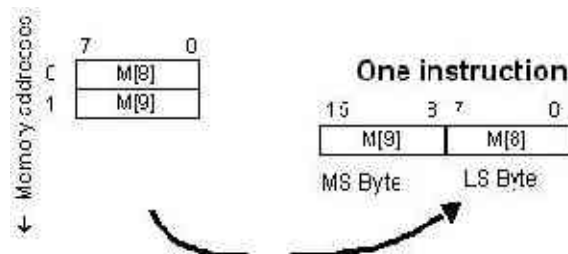
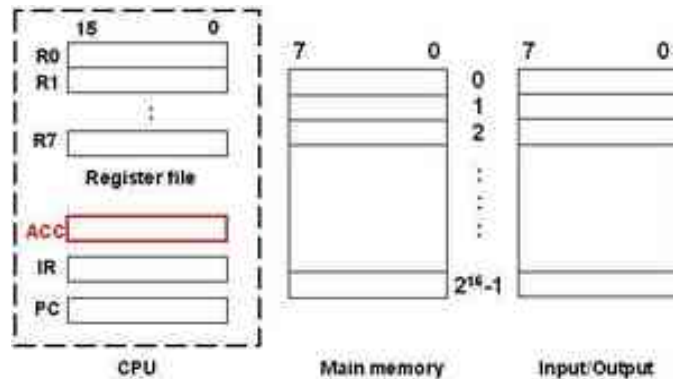
Where $\odot$ is used to represent concatenation. The memory word access and copy to a register is shown in the figure.

Features

The following features characterize the modified EAGLE processor.

- Instruction length is variable. Instructions are either 8 bits or 16 long, i.e., instruction size is either half a word or 1 word.
- The instructions may have either one or two operands.
- The only way to access memory is through load and store instructions
- Limited addressing modes are supported

Note that these properties are the same as the original EAGLE processor



Instruction formats

There are four instruction format types in the modified EAGLE processor as well. These are

Register	Code	Register	Code
R0	000	R4	100
R1	001	R5	101
R2	010	R6	110
R3	011	R7	111

Encoding of the General Purpose Registers

The encoding for the eight GPRs is shown in the table. These are binary codes assigned to the registers that will be used in place of the ra, rb in the actual instructions of the modified processor **EAGLE**.

ISA for the Modified EAGLE

(16-bit registers, 16-bit ACC, PC and IR, 8-bit wide memory, 256 I/O ports)

Mnemonic	Op-code	Operand Regs	Constant n bits	Format	Behavioral RTL
Clear	00111				
addi	00100	ra	C1	N	$ACC \leftarrow R[ra] + (R[C1] \ll 7) \oplus CC1$
subi	00101	ra	C1	N	$ACC \leftarrow R[ra] - (R[C1] \ll 7) \oplus CC1$
shli	01010	ra	C1	N	$R[ra] \leftarrow R[ra] \ll (15-d), d = CC1 \oplus 001$
shri	01011	ra	C1	N	$R[ra] \leftarrow (R[ra] \gg R[ra]) \gg (15-d), d = CC1$
andi	01100	ra	C1	N	$ACC \leftarrow R[ra] \& (R[C1] \ll 7) \oplus CC1$
ori	01101	ra	C1	N	$ACC \leftarrow R[ra] \vee (R[C1] \ll 7) \oplus CC1$
lwr	01110	ra	C1	N	$R[ra] \leftarrow (R[ra] \ll (15-d) \oplus R[ra]) \gg (15-d), d = CC1$
sw	01000	ra	C1	N	$R[ra] \leftarrow R[C1]$
store	00010	ra	C1	N	$ACC \leftarrow M[R[ra] + (R[C1] \ll 7) \oplus CC1]$
load	00100	ra	C1	N	$R[ra] \leftarrow (R[C1] \ll 7) \oplus CC1$
lwr	00101	ra	C1	N	$IR[CC1] \leftarrow R[ra]$
store	00111	ra	C1	N	$M[R[ra] + (R[C1] \ll 7) \oplus CC1] \leftarrow ACC$
store	00011		C1	W	$ACC \leftarrow (R[C1] \ll 7) \oplus CC1$
ir	01000		C1	W	$PC \leftarrow PC + (R[C1] \ll 7) \oplus CC1$
lwr	01001		C1	W	$(d=1): PC \leftarrow PC + (R[C1] \ll 7) \oplus CC1$
lwr	01010		C1	W	$(d=0): PC \leftarrow PC + (R[C1] \ll 7) \oplus CC1$
lwr	01011		C1	W	$(d=0): PC \leftarrow PC + (R[C1] \ll 7) \oplus CC1$
lwr	01100		C1	W	$(d=1): PC \leftarrow PC + (R[C1] \ll 7) \oplus CC1$
add	00000	ra	-	N	$ACC \leftarrow ACC + R[ra]$
sub	00001	ra	-	N	$ACC \leftarrow ACC - R[ra]$
lwr	00010	ra	-	N	$ACC \leftarrow (R[ra] \oplus ACC) \oplus R[ra]$
lwr	00011	ra	-	N	$R[ra] \leftarrow (R[ra] \oplus ACC) \oplus R[ra]$
and	00001	ra	-	N	$R[ra] \oplus ACC \leftarrow R[ra] \wedge ACC$
and	01000	ra	-	N	$ACC \leftarrow ACC \& R[ra]$
or	01001	ra	-	N	$ACC \leftarrow ACC \vee R[ra]$
lwr	01111	ra	-	N	$ACC \leftarrow S[R[ra]]$
lwr	10000	ra	-	N	$R[ra] \leftarrow ACC$
lwr	00110	ra	-	N	$ACC \leftarrow R[ra]$
lwr	00110		-	2	$ACC \leftarrow R$
lwr	01101		-	2	$IR[54] \leftarrow R$
lwr	01110		-	2	
lwr	01111		-	2	IR[55]

Symbol	Meaning	Symbol	Meaning
α	Replication	%	Remainder after integer division
©	Concatenation	&	Logical AND
:	Conditional constructs (IF-THEN)	~	Logical OR
;	Sequential constructs	!	Logical NOT or complement
,	Concurrent constructs	$\leftarrow$	LOAD or assignment operator

Lecture No. 10

The FALCON-E and ISA Comparison

Reading Material

Handouts

Slides

Summary

- The FALCON-E
- Instruction Set Architecture Comparison

THE FALCON-E

Introduction

FALCON stands for First Architecture for Learning Computer Organization and Networks. We are already familiar with our example processor, the FALCON-A, which was the first version of the FALCON processor. In this section we will develop a new version of the processor. Like its predecessor, the FALCON-E is a General-Purpose Register machine that is simple, yet is able to elucidate the fundamentals of computer design and architecture.

The FALCON-E is characterized by the following

- **Eight General Purpose Registers (GPRs)**, named R0, R1...R7. Each registers is 4 bytes long (32-bit registers).
- **Two special purposes registers**, named BP and SP. These registers are also 32-bit in length.
- **Two special registers**, the Program Counter (PC) and the Instruction Register (IR). PC points to the next instruction to be executed, and the IR holds the current instruction.
- **Memory word size is 32 bits (4 bytes).**
- **Memory space is 2^{32} bytes**
- Memory is organized as 1-byte cells, and hence it is $2^{32} \times 8$ bits.
- Memory is accessed in 32-bit words (4-byte chunks, or 4 consecutive cells)
- Byte storage format is little endian.

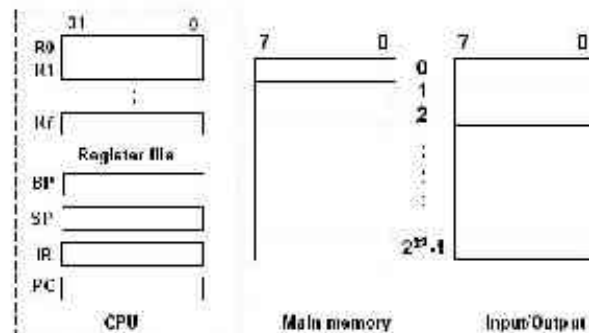


Fig. Programmer's View

Programmer's view of the FALCON-E

The programmer's view of the FALCON-E is shown in the given figure.

FALCON-E Notation

We take a brief look at the notation that we will employ for the FALCON-E.

Register contents are referred to in a similar fashion as the FALCON-A, i.e. the register name in square brackets. So R[3] means contents of register R3.

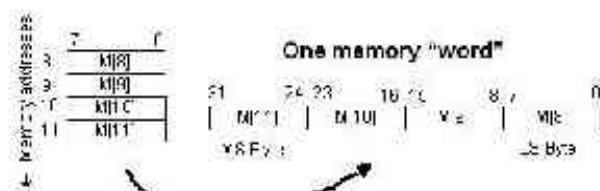


Fig. FALCON-E Notation

Memory contents (or the memory location) can be referred to in a similar way. Therefore, $M[8]$ means contents of memory location 8.

A **memory word** is stored in the memory in the little endian format. This means that the least significant byte is stored first (or the little end comes first!). For instance, a memory word at address 8 is defined as the 32 bits at addresses 11, 10, 9, and 8 (little-endian). So we can employ a special notation to refer to the memory words. Again, we will employ $\odot$ as the concatenation operator. In our notation for the FALCON-E, the memory word stored at address 8 is represented as:

$M[8]<31...0>:=M[11]\odot M[10]\odot M[9]\odot M[8]$

The shown figure will make this easier to understand.

FALCON-E Features

The following features characterize the FALCON-E

- Fixed instruction size, which is 32 bits. So the instruction size is 1 word.
- All ALU instructions have three operands
- Memory access is possible only through the load and store instructions. Also, only a limited addressing modes are supported by the FALCON-E

FALCON-E Instruction Formats

Four different instruction formats are supported by the FALCON-E. These are

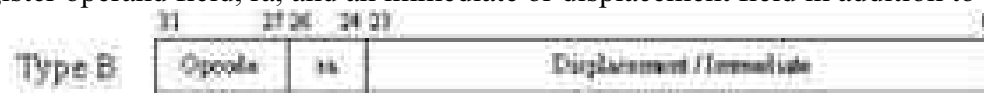
Type A instructions

The type A instructions have 5 bits reserved for the operation code (abbreviated op-code), and the rest of the bits are either not used or specify a displacement.



Type B instructions

The type B instructions also have 5 bits (27 through 31) reserved for the op-code. There is a register operand field, ra, and an immediate or displacement field in addition to the op-code field.



Type C instructions

Type C instructions have the 5-bit op-code field, two 3-bit operand registers (rb is the source register, ra is the destination register), a 17-bit immediate or displacement field, as well as a 3-bit function field. The function field is used to differentiate between instructions that may have the same op-code, but different operations.



Type D instructions

Type D instructions have the 5-bit op-code field, three 3-bit operand registers, 14 bits are unused, and a 3-bit function field.



Encoding for the General Purpose Registers (GPRs)

In the instruction formats discussed above, we used register operands ra, rb and rc. It is important to know that these are merely placeholders, and not the real register names. In an actual instruction, any one of the 8 registers of our general-purpose register file may be used. We need

to encode our registers so we can refer to them in an instruction. Note that we have reserved 3 bits for each of the register field. This is because we have 8 registers to represent, and they can be completely represented by 3 bits, since $2^3 = 8$. The following table shows the binary encoding of the general-purpose registers.

Register	Code	Register	Code
R0	000	R4	100
R1	001	R5	101
R2	010	R6	110
R3	011	R7	111

Fig. Encoding of the GPRs

There are two more special registers that we need to represent; the SP and the BP. We will use these registers in place of the operand register rb in the **load** and **store** instructions only, and therefore, we may encode these as

Register	Code
SP	000
BP	001

Fig. Special Registers Encoding

Instructions, Instruction Formats

The following is a brief introduction to the various instructions of the FALCON-E, categorized with respect to the instruction formats.

Type A instructions

Four instructions of the FALCON-E belong to type A. These are

- **nop** (op-code = 0)

This instruction instructs the processor to do nothing. It is generally useful in pipelining. We will study more on pipelining later in the course.

- **ret** (op-code = 15)

The return instruction is used to return control to the normal flow of a program after an interrupt or a procedure call concludes

- **iret** (op-code = 17)

The **iret** instruction instructs the processor to return control to the address specified by the immediate field of the instruction. Setting the program counter to the specified address returns control.

- **near jmp** (op-code = 18)

A near jump is a PC-relative jump. The PC value is incremented (or decremented) by the immediate field value to take the jump.

Type B instructions

Five instructions belong to the type B format of instructions. These are:

- **push** (op-code = 8)
This instruction is used to push the contents of a register onto the stack. For instance, the instruction,
push R4
will push the contents of register R4 on top of the stack
- **pop** (op-code = 9)
The pop instruction is used to pop a value from the top of the stack, and the value is read into a register. For example, the instruction
pop R7
will pop the upper-most element of the stack and store the value in register R7
- **ld** (op-code = 10)
This instruction with op-code (10) loads a memory word from the address specified by the immediate field value. This word is brought into the operand register ra. For example, the instruction,
ld R7, 1254h
will load the contents of the memory at the address 1254h into the register R7.
- **st** (op-code = 12)
The store instruction of (opcode 12) stores a value contained in the register operand into the memory location specified by the immediate operand field. For example, in
st R7, 1254h
the contents of register R7 are saved to the memory location 1254h.

Type C instructions

There are four data transfer instructions, as well as nine ALU instructions that belong to type C instruction format of the FALCON-E. The data transfer instructions are

- **lds** (op-code = 4)
The load instruction with op-code (4) loads a register from the memory, after calculating the address of the memory location that is to be accessed. The effective address of the memory location to be read is calculated by adding the immediate value to the value stored by the register rb. For instance, in the example below, the immediate value 56 is added to the value stored by the register R4, and the resultant value is the address of the memory location which is read
lds R3, R4(56)
In RTL, this can be shown as
 $R[3] \leftarrow M[R[4] + 56]$
- **sts** (op-code = 5)
This instruction is used to store the register contents to the memory location, by first calculating the effective memory address. The address calculation is similar to the lds instruction. An example:
sts R3, R4 (56)
In RTL, this is shown as
 $M[R[4] + 56] \leftarrow R[3]$
- **in** (op-code = 6)

This instruction is to load a register from an input/output device. The effective address of the I/O device has to be calculated before it is accessed to read the word into the destination register ra, as shown in the example:

in R5, R4(100)

In RTL:

$R[5] \leftarrow IO[R[4]+100]$

- **out** (op-code = 7)

This instruction is used to write / store the register contents into an input/output device. Again, the effective address calculation has to be carried out to evaluate the destination I/O address before the write can take place. For example,

out R8, R6 (36)

RTL representation of this is $IO[R[6]+36] \leftarrow R[8]$

Three of the ALU instructions that belong to type C format are

- **addi** (op-code = 2)

The addi instruction is to add a constant to the value of operand register rb, and assign the result to the destination register ra. For example, in the following instruction, 56 is added to the value of register R4, and result is assigned to the register R3.

addi R3, R4, 56

In RTL this can be shown as $R[3] \leftarrow R[4]+56$

Note that if the immediate constant specified was a negative number, then this would become a subtract operation. 

- **andi** (op-code = 2)

This instruction is to calculate the logical AND of the immediate value and the rb register value. The result is assigned to destination register ra. For instance

andi R3, R4, 56

$R[3] \leftarrow R[4] \& 56$

Note that the logical AND is represented by the symbol '&'

- **ori** (op-code = 2)

This instruction calculates the logical OR of the immediate field and the value in operand register rb. The result is assigned to the destination register ra. Following is an example:

ori R3, R4, 56

The RTL representation of this instruction:

$R[3] \leftarrow R[4] \sim 56$

Note that the symbol '~' is used to represent logical OR.

Type D Instructions

Four of the instructions that belong to this instruction format type are the ALU instructions shown below. There are other instructions of this type as well, listed in the tables at the end of this section.

- **add** (op-code = 1)

This instruction is used to add two numbers. The numbers are stored in the registers specified by rb and rc. Result is stored into register ra. For instance, the instruction, add

R3, R5, R6

adds the numbers in register R5, R6, storing the result in R3. In RTL, this is given by $R[3] \leftarrow R[5] + R[6]$

- **sub** (op-code = 1)

This instruction is used to carry out 2's complement subtraction. Again, register addressing mode is used, as shown in the example instruction

sub R3, R5, R6

RTL representation of this is $R[3] \leftarrow R[5] - R[6]$

- **and** (op-code = 1)

For carrying out logical AND operation on the values stored in registers, this instruction is employed. For instance

and R8, R3, R4

In RTL, we can write this as $R[8] \leftarrow R[3] \& R[4]$

- **or** (op-code = 1)

For evaluating logical OR of values stored in two registers, we use this instruction. An example is

or R8, R3, R4

In RTL, this is

$R[8] \leftarrow R[3] \sim R[4]$

Falcon-E Instruction Summary

The following are the tables that list the instructions that form the instruction set of the FALCON-E. These instructions have been grouped with respect to the functionality they provide.

Control Instructions	Mnemonic	Opcode		Function	
		Dec	Bin	Dec	Bin
No Operation	nop	0	00000	-	

Fig. Control Instructions

Arithmetic Instructions	Mnemonic	Opcode		Function	
		Dec	Bin	Dec	Bin
Add	add	1	00001	0	0000
Add Immediate	addi	2	00010	0	0000
Subtract	sub	1	00001	1	0001
Subtract Immediate	subi	2	00010	1	0001
Multiply	mul	1	00001	2	0010
Multiply Immediate	mulh	2	00010	2	0010
Divide	div	1	00001	3	0011
Divide Immediate	divi	2	00010	3	0011

Fig. Arithmetic Instructions

Data Transfer Instructions	Mnemonic	Opcode		Function	
		Dec	Bin	Dec	Bin
Move Immediate to GPR	mov	3	00011	-	-
Load Special Purpose Register from GPR	ldr	4	00100	-	-
Store Special Purpose Register to GPR	str	5	00101	-	-
Load Register from IO	ldi	6	00110	-	-
Store Register to IO	sti	7	00111	-	-
Push GPR to Stack	push	8	01000	-	-
Pop GPR from Stack	pop	9	01001	-	-
Load GPR from Memory (Direct Addressing)	ld	10	01010	-	-
Load GPR from Memory (Displacement Addressing)	ldr	11	01011	-	-
Store GPR to Memory (Direct Addressing)	st	12	01100	-	-
Store GPR to Memory (Displacement Addressing)	str	13	01101	-	-

Fig. Data Transfer Instructions

Procedure Calls/Interrupts	Mnemonic	Opcode		Function	
		Dec	Bin	Dec	Bin
Call	call	14	01110	-	-
Return	ret	15	01111	-	-
Interrupt	int	16	10000	-	-
Interrupt Return	intret	17	10001	-	-

Fig. Procedure Calls/Interrupts

Branch Instructions	Mnemonic	Opcode		Function	
		Dec	Bin	Dec	Bin
Near Jump (Relative)	jmp	19	10010	+	
Far Jump (Direct)	jmp	19	10011	-	
Branch If Equal (Relative)	bne	20	10100	0	0000
Branch If Equal (Direct)	bne	21	10101	0	0000
Branch If Not Equal (Relative)	bne	20	10100	1	0001
Branch If Not Equal (Direct)	bne	21	10101	1	0001
Branch If Less (Relative)	bl	20	10100	2	0010
Branch If Less (Direct)	bl	21	10101	2	0010
Branch If Greater (Relative)	bg	20	10100	3	0011
Branch If Greater (Direct)	bg	21	10101	3	0011

Fig. Branch Instructions

Instruction Set Architecture Comparison

In this lecture, we compare the instruction set architectures of the various processors we have described/ designed up till now. These processors are:

- EAGLE
- FALCON-A
- FALCON-E
- SRC

Classifying Instruction Set Architectures

In the design of the ISA, the choice of some of the parameters can critically affect the code density (which is the number of instructions required to complete a given task), cycles per instruction (as some instructions may take more than one clock cycle, and the number of cycles per instruction varies from instruction to instruction, architecture to architecture), and cycle time (the total cycle time to execute a given piece of code). Classification of different architectures is based on the following parameters.

Operand storage in CPU	Where are they stored other than memory?
Number of explicit operands in an instruction	One, two or three operands?
Addressing Modes	How the effective address for operands is calculated?
Operations	What operation are possible and what are the choices for the opcodes?
Type and size of operands	How the size is specified for operands?

Fig. ISA Comparison Parameters

Instruction Length

With reference to the instruction lengths in a particular ISA, there are two decisions to be made; whether the instruction will be fixed in length or variable, and what will be the instruction length or the range (in case of variable instruction lengths).

Fixed versus variable

Fixed instruction lengths are desirable when simplicity of design is a goal. It provides ease of implementation for assembling and pipelining. However, fixed instruction length can be wasteful in terms of code density. All the RISC machines use fixed instruction length format

Instruction Length

The required instruction length mainly depends on the number of instruction required to be in the instruction set of a processor (the greater the number of instructions supported, the more bits are required to encode the operation code), the size of the register file (greater the number of registers in the register file, more is the number of bits required to encode these in an instruction), the number of operands supported in instructions (as obviously, it will require more bits to encode a greater number of operands in an instruction), the size of immediate operand field (the greater the size, the more the range of values that can be specified by the immediate operand) and finally, the code density (which implies how many instructions can be encoded in a given number of bits). A summary of the instruction lengths of our processors is given in the table below.

EAGLE	FALCON-A	FALCON-E	SRC
Variable 8 bits or 16 bits	Fixed 16 bits	Fixed 32 bits	Fixed 32 bits

Fig. Instruction Length

Instruction types and sub-types

The given table summarizes the number of instruction types and sub-types of the processors we have studied. We have already studied these instruction types, and their sub-types in detail in the related sections.

	EAGLE	FALCON-A	FALCON-E	SRC
Types	4	4	4	4
Sub-types	1	2	4	3

Number of operands in the instructions

The number of operands that may be required in an instruction depends on the type of operation to be performed by that instruction; some instruction may have no operands, other may have up to 3. But a limit on the maximum number of operands for the instruction set of a processor needs to be defined explicitly, as it affects the instruction length and code density. The maximum number of operands supported by the instruction set of each processor under study is given in the given table. So FALCON-A, FALCON-E and the SRC processors may have 3, 2, 1 or no operands, depending on the instruction. EAGLE has a maximum number of 2 operands; it may have one operand or no operands in an instruction.

EAGLE	FALCON-A	FALCON-E	SRC
2	3	3	3

Fig. Number of Operands per instructions

Explicit operand specification in an instruction gives flexibility in storage. Implicit operands like an accumulator or a stack reduces the instruction size, as they need not be coded into the instruction. Instructions of the processor EAGLE have implicit operands, and we saw that the result is automatically stored in the accumulator, without the accumulator being specified as a destination operand in the instruction.

Number and Size of General Purpose Registers

While designing a processor, another decision that has to be made is about the number of registers present in the register file, and the size of the registers.

Increasing the number of registers in the register file of the CPU will decrease the memory traffic, which is a desirable attribute, as memory accesses take relatively much longer time than register access. Memory traffic decreases as the number of registers is increased, as variables are copied into the registers and these do not have to be accessed from memory over and over again. If there is a small number of registers, the values stored previously will have to be saved back to memory to bring in the new values; more registers will solve the problem of swapping in, swapping out. However, a very large register file is not feasible, as it will require more bits of the instruction to encode these registers. The size of the registers affects the range of values that can be stored in the registers.

The number of registers in the register file, along with the size of the registers, for each of the processors under study, is in the given table.

EAGLE	FALCON-A	FALCON-E	SRC
Eight registers, 16 bit wide	Eight registers, 16 bit wide	Eight registers, 32 bit wide	Thirty-two registers 32 bit wide

Fig. Number and size of GPRS

Memory specifications

Memory design is an integral part of the processor design. We need to decide on the memory space that will be available to the processor, how the memory will be organized, memory word size, memory access bus width, and the storage format used to store words in memory. The memory specifications for the processor under comparison are:

Memory Specs.	EAGLE	FALCON-A	FALCON-E	SRC
Memory Space	2^{16}	2^{16}	2^{32}	2^{32}
Memory Organization	$2^{16} * 8$	$2^{16} * 8$	$2^{32} * 8$	$2^{32} * 8$
Memory Word Size	16 bit	16 bit	32 bit	32 bit
Memory Access	16 bits	16 bits	32 bits	32 bits
Memory Storage	Little-Endian	Big Endian	Little-Endian	Big Endian

Fig. Memory Specifications

Data transfer instructions

Data needs to be transferred between storage devices for processing. Data transfers may include loading, storing back or copying of the data. The different ways in which data transfers may take place have their related advantages and disadvantages. These are listed in the given table.

Data Transfer	Advantage	Disadvantage
Register to Register	Simple, faster, constant CPI, Easier to pipeline.	Higher instruction count, longer program codes
Register to Memory	Separate load instruction eliminated, good code density	Variable CPI due to different operand locations
Memory to Memory	Most compact, small number of registers required	Variable CPI, variable instruction size, memory bottleneck.

Fig: Data Transfer Modes

Following are the data transfer instructions included in the instruction sets of our processors.

Register to register transfers

As we can see from the given table on the next page, in the processor EAGLE, register to register transfers are of two types only: register to accumulator, or accumulator to register. Accumulator is a special-purpose register.

FALCON-A has a **mov** instruction, which can be used to move data of any register to any other register. FALCON-E has the instructions 'lds' and 'sts' which are used to load/store a register from/to memory after effective address calculation.

SRC does not provide any instruction for data movement between general-purpose registers. However, this can be accomplished indirectly, by adopting either of the following two approaches:

- A register's contents can be loaded into another register via memory. First storing the content of a register to a particular memory location, and then reading the contents of the memory from that location into the register we want to copy the value to can achieve this. However, this method is very inefficient, as it requires memory accesses, which are inherently slow operations.
- A better method is to use the addi instruction with the constant set to 0.

Data Transfer Instructions

Instructions	EAGLE	FALCON-A	FALCON-E	SRC
Register to Register	a2r, r2a	mov	lds, sts	lar (only from PC)
Register to Memory	ldacc, stacc	load, store	ld, st	ld, st
Memory to Memory	-	-	-	-

Register to memory

EAGLE has instructions to load values from memory to the special purpose register, names the accumulator, as well as saving values from the accumulator to memory. Other register to memory transfers is not possible in the EAGLE processor. FALCON-A, FALOCN-E and the SRC have simple load, store instructions and all register-memory transfers are supported.

Memory to memory

In any of the processors under study, memory-to-memory transfers are not supported. However, in other processors, these may be a possibility.

Control Flow Instructions

All processors have instructions to control the flow of programs in execution. The general control flow instructions available in most processors are:

- Branches (conditional)
- Jumps (unconditional)
- Calls (procedure calls)
- Returns (procedure returns)

Conditional Branches

Whereas jumps, calls and call returns changes the control flow in a specific order, branches depend on some conditions; if the conditions are met, the branch may be taken, otherwise the program flow may continue linearly. The branch conditions may be specified by any of the following methods:

- Condition codes
- Condition register
- Comparison and branching

Condition codes

The ALU may contain some special bits (also called flags), which may have been set (or raised) under some special circumstances. For instance, a flag may be raised if there is an overflow in the addition results of two register values, or if a number is negative. An instruction can then be ordered in the program that may change the flow depending on any of these flag's values. The EAGLE processor uses these condition codes for branch condition evaluation.

Condition register

A special register is required to act as a branch register, and any other arbitrary register (that is specified in the branch instruction), is compared against that register, and the branching decision is based on the comparison result of these two registers. None of the processors under our study use this mode of conditional branching.

Compare and branch

In this mode of conditional branching, comparison is made part of the branching instruction. Therefore, it is somewhat more complex than the other two modes. All the processors we are studying use this mode of conditional branching.

Size of jumps

Jumps are deviations from the linear program flow by a specified constant. All our processors, except the SRC, support PC-relative jumps. The displacement (or the jump) relative to the PC is specified by the constant field in the instruction. If the constant field is wider (i.e. there are more

bits reserved for the constant field in the instruction), the jump can be of a larger magnitude. Shown table specifies the displacement size for various processors.

Processor	Displacement size
EAGLE	8 bits for both conditional and unconditional.
FALCON-A	8 bits for both conditional and unconditional.
FALCON-E	27 bits (unconditional jump), 21 or 32 bits (conditional jumps)
SRC	32 bits for both conditional and unconditional jumps.

Fig. Size of Jumps

Addressing Modes

All processors support a variety of addressing modes. An addressing mode is the method by which architectures specify the address of an object they will access. The object may be a constant, a register or a location in memory.

Common addressing modes are

- **Immediate**

An immediate field may be provided in instructions, and a constant value may be given in this immediate field, e.g. **123** is an immediate value.

- **Register**

A register may contain the value we refer to in an instruction, for instance, register **R4** may contain the value being referred to.

- **Direct**

By direct addressing mode, we mean the constant field may specify the location of the memory we want to refer to. For instance, **[123]** will directly refer to the memory location 123's contents.

- **Register Indirect**

A register may contain the address of memory location to which we want to refer to, for example, **M [R3]**.

- **Displacement**

In this addressing mode, the constant value specified by the immediate field is added to the register value, and the resultant is the index of memory location that is referred to, e.g. **M [R3+123]**

- **Relative**

Relative addressing mode implies PC-relative addressing, for example, **[PC+123]** will refer to the memory location that is 123 words farther than the memory index currently stored in the program counter.

- **Indexed or scaled**

The values contained in two registers are added and the resultant value is the index to the memory location we refer to, in the indexed addressing mode. For example, **M [[R1]+[R2]]**. In the scaled addressing mode, a register value may be scaled as it is added to the value of the other register to obtain the index of memory location to be referred to.

- **Auto increment/ decrement**

In the auto increment mode, the value held in a register is used as the index to memory location that holds the value of operand. After the operand's value is retrieved, the

register value is automatically increased by 1 (or by any specified constant). e.g. **M [R4]+**, or **M [R4]+d**. In the auto decrement mode, the register value is first decremented and then used as a reference to the memory location that referred to in the instruction, e.g. **-M [R4]**.

As may be obvious to the reader, some of these addressing modes are quite simple, others are relatively complex. The complex addressing modes (such as the indexed) reduce the instruction count (thus improving code density), at the cost of more complex implementation.

The given table lists the addressing modes supported by the processors we are studying. Note that the register-addressing mode is a special case of the relative addressing mode, with the constant equal to 0, and only the PC can be used as a source. Also note that, in the shown table, relative implies PC-relative.

EAGLE	FALCON-A	FALCON-E	SRC
Immediate	Immediate	Immediate	Immediate
-	-	Direct	Direct
Register	Register	Register	Register *
Register Indirect	Register Indirect	Register Indirect	Register Indirect
-	-	-	Relative**
Displacement	Displacement	Displacement	Displacement

Fig. Addressing Modes Comparison

Displacement addressing mode

We have already talked about the displacement-addressing mode. We look at this addressing mode at length now.

The displacement-addressing mode is the most common of the addressing mode used in general

Size of displacement field

Processor	Number of bits in displacement field
SRC	17 or 22 bits depending on the instruction type.
FALCON-E	21 or 24 bits depending on the instruction type.
FALCON-A	5 bits for load and store instruction
EAGLE	8 bits for ldata and stacc instructions

purpose processors. Some other modes such as the indexed based plus index, scaled and register indirect are all slightly modified forms of the displacement-addressing mode. The size of

displacement plays a key role in efficient address calculation. The following table specifies the size of the displacement field in different processors under study. The given table lists the size of the immediate field in our processors.

Processor	Number of bits in the immediate field
EAGLE	8 bits
FALCON-A	5 or 8 bits
FALCON-E	17 or 24 bits depending on the instruction
SRC	17 or 22 bits

Fig. Immediate Field Bits Comparison

Instructions common to all Instruction Set Architectures

In this section we have listed the instructions that are common to the Instruction Set Architectures of all the processors under our study.

- **Arithmetic Instructions**
add, addi & sub.
- **Logic Instructions**
and, andi, or, ori, not.
- **Shift Instructions.**
Right shift, left shift & arithmetic right shift.
- **Data movement Instructions.**
Load and store instructions.
- **Control Instructions**
Conditional and unconditional branches, nop & reset.

The following tables list the assembly language instruction codes of these common instructions for all the processors under comparison.

Common Arithmetic Instructions				
Instruction	EAGLE	FALCON-A	FALCON-E	SRC
Add	add	add	add	add
Add Immediate	addi	addi	addi	addi
Subtract	sub	sub	sub	sub
Subtract Immediate	subi	subi	subi	-
Multiply	mul	mul	mul	-
Divide	div	div	div	-

Common data movement Instructions

Instruction	EAGLE	FALCON-A	FALCON-E	SRC
Load	ldacc	load	ld	ld
Store	stacc	store	st	st
Move	mov	mov	-	-
Move immediate	movi	movi	movi	li
In	in	in	in	-
Out	out	out	out	-

Common Logical Instructions

Instruction	EAGLE	FALCON-A	FALCON-E	SRC
And	and	and	and	and
And immediate	andi	andi	andi	andi
Or	or	or	or	or
Or immediate	ori	ori	ori	ori
Not	not	not	not	not
Neg	neg	neg	-	-

Common Shift Instructions

Instruction	EAGLE	FALCON-A	FALCON-E	SRC
Shift right	shifr	shifr	-	shr
Shift right immediate	-	-	srai	shr
Circular shift	-	-	rol	shc
Shift left	shifl	shifl	-	shl
Shift right arithmetic	asr	asr	sra	shra

Common Branch Instructions

Instruction	EAGLE	FALCON-A	FALCON-E	SRC
Unconditional branch	br	jump	jmp	br
Branch if zero	brz	jz	-	brz
Branch if non-zero	brnz	jnz	-	brnz
Branch if positive	brp	jpl	-	brpl
Branch if negative	brn	jml	-	brmi

Common Call and Interrupt Instructions

Instruction	EAGLE	FALCON-A	FALCON-E	SRC
Procedure call	-	call	call	brl
Interrupt	-	int	int	?
Interrupt return	-	iret	iret	?

Common Control Instructions

Instruction	EAGLE	FALCON-A	FALCON-E	SRC
No operation	nop	nop	nop	nop
Halt	halt	halt	-	stop
Reset	reset	reset	-	-

Instructions unique to each processor

Now we take a look at the instructions that are unique to each of the processors we are studying.

EAGLE

The EAGLE processor has a minimal instruction set. Following are the instructions that are unique only to the EAGLE processor. Note that these instructions are unique only with reference to the processor set under our study; some other processors may have these instructions. 

- **movia**

This instruction is for moving the immediate value to the accumulator (the special purpose register)

- **a2r**

This instruction is for moving the contents of the accumulator to a register

- **r2a**

For moving register contents to the accumulator

- **cla**

For clearing (setting to zero) the value in the accumulator

FALCON-A

There is only one instruction unique to the FALCON-A processor;

- **ret**

This instruction is used to return control to a calling procedure. The calling procedure may save the PC value in a register ra, and when this instruction is called, the PC value is restored. In RTL, we write this as

PC R [ra];

FALCON-E

The instructions unique to the FALCON-E processor are listed:

- **push**

To push the contents of a specified general purpose register to the stack

- **pop**

To pop the value that is at the top of the stack

- **ldr**

To load a register with memory contents using displacement addressing mode

- **str**

To store a register value into memory, using displacement addressing mode

- **bl**

To branch if source operand is less than target address

- **bg**

To branch if source operand is greater than target address

- **mul**

To multiply an immediate value with a value stored in a register

- **div**

To divide a register value by the immediate value

- **xor, xori**

To evaluate logical ‘exclusive or’

- **ror, rori**

SRC

Following are the instructions that are unique to the SRC processor, among of the processors under study

- **ldr**

To load register from memory using PC-relative address

- **lar**

To load a register with a word from memory using relative address

- **str**

To store register value to memory using relative address

- **brlnv**

This instruction is to tell the processor to ‘never branch’ at that point in program. The instruction saves the program counter’s contents to the register specified

- **brlpl**

This instruction instructs the processor to branch to the location specified by a register given in the instruction, if the condition register’s value is positive. Return address is saved before branching.

- **brlmi**

This instruction instructs the processor to branch to the location specified by a register given in the instruction, if the condition register’s value is negative. Return address is saved before branching.

- **brl zr**

This instruction instructs the processor to branch to the location specified by a register given in the instruction, if the condition register’s value equals zero. Return address is saved before branching.

- **brlnz**

This instruction instructs the processor to branch to the location specified by a register given in the instruction, if the condition register’s value does not equal zero. Return address is saved before branching.

Problem Comparison

Given is the code for a simple C statement:

$a = (b - 2) + 4c$

The given table gives its implementation in all the four processors under comparison. Note that this table highlights the code density for each of the processors; EAGLE, which has relatively fewer specialized instructions, and so it takes more instructions to carry out this operation as compared with the rest of the processors

EAGLE	FALCON-A	FALCON-E	SRC
.org 100 a .dw 1	.org 100 a .dw 1	.org 100 a .dw 1	.org 100 a .dw 1
.org 200 ltdcc b	.org 200	.org 200	.org 200
ldr r1	load r1, b	ld r1, b	ld r1, b
subi r1, 2	subi r2, r1, 2	subi r2, r1, 2	addi r2, r1, -2
ldr r1	load r3, c	ld r3, c	ld r3, c
ltdcc c	shri r3, r3, 2	mulh r3, r3, 4	shl r3, r3, 2
ldr r2	add r4, r2, r3	add r4, r3, r2	add r4, r2, r3
shl r2, 2	store r4, a	store r4, a	st r4, a
r2a r2			
add r1			
stacc a			

Fig. Problem Comparison

Lecture No. 11

CISC and RISC

Reading Material

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

Chapter 3
3.3, 3.4

Summary

- A CISC microprocessor: The Motorola MC68000
- A RISC Architecture: The SPARC

Material of this Lecture is included in the above-mentioned sections of Chapter 3.

Lecture No. 12

CPU Design

Reading Material

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

Chapter 4
4.1, 4.2, 4.3

Summary

- The design process
- A Uni-Bus implementation for the SRC
- Structural RTL for the SRC instructions

Central Processing Unit Design

This module will explore the design of the central processing unit from the logic designer's view. A unibus implementation of the SRC is discussed in detail along with the Data Path Design and the Control Unit Design. The topics covered in this module are outlined below:

- The Design Process
- Unibus Implementation of the SRC
- Structural RTL for the SRC
- Logic Design for one bus SRC
- The Control Unit
 - 2-bus and 3-bus designs
 - The machine reset
 - The machine exceptions

As we progress through this list of topics, we will learn how to convert the earlier specified behavioral RTL into a concrete structural RTL. We will also learn how to interconnect various programmer visible registers to get a complete data path and how to incorporate various control signals into it. Finally, we will add the machine reset and exception capability to our processor.

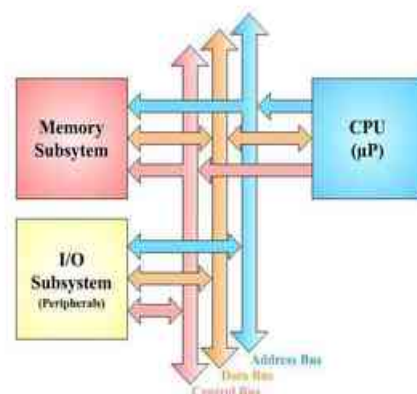
The design process

The design process of a processor starts with the specification of the behavioral RTL for its instruction set. This abstract description is then converted into structural RTL which shows the actual implementation details. Since the processor can be divided into two main sub-systems, the data path and the control unit, we can split the design procedure into two phases.

1. The data path design
2. The control unit design

It is important that the design activity of these important components of the processor be carried out with the pros and cons of adopting different approaches in mind.

As we know, the execution time is dependent on the following three factors. $ET = IC \times CPI \times T$



Block Diagram of Computer System

During the design procedure we specify the implementation details at an advanced level. These details can affect the clock cycle per instruction and the clock cycle time. Hence following things should be kept in mind during the design phase.

- Effect on overall performance
- Amount of control hardware
- Development time

Processor Design

Let us take a look at the steps involved in the processor design procedure.

1. ISA Design

The first step in designing a processor is the specification of the instruction set of the processor. ISA design includes decisions involving number and size of instructions, formats, addressing modes, memory organization and the programmer's view of the CPU i.e. the number and size of general and special purpose registers.

2. Behavioral RTL Description

In this step, the behavior of processor in response to the specific instructions is described in register transfer language. This abstract description is not bound to any specific implementation of the processor. It presents only those static (registers) and dynamic aspects (operations) of the machine that are necessary to understand its functionality. The unit of activity here is the instruction execution unlike the clock cycle in actual case. The functionality of all the instructions is described here in special register transfer notation.

3. Implementation of the Data Path

The data path design involves decisions like the placement and interconnection of various registers, the type of flip-flops to be used and the number and kind of the interconnection buses. All these decisions affect the number and speed of register transfers during an operation. The structure of the ALU and the design of the memory-to-CPU interface also need to be decided at this stage. Then there are the control signals that form the interface between the data path and the control unit. These control signals move data onto buses, enable and disable flip-flops, specify the ALU functions and control the buses and memory operations. Hence an integral part of the data path design is the seamless embedding of the control signals into it.

4. Structural RTL Description

In accordance with the chosen data path implementation, the structural RTL for every instruction is described in this step. The structural RTL is formed according to the proposed micro-architecture which includes many hidden temporary registers necessary for instruction execution. Since the structural RTL shows the actual implementation steps, it should satisfy the time and space requirements of the CPU as specified by the clocking interval and the number of registers and buses in the data path.

5. Control Unit Design

The control unit design is a rather tricky process as it involves timing and synchronization issues besides the usual combinational logic used in the data path design. Additionally, there are two different approaches to the control unit design; it can be either hard-wired or micro-programmed. However, the task can be made simpler by dividing the design procedure into smaller steps as follows.

- a) Analyze the structural RTL and prepare a list of control signals to be activated during the execution of each RTL statement.
- b) Develop logic circuits necessary to generate the control signals
- c) Tie everything together to complete the design of the control unit.

Processor Design

A Uni-bus Data Path Implementation for the SRC

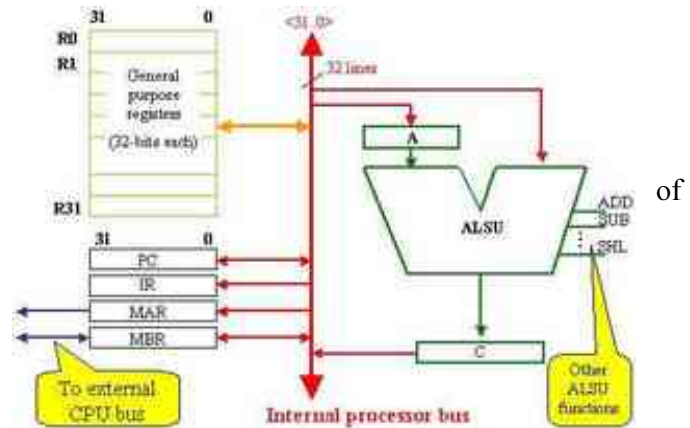
In this section, we will discuss the uni-bus implementation of the data path for the SRC. But before we go onto the design phase, we will discuss what a data path is. After the discussion of the data path design, we will discuss the timing step generation, which makes possible the synchronization of the data path functions.

The Data Path

The data path is the arithmetic portion of the Von Neumann architecture. It consists of registers, internal buses, arithmetic units and shifters. We have already discussed the decisions involved in designing the data path. Now we shall have an overview of the 1-Bus SRC data path design. As the name suggests, this implementation employs a single bus for data flow. After that we develop each of its blocks in greater detail and present the gate level implementation.

Overview of the Unibus SRC Data Path

The 1-bus implementation of the SRC data path is shown in the figure given. The control signals are omitted here for the sake simplicity. Following units are present in the SRC data path.



1. The Register File

The general-purpose register file includes 32 registers R0 to R31 each 32 bit wide. These registers communicate with other components via the internal processor bus.

2. MAR

The Memory Address Register takes input from the ALSU as the address of the memory location to be accessed and transfers the memory contents on that location onto the memory sub-system.

3. MBR

The Memory Buffer Register has a bi-directional connection with both the memory sub-system and the registers and ALSU. It holds the data during its transmission to and from memory.

4. PC

The Program Counter holds the address of the next instruction to be executed. Its value is incremented after loading of each instruction. The value in PC can also be changed based on a branch decision in ALSU. Therefore, it has a bi-directional connection with the internal processor bus.

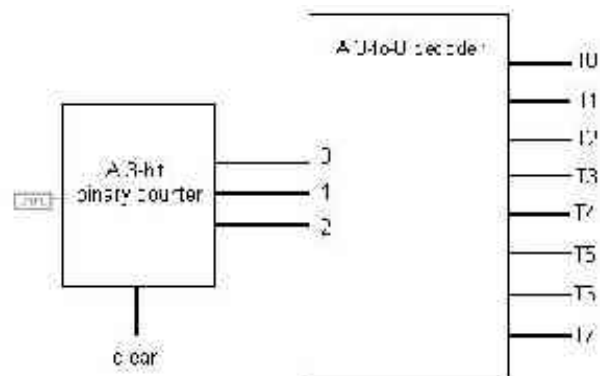
5. IR

The Instruction Register holds the instruction that is being executed. The instruction fields are extracted from the IR and transferred to the appropriate registers according to the external circuitry (not shown in this diagram).

6. Registers A and C

The registers A and C are required to hold an operand or result value while the bus is busy transmitting some other value. Both these registers are programmer invisible.

7. ALSU



There is a 32-bit Arithmetic Logic Shift Unit, as shown in the diagram. It takes input from memory or registers via the bus, computes the result according to the control signals applied to it, and places it in the register C, from where it is finally transferred to its destination.

Timing Step Generator To ensure the correct and controlled execution of instructions in a program, and all the related operations, a timing device is required. This is to ensure that the operations of essentially different instructions do not mix up in time. There exists a ‘timing step generator’ that provides mutually exclusive and sequential timing intervals. This is analogous to the clock cycles in the actual processor. A possible implementation of the timing step generator is shown in the figure.

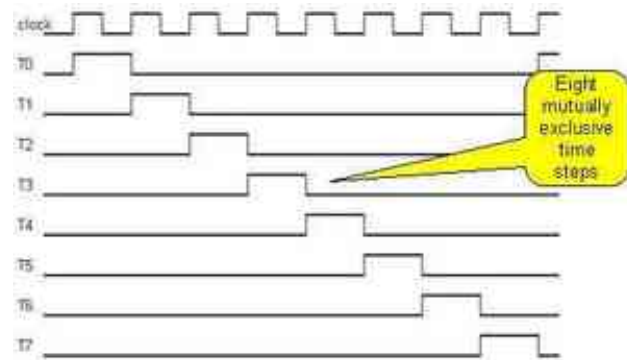
Each mutually exclusive step is carried out in one timing interval. The timing intervals can be named T0, T1...T7. The given figure is helpful in understanding the ‘mutual exclusiveness in time’ of these timing intervals.

Processor design

Structural RTL descriptions of selected SRC instructions

Structural RTL for the SRC

The structural RTL describes how a particular operation is performed using a specific hardware implementation. In order to present the structural RTL we assume that there exists a “timing step generator”, which provides mutually exclusive and sequential timing intervals, analogous to the clock cycles in actual processor.



Structural RTL for Instruction Fetch

The instruction fetch procedure takes three time steps as shown in the table. During the first time step, T0, address of the instruction is moved to the Memory Address Register (MAR) and value of PC is incremented. In T1 the instruction is brought from the memory into the Memory Buffer Register (MBR), and the incremented PC is updated. In the third and final time-step of the instruction fetch phase, the instruction from the memory buffer register is written into the IR for execution. What follows the instruction fetch phase, is the instruction execution phase. The number of timing steps taken by the execution phase generally depends on the type and function of instruction. The more complex the instruction and its implementation, the more timing steps it will require to complete execution. In the following discussion, we will take a look at various types of instructions, related timing steps requirements and data path implementations of these in terms of the structural RTL.

Step	RTL
T0	MAR ← PC; PC ← PC + 1;
T1	MBR ← M[MAR]; PC ← PC;
T2	IR ← MBR;

Structural RTL for Arithmetic/Logic Instructions

The arithmetic/logic instructions come in two formats, one with the immediate operand and the other with register operand. Examples of both are shown in the following tables.

Register-to-Register sub

Register-to-register subtract (or sub) will take three timing steps to complete execution, as shown in the table. Here we have assumed that the instruction given is:

step	RTL
T0-T2	Instruction fetch
T3	A ← R[ra];
T4	C ← A - R[rc];
T5	R[ra] ← C;

sub ra, rb, rc

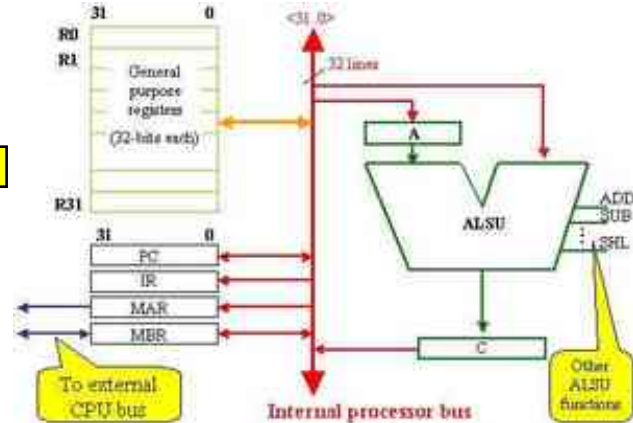
Here we assume that the instruction fetch process has taken up the first three timing steps. In step T3 the internal register A receives the contents of the register rb. In the next timing step, the value of register rc is subtracted (since the op-code is sub) from A. In the final step, this result is transferred into the destination register ra. This concludes the instruction fetch-execute cycle and at the end of it, the timing step generator is initialized to T0. The given figure refreshes our knowledge of the data path. Notice that we can visualize how the steps that we have just outlined can be carried out, if appropriate control signals are applied at the appropriate timing.

As will be obvious, control signals need to be applied to the ALSU, based on the decoding of the op-code field of an instruction. The given table lists these control signals:

Note that we have used uppercase alphabets

for naming the ALSU functions. This is to differentiate these control signals from the actual operation-code mnemonics we have been using for the instructions.

The SHL, SHR, SHC and the SHRA functions are listed assuming that a barrel shifter is available to the processor with signals to differentiate between the various types of shifts that are to be performed.



Structural RTL for Register-to-Register add

To enhance our understanding of the instruction execution phase implementation, we will now take a look at some more instructions of the SRC. The structural RTL for a simple add instruction **add ra, rb, rc** is given in table. The first three instruction fetch steps are common to all instructions. Execution of instruction starts from step T3 where the first operand is moved to register A. The second step involves computation of the sum and result is transferred to the destination in step T5. Hence the complete execution of the add instruction takes 6 time steps. Other arithmetic/logic instructions having the similar structural RTL are “**sub**”, “**and**” and “**or**”. The only difference is in the T4 step where the sign changes to (-), (^), or (~) according to the opcode.

ALSU Function	Needed for the following instructions/operations
ADD	add, addi, address calculation for disp and rel
SUB	sub
NEG	neg; applies to the D input of the ALSU
AND	and, andi
OR	or, ori
NOT	not; applies to the E input of the ALSU
SHL	shl
SHR	shr
SHC	shc
SHRA	shra
LD	to load from the bus directly into C
INC	to increment the PC by 4; applies to the D input

Step	RTL
I0-I2	Instruction fetch
T3	$A \leftarrow R[rb];$
T4	$C \leftarrow A + R[rc];$
T5	$R[ra] \leftarrow C;$

Structural RTL for the not instruction

The first three steps T0 to T2 are used up in fetching the instruction as usual. In step T3, the value of the operand specified by the register is brought into the ALSU, which will use the control function NOT, negate the value (i.e. invert it), and the result moves to the register C. In the time step

Step	RTL
I0-I2	Instruction fetch
T3	$C \leftarrow \neg(R[rc]);$
T4	$R[ra] \leftarrow C;$

R4, this result is assigned to the destination register through the internal bus. Note that we need control signals to coordinate all of this; a control signal to allow reading of the instruction-specified source register in T3, control signal for the selection of appropriate function to be

carried out at the ALSU, and control signal to allow only the instruction-specified destination register to read the result value from the data bus.

The table shown outlines these steps for the instruction: **not ra, rb**

Structural RTL for the addi instruction

Again, the first three time steps are for the instruction fetch. Next, the first operand is brought into ALSU in step T3 through register A. The step T4 is of interest here as the second operand c2 is extracted from the instruction in IR register, sign extended to 32 bits, added to the first operand and written into the result register C. The execution of instruction completes in step T5 when the result is written into the destination register. The sign extension is assumed to be carried out in the ALSU as no separate extension unit is provided.

Sign extension for 17-bit c2 is the same as: $(15aIR<16> \odot IR<16..0>)$

Sign extension for 22-bit c1 is the same as: $(10aIR<21> \odot IR<21..0>)$

The given table outlines the time steps for the instruction addi:

Other instructions that have the same structural RTL are **subi**, **andi** and **ori**.

RTL for the load (ld) and store (st) instructions

The syntax of load instructions is:

ld ra, c2(rb)

And the syntax of store instructions is:

st ra, c2(rb)

Step	RTL	
T0-T2	Instruction fetch	
T3	$A \leftarrow R[rb]$	
T4	$C \leftarrow A + c2(\text{sign extend})$	

Step	RTL for ld	RTL for st
T0-T2	Instruction fetch	Instruction fetch
T3	$A \leftarrow ((rb = 0) : 0, (rb \neq 0) : R[rb])$	$A \leftarrow ((rb = 0) : 0, (rb \neq 0) : R[rb])$
T4	$C \leftarrow A + (15aIR<16> \odot IR<16..0>)$	$C \leftarrow A + (15aIR<16> \odot IR<16..0>)$
T5	$MAR \leftarrow C$	$MAR \leftarrow C$
T6	$MBR \leftarrow M[MAR]$	$MBR \leftarrow R[ra]$
T7	$R[ra] \leftarrow MBR$	$M[MAR] \leftarrow MBR$

The given table outlines the time steps in fetching

and executing a load and a store instruction. Note that the first 6 time steps (T0 to T5) for both the instructions are the same.

The first three steps are those of instruction fetch. Next, the register A gets the value of register rb, in case it is not zero. In time step T4, the constant is sign-extended, and added to the value of register A using the ALSU. The result is assigned to register C. Note that in the RTL outlined above; we are sign extending a field of the Instruction Register (32-bit). It is so because this field is the constant field in the instruction, and the Instruction Register holds the instruction in execution. In step T5, the value in C is transferred to the Memory Address Register (MAR). This completes the effective address calculation of the memory location to be accessed for the load/store operation. If it is a load instruction in time step T6, the corresponding memory location is accessed and result is stored in Memory Buffer Register (MBR). In step T7, the result is transferred to the destination register ra using the data bus. If the instruction is to store the value of a register, the time step T6 is used to store the value of the register to the MBR. In the next and final step, the value stored in MBR is stored in the memory location indexed by the MAR. We can look at the data-path figure and visualize how all these steps can take place by applying appropriate control signals. Note that, if more time steps are required, then a counter with more bits and a larger decoder can be used, e.g., a 4-bit counter along with a 4-to-16 decoder can produce up to 16 time steps.

Lecture No. 13

Structural RTL Description of the FALCON-A

Reading Material

Vincent P. Heuring & Harry F. Jordan Computer Systems Design and Architecture

Chapter 4
4.2.2, slides

Summary

- Structural RTL Description of the SRC (continued...)
- Structural RTL Description of the FALCON-A

This lecture is a continuation of the previous lecture.

Structural RTL for branch instructions

Let us take a look at the structural RTL for branch instructions. We know that there are several variations of the branch instructions including unconditional branch and different conditional branches. We look at the RTL for ‘branch if zero’ (brzr) and ‘branch and link if zero’ brlzr’ conditional branches.

The syntax for the branch if zero (brzr) is:

brzr rb, rc

As you may recall, this instruction instructs the processor to branch to the instruction at the address held in register rb, if the value stored in register rc is zero. Time steps for this instruction are outlined the table.

The first three steps are of the instruction fetch phase. Next, the value of register rc is checked and depending

on the result, the condition flag CON is set. In time step T4, the program counter is set to the register rb value, depending on the CON bit (the condition flag). The syntax for the branch and link if zero (brlzl) is:

brlzl ra, rb, rc

This instruction is the same as the instruction **brzr** but additionally the return address is saved (linking procedure). The time steps for this instruction are shown in the table.

Notice that the steps for this instruction are the same as the instruction brzr with an additional step after the condition bit is set; the current value of the program counter is saved to register ra.

Step	RTL
T0-T2	Instruction Fetch
T3	CON ← cond(R[rc]);
T4	CON: PC ← R[rb];

in
is

Step	RTL
T0-T2	Instruction Fetch
T3	CON ← cond(R[rc]);
T4	CON: R[ra] ← PC;
T5	CON: PC ← R[rb];

Structural RTL for shift instructions

Shift instructions are rather complicated in the sense that they require extra hardware to hold and decrement the count. For an ALSU that can perform only single bit shifts, the data must be repeatedly cycled through the ALSU and the count decremented until it reaches zero. This approach presents some timing problems, which can be overcome by employing multiple-bit shifts using a barrel shifter.

The structural RTL for **shr ra, rb, rc** or **shr ra, rb, c3** is given in the corresponding

table shown. Here n represents a 5-bit register; IR bits 0 to 4 are copied in to it. N is the decimal value of the number in this register. The actual shifting is being done in step T5. Other instructions that will have similar tables are: **shl, shc, shra** e.g., for **shra**, T5 will have $C \leftarrow (N \& 0) \oplus R[rb] \langle 31..N \rangle$;

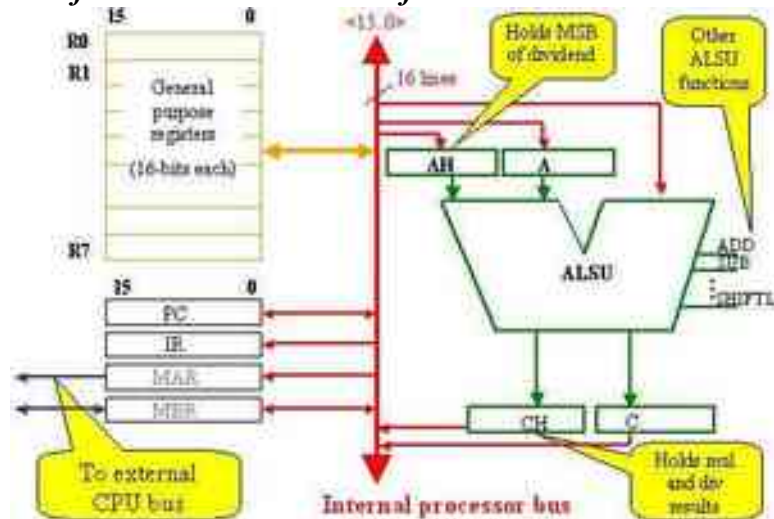
Step	RTL
T0-T2	Instruction fetch
T3	$n \langle 4..0 \rangle \leftarrow IR \langle 4..0 \rangle$;
T4	$(N = 0) : (n \langle 4..0 \rangle \leftarrow R[rc] \langle 4..0 \rangle)$;
T5	$C \leftarrow (N \& 0) \oplus R[rb] \langle 31..N \rangle$;
T6	$R[ra] \leftarrow C$;

Structural RTL Description of FALCON-A Instructions

Uni-bus data path implementation

Comparing the uni-bus implementation of FALCON-A with that of SRC results in the following differences:

- **FALCON-A processor bus has 16 lines or is 16-bits wide while that of SRC is 32-bits wide.**
- **All registers of FALCON-A are of 16-bits while in case of SRC all registers are 32-bits.**
- **Number of registers in FALCON-A are 8 while in SRC the number of registers is 32.**
- **Special registers i.e. Program Counter (PC) and Instruction Register (IR) are 16-bit registers while in SRC these are 32-bits.**
- Memory Address Register (MAR) and Memory Buffer Register (MBR) are also of 16-bits while in SRC these are of 32-bits.
- MAR and MBR are dual port registers. At one side they are connected to internal bus and at other side to external memory in order to point to a particular address for reading or writing data from or to the memory and MBR would get the data from the memory.



ALSU functions needed

ALSU of FALCON-A has slightly different functions. These functions are given in the table.

Note that mul and div are two significant instructions in this instruction set. So whenever one of these instructions is activated, the ALSU unit would take the operand from its input and provide the output immediately, if we neglect the propagation delays to its output. In case of

FALCON-A, we have two registers A and AH each of 16-bits. AH would contain the higher 16-bits or most significant 16-bits of a 32-bit operand. This

means that the ALSU provides the facility of using 32-bit operand in certain instructions. At the output of ALSU we could have a 32-bit result and that cannot be saved in just one register C so we need to have another one that is CH. CH can store the most significant 16-bits of the result.

ALSU Function	Needed for the following Instructions/operations
ADD	add, addi
SUB	sub, subi
MUL	mul
DIV	div
AND	and, andi
OR	or, ori
NOT	not, noti
SHIFTL	shll
SHIFTR	shrr
ASR	asr
C=0	to load from the bus directly into C
C=1	to implement the F2 by 2; supplies to the D input

Why do we need to add AH and CH?

This is because we have mul and div instructions in the instruction set of the FALCON-A. So for that case, we can implement the div instruction in which, at the input, one of the operand which is dividend would be 32-bits or in case of mul instruction the output which is the result of multiplication of two 16-bit numbers, would be 32-bit that could be placed in C and CH. The data in these 2 registers will be concatenated and so would be the input operand in two registers AH and A. Conceptually one could consider the A and AH together to represent 32-bit operand.

Step	RTL
T0-T2	Instruction fetch
T3	$A \leftarrow R[rb]$
T4	$C \leftarrow A - R[rc]$
T5	$R[ra] \leftarrow C;$

Structural RTL for subtract instruction sub ra, rb, rc

In sub instruction three registers are involved. The first three steps will fetch the sub instruction and in T3, T4, T5 the steps for execution of the sub instruction will be performed.

Step	RTL
T0-T2	Instruction fetch
T3	$A \leftarrow R[rb];$
T4	$C \leftarrow A - R[rc];$
T5	$R[ra] \leftarrow C;$

Structural RTL for addition instruction add ra, rb, rc

The table of add instruction is almost same as of sub instruction except in timing step T4 we have + sign for addition instead of – sign as in sub instruction. Other instructions that belong to the same group are ‘and’, ‘or’ and ‘sub’.

Structural RTL for multiplication instruction

mul ra, rb, rc

This instruction is only present in this processor and not in SRC. The first three steps are exactly same as of other instructions and would fetch the mul instruction. In step T3 we will bring the contents of register R

[rb] in the buffer register A at the input of ALSU. In step T4 we take the multiplication of A with the contents of R[rc] and put it at the output of the ALSU in two registers C and CH. CH would contain the higher 16-bits while register C would contain the lower 16-bits. Now these two registers cannot transfer the data in one bus cycle to the registers, since the width is 16-bits.

So we need to have 2 timing steps, in T5 we transfer the higher byte to register R[0] and in T6 the lower 16-bits are transferred to the placeholder R[a]. As a result of multiplication instruction we need 3 timing steps for Instruction Fetch and 4 timing steps for Instruction Execution and 7 steps altogether.

Step	RTL
T0-T2	Instruction fetch
T3	$A \leftarrow R[rb];$
T4	$C \parallel CH \leftarrow A * R[rc];$
T5	$R[0] \leftarrow CH;$
T6	$R[ra] \leftarrow C;$

Structural RTL for division instruction

div ra, rb, rc

In this instruction first three steps are the same. In step T3 the contents of register rb are placed in buffer register A and in step T4 we take the contents of register R[0] in to the register AH. We assume before using the divide instruction that we will place the higher 16-bits of dividend to register R[0]. Now in T5 the actual division takes place in two concurrent operations. We have the dividend at the input of ALSU unit represented by concatenation of AH and

Step	RTL
T0-T2	Instruction fetch
T3	$A \leftarrow R[rb];$
T4	$AH \leftarrow R[0];$
T5	$CH \leftarrow (AH \parallel A) \% R[rc]; C \leftarrow (AH \parallel A) / R[rc];$
T6	$R[ra] \leftarrow C;$
T7	$R[0] \leftarrow CH;$

A. Now as a result of division instruction, the first operation would take the remainder. This means divide AH concatenated with A with the contents given in register rc and the remainder is placed in register CH at the output of ALSU. The quotient is placed in C. In T6 we take C to the register R[ra] and in T7 remainder available in CH is taken to the default register R[0] through the bus. In divide instruction 5 timing steps are required to execute the instruction while 3 to fetch the instruction.

Note: Corresponding to mul and div instruction one should be careful about the additional register R[0] that it should be properly loaded prior to use the instructions e.g. if in the divide instruction we don't have the appropriate data available in R[0] the result of divide instruction would be wrong.

Structural RTL for not instruction

not ra, rb

In this instruction first three steps will fetch the instruction. In T3 we perform the not operation of contents in R[rb] and transfer them in to the buffer register C. It is simply the one's complement changing of 0's to 1's and 1's to 0's. In timing step T4 we take

Step	RTL
T0-T2	Instruction fetch
T3	$C \leftarrow \neg(R[rb]);$
T4	$R[ra] \leftarrow C;$

the contents of register C and transfer to register R[ra] through the bus as shown in its corresponding table.

Structural RTL for add immediate instruction addi ra, rb, c1

In this instruction c1 is a constant as a part of the instruction. First three steps are for Instruction Fetch operation. In T3

we take the contents of register R [rb] in to the buffer register A. In

T4 we add up the contents of A with the constant c1 after sign extension and bring it to C.

Sign extension of 5-bit c1 and 8-bit constant c2

Sign extension for 5-bit c1 is:

$$(11aIR<4> \odot IR<4..0>)$$

We have immediate constant c1 in the form of lower 5-bits and bit number 4 indicates the sign bit. We just copy it to the left most 11 positions to make it a 16-bit number.

Sign extension for 8-bit c2 is:

$$(8aIR<7> \odot IR<7..0>)$$

In the same way for constant c2 we need to place the sign bit to the left most 8 position to make it 16-bit number.

Structural RTL for the load and store instruction

Tables for load and store instructions are same as SRC except a slight difference in the notation. So when we have square brackets [R [rb]+c1], it corresponds to the base address in R[rb] and an offset taken from c1.

Step	RTL for ld	RTL for st
T0-T2	Instruction fetch	Instruction fetch
T3	$A \leftarrow R[rb]$	$A \leftarrow R[rb]$
T4	$C \leftarrow A + (IR[4] \odot IR[4..0])$	$C \leftarrow A + (IR[4] \odot IR[4..0])$
T5	$MEM \leftarrow C$	$MEM \leftarrow C$
T6	$R[ra] \leftarrow MEM$	$MEM \leftarrow R[ra]$
T7	$R[ra] \leftarrow MEM$	$MEM \leftarrow MEM$

Structural RTL for conditional jump instructions

jz ra, [c2]

In first three steps of this table, the instruction is fetched. In T3 we set a 1-bit register “CON” to true if the condition is met.

How do we test the condition?

This is tested by the contents given by the register ra. So condition within square brackets is R[ra]. This means test the data given in register ra. There are different possibilities and so the data could be positive, negative or zero. For this particular instruction it would be tested if the data were zero. If the data were zero, the “CON” would be 1.

In T4 we just take the contents of the PC into the buffer register A. In T5 we add up the contents of A to the constant c2 after sign extension. This addition will give us the effective address to which a jump would be taken. In T6, this value is copied to the PC.

Step	RTL
T0-T2	Instruction Fetch
T3	$CON \leftarrow \text{cond}(R[ra]);$
T4	$A \leftarrow PC;$
T5	$C \leftarrow A + c2(\text{sign extend});$
T6	$PC \leftarrow C;$

In FALCON-A, the number of conditional jumps is more than in SRC. Some of which are shown below:

- **jz (op-code= 19) jump if zero**
jz r3, [4] (R[3]=0): PC ← PC + 2;
- **jnz (op-code= 18) jump if not zero**
jnz r4, [variable] (R[4]≠0): PC ← PC + variable;
- **jpl (op-code= 16) jump if positive**
jpl r3, [label] (R[3] ≥ 0): PC ← PC + (label - PC);
- **jmi (op-code= 17) jump if negative**
jmi r7, [address] (R[7] < 0): PC ← PC + address;

The unconditional jump instruction will be explained in the next lecture.

Lecture No. 14

External FALCON-A CPU

Reading Material

Handouts

Slides

Summary

- Structural RTL Description of the FALCON-A (continued...)
- External FALCON-A CPU Interface

This lecture is a continuation of the previous lecture.

Un-conditional jump instruction jump (op-code= 20)

In the un-conditional jump with op-code 20, the op-code is followed by a 3-bit identifier for register ra and then followed by an 8-bit constant c2.

Forms allowed by the assembler to define the jump are as follows:

jump [ra + constant]
jump [ra + variable]
jump [ra + address]
jump [ra + label]

For all the above instructions:

$(ra=0):PC \leftarrow PC + (8\alpha C2<7>) \odot C2<7..0>$,
 $(ra \neq 0):PC \leftarrow R[ra] + (8\alpha C2<7>) \odot C2<7..0>$; ⁴

In the case of a constant, variable, an address or (label-PC) the jump ranges from -128 to 127 because of the restriction on 8-bit constant c2. Now, for example if we have jump [r0+a], it means jump to a. On the other hand if we have jump [- r2] that is not allowed by the assembler. The target address should be even because we have each instruction with 2 bytes. So the types available for the un-conditional jumps are either direct, indirect, PC-relative or register relative. In the case of direct jump the constant c2 would define the target address and in the case of indirect jump constant c2 would define the indirect location of memory from where we could find out the address to jump. While in the case of PC-relative if the contents of register ra are zero then we have near jump and the type of jump for this would be PC-relative. If ra is not be zero then we have a far jump and the contents of register ra will be added with the constant c2 after sign-extension to determine the jump address.

⁴ c2 is computed by sign extending the constant, variable, address or (label-PC)

Structural RTL description for un-conditional jump instruction

jump [ra+c2]

In first three steps, T0-T2, we would fetch the jump instruction, while in T3 we would either take the contents of PC and place them in a temporary register A if the condition given in jump instruction is true, that is if the ra field is zero, otherwise we would place the contents of register ra in the temporary register A. Comma ‘,’ indicates that these two instructions are concurrent and only one of them would execute at a time. If the ra field is zero then it would be PC-relative jump otherwise it would be register-relative jump. In step T4 we would add the constant c2 after sign-extension to the contents of temporary register A. As a result we would have the effective address in the buffer register C, to which we need to jump. In step T5 we will take the contents of C and load it in the PC, which would be the required address for the jump.

Step	RTL
T0-T2	Instruction Fetch
T3	$(ra=0); A \leftarrow PC, (ra \neq 0); A \leftarrow R[ra];$
T4	$C \leftarrow A + c2(\text{sign extend});$
T5	$PC \leftarrow C;$

Structural RTL for the shift instruction

shiftr ra, rb, c1

First three steps would fetch the shift instruction. c1 is the count field. It is a 5-bit constant and is obtained from the lower 5-bits of the instruction register IR. In step T3 we would load the 5-bit register ‘n’ from the count field or the lower 5-bits of the IR and then in T4 depending upon the

Step	RTL
T0-T2	Instruction fetch
T3	$n \leftarrow IR[4:0];$
T4	$C \leftarrow (N=0) \odot R[rb] \ll 15-N;$
T5	$R[ra] \leftarrow C;$

value of ‘N’ which indicates the decimal value of ‘n’, we would take the contents of register rb and shift right by N-bits which would indicate how many shifts are to be performed. ‘n’ indicates the register while ‘N’ indicates the decimal value of the bits present in the register ‘n’. So as a result we need to copy the zeros to the left most bits, this shows that zeros are replicated ‘N’ times and are concatenated with the shifted bits that are actually 15...N. In T5, we take the contents from C through the bus and feed it to the register ra which is the destination register. Other instructions that would have similar tables are ‘shifl’ and ‘asr’.

In case of asr, when the data is shifted right, instead of copying zeros on the left side, we would copy the sign bit from the original data to the left-most position.

Other instructions

Other instructions are mov, call and ret. Note that these instructions were not available with the SRC processor.



Structural RTL for the mov instruction

mov ra, rb

In mov instruction the data in register rb, which is the source register, is to be moved in the register ra, which is the destination register. In first three steps, mov instruction is fetched. In step T3 the contents of register rb are placed in buffer register C through the ALSU unit while in step T4 the buffer register C transfers the data to register ra through internal uni-bus.

Step	RTL
T0-T2	Instruction fetch
T3	$C \leftarrow R[rb];$
T4	$R[ra] \leftarrow C;$

Structural RTL for the mov immediate instruction

movi ra, c2

In this instruction ra is the destination register and constant c2 is to be moved in the ra. First three steps would fetch the move immediate instruction. In step T3 we would take the constant c2 and place it into the buffer register C. Buffer register C is 16-bit register and c2 is 8-bit constant so we need to concatenate the remaining leftmost bits with the sign bit which is bit '7' shown within angle brackets. This sign bit which is the most significant bit would be '1' if the number is negative and '0' if the number is positive. So depending upon this sign bit the remaining 8-bits are replicated with this sign bit to make a 16-bit constant to be placed in the buffer register C. In step T4 the contents of C are taken to the destination register ra.

Step	RTL
T0-T2	Instruction fetch
T3	$C \leftarrow (8@c2<7>) @ c2<7..0>;$
T4	$R[ra] \leftarrow C;$

In case of FALCON-A, 'in' and 'out' instructions are present which are not present in the SRC processor. So, for this we assume that there would be interconnection with the input and output addresses up to 0..255.

Structural RTL for the in instruction

in ra, c2

First three steps would fetch the instruction. In step T3 we take the IO [c2] which indicates that go to IO address indicated by c2 which is a positive constant in this case and then data would be taken to the buffer register C. In step T4 we would transfer the data from C to the destination register ra.

Step	RTL
T0-T2	Instruction fetch
T3	$C \leftarrow IO[c2];$
T4	$R[ra] \leftarrow C$

Structural RTL for the out instruction

out ra, c2

This instruction is opposite to the 'in' instruction. First three instructions would fetch the instruction. In step T3 the contents of register ra are placed in to the buffer register C and then in Step T4 from C the data is placed at the output port indicated by the c2 constant. So this

Step	RTL
T0-T2	Instruction fetch
T3	$C \leftarrow R[ra];$
T4	$IO[c2] \leftarrow C$

instruction is just opposite to the ‘in’ instruction.

Structural RTL for the call instruction

call ra, rb

In this instruction we need to give the control to the procedure, sub-routine or to another address specified in the program. First three steps would fetch the call instruction. In step T3 we store the present contents of PC in to the buffer register C and then from C we transfer the data to the register ra in step T4. As a result register ra would contain the original contents of PC and this would be a pointer to come back after executing the sub-routine and it would be later used by a return instruction. In step T5 we take the contents of register rb, which would actually indicate to the point where we want to go. So in step T6 the contents of C are placed in

Step	RTL
T0-T2	Instruction Fetch
T3	$C \leftarrow PC;$
T4	$R[ra] \leftarrow C;$
T5	$C \leftarrow R[rb];$
T6	$PC \leftarrow C;$

PC and as a result PC would indicate the position in the memory from where new execution has to begin.

Structural RTL for return instruction

ret ra

After instruction fetch in first 3 steps T0-T2, the register data in ra is placed in the buffer register C through ALSU unit. PC is loaded with contents of this buffer register in step T4. Assuming that bus activity is synchronized, appropriate control signals are available to us now.

Step	RTL
T0-T2	Instruction Fetch
T3	$C \leftarrow R[ra];$
T4	$PC \leftarrow C;$

Control signals required at different timing steps of FALCON-A instructions

The following table shows the details of the control signals needed. The first column is the time step, as before. In the second column the structural RTLs for the particular step is given, and the corresponding

Step	RTL	Control Signals
T0	$MAR \leftarrow PC; C \leftarrow PC + 2;$	PCout, LMAR, INC2, LC
T1	$MBR \leftarrow M[MAR]; PC \leftarrow C;$	LMBR, MRead, MARout, Cout, LPC
T2	$IR \leftarrow MBR;$	MBRout, LIR
T3	Instruction Execution	

control signals are shown in the third column. Internal bus is active in step T0, causing the contents of the PC to be placed in the Memory Address register MAR and simultaneously the PC is incremented by 2 and placed it in the buffer register C. Recalling previous lectures, to write data in to a particular register we need to enable the load signal. In case of fetch instruction in step T0, control signal LMAR is enabled to cause the data from internal bus to be written in to the address register. To provide data to the bus through tri-state buffers we need to activate the ‘out’ control signal named as ‘PCout’, making contents of the PC available to the ALSU and so control unit provides the increment signal ‘INC2’ to increment the PC. As the ALSU is the combinational circuit, the PCout signal causes the contents over the 2nd input of ALSU incremented by 2 and so the data is available in buffer register C. Control signal “LC” is required

to write data into the buffer register C from the ALSU output. Now note that 'INC2' is one of the ALSU functions and also it is a control signal. So knowing the control signals, which need to be activated at a particular step, is very important.

So, at step T0 the control signal 'PCout' is activated to provide data to the internal bus. Now control signal 'LMAR' causes the data from the bus to be read into the register MAR. The ALSU function 'INC2' increments the PC to 2 and the output are stored in the buffer register C by the control signal 'LC'. The data from memory location addressed by MAR is read into Memory Buffer Register MBR in the next timing step T1. In the mean time there is no activity on the internal bus, the output from the buffer register C (the incremented value of the PC) is placed in the PC through bus. For this the control signal 'LPC' is activated.

To enable tri-state buffer of Memory Address Register MAR, we need control signal 'MARout'. Another control signal is required in step T1 to enable memory read i.e. 'MRead'. In order to enable buffer register C to provide its data to the bus we need 'Cout' control signal and in order to enable the PC to read from C we need to enable its load signal, which is 'LPC'. To read data coming from memory into the Memory Buffer Register MBR, 'LMBR' control signal is enabled. So in T2 we need 5 control signals, as shown.

In T2, the instruction register IR is loaded with data from the MBR, so we need two-control signals, 'MBRout' to enable its tri-state buffers and the other signal required is the load signal for IR register 'LIR'. Fetch operation is completed in steps T0-T2 and appropriate control signals are generated. Those control signals, which are not shown, would remain de-activated. All control signals are activated simultaneously so the order of these controls signals is immaterial. Recall that in SRC the fetch operation is implemented in the same way, but 'INC4' is used instead of 'INC2' because the instruction length is 4 bytes.

Now we take a look at other examples for control signals required during execution phase.

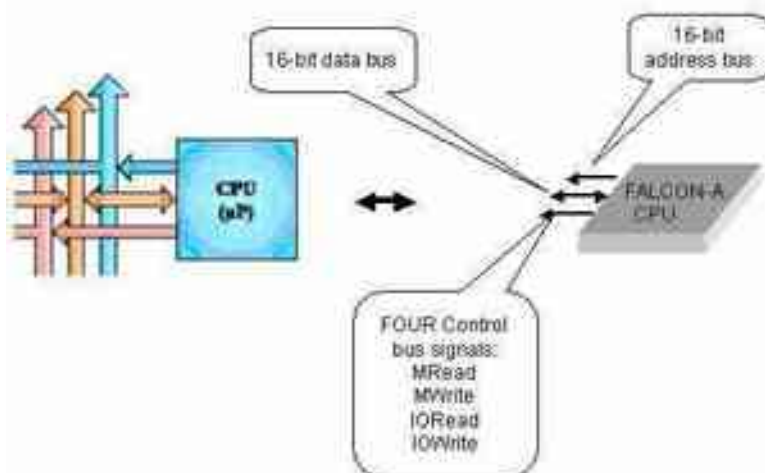
For various instructions, we will define other control signals needed in the execution phase of each instruction but fetch cycle will be the same for all instructions.

Another important fact is the interface of the CPU with an external memory and the I/O depending upon whether the I/O is memory mapped or non-memory mapped. The processor will generate some control signals, used by the memory or I/O to read/write data to/from the I/O devices or from the memory. Another assumption is that the memory read is fast enough. Therefore data from memory must be available to the processor in a fixed time interval, which in this particular example is T2.

For a slow data transfer, the concept of handshaking is used. Some idle states are introduced and buffer is prepared until the data is available. But for simplicity, we will assume that memory is fast enough and data is available in buffer register MBR to the CPU.

External FALCON-A CPU Interface

This figure is a symbolic representation of the FALCON-A in the form of a chip. The external interface consists of a 16-bit address bus, a 16-bit data bus and a control bus on which different control signals like MRead, MWrite, IORead, IOWrite are present.



Example Problem

Instruction	RTL equivalent	Address Bus <15,0>	Data Bus <15,0>	PC/PC+1	PC/PC+2
load r7, [12+r5]					
addi r2, r4, 31					
jump [52]					
store r1, [r3+17]					
subi r5, r7, 10					
shlri r2, r6, 4					
movi r3, r2					
li r4, [-32]					

Memory Address	Memory Content
0020h	D2h
0021h	96h
0022h	49h
0023h	2Fh
...	...
C300h	44h
C301h	33h
C302h	E3h
C303h	D5h

C340h	51h
C341h	CAh
C343h	D5h
C344h	E2h
...	...
1240h	07h
1241h	6Sh
1242h	E5h
1243h	3Dh

- (a) What will be the logic levels on the external FALCON-A buses when each of the given FALCON-A instruction is executing on the processor? Complete the table given. All numbers are in the decimal number system, unless noted otherwise.
- (b) Specify memory-addressing modes for each of the FALCON-A instructions given.

Assumptions

For this particular example we will assume that all memory contents are properly aligned, i.e. memory addresses start at address divisible by 2.

PC= C348h

This table contains a partial memory map showing the addresses and the corresponding data values.

The next table shows the register map showing the contents of all the CPU registers.

Another important thing to note is that memory storage is big-endian.

Register Name	Content
R[0]	A54Bh
R[1]	4CB8h
R[2]	492Fh
R[3]	C2EFh
R[4]	2301h
R[5]	1234h
R[6]	0020h
R[7]	2D7Fh

Solution:

FALCON-A Instruction	RTL equivalent	Address Bus ^a <15..0>	Data Bus <15..0>	M R	M W
load r7, [r5+12]	$R[7] \leftarrow M[12+R[5]]$	1240h	0785h	1	0
addi r2, r4, 31	$R[2] \leftarrow R[4]+31$	Unknown	????	?	?
jump [52]	$PC \leftarrow PC+52$	Unknown	????	?	?
store r1, [r3+17]	$M[R[3]+17] \leftarrow R[1]$	C300h	4CB8h	0	1
sub r5, r7, r6	$R[5] \leftarrow R[7]-R[6]$	Unknown	????	?	?
shiftr r2, r6, 4	$R[2] \leftarrow (4 \times 0) \circ R[6] \langle 15..4 \rangle$	Unknown	????	?	?
mov r3, r2	$R[3] \leftarrow R[2]$	Unknown	????	?	?
jz r4, [-32]	$R[4]=0, PC \leftarrow PC-32$	Unknown	????	?	?

In this table the second column contains the RTL descriptions of the instructions. We have to specify the address bus and data bus contents for each instruction execution. For load instruction the contents of register r5+12 are placed on the address bus. From register map shown in the previous table we can see that the contents of r5 are 1234h. Now contents of r5 are added with displacement value 12 in decimal .In other words the address bus will carry the hexadecimal value 1234h+ Ch = 1240h.Now for load instruction, the contents of memory location at address 1240h will be placed on the data bus. From the memory map shown in the previous table we can see that memory location 1240h contains 785h. Now to read this data from this location, MRead control signal will be activated shown by 1 in the next column and MWrite would be 0.Similarly RTL description is given for the 2nd instruction. In this instruction, only registers are involved so

there is no need to activate external bus. So data bus, address bus and control bus columns will contain '?' or 'unknown'. The next instruction is jump. Here PC is incremented by the jump offset, which is 52 in this case. As before, the external bus will remain inactive and control signals will be zero. The next instruction is store. Its RTL description is given. For store instruction, the register contents have to be placed at memory location addressed by R [3] +17. As this is a memory write operation, the MWrite will be 1 and MRead will be zero. Now the effective address will be determined by adding the contents of R [3] with the displacement value 17 after its conversion to the hexadecimal. The resulting effective address would be C300h. In this way we can complete the table for other instructions.

Addressing Modes

This table lists the addressing mode for each instruction given in the previous example.

FALCON-A Instruction	Addressing Mode
load r7, [r5+12]	Displacement
addi r2, r4, 31	Immediate
jump [52]	Relative
store r1, [r3+17]	Displacement
sub r5, r7, r6	Register
shiftr r2, r6, 4	Register
mov r3, r2	Register
jz r4, [-32]	Relative

Lecture No. 15

Logic Design and Control Signals Generation in SRC

Reading Material

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

Chapter 4
4.4

Summary

- Logic Design for the Uni-bus SRC
- Control Signals Generation in SRC

Logic Design for the Uni-bus SRC

In the previous sections, we have looked at both the behavioral and structural RTL for the SRC. We saw that there is a need for some control circuitry for ensuring the proper and synchronized functioning of the components of the data path, to enable it to carry out the instructions that are part of the Instruction Set Architecture of the SRC. The control unit components and related signals make up the control path. In this section, we will talk about

- Identifying the control signals required
- The external CPU interface
- Memory Address Register (MAR), and Memory Buffer Register (MBR) circuitry
- Register Connections

We will also take a look at how sign extension is performed. This study will help us understand how the entire framework works together to ensure that the operations of a simple computer like the SRC are carried out in a smooth and consistent fashion.

Identifying control signals

For any of the instructions that are a part of the instruction set of the SRC, there are certain control signals required; these control signals may be to select the appropriate function for the ALU to be performed, to select the appropriate registers, or the appropriate memory location.

Any instruction that is to be executed is first fetched into the CPU. We look at the control signals that are required for the fetch operation.

Control signals for the fetch operation

Table 1 lists the control signals that are needed to ensure the synchronized register transfers in the instruction fetch phase.

Note that we use uppercase for control signals as we have been using lowercase for the instruction mnemonics, and we want to distinguish between the two. Also note that control signals during each time slot are activated simultaneously, and

Step	RTL	Control Signals
T0	$MAR \leftarrow PC, PC \leftarrow PC + 4$	$PC_{out}, MAR_{in}, INC4, LDC$
T1	$MBR \leftarrow M[MAR], PC \leftarrow G$	$LM35, MBR_{in}, MAR_{out},$ $Cont, LDC$
T2	$IR \leftarrow MBR$	MBR_{out}, LIR

Table:1

that the control signals for successive time slots are activated in sequence. If a particular control signal is not shown, its value is zero.

As shown in the Table: 1, some control signals are to let register values to be written onto buses, or read from the buses. Similarly, some signals are required to read/ write memory contents onto the bus. The memory is assumed to be fast enough to respond during a given time slot; if that is not true, wait states have to be inserted. We require four control signals to be issued in the time step T0:

PCout: This control signal allows the contents of the Program Counter register to be written onto the internal processor bus.

LMAR: This signal enables write onto the memory address register (MAR), thus the value of PC that is on the bus, is copied into this register

INC4: It lets the PC value to be incremented by 4 in the ALSU, and result to be stored in C. Notice that the value of PC has been received by the ALSU as an operand. This control signal allows the constant 4 to be added to it. The ALSU is assumed to include an INC4 function

LC: This enables the input to the register C for writing the incremented value of PC onto it.

During the time step T1, the following control signals are applied:

LMBR: This enables the “write” for the register MBR. When this signal is activated, whatever value is on the bus, can be written into the MBR.

MRead: Allow memory word to be gated from the external CPU data bus into the MBR.

MARout: This signal enables the tri-state buffers at the output of MAR.

Cout: This will enable writing of the contents of register C onto the processor’s internal data bus.

LPC: This will enable the input to the PC for receiving a value that is currently on the internal processor bus. Thus the PC will receive an incremented value.

At the final time step, T2, of the instruction fetch phase, the following control signals are issued:

MBRout: To enable the tri-state buffers with the **MBR**.

LIR: To allow the **IR** read the value from the internal bus. Thus the instruction stored in the **MBR** is read into the Instruction Register (IR).

Uni-bus SRC implementation

The uni-bus implementation of the SRC data path is given in the Fig.1. We can now visualize

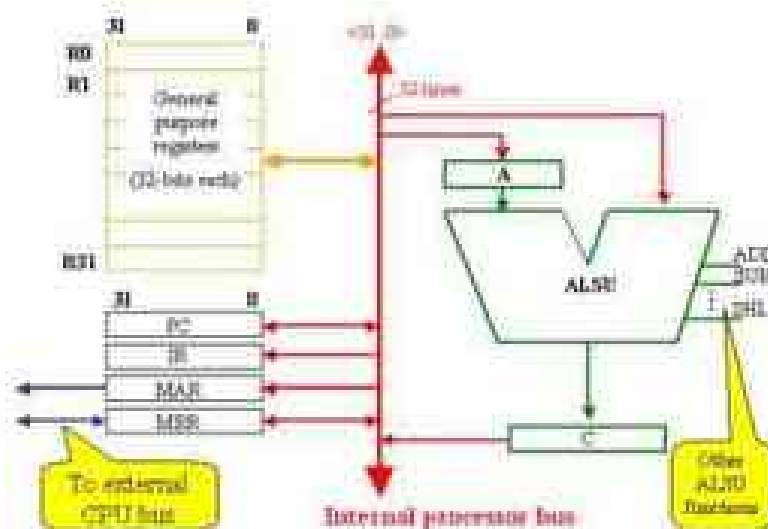


Fig.1

how the control signals in mutually exclusive time steps will allow the coordinated working of instruction fetch cycle.

Similar control signals will allow the instruction execution as well. We have already mentioned the external CPU buses that read from the memory and write back to it. In the given figure, we had not shown these external (address and data buses) in detail. Fig.2 will help us understand this external interface.

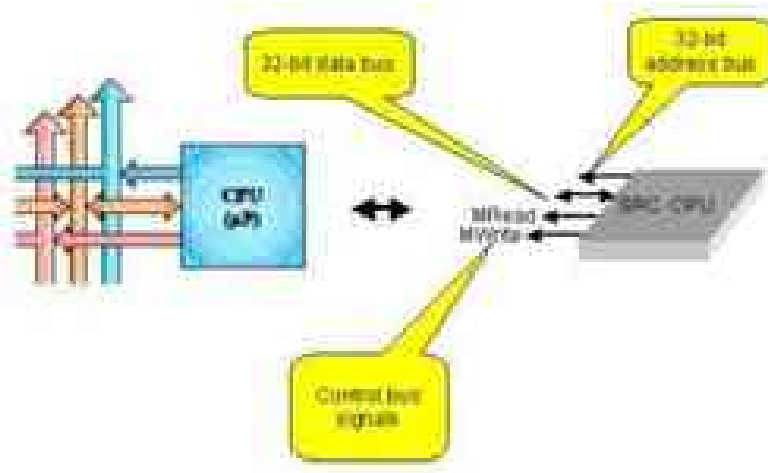


Fig.2

External CPU bus activity

Let us take up a sample problem to further enhance our understanding of the external CPU interface. As mentioned earlier, this interface consists of the data bus/ address bus, and control signals for enabling memory read and write.

Example problem:

- What will be the logic levels on the external SRC buses when each of the given SRC instruction is executing on the processor? Complete Table: 2. all numbers are in the decimal number system, unless noted otherwise.
- Specify memory addressing modes for each of the SRC instructions given in Table: 2.

Instruction	RTL equivalent	Address Bus <31..0>	Data Bus <31..0>	MRead	MWrite
ld r7, 12(r5)					
ld r2, r2					
ld r9, r2					
ldr r12, -r2					
ldr r3, 0					
st r2, 0(r6)					
str r3, -8					
st r9, r2					

Table 2

Assumptions:

- All memory content is aligned properly.

In other words, all the memory accesses start at addresses divisible by 4. Value in the PC = 000DC348h

(Note that the SRC uses the big-endian storage format). 

Memory map with assumed values

Memory Address	Memory Content	Memory Address	Memory Content
00000000h	D2h	000DC348h	11h
00000020h	90L	000DC34Ch	0Ah
00000022h	40h	000DC34Eh	D5h
00000024h	2Fh	000DC34Fh	82h
...	...	...	...
000DC300h	44h	00AB1240h	07h
000DC304h	23L	00AD124Ch	35h
000DC308h	E3h	00AB124Eh	52h
000DC30Ch	D5h	00AB124Fh	10h

Fig.3

Register map with assumed values

Register Name	Content
R[0]	0012A54Bh
R[1]	10234CB8h
R[2]	D296492Fh
R[3]	001400CDh
R[4]	B7432301h
R[5]	00AB1234h
R[6]	00000020h
R[7]	01432D7Fh
R[8]	00B94821h
R[9]	00CDA7A3h
R[10]	0031A0F0h
R[11]	0012A246h
R[12]	000FAB17h

Fig.4

Solution Part (a):

Instruction	RTL equivalent	Address Bus <A>_0</th>	Data Bus <D>_0</th>	Address	Write
ld r7, 12(r5)	$R[7] \leftarrow M[12 + R[5]]$	00400240 h	07850530 h	1	0
ld r2, 32	$R[2] \leftarrow M[32]$	00000020 h	02786490 h	1	0
la r9, 32	$R[9] \leftarrow 32$	Unknown	Unknown	7	7
ldr r12, -4	$R[12] \leftarrow M[PC - 4]$	00000000 h	44278300 h	1	0
lar r3, 0	$R[3] \leftarrow PC$	Unknown	Unknown	7	7
st r2, 0(r6)	$M[R[6]] \leftarrow R[2]$	00000020 h	02786490 h	0	1
str r3, -8	$M[PC - 8] \leftarrow R[3]$	00000000 h	00140000 h	0	1
st r4, 32	$M[32] \leftarrow R[4]$	00000020 h	07850530 h	0	1

Solution part (b):

SRC Instruction	Addressing Mode
ld r7, 12(r5)	Displacement
ld r2, 32	Direct
la r9, 32	Immediate
ldr r12, -4	PC Relative
lar r3, 0	Register
st r2, 0(r6)	Register Indirect
str r3, -8	PC Relative
st r4, 32	Register Direct

Fig:5

Notes:

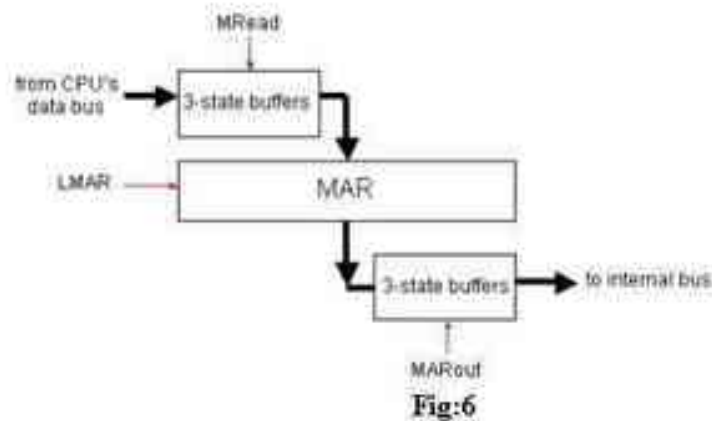
* Relative addressing is always PC relative in the SRC

*** Displacement addressing mode is the same as Based or Indexed in the SRC. It is also the same as Register Relative addressing mode

Memory address register circuitry

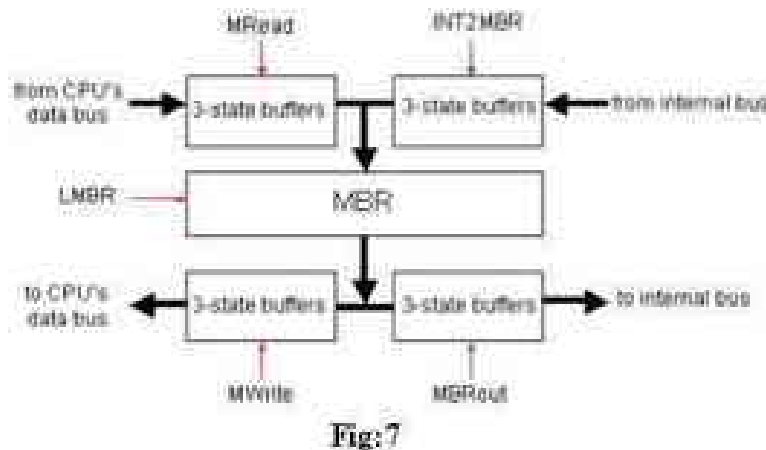
We have already talked about the functionality of the **MAR**. It provides a temporary storage for the address of memory location to be accessed. We now take a detailed look at how it is interconnected with other components. The MAR is connected directly to the CPU internal bus, from which it is loaded (receives a value). The LMAR signal causes the contents of the internal CPU bus to be loaded into the MAR. It writes onto the CPU external address bus. The MARout signal causes the contents of the MAR to be placed on the address bus. Thus, it provides the addresses for the memory and I/O devices over the CPU's address bus. A set of tri-state buffers

is provided with these connections; the tri-state buffers are controlled by the control signals, which in turn are issued when the corresponding instruction is decoded. The whole circuitry is shown in Fig.6.



Memory buffer register circuitry

The Memory Buffer Register (MBR) holds the value read from the memory or I/O device. It is possible to load the MBR from the internal CPU bus or from the external CPU data bus. The MBR also drives the internal CPU bus as well as the external CPU data bus. Similar to the MAR register, tri-state buffers are provided at the connection points of the MBR, as illustrated in the Fig.7.



Register connections

The register file containing the General Purpose Registers is programmer visible. Instructions may refer to any of these registers, as source operands in an operation or as the destination registers. Appropriate circuitry is needed to enable the specified register for read/ write. Intuitively, we can tell that we require connections of the register to the CPU internal bus, and we need control signals that will enable specified registers to be read/ write enabled as a corresponding instruction is decoded. Fig.8 illustrates the register connections and the control signals generation in the uni-bus data path of the SRC. We can see from this figure that the ra, rb and rc fields of the Instruction Register specify the destination and source registers. The control signals RAE, RBE and RCE can be applied to select any of the ra, rb or rc field respectively to apply its contents to the input of 5-to-32 decoder. Through the decoder, we get the signal for the specific register to be accessed. The BUS2R control signal is activated if it is desired to write into the register. On the other hand, if the register contents are to be written to the bus, the control signal R2BUS is activated.

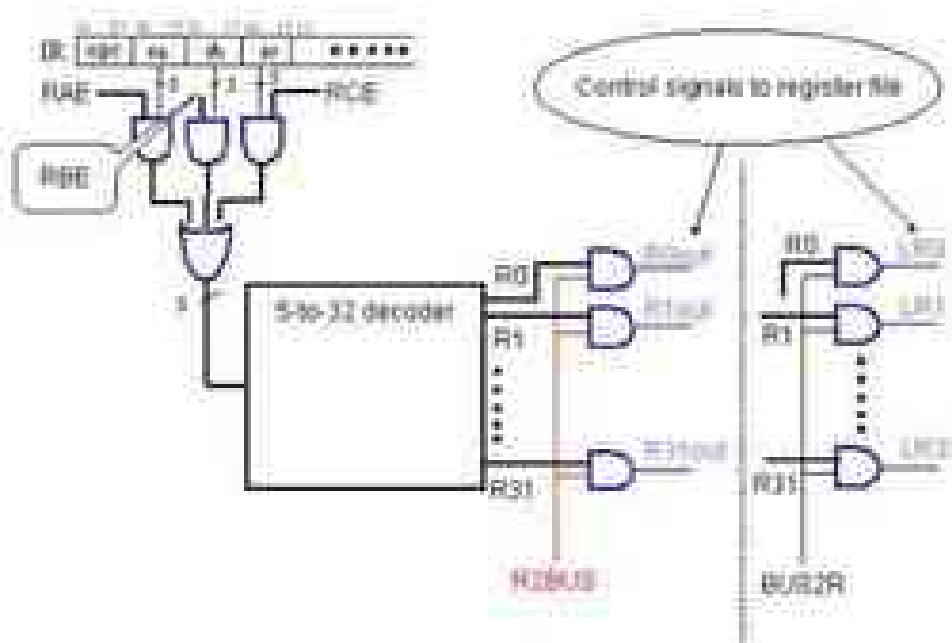


Fig.8

Alternate control circuitry for register selection

Fig.9 illustrates an alternate circuitry that implements the register connections with the internal processor bus, the instruction register fields, and the control signals required to coordinate the appropriate read/write for these registers. Note that this implementation is somewhat similar to our earlier implementation with a few differences. It illustrates the fact that the implementations we have presented are not necessarily the only solutions, and that there may be other possibilities.

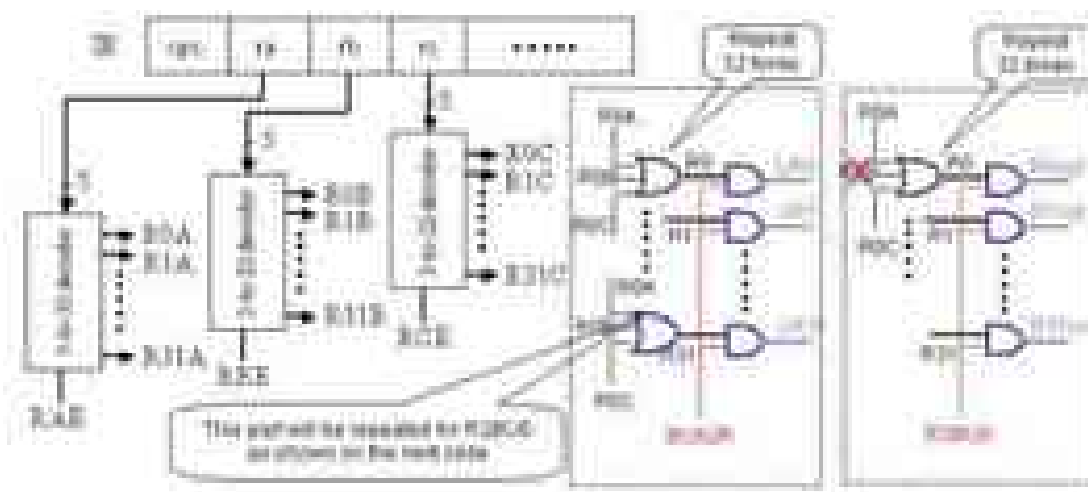


Fig.9

In this alternate circuitry, there is a separate 5-to-32 decoder for each of the register fields of the instruction register. The output of these decoders is allowed to be read out and enables the decoded register, if the control signal (RAE, RBE or RCE) is active.

Control signals Generation in SRC

We take a few example instructions to study the control signals that are required in the instruction execution phase.

Control signals for the add instruction

The add instruction has the following syntax:

add ra, rb, rc

Table: 4 lists the control signals that are applied at each of the time steps. The first three steps are of the instruction fetch phase, and we have already discussed the control signals applied at this phase.

Step	RTL	Control Signals
T0 – T2	Instruction Fetch	As before
T3	$A \leftarrow R[rb]$	RBE, R2BUS, LA
T4	$C \leftarrow A + R[rc]$	RCE, R2BUS, ADD, LC
T5	$R[ra] \leftarrow C$	Cout, RAE, BUS2R

At time step T3, the control RBE is applied, which will enable the register rb to write its contents onto the internal CPU bus, as it is decoded. The writing from the register onto the bus is enabled by the control signal R2BUS. Control signal LA allows the bus contents to be transferred to the register A (which will supply it to the ALSU). At time step T4, the control signals applied are RCE, R2BUS, ADD, LC, to respectively enable the register rc, enable the register to write onto the internal CPU bus (which will supply the second operand to the ALSU from the bus), select the add function of the ALSU (which will add the values) and enable register C (so the result of the addition operation is stored in the register C). Similarly in T5, signals Cout, RAE and BUS2R are activated.

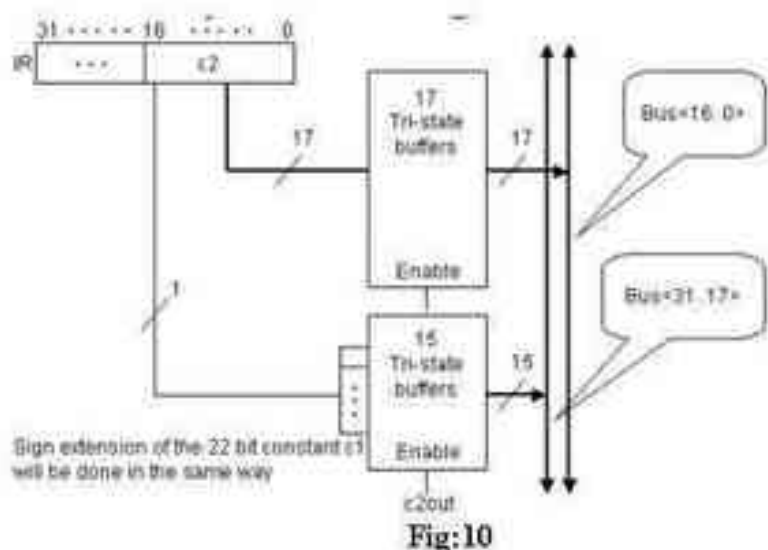
Sign extension

When we copy constant values to registers that are 32 bits wide, we need to sign extend the values first. These values are in the 2's complement form, and to sign-extend these values, we need to copy the most significant bit to all the additional bits in the register.

We consider the field c2, which is a 17 bit constant. Sign extension of c2 requires that we copy c2<16> to all the left-most bits of the destination register, in addition to copying the original constant values to the register.

This means that bus<31..17>

should be the same as c2<16>. A 15 line tri-state buffer can perform this sign extension. So we apply c2<16> to all the inputs of this tri-state buffer as illustrated in the Fig.10.



Structural RTL for the addi instruction

We now return to our study of the control signals required in the instruction execute phase. We have already looked at the add instruction and the corresponding signals. Now we take a look at the **addi** (add immediate) instruction, which has the following syntax:

addi ra, rb, c2

Table: 5 lists the RTL and the control signals for the **addi** instruction:

Step	RTL for addi	Control signals
T0-T2	instruction fetch	As before
T3	$A \leftarrow R[r_b]$	RBE, R2BUB, LA
T4	$C \leftarrow A + c2(\text{sign extend})$	c2out, ADD, LC
T5	$R[r_a] \leftarrow C$	Cout, RAE, BU2R

Table:5

The table shows that the control signals for the addi instruction are the same as the add instruction, except in the time step T4. At this time step, the control signals that are applied are c2out, ADD and LC, to respectively do the following:

Enable the read of the constant c2 (which is sign extended) onto the internal processor bus. Add the values using the ALSU and finally assign the result to register C by enabling write for this register.

To place a 0 on the bus

When the field rb is zero, for instance, in the **load** and **store** instructions, we need to place a zero on the bus. The given circuit in Fig.11 can be used to do this.

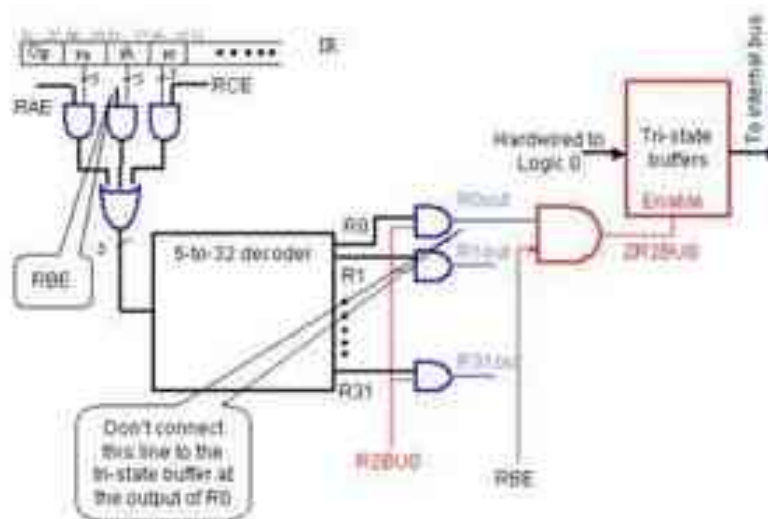


Fig:11

Note that, by default, the value of register R0 is 0 in some cases. So, when the selected register turns out to be 0 (as rb field is 0), the line connecting the output of the register R0 is not enabled, and instead a hardwired 0 is output from the tri-state buffer onto the CPU internal bus. An alternate circuitry for achieving the same is shown in the Fig.12.

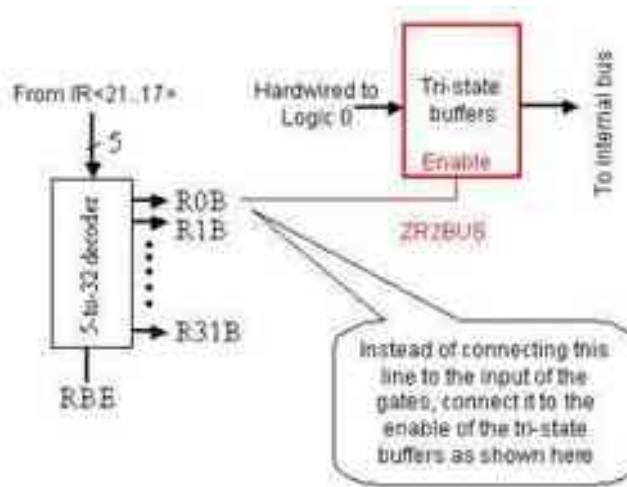


Fig:12

Control signals for the ld instruction

Now we take a look at the control signals for the **load** instruction. The syntax of the instruction is:

ld ra, c2 (rb)

Table: 6 outlines the control signals as well as the RTL for the **load** instruction in the SRC.

The first three steps are of the instruction fetch

Step	RTL for ld	Control Signals
T0-T2	Instruction fetch	AE before
T3	$A \leftarrow ((c2 = 0) ? 0 : (rb \neq 0) ? R[rb] : 0)$	RBE, R2BUS, LA
T4	$C \leftarrow A + (1 \text{ if } R[15] = 1 \text{ else } 0) \times R[15:0]$	C2out, ADD, LC
T5	$MAR \leftarrow C$	Cout, LMAR
T6	$MBR \leftarrow M[MAR]$	MBRout, MBRin, LMBR
T7	$R[ra] \leftarrow MBR$	MBRout, RAE, BUS2R

Table:6

RBE is issued to allow the register rb value to be read **R2BUS** to allow the bus to read from the selected register

LA to allow write onto the register A. This will allow the CPU bus contents to be written to the register A.

At step T4 the control signals are:

c2out to allow the sign extended value of field c2 to be written to the internal CPU bus **ADD** to instruct the ALSU to perform the add function.

LC to let the result of the ALSU function be stored in register C by enabling write of register C. Control signals issued at step T5:

Cout is to read the register C, this copies the value in C to the internal CPU bus.

LMAR to enable write of the Memory Address Register (which will copy the value present on the bus to MAR). This is the effective address of memory location that is to be accessed to read (load) the memory word.

During the time step T6:

MARout to read onto the external CPU bus (the address bus, to be more specific), the value stored in the MAR. This value is an index to memory location that is to be accessed.

MRead to enable memory read at the specified location, this loads the memory word at the specified location onto the CPU external data bus.

LMBR is the control signal to enable write of the MBR (Memory Buffer Register). It will obtain its value from the CPU external data bus. Finally, the control signals issued at the time step T7 are:

MBRout is the control signal to allow the contents of the MBR to be read out onto the CPU internal bus.

RAE is the control signal for the destination register field ra. It will let the actual index of the ra register be encoded, and

BUS2R will let the appropriate destination register be written to with the value on the CPU internal bus.

Lecture No. 16

Control Unit Design

Reading Material

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

Chapter 4
4.2.2, 4.6.1

Summary

- Control Signals Generation in SRC (continued...)
- The Control Unit
- 2-Bus Implementation of the SRC Data Path

This section of lecture 16 is a continuation of the previous lecture.

Control signals for the store instruction $st\ ra, c2(rb)$

The store time step operations are similar to the load instruction, with the exception of steps T6 and T7. However, one can easily interpret these now. These are outlined in the given table.

Step	RTL for st	Control Signals
T0-T2	Instruction Fetch	As before
T3	$A \leftarrow ((rb \ll 2) (D \ll 8))$ $R[rb]$	RBE, R2BUS, LADDR, LA
T4	$C \leftarrow A + (1 \ll 16) \ll 10$ $(R[15, 0] \gg 1)$	CADDR, ADDR, LC
T5	$MAP \leftarrow C$	COUT, LMAP
T6	$MBR \leftarrow R[rb]$	RAE, R2BUS, INT2MER, LMBR
T7	$M[MAP] \leftarrow MBR$	MAPRWE, MWRITE

Control signals for the branch and branch link instructions

Branch instructions can be either be simple branches or link-and-then-branch type. The syntax for the branch instructions is

brzr rb, rc

This is the branch and zero instruction we looked at earlier. The control signals for this instruction are:

As usual, the first three steps are for the instruction fetch phase. Next, the following control signals are issued:

Step	RTL for br	Control signals
T0-T2	Instruction Fetch	As before
T3	$CON \leftarrow cond \ll R[rc]$	LOON, RCE, R2BUS
T4	$CON \cdot PC \leftarrow R[rb]$	RBE, R2BUS, LPC (if $CON=1$)

LCON to enable the CON circuitry to operate, and instruct it to check for the appropriate condition (whether it is branch if zero, or branch if not equal to zero, etc.) **RCE** to allow the register rc value to be read.

R2BUS allows the bus to read from the selected register.

At step T4:

RBE to allow the register rb value to be read. rb value is the branch target address.

R2BUS allows the bus to read from the selected register.

LPC (if CON=1): this control signal is issued conditionally, i.e. only if CON is 1, to enable the write for the program counter. CON is set to 1 only if the specified condition is met. In this way, if the condition is met, the program counter is set to the branch address.

Branch and link instructions

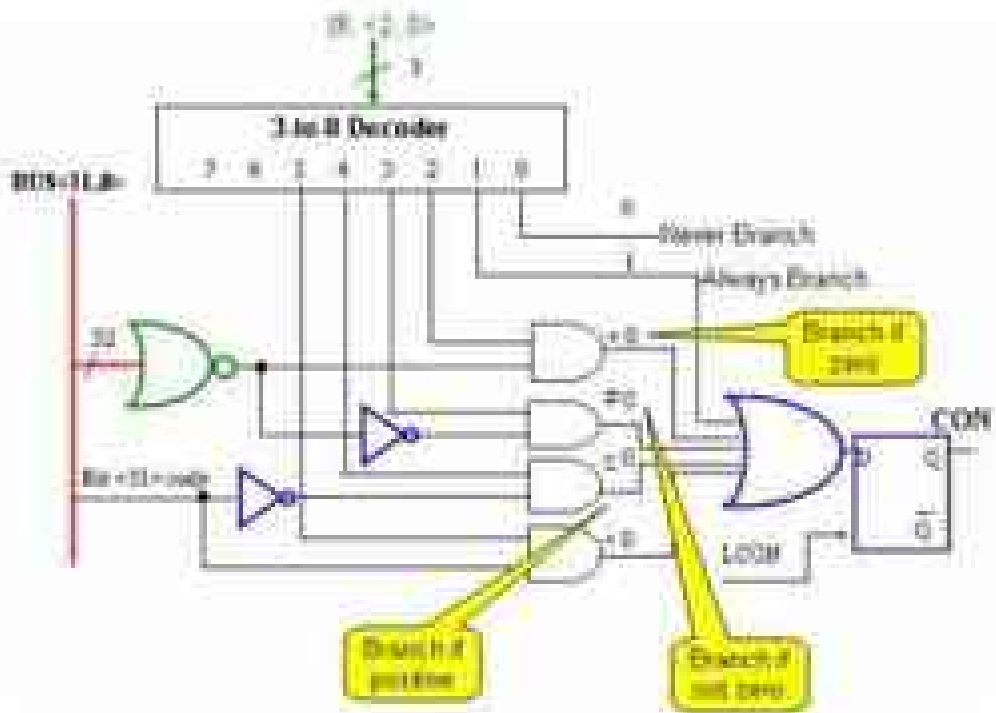
The branch and link instruction is similar to the branch instruction, with an additional step, T4. Step T4 of the simple conditional branch instruction becomes the step T5 in this case.

Step	RTL	Control signals
T0-T2	Instruction Fetch	As before
T3	$CON \leftarrow cond(R[rt])$	LDON, RCE, R2BUS
T4	$CON, R[ra] \leftarrow PC$	R4E, BUS2R, PCOUT (if $CON \neq 1$)
T5	$CON, PC \leftarrow R[rt]$	RBE, R2BUS, LPC (if $CON \neq 1$)

The syntax of the instruction ‘branch and link if zero’ is

brl **ra**, **rb**, **rc**

Table that lists the RTL and control signals for the store instruction of the SRC is given: The circuitry that enables the condition checking for the conditional branches in the SRC is illustrated in the following figure:

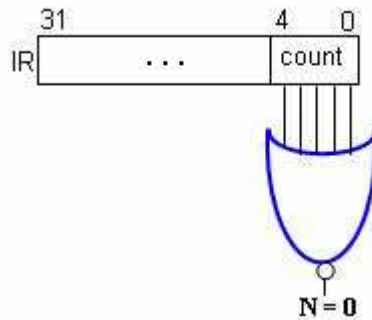


Control signals for the shift right instruction

The given table illustrates the RTL and the control signals for the shift right ‘shr’ instruction. This is implemented by applying the five bits of n ($nb4$, $nb3$, $nb2$, $nb1$, $nb0$) to the select inputs of the barrel shifter and activating the control signal SHR as explained in an earlier lecture.

Step	RTL for shr	Control signals
T0-T2	Instruction Fetch	As before
T3	$n<4..0> \leftarrow IR<4..0>;$	LN
T4	$(N = 0) : (n<4..0> \leftarrow R[rc]<4..0>);$	LN($N=0$), RCE, R2BUS
T5	$C \leftarrow (N \neq 0) \text{ OR } [rb]<31..N>;$	LC, SHR(N)
T6	$R[ra] \leftarrow C;$	Count, RAE, BUS2R

Generating the Test Condition $N=0$



The Control Unit

The control unit is responsible for generating control signals as well as the timing signals. Hence the control unit is responsible for the synchronization of internal as well as external events. By means of the control signals, the control unit instructs the data path what to do in every clock cycle during the execution of instructions.

Control Unit Design

Since the control unit performs quite complex tasks, its design must be done very carefully. Most errors in processor design are in the Control Unit design phase. There are primarily two approaches to design a control unit.

1. Hardwired approach
2. Micro programming

Hardwired approach is relatively faster, however, the final circuit is quite complex. The micro-programmed implementation is usually slow, but it is much more flexible.

“Finite-state machine” concepts are usually used to represent the CU. Every state corresponds to one “clock cycle” i.e., 1 state per clock. In other words each timing step could be considered as just 1 state and therefore from one timing step to other timing step, the state would change. Now, if we consider the control unit as a black box, then there would be four sets of inputs to the control unit. These are as follows:

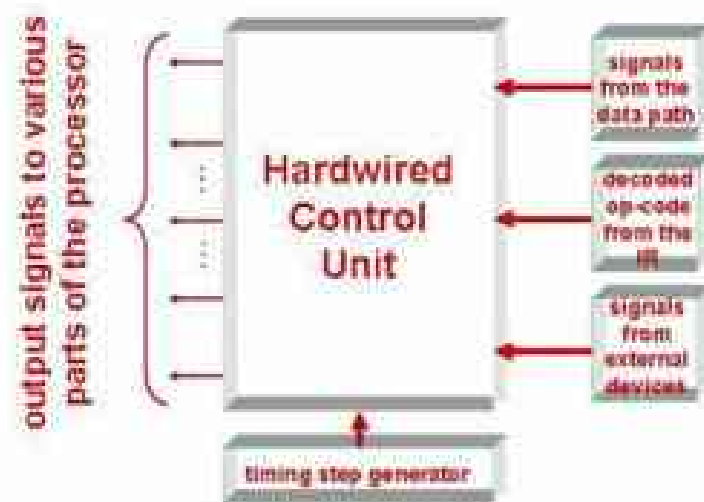
1. The output of timing step generator (There are 8 disjoint timing steps in our example T0-T7).
2. Op-code (op-code is first given to the decoder and the output of the decoder is given to the control unit).
3. Data path generated signals, like the “CON” control signal,
4. Signals from external events, like “Interrupt” generated by the Interrupt generator.

The complexity of the control is a function of the

- Number of states
- Number of inputs to the CU
- Number of the outputs generated by the CU

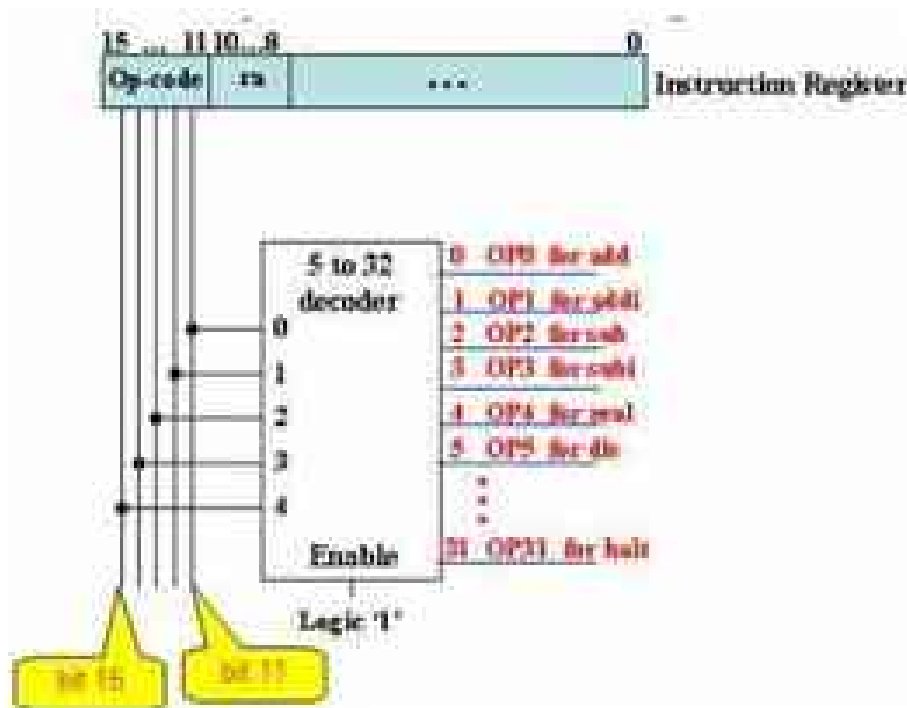
Hardwired Implementation of the Control Unit

The accompanying block diagram shows the inputs to the control unit. The output control signals generated from control unit to the various parts of the processor are also shown in the figure.



Example Control Unit for the FALCON-A

The following figure shows how the operation code (op-code) field of the Instruction Register is decoded to generate a set of signals for the Control unit.



This is an example for the FALCON-A processor where the instruction is 16-bit long. Similar concepts will apply to the SRC, in which case the instruction word is 32 bits and IR <31...27> contains the op-code. Similar concepts will apply to the SRC, in which case the instruction word is 32 bits and IR<31..27> contains the opcode. The most significant 5 bits represent the op-code. These 5-bits from the IR are fed to a 5-to-32 decoder. These 32 outputs are numbered from 0-to-31 and named as op0, op1 up to op31. Only one of these 32 outputs will be active at a given time. The active output will correspond to instruction executing on the processor.

To design a control unit, the next step is to write the Boolean Equations. For this we need to browse through the structural descriptions to see which particular control signals occur in different timing steps. So, for each instruction we have one such table defining structural RTL and the control signals generated at each timing step. After browsing we need to check that which control signal is activated under which condition. Finally we need to write the expression in the form of a logical expression as the logical combination of “AND” and “OR” of different control signals. The given table shows Boolean Equations for some example control signals.

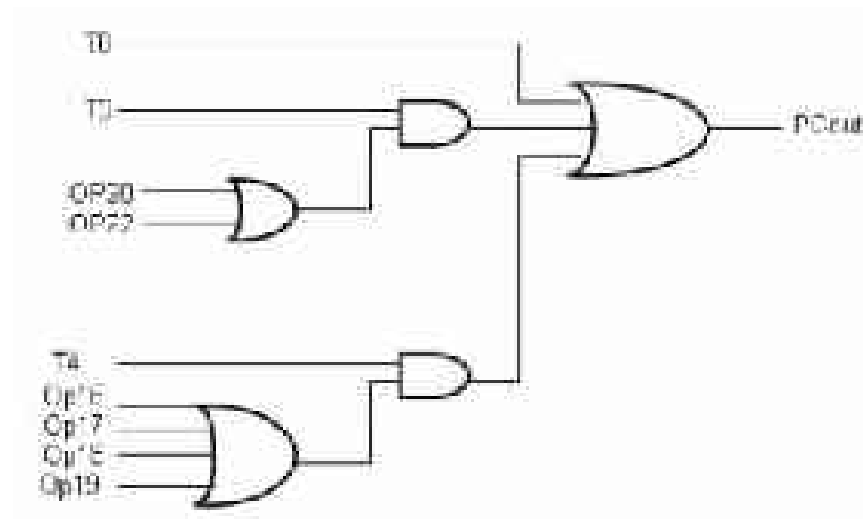
Step	RTL	Control Signals
T0	MAR ← PC;	PCout, LMAR, C=B;
T1	MBR ← M[MAR], PC ← PC + 4;	PCout, INC4, LPC, MRead, MARout, LMBR;
T2	IR ← MBR;	MBRout, C=B, LIR;
T3	Instruction Execution	

For example, PCout would be active in every T0 timing step. Then in timing interval T3 the output of the PC would be activated if the op-code is 20 or 22 which represent jump and sub-routine call. In step T4 if the op-code is 16, 17, 18 or 19, again we need PCout activated and these 4 instructions correspond to the conditional jumps. We can say that in other words in step

T1, PCout is always activated “OR” in T3 it is activated if the instruction is either jump or sub-routine call “OR” in T4 if there is one of the conditional jumps. We can write an equation for it as

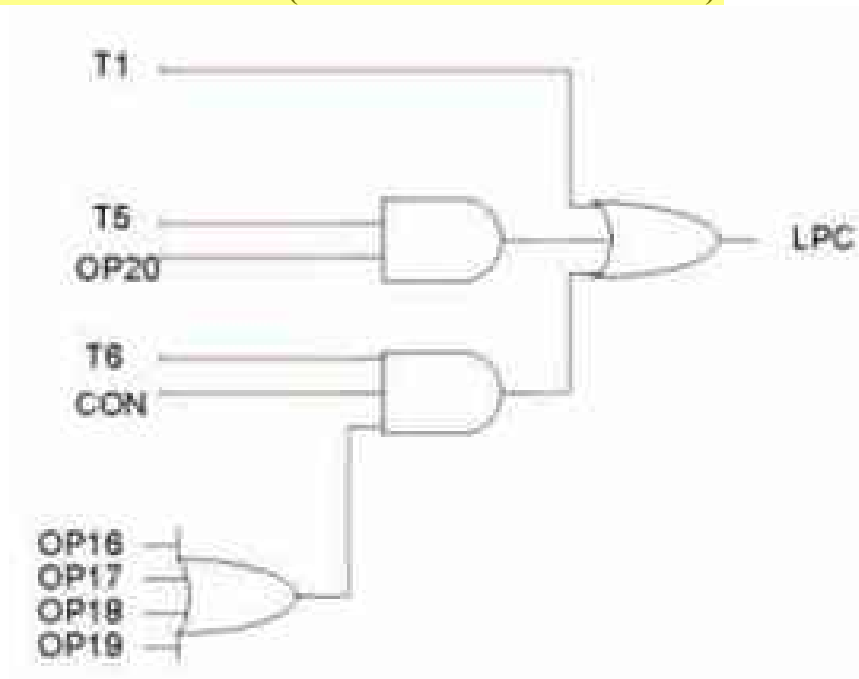
$$\text{PCout} = T0 + T3.(OP20 + OP22) + T4.(OP16 + OP17 + OP18 + OP19)$$

In the form of logic circuit the implementation is shown in the figure. We can see that we “OR” the op-ode 20 and 22 and “AND” it with T3, then “OR” all the op16 up to op19 and “AND” it with T4, then T0 and the “AND” outputs of T3 and T4 are “OR” together to obtain the PCout.



In the same way the logic circuit for LPC control signal is as shown and the equation would be :

$$\text{LPC} = T1 + T5.OP20 + T6.CON.(OP16 + OP17 + OP18 + OP19)$$



We can formulate Boolean equations and draw logic circuits for other control signals in the same way.

Effect of using “real” Gates

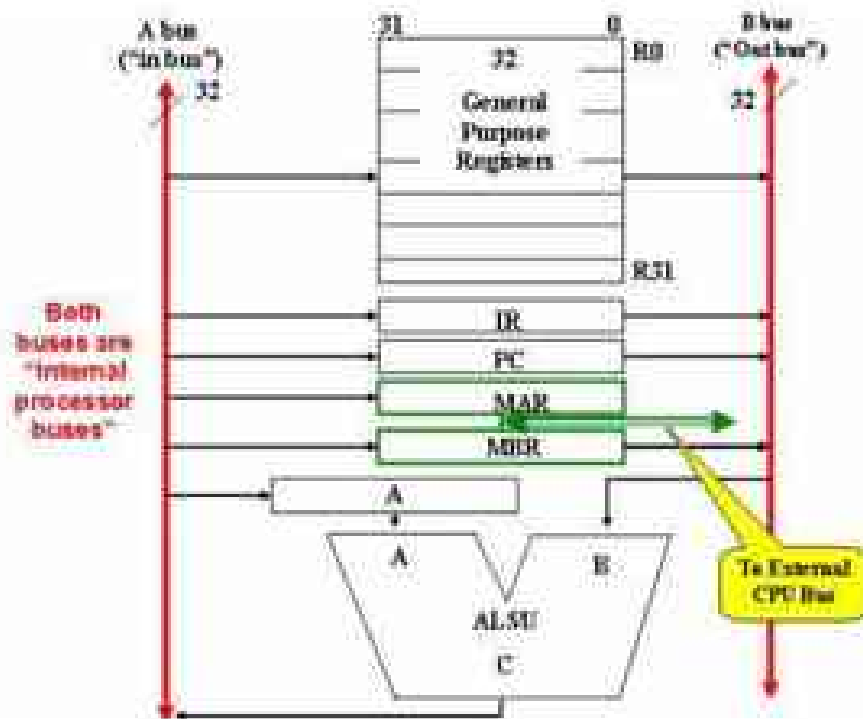
We have assumed so far that the gates are ideal and that there is no propagation delay. In designing the control unit, the propagation delays for the gates cannot be neglected. In particular, if different gates are cascaded, the output of one gate forms the input of other. The propagation delays would add up. This, in turn would place an upper limit on the frequency of the clock which controls the generation of the timing intervals $T_0, T_1 \dots T_7$. So, we cannot arbitrarily increase the frequency of this clock. As an example consider the transfer of the contents of a register R1 to a register R2. The minimum time required to perform this transfer is given by

$$t_{\min} = t_g + t_{bp} + t_{comb} + t_1$$

The details are explained in the text with reference to Fig 4.10. Thus, the maximum clock frequency based on this transfer will be $1/t_{\min}$. Students are encouraged to study example 4.1 of the text.

2-Bus Implementation of the SRC Data Path

In the previous sections, we studied the uni-bus implementation of the data path in the SRC. Now we present a 2-bus implementation of the data path in the SRC. We observe from this figure that there is a bus provided for data that is to be written to a component. This bus is named the ‘in’ bus. Another bus is provided for reading out the values from these components. It is called the ‘out’ bus.



Structural RTL for the ‘sub’ instruction using the 2-bus data path implementation

Next, we look at the structural RTL as well as the control signals that are issued in sequence for instruction execution in a 2-bus implementation of the data path. The given table illustrates the Register Transfer Language representation of the operations for carrying out instruction fetch, and execution for the sub instruction.

	Step	RTL
Instruction Fetch	T0	$MAR \leftarrow PC;$
	T1	$MBR \leftarrow M[MAR], PC \leftarrow PC + 4;$
	T2	$IR \leftarrow MBR;$
Instruction Execute	T3	$A \leftarrow R[rb];$
	T4	$R[ra] \leftarrow A - R[rc];$

The first three steps belong to the instruction fetch phase; the instruction to be executed is fetched into the Instruction Register and the PC value is incremented to point to the next-in-line instruction. At step T3, the register R[rb] value is written to register A. At the time step T4, the subtracted result from the ALSU is assigned to the destination register R[ra]. Notice that we did not need to store the result in a temporary register due to the availability of two buses in place of one. At the end of this sequence, the timing step generator is initialized to T0.

Control signals for the fetch operation

The control signals for the instruction fetch phase are shown in the table. A brief explanation is given below:

Step	RTL	Control Signals
T0	$MAR \leftarrow PC;$	PCout, LMAR, C=B;
T1	$MBR \leftarrow M[MAR],$ $PC \leftarrow PC + 4;$	PCout, INC4, LPC, MRead, MARout, LMBR;
T2	$IR \leftarrow MBR;$	MBRout, C=B, LIR;
T3	Instruction Execution	

At time step **T0**, the following control signals are issued:

- **PCout:** This will enable read of the Program Counter, and so its value will be transferred onto the 'out' bus
- **LMAR:** To enable the load for MAR
- **C=B:** This instruction is used to copy the value on the 'out' bus to the 'in' bus, so it can be loaded into the Memory Address Register. We can observe in the data-path implementation figure given earlier that, at any time, the value on the 'out' bus makes up the operand B for the ALSU. The result C of ALSU is connected to the "in" bus, and therefore, the contents transfer from one bus to the other can take place.

At time step T1:

- **PCout:** Again, this will enable read of the Program Counter, and so its value will be transferred onto the CPU internal 'out' bus
- **INC4:** To instruct the ALSU to perform the increment-by-four operation.
- **LPC:** This control signal will enable write of the Program Counter, thus the new, incremented value can be written into the PC if it is made available on the "in" bus. Note that the ALSU is assumed to include an INC4 function.
- **MRead:** To enable memory word read.
- **MARout:** To supply the address of memory word to be accessed by allowing the contents of the MAR (memory address register) to be written onto the CPU external (address) bus.
- **LMBR:** The memory word is stored in the register MBR (memory buffer register) by applying this control signal to enable the write of the MBR.

At time step T2:

- **MBRout:** The contents of the Memory Buffer Register are read out onto the 'out' bus, by means of applying this signal, as it enables the read for the MBR.
- **C=B:** Once again, this signal is used to copy the value from the 'out' bus to the 'in' bus, so it can be loaded into the Memory Address Register.
- **LIR:** This instruction will enable the write of the Instruction Register. Hence the instruction that is on the 'in' bus is loaded into this register.

At time step T3, the execution may begin, and the control signals issued at this stage depend on the actual instruction encountered. The control signals issued for the instruction fetch phase are the same for all the instructions.

Note that, we assume the memory to be fast enough to respond during a given time slot. If that is not true, wait states have to be inserted. Also keep in mind that the control signals during each time slot are activated simultaneously, while those for successive time slots are activated in sequence. If a particular control signal is not shown, its value is zero.



Lecture No. 17

Machine Reset and Machine Exceptions

Reading Material

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

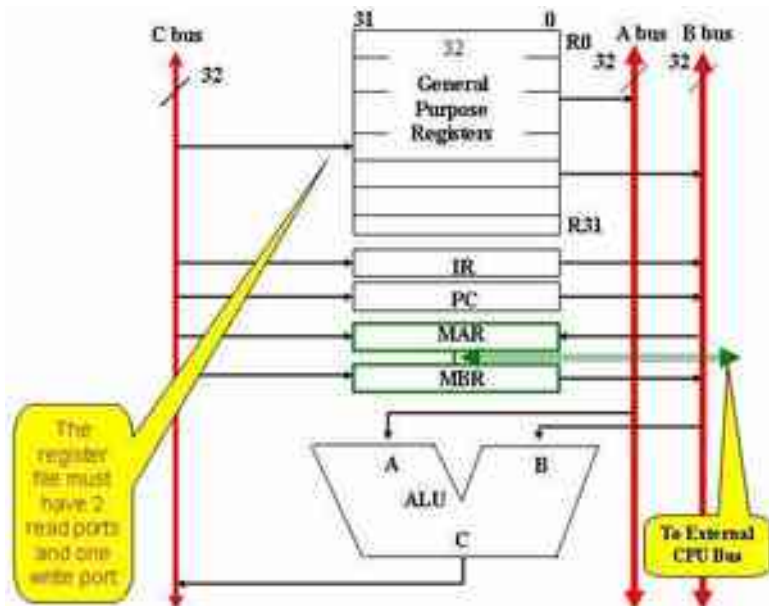
Chapter 4
4.6.2, 4.7, 4.8

Summary

- 3-bus implementation for the SRC
- The Machine Reset
- Machine Exceptions

A 3-bus Implementation for the SRC

Let us now look at a 3-bus implementation of the data-path for the SRC as shown in the figure. Two buses, 'A' and 'B' bus for reading, and a bus 'C' for writing, are part of this implementation. Hence all the special purpose as well as the general purpose registers have two read ports and one write port.



Structural RTL for the Subtract Instruction using the 3-bus Data Path Implementation

We now consider how instructions are fetched and executed in 3-bus architecture. For this purpose, the same 'sub' instruction example is followed.

The syntax of the subtract instructions is

sub ra, rb, rc

The structural RTL for implementing this instruction is given in the table. We observe that in this table, only two time steps are required for the instruction fetch phase.

At time step T0, the Memory Address Register receives the value of the Program Counter. This

	Step	RTL
Instruction Fetch	T0	MAR ← PC; MBR ← M[MAR]; PC ← PC + 4;
	T1	IR ← MBR;
Instruction Execute	T2	R[ra] ← R[rb] - R[rc].

is done in the initial phase of the time step T0. Then, the Memory Buffer Register receives the memory word indexed by the MAR, and the PC value is incremented. At time step T1, the instruction register is assigned the instruction word that was loaded into the MBR in the previous time step. This concludes the instruction fetch and now the instruction execution can commence.

In the next time step, T2, the instruction is executed by subtracting the values of register rc from rb, and assigning the result to the register ra.

At the end of each sequence, the timing step generator is initialized to T0

Control Signals for the Fetch Operation

The given table lists the control signals in the instruction fetch phase. The control signals for the execute phase can be written in a similar fashion.

Step	RTL	Control Signals
T0	$MAR \leftarrow PC$, $MBR \leftarrow M[MAR]$, $PC \leftarrow PC + 4$	PC_{out} , $INC4$, LPC , $LMAR$, $MRead$
T1	$IR \leftarrow MBR$	MBR_{out} , $C \leftarrow B$, LR
T2	Instruction_Execution	

The Machine Reset

In this section, we will discuss the following

- Reset operation
- Behavioral RTL for SRC reset
- Structural RTL for SRC reset

The reset operation

Reset operation is required to change the processor's state to a known, defined value. The two essential features of a reset instruction are clearing the control step counter and reloading the PC to a predefined value. The control step counter is set to zero so that operation is restarted from the instruction fetch phase of the next instruction. The PC is reloaded with a predefined value usually to execute a specific recovery or initializing program.

In most implementations the reset instruction also clears the interrupt enable flags so as to disable interrupts during the initialization operation. If a condition code register is present, the reset instruction usually clears it, so as to clear any effects of previously executed instructions. The external flags and processor state registers are usually cleared too.

The reset instruction is mainly used for debugging purposes, as most processors halt operations immediately or within a few cycles of receiving the reset instruction. The processors state may then be examined in its halted state.

Some processors have two types of reset operations. Soft reset implies initializing PC and interrupt flags. Hard reset initializes other processor state registers in addition to PC and interrupts enable flags. The software reset instruction asserts the external reset pin of the processor.

Reset operation in SRC

Hard Reset

The SRC should perform a hard reset upon receiving a start (Strt) signal. This initializes the PC and the general registers.

Soft Reset

The SRC should perform a soft reset upon receiving a reset (rst) signal. The soft reset results in initialization of PC only.

The reset signal in SRC is assumed to be external and asynchronous.

PC Initialization

There are basically two approaches to initialize a PC.

1. Direct Approach

The PC is loaded with the address of the startup routine upon resetting.

2. Indirect Approach

The PC is initialized with the address where the address of the startup routine is located. The reset instruction loads the PC with the address of a jump instruction. The jump instruction in turn contains the address of the required routine.

An example of a reset operation is found in the 8086 processor. Upon receiving the reset instruction the 8086 initializes its PC with the address FFFF0H. This memory location contains a jump instruction to the bootstrap loader program. This program provides the system initialization

Behavioral RTL for SRC Reset

The original behavioral RTL for SRC without any reset operation is:

```
Instruction_Fetch := (! Run & Strt : (Run ← 1; instruction_Fetch,
                                Run : (IR ← M [PC]; PC ← PC+4; instruction_execution)),
                    instruction_execution := (ld (:=op=1...)) ;
```

This recursive definition implies that each instruction at the address supplied by PC is executed.

The modified RTL after adding the reset capability is

```
Instruction_Fetch := (! Run & Strt : (Run ← 1,
                                PC, R [0...31] ← 0),
                    Run & !Rst : (IR ← M [PC],
                                PC ← PC+4, instruction_execution);
                    Run & Rst : (Rst ← 0, PC ← 0);
                    instruction_Fetch),
```

The modified definition includes testing the value of the “rst” signal after execution of each instruction. The processor may not be halted in the midst of an instruction *in the RTL definition*

To actually implement these changes in the SRC, the following modifications are required to add the reset operation to the structural RTL for SRC:

- A check for the reset signal on each clock cycle
- A control signal for clearing the PC
- A control signal to load zero to control step counter

Example: The sub instruction with RESET processing

To actually reset the processor in the midst of an instruction, the “Rst” condition must be tested after each clock cycle.

Let us examine the contents of each phase in the given table. In step T0, if the Rst signal is not asserted, the address of the new instruction is delivered to memory and the value of PC is incremented by 4 and stored in another register. If the “Rst” signal is asserted, the “Rst” signal is immediately cleared, the PC is cleared to zero and T, the step counter is also set to zero. This behavior (in case of ‘Rst’ assertion) is the same for all steps. In step T1, if the rst signal is not asserted, the value stored at the delivered memory word is stored in the memory data register and the PC is set to its incremented value.

In step T2, the stored memory data is transferred to the instruction register.

In step T3, the register operand values are read.

In step T4, the mathematical operation is executed.

In step T5, the calculated value is written back to register file.

During all these steps if the Rst signal is asserted, the value of PC is set to 0 and the value of the step counter is also set to zero.

Step	RTN	Control Sequence
T0	$IRst: (MA \leftarrow PC, C \leftarrow PC+4),$ $Rst: (Rst \leftarrow 0, PC \leftarrow 0, T \leftarrow 0)$	$IRst: (PC_{new} \leftarrow LMAR, INC4, LC,$ $MRead), Rst: (ClrPC, Goto0),$
T1	$IRst: (MD \leftarrow M[MA], PC \leftarrow C),$ $Rst: (Rst \leftarrow 0, PC \leftarrow 0, T \leftarrow 0)$	$IRst: (C_{new} \leftarrow LPC, Wait),$ $Rst: (ClrPC, Goto0),$
T2	$IRst: (IR \leftarrow MD),$ $Rst: (Rst \leftarrow 0, PC \leftarrow 0, T \leftarrow 0)$	$IRst: (MBR_{new} \leftarrow LIR),$ $Rst: (ClrPC, Goto0),$
T3	$IRst: (A \leftarrow R[ra]),$ $Rst: (Rst \leftarrow 0, PC \leftarrow 0, T \leftarrow 0)$	$IRst: (RBE, R2BUS, LA),$ $Rst: (ClrPC, Goto0),$
T4	$IRst: (C \leftarrow A - R[ra]),$ $Rst: (Rst \leftarrow 0, PC \leftarrow 0, T \leftarrow 0)$	$IRst: (PCE, R2BUS, SUB, LC),$ $Rst: (ClrPC, Goto0),$
T5	$IRst: (R[ra] \leftarrow C),$ $Rst: (Rst \leftarrow 0, PC \leftarrow 0, T \leftarrow 0)$	$IRst: (LC \leftarrow RAE, BUS2R \leftarrow End),$ $Rst: (ClrPC, Goto0),$

Machine Exceptions

- Anything that interrupts the normal flow of execution of instructions in the processor is called an exception.
- Exceptions may be generated by an external or internal event such as a mouse click or an attempt to divide by zero etc.
- External exceptions or interrupts are generally asynchronous (do not depend on the system clock) while internal exceptions are synchronous (paced by internal clock)
- The exception process allows instruction flow to be modified, in response to internal or external events or anomalies. The normal sequence of execution is interrupted when an exception is thrown.

Exception Processing

A generalized exception handler should include the following mechanisms:

- Logic to resolve priority conflicts.** In case of nested exceptions or an exception occurring while another is being handled the processor must be able to decide which exception bears the higher priority so as to handle it first. For example, an exception raised by a timer interrupt might have a higher priority than keyboard input.
- Identification of interrupting device.** The processor must be able to identify the interrupting device that it can load the appropriate exception handler routine. There are two basic approaches for managing this identification: exception vectors and

“information” register. The exception vector contains the address of the exception handling routine. The interrupting process fills the exception vector as soon as the interruption is acknowledged. The disadvantage of this approach is that a lot of space may be taken up by vectors and exception handler codes.

3. In the information register, only one general purpose exception handler is used. The PC is saved and the address of the general purpose register is loaded into the PC. The interrupting process must fill the information register with information to allow identification of the cause and type of exception.
4. **Saving the processor state.** As stated earlier the processor state must be saved before jumping to the exception handler routine. The state includes the current value of the PC, general purpose registers, condition vector and external flags.
5. **Exception disabling during critical operation.** The processor must disable interrupts while it is switching context from the interrupted process to the interrupting process, so that another exception might not disrupt the transition.

Examples of Exceptions

- **Reset Exception**
Reset operation is treated as an exception by some machines e.g. SPARC and MC68000.
- **Machine Check**
This is an external exception caused by memory failure
- **Data Access Exception**
This exception is generated by memory management unit to protect against illegal accesses.
- **Instruction Access Exception**
Similar to data access exception
- **Alignment Exception**
Generated to block misaligned data access

Types of Exception

- **Program Exceptions**
These are exceptions raised during the process of decoding and executing the instruction. Examples are illegal instruction, raised in response to executing an instruction which does not belong to the instruction set. Another example would be the privileged instruction exception.
- **Hardware Exceptions**
There are various kinds of hardware exceptions. An example would be of a timer which raises an exception when it has counted down to zero.
- **Trace and debugging Exceptions**
Variable trace and debugging is a tricky task. An easy approach to make it possible is through the use of traps. The exception handler which would be called after each instruction execution allows examination of the program variables.
- **Non-Maskable Exceptions**
These are high priority exceptions reserved for events with catastrophic consequences such as power loss. These exceptions cannot be suppressed by the processor under any condition. In case of a power loss the processor might try to save the system state to the hard drive, or alert an alternate power supply.
- **Interrupts (External Exceptions)**
Exception handlers may be written for external interrupts, thus allowing programs to respond to external events such as keyboard or mouse events.

Lecture No. 18

Pipelining

Reading Material

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

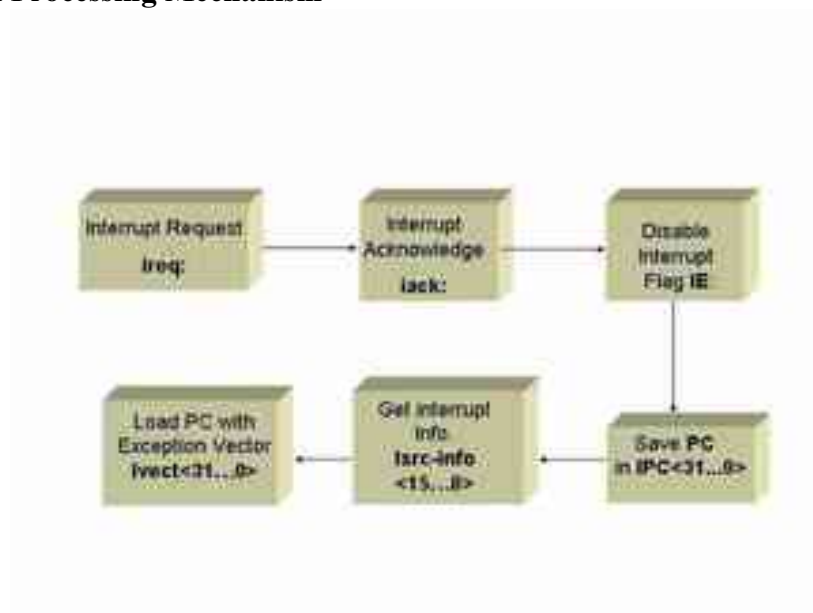
Chapter 4
4.8

Summary

- SRC Exception Processing Mechanism
- Introduction to Pipelining
- Complications Related to Pipelining
- Pipeline Design Requirements

Correction: Please note that the phrase “instruction fetch” should be used where the speaker has used “instruction interpretation”.

SRC Exception Processing Mechanism



The following tables on the next few pages summarize the changes needed in the SRC description for including exceptions:

Behavioral RTL for Exception Processing

Instruction_Fetch:=

(!Run&Strt: Run $\leftarrow$ 1,	Start
Run & !(ireq&IE):(IR $\leftarrow$ M[PC],	Normal Fetch
PC $\leftarrow$ PC + 4;	
Instruction_Execution),	
Run&(ireq&IE): (IPC $\leftarrow$ PC<31..0>,	Interrupt, PC copied
II<15..0> $\leftarrow$ Isrc_info<15..0>,	II is loaded with the info. PC loaded with
IE $\leftarrow$ 0: PC $\leftarrow$ Ivect<31..0>,	new address
iack $\leftarrow$ 1; iack $\leftarrow$ 0),	
Instruction_Fetch);	

Additional Instructions to Support Interrupts

Mnemonic	Behavioral RTL	Meaning
svi (op=16)	$R[ra]<15..0> \leftarrow II<15..0>$, $R[rb] \leftarrow IPC<31..0>$;	Save II and IPC
ri (op=17)	$II<15..0> \leftarrow R[ra]<15..0>$, $IPC<31..0> \leftarrow R[rb]$;	Restore II and IPC
een (op=10)	IE $\leftarrow$ 1;	Exception enable
edi (op=11)	IE $\leftarrow$ 0;	Exception disable
rfi (op=30)	PC $\leftarrow$ IPC, IE $\leftarrow$ 1;	Return from interrupt

Structural RTL for the Fetch Phase including Exception Processing

Step	Structural RTL for the 1-bus SRC
T0	$!(ireq \& IE): (MA \leftarrow PC, C \leftarrow PC + 4);$ $(ireq \& IE): (IPC \leftarrow PC, II \leftarrow Isrc_info,$ $IE \leftarrow 0, PC \leftarrow (22\alpha \ 0) \oplus (Isrc_vect \langle 7..0 \rangle) \oplus 00, iack \leftarrow 1;$ $iack \leftarrow 0, End);$
T1	$MD \leftarrow M[MA], PC \leftarrow C;$
T2	$IR \leftarrow MD;$
T3	Instruction_Execution;

Combining the RTL for Reset and Exception Instruction_Fetch:=

RTL	Event
$(Run \& !Rst \& !(ireq \& IE)): (IR \leftarrow M[PC], PC \leftarrow PC + 4;$ Instruction_Execution),	Normal
$Run \& Rst: (Rst \leftarrow 0, IE \leftarrow 0, PC \leftarrow 0; Instruction_Fetch),$	Fetch
$!Run \& Strt: (Run \leftarrow 1, PC \leftarrow 0, R[0..31] \leftarrow 0; Instruction_Fetch),$	Soft Reset
$Run \& !Rst \& (ireq \& IE): (IPC \leftarrow PC \langle 31..0 \rangle,$	Hard Reset
$II \langle 15..0 \rangle \leftarrow Isrc_info \langle 15..0 \rangle, IE \leftarrow 0, PC \leftarrow Ivect \langle 31..0 \rangle,$ $iack \leftarrow 1; iack \leftarrow 0; Instruction_Fetch);$	Interrupt

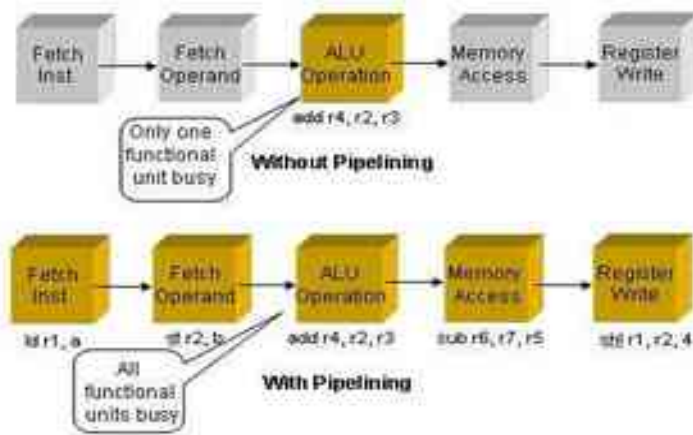
Introduction to Pipelining

Pipelining is a technique of overlapping multiple instructions in time. A pipelined processor issues a new instruction before the previous instruction completes. This results in a larger number of operations performed per unit of time. This approach also results in a more efficient usage of all the functional units present in the processor, hence leading to a higher overall throughput. As an example, many shorter integer instructions may be executed along with a

longer floating point multiply instruction, thus employing the floating point unit simultaneously with the integer unit.

Executing machine instructions with and without pipelining

We start by assuming that a given processor can be split in to five different stages as shown in the diagram below, and as explained later in this section. Each stage receives its input from the previous stage and provides its result to the next stage. It can be easily seen from the diagram that in case of a non-pipelined machine there is a single instruction **add r4, r2, r3** being processed at a given time, while in a pipelined machine, five different instructions are being processed simultaneously. An implied assumption in this case is that at the end of each stage, we have some sort of a storage place (like temporary registers) to hold the results of the present stage till they are used by the next stage.



Description of the Pipeline Stages

In the following paragraphs, we discuss the pipeline stages mentioned in the previous example.

1. Instruction fetch

As the name implies, the instruction is fetched from the instruction memory in this stage. The fetched instruction bits are loaded into a temporary pipeline register.

2. Instruction decode/operand fetch

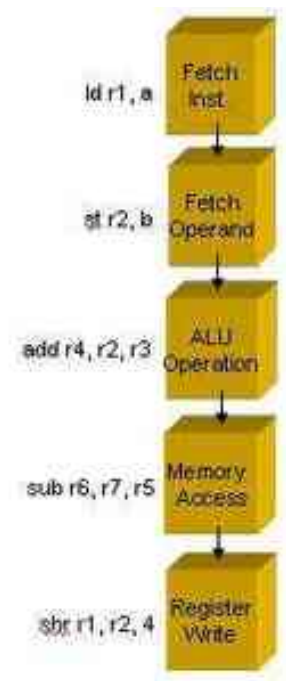
In this stage the operands for the instruction are fetched from the register file. If the instruction is **add r1, r2, r3** the registers r2 and r3 will be read into the temporary pipeline registers.

3. ALU⁵ operation

In this stage, the fetched operand values are fed into the ALU along with the function which is required such as addition, subtraction, etc. The result is stored into temporary pipeline registers. In case of a memory access such as a load or a store instruction, the ALU calculates the effective memory address in this stage.

4. Memory access

For a load instruction, a memory read operation takes place. For a store instruction, a memory write operation is performed. If there is no memory access involved in the instruction, this stage is simply bypassed.



⁵ The ALU is also called the ALSU in some cases, in particular, where its “shifting” capabilities need to be highlighted. ALSU stands for Arithmetic Logic Shift Unit.

5. Register write

The result is stored in the destination register in this stage.

Latency & throughput

Latency is defined as the time required to process a single instruction, while throughput is defined as the number of instructions processed per second. Pipelining cannot lower the latency of a single instruction; however, it does increase the throughput. With respect to the example discussed earlier, in a non-pipelined machine there would be one instruction processed after an average of 5 cycles, while in a pipelined machine, instructions are completed after each and every cycle (in the steady-state, of course!!!). Hence, the overall time required to execute the program is reduced.

Remember that the performance gain in a pipeline is limited by the slowest stage in the pipeline.



Complications Related to Pipelining

Certain complications may arise from pipelining a processor. They are explained below:

Data dependence

This refers to the situation when an instruction in one stage of the pipeline uses the results of an instruction in the previous stage. As an example let us consider the following two instructions

...

S1: add r3, r2, r1

S2: sub r4, r5, r3

...

There is a data-dependence among the above two instructions. The register R3 is being written to in the instruction S1, while it is being read from in the instruction S2. If the instruction S2 is executed before instruction S1 is completed, it would result in an incorrect value of R3 being used.

Resolving the dependency

There are two methods to remedy this situation:

1. Pipeline stalls

These are inserted into the pipeline to block instructions from entering the pipeline until some instructions in the later part of the pipeline have completed execution. Hence our modified code would become

...

S1: add r3, r2, r1

stall⁶

stall

stall

S2: sub r4, r5, r3

...

⁶ A pipeline stall can be achieved by using the **nop** instruction.

2. Data forwarding

When using data forwarding, special hardware is added to the processor, which allows the results of a particular pipeline stage to be transferred directly to another stage in the pipeline where they are required. Data may be forwarded directly from the execute stage of one instruction to the decode stage of the next instruction. Considering the above example, S1 will be in the execute stage when S2 will be decoded. Using a comparator we can determine that the destination operand of S1 and source operand of S2 are the same. So, the result of S1 may be directly forwarded to the decode stage.

Other complications include the “branch delay” and the “load delay”. These are explained below:

Branch delay

Branches can cause problems for pipelined processors. It is difficult to predict whether a branch will be taken or not before the branch condition is tested. Hence if we treat a branch instruction like any normal instruction, the instructions following the branch will be loaded in the stages following the stage which carries the branch instruction. If the branch is taken, then those instructions would need to be removed from the pipeline and their effects if any, will have to be undone. An alternate method is to introduce stalls, or **nop** instructions, after the branch instruction.

Load delay

Another problem surfaces when a value is loaded into a register and then immediately used in the next operation. Consider the following example:

```
...  
S1: load r2, 34(r1)  
S2: add r5, r2, r3  
...
```

In the above code, the “correct” value of R2 will be available after the memory access stage in the instruction S1. Hence even with data forwarding a stall will need to be placed between S1 and S2, so that S2 fetches its operands only after the memory access for S1 has been made.

Pipeline Design Requirements

For a pipelined design, it is important that the overall meaning of the program remains unchanged, i.e., the program should produce the same results as it would produce on a non-pipelined machine. It is also preferred that the data and instruction memories are separate so that instructions may be fetched while the register values are being stored and/or loaded from data memory. There should be a single data path so as not to complicate the flow of instructions and maintain the order of program execution. There should be a three port register file so that if the register write and register read stages overlap, they can be performed in parallel, i.e., the two register operands may be read while the destination register may be written. The data should be latched in between each pipeline stage using temporary pipeline registers. Since the clock cycle depends on the slowest pipeline stage, the ALU operations must be able to complete quickly so that the cycle time is not increased for the rest of the pipeline.

Designing a pipelined implementation

In this section we will discuss the various steps involved in designing a pipeline. Broadly speaking they may be categorized into three parts:

1. Adapting the instructions to pipelined execution

The instruction set of a non-pipelined processor is generally different from that of a pipelined processor. The instructions in a pipelined processor should have clear and definite phases, e.g., **add r1, r2, r3**. To execute this instruction, the processor must first fetch it from memory, after which it would need to read the registers, after which the actual addition takes place followed by writing the results back to the destination register. Usually register-register architecture is adopted in the case of pipelined processors so that there are no complex instructions involving operands from both memory and registers. An instruction like **add r1, r2, a** would need to execute the memory access stage before the operands may be fed to the ALU. Such flexibility is not available in a pipelined architecture.

2. Designing the pipelined data path

Once a particular instruction set has been chosen, an appropriate data path needs to be designed for the processor. The data path is a specification of the steps that need to be followed to execute an instruction. Consider our two examples above

For the instruction **add r1, r2, r3**: *Instruction Fetch – Register Read – Execute – Register Write*,

Whereas for the instruction **add r1, r2, a** (remember **a** represents a memory address), we have 
Instruction Fetch – Register Read – Memory Access – Execute – Register Write,

The data path is defined in terms of registers placed in between these stages. It specifies how the data will flow through these registers during the execution of an instruction. The data path becomes more complex if forwarding or bypassing mechanism is added to the processor.

3. Generating control signals

Control signals are required to regulate and direct the flow of data and instruction bits through the data path. Digital logic is required to generate these control signals.

Lecture No. 19

Pipelined SRC

Reading Material

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

Chapter 5
5.1.3

Summary

- Pipelined Version of the SRC
- Adapting SRC instructions for Pipelined Execution
- Control Signals for Pipelined SRC

Pipelined Version of the SRC

In this lecture, a pipelined version of the SRC is presented. The SRC uses a five-stage pipeline. Those five stages are given below:

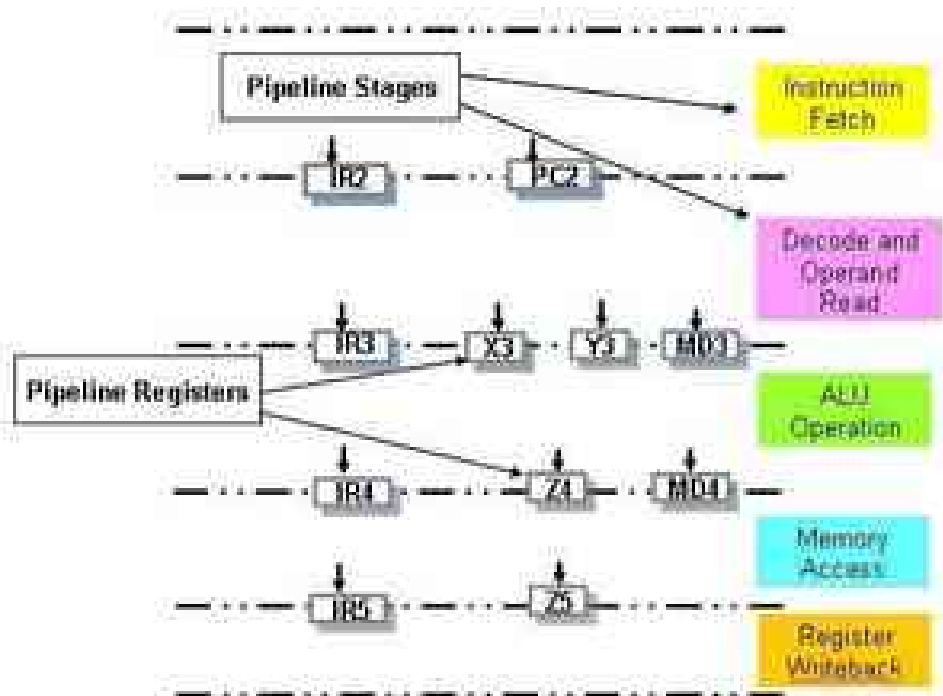
1. Instruction Fetch
2. Instruction decode/operand fetch
3. ALU operation
4. Memory access
5. Register write

As shown in the next diagram, there are several registers between each stage.

After the instruction has been fetched, it is stored in **IR2** and the incremented value of the program counter is held in **PC2**. When the register values have been read, the first register value is stored in **X3**, and the second register value is stored in **Y3**. **IR3** holds the opcode and ra. If it is a store to memory instruction, **MD3** holds the register value to be stored.

After the instruction has been executed in the ALU, the register **Z4** holds the result. The op-code and ra are passed on to **IR4**. During the write back stage, the register **Z5** holds the value to be stored back into the register, while the op-code and ra are passed into **IR5**. There are also two separate memories and several multiplexers involved in the pipeline operation. These will be shown at appropriate places in later figures.

The number after a particular register name indicates the stage where the value of this register is used.



Adapting SRC Instructions for Pipelined Execution

As mentioned earlier, the SRC instructions fall into the following three categories:

1. ALU Instructions
2. Load/Store instructions
3. Branch Instructions

We will now discuss how to design a common pipeline for all three categories of instructions.

1. ALU instructions

ALU instructions are usually of the form:

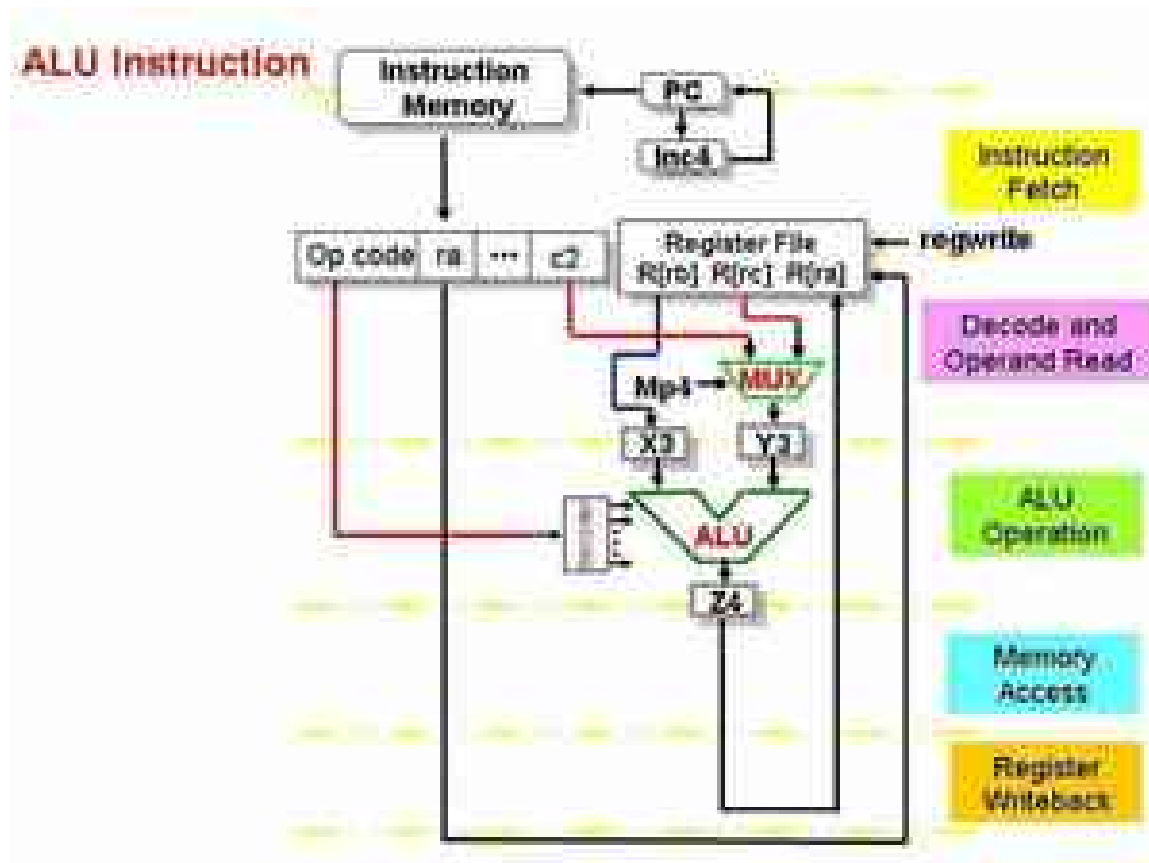
op-code ra, rb, rc

or

op-code ra, rb, constant.

In the diagram shown, X3 and Y3 are temporary registers to hold the values between pipeline stages. X3 is loaded with operand value from the register file. Y3 is loaded with either a register value from the register file or a constant from the instruction. The operands are then available to the ALU. The ALU function is determined by decoding the op-code bits. The result of the ALU operation is stored in register Z4, and then stored in the destination register in the register write back stage. There is no activity in the memory access stage for ALU instructions. Note that Z5, IR3, IR4, and IR5 are not shown explicitly in the figure. The purpose of not including these registers is to keep the drawing simple. However, these registers will transfer values as

instructions progress through the pipeline. This comment also applies to some other figures in this discussion.



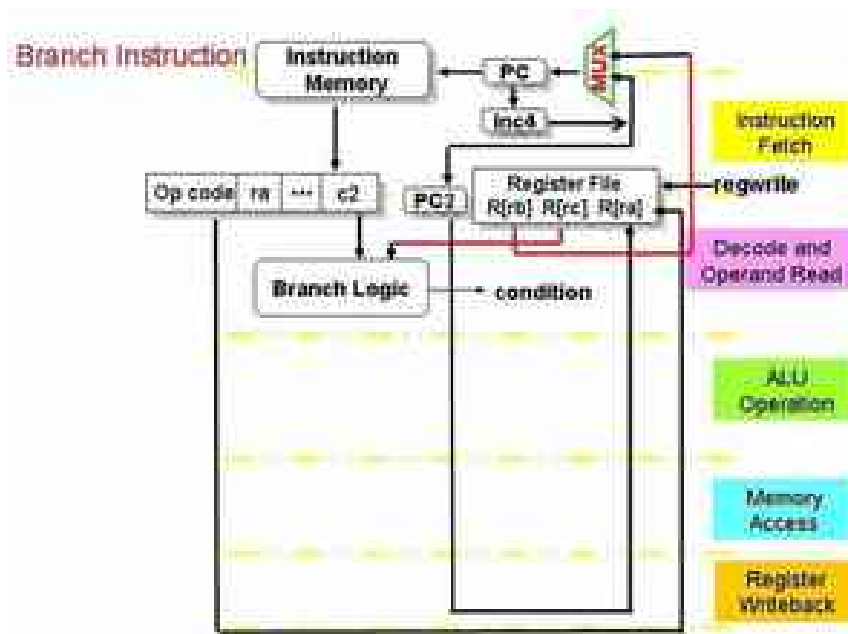
2. Load/Store instructions

Load/Store instructions are usually of the form:

op-code ra, constant(rb)

The instruction is loaded into **IR2** and the incremented value of the PC is loaded in **PC2**. In the next stage, **X3** is loaded with the value in **PC2** if the relative addressing mode is used, or the value in **rb** if the displacement addressing mode is used. Similarly, **C1** is transferred to **Y3** for the relative addressing mode, and **c2** is transferred to **Y3** for the displacement addressing mode. The store instruction is completed once memory access has been made and the memory location has been written to. The load instruction is completed once the loaded value is transferred back to the register file. The following figure shows the schematic for a load instruction. A similar schematic can be drawn for the store instruction.

Branch Instructions usually involve calculating the target address and evaluating a condition. The condition is evaluated based on the c2 field of the IR and by using the value in R[rc]. If the condition is true, the PC is loaded with the value in R[rb], otherwise it is incremented by 4 as usual. The following figure shows these details.



The pipelined data path implementation diagrams shown earlier for the three SRC instruction categories must be combined and refined to get a working system. These details get complicated very quickly. A detailed combined diagram is shown in Figure 5.7 of the text book.

Control Signals for the Pipelined SRC

We define the following signals for the SRC by grouping similar op-codes:

Control signals for pipeline stages

- `branch = br - brl`
- `cond = ((R2<2_0>=1) - (R2<2_1>=1) & (R2<0>=R[rc]=0)) =`
`((R2<2_1>=2) & (R2<0>=R[rc]=31*))`
- `sh = shr - shra - shl - snc`
- `alu = add - addi - sub - neg - and - andi - or - ori - nci - sh`
- `imm = add - andi - ori - (sh & (R<4_0>=0))`
- `load = ld - ldr`
- `ladr = la - lar`
- `store = st - str`
- `rl = load - ladr - store`
- `regwrite = load - ladr - brl - alu`
- `dsp = ld - st - la`
- `rt = ldr - str - lar`

In most cases, the signals defined above are used in the same stage where they are generated. If that is not the case, a number used after the signal name indicates the stage where the signal is generated.

Using these definitions, we can develop RTL statements for describing the pipeline activity as well as the equations for the multiplexer select signals for different stages of the pipeline. This is shown in the next diagram.

Control Signals for different pipeline Stages

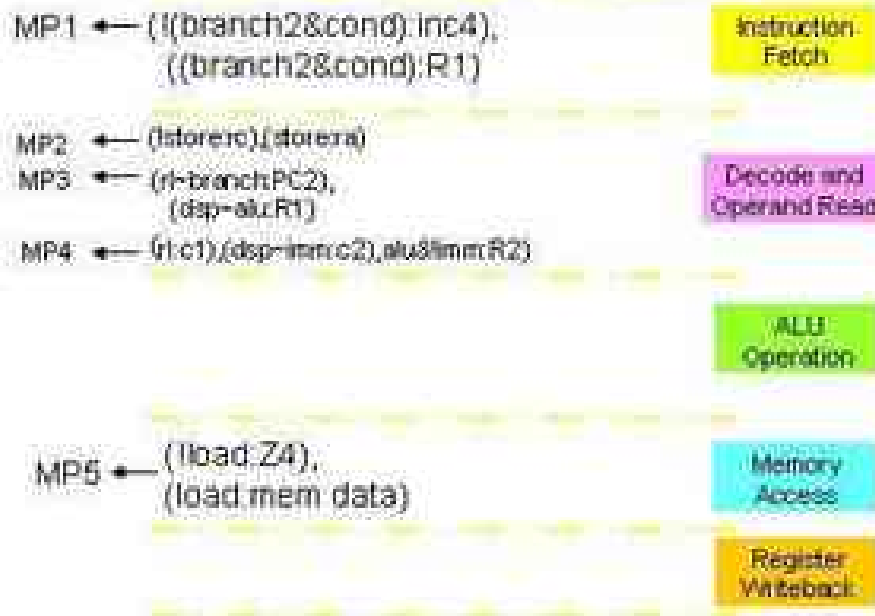
Consider the RTL description of the Mp1 signal, which controls the input to the PC. It simply means that if the branch and cond signals are not activated, then the PC is incremented by 4, otherwise if both are activated then the value of R1 is copied in to the PC.

The multiplexer Mp2 is used to decide which registers are read from the register file. If the store signal is activated then R[rb] from the instruction bits is read from the register file so that its value may be stored into memory, otherwise R[rc] is read from the register file.

The multiplexer Mp3 is used to decide which registers are read from the register file for operand 2. If either rl or branch is activated then the updated value of PC2 is transferred to X3, otherwise if dsp or alu is activated, the value of R[ra] from the register file is transferred to the x3. In the same way, multiplexer Mp4 is used to select an input from Y3.

In the same way, multiplexer Mp4 is used to select an input for Y3.

Control signals for pipeline stages



The multiplexer MP5 is used to decide which value is transferred to be written back to the register file. If the load signal is activated data from memory is transferred to Z5, however if the load signal is not activated then data from Z4 (which is the result of ALU) is transferred to Z5 which is then written back to the register file.

Lecture No. 20

Hazards in Pipelining

Reading Material

Vincent P. Heuring & Harry F. Jordan Computer Systems Design and Architecture

Chapter 5
5.1.5, 5.1.6

Summary

- Structural RTL for Pipeline Stages
- Instruction Propagation Through the Pipeline
- Pipeline Hazards
- Data Dependence Distance
- Data Forwarding
- Compiler Solution to Hazards
- SRC Hazard Detection and Correction
- RTL for Hazard Detection and Pipeline Stall

Structural RTL for Pipeline Stages

The Register Transfer Language for each phase is given as follows:

Instruction Fetch

$IR2 \leftarrow M[PC];$
 $PC2 \leftarrow PC+4;$

Instruction Decode & Operand fetch

$X3 \leftarrow I-s2:(rel2:PC2, disp2:(rb=0):?, (rb!=0):R[rb]), brl2:PC2, alu2:R[rb],$
 $Y3 \leftarrow I-s2:(rel2:c1, disp2:c2), alu2:(imm2:c2, !imm2:R[rc]),$
 $MD3 \leftarrow store2:R[ra], IR3 \leftarrow IR2, stop2:Run \leftarrow 0,$
 $PC \leftarrow !branch2:PC+4, branch2:(cond(IR2, R[rc]):R[rb], !cond(IR2, R[rc]):PC+4;$

ALU operation

$Z4 \leftarrow (I-s3: X3+Y3, brl3: X3, Alu3: X3 \text{ op } Y3,$
 $MD4 \leftarrow MD3,$
 $IR4 \leftarrow IR3;$

Memory access

$Z5 \leftarrow (load4: M[Z4], laddr4 \sim branch4 \sim alu4:Z4),$
 $store4: (M[Z4] \leftarrow MD4),$
 $IR5 \leftarrow IR4;$

Write back

regwrite5: (R[ra] $\leftarrow$ Z5);

Instruction Propagation through the Pipeline

Consider the following SRC code segment flowing through the pipeline. The instructions along with their addresses are

```
200:    add r1, r2, r3
204:    ld r5, [4(r7)]
208:    br r6
212:    str r4, 56
...
400
```

We shall review how this chunk of code is executed.

First Clock Cycle

Add instruction enters the pipeline in the first cycle. The value in PC is incremented from 200 to 204.

Second Clock Cycle

Add moves to decode stage. Its operands are fetched from the register file and moved to X3 and Y3 at the end of clock cycle, meanwhile the Instruction ld r5, [4+r7] is fetched in the first stage and the PC value is incremented from 204 to 208.

Third Clock Cycle

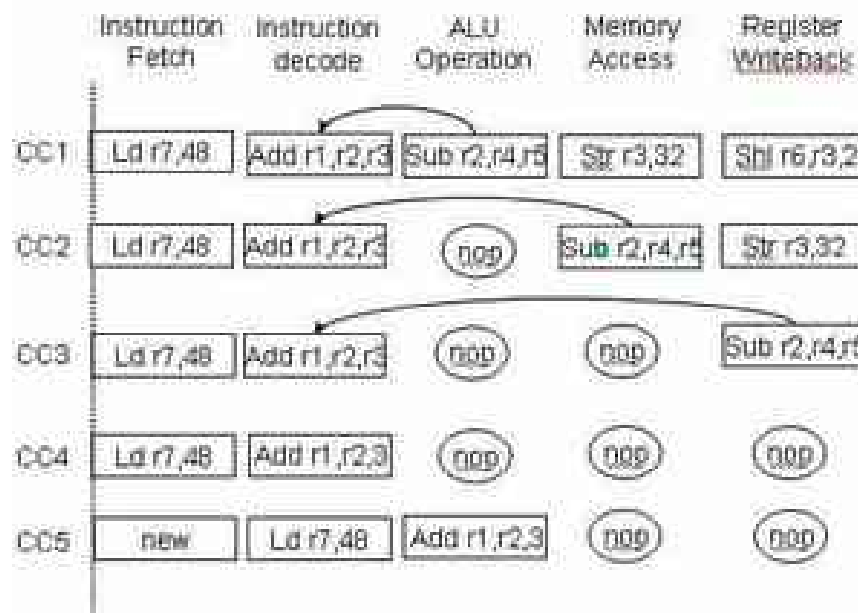
Add instruction moves to the execute stage, the results are written to Z4 on the trailing edge of the clock. Ld instruction moves to decode stage. The operands are fetched to calculate the displacement address. Br instruction enters the pipeline. The value in PC is incremented from 208 to 212.

Fourth Clock Cycle

Add does not access memory. The result is written to Z5 at the trailing edge of clock. The address is being calculated here for ld. The results are written to Z4. Br is in the decode stage. Since this branch is always true, the contents of PC are modified to new address. Str instruction enters the pipeline. The value in PC is incremented from 212 to 216.

Fifth Clock Cycle

The result of addition is written into register r1. Add instruction completes. Ld accesses data memory at the address specified in Z4 and result stored in Z5 at falling edge of clock. Br instruction just propagates through this stage without any calculation. Str is in the decode stage. The operands are being fetched for address calculation to X3 and Y3. The instruction at address 400 enters the pipeline. The value in PC is incremented from 400 to 404.



Pipeline Hazards

The instructions in the pipeline at any given time are being executed in parallel. This parallel execution leads to the problem of instruction dependence. A hazard occurs when an instruction depends on the result of previous instruction that is not yet complete.

Classification of Hazards

There are three categories of hazards

1. Branch Hazard
2. Structural Hazard
3. Data Hazard

Branch hazards

The instruction following a branch is always executed whether or not the branch is taken. This is called the branch delay slot. The compiler might issue a nop instruction in the branch delay slot. Branch delays cannot be avoided by forwarding schemes.

Structural hazards

A structural hazard occurs when attempting to access the same resource in different ways at the same time. It occurs when the hardware is not enough to implement pipelining properly e.g. when the machine does not support separate data and instruction memories.

Data hazards

Data hazard occur when an instruction attempts to access some data value that has not yet been updated by the previous instruction. An example of this RAW (read after write) data hazard is;

```
200:    add r2, r3, r4
204:    sub r7, r2, r6
```

The register r2 is written in clock cycle 5 hence the sub instruction cannot proceed beyond stage 2 until the add instruction leaves the pipeline.

Data Hazard Detection & Correction

Data hazards can be detected easily as they occur when the destination register of an instruction is the same as the source register of another instruction in close proximity. To remedy this situation, dependent instructions may be delayed or stalled until the ones ahead complete. Data can also be forwarded to the next instruction before the current instruction completes, however this requires forwarding hardware and logic. Data can be forwarded to the next instruction from the stage where it is available without waiting for the completion of the instruction. Data is normally required at stage 2 (operand fetch) however data is earliest available at stage 3 output (ALU result) or stage 4 output (memory access result). Hence the forwarding logic should be able to transfer data from stage 3 to stage 2 or from stage 4 to stage 2.

Data Dependence Distance

Designing a data forwarding unit requires the study of dependence distances. Without forwarding, the minimum spacing required between two data dependent instructions to avoid hazard is four. The load instruction has a minimum distance of two from all other instructions except branch. Branch delays cannot be removed even with forwarding.

Table 5.1 of the text shows numbers related to dependence distances with respect to some important instruction categories.

Compiler Solution to Hazards

Hazards can be detected by the compiler, by analyzing the instruction sequences and dependencies. The compiler can insert bubbles (nop instruction) between two instructions that form a hazard, or it could reorder instructions so as to put sufficient distance between dependent instructions. The compiler solution to hazards is complex, expensive and not very efficient as compared to the hardware solution.

SRC Hazard Detection and Correction

The SRC uses a hazard detection unit. The hazard can be resolved using either pipeline stalls or by data forwarding.

Pipeline stalls

Consider the following sequence of instructions going through the SRC pipeline

```
200:    shl r6, r3, 2
204:    str r3, 32
208:    sub r2, r4,r5
212:    add r1,r2,r3
216:    ld r7, 48
```

There is a data hazard between instruction three and four that can be resolved by using pipeline stalls or bubbles

When using pipeline stalls, nop instructions are placed in between dependent instructions. The logic behind this scheme is that if opcode in stage 2 and 3 are both alu, and if ra in stage 3 is the same as rb or rc in stage 2, then a pause signal is issued to insert a bubble between stage 3 and 2. Similar logic is used for detecting hazards between stage 2 and 4 and stage 4 and 5.

Data Forwarding

By adding data forwarding mechanism to the SRC data path, the stalls can be completely eliminated at least for the ALU instructions. The hazard detection is required between stages 3 and 4, and between stages 3 and 5. The testing and forwarding circuits employ wider IRs to store the data required in later stages. The logic behind this method is that if the ALU is activated for both 3 and 5 and ra in 5 is the same as rb in 3 then Z5 which hold the currently loaded or calculated result is directly forwarded to X3. Similarly, if both are ALU operations and instruction in stage 3 does not employ immediate operands then value of Z5 is transferred to Y3. Similar logic is used to forward data between stage 3 and 4.

RTL for Hazard Detection and Pipeline Stall

The following RTL expression detects data hazard between stage 2 and 3, then stalls stage 1 and 2 by inserting a bubble in stage 3

$$\text{alu3\&alu2\&((ra3=rb2)\sim((ra3=rc2)\&!imm2))}:$$
$$(\text{pause2}, \text{pause1}, \text{op3} \leftarrow 0)$$

Meaning:

If opcode in stage 2 and 3 are both ALU, and if ra in stage 3 is same as rb or rc in stage 2, issue a pause signal to insert a bubble between stage 3 and 2.

Following is the complete RTL for detecting hazards among ALU instructions in different stages of the pipeline

Data Hazard (between)

RTL for detection and stalling

Stage 2 and 3

$$\text{alu3\&alu2\&((ra3=rb2)\sim((ra3=rc2)\&!imm2))}:$$
$$(\text{pause2}, \text{pause1}, \text{op3} \leftarrow 0)$$

Stage 2 and 4

$$\text{alu4\&alu2\&((ra4=rb2)\sim((ra4=rc2)\&!imm2))}:$$
$$(\text{pause2}, \text{pause1}, \text{op3} \leftarrow 0)$$

Stage 2 and 5

$$\text{alu5\&alu2\&((ra5=rb2)\sim((ra5=rc2)\&!imm2))}:$$
$$(\text{pause2}, \text{pause1}, \text{op3} \leftarrow 0)$$

Lecture No. 21

Instruction Level Parallelism

Reading Material

Vincent P. Heuring & Harry F. Jordan Computer Systems Design and Architecture

Chapter 5
5.2

Summary

- Data Forwarding Hardware
- Instruction Level Parallelism
- Difference between Pipelining and Instruction-Level Parallelism
- Superscalar Architecture
- Superscalar Design
- VLIW Architecture

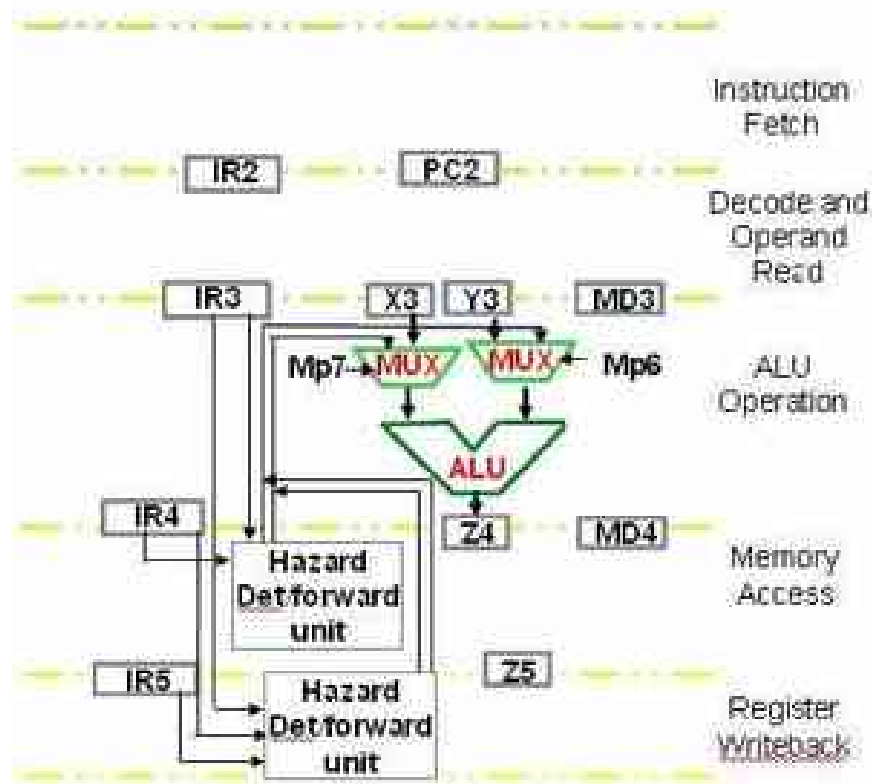
Maximum Distance between two instructions

Example

Read page no. 219 of Computer System Design and Architecture (Vincent P.Heuring, Harry F. Jordan)

Data forwarding Hardware

The concept of data forwarding was introduced in the previous lecture.



RTL for data forwarding in case of ALU instructions

Dependence RTL

Stage 3-5	alu5&alu3:((ra5=rb3):X←Z5,
	(ra5=rc3)&!imm3: Y ← Z5);
Stage 3-4	alu4&alu3:((ra4=rb3):X←Z4,
	(ra4=rc3)&!imm3: Y ← Z4);

Instruction-Level Parallelism

Increasing a processor's throughput

There are two ways to increase the number of instructions executed in a given time by a processor

- By increasing the clock speed
- By increasing the number of instructions that can execute in parallel

Increasing the clock speed

- Increasing the clock speed is an IC design issue and depends on the advancements in chip technology.
- The computer architect or logic designer can not thus manipulate clock speeds to increase the throughput of the processor.

Increasing parallel execution of instructions

The computer architect cannot increase the clock speed of a microprocessor however he/she can increase the number of instructions processed per unit time. In pipelining we discussed that a number of instructions are executed in a staggered fashion, i.e. various instructions are simultaneously executing in different segments of the pipeline. Taking this concept a step further we have multiple data paths hence multiple pipelines can execute simultaneously. There are two main categories of these kinds of parallel instruction processors VLIW (very long instruction word) and superscalar.

The two approaches to achieve instruction-level parallelism are

- **Superscalar Architecture**
A scalar processor that can issue multiple instructions simultaneously is said to be superscalar
- **VLIW Architecture**
A VLIW processor is based on a very long instruction word. VLIW relies on instruction scheduling by the compiler. The compiler forms instruction packets which can run in parallel without dependencies.

Difference between Pipelining and Instruction-Level Parallelism

Pipelining

Single functional unit

Instructions are issued sequentially

Throughput increased by overlapping the instruction execution

Very little extra hardware required to implement pipelining

Instruction-Level Parallelism

Multiple functional units

Instructions are issued in parallel

Instructions are not overlapped but executed in parallel in multiple functional units

Multiple functional units within the CPU are required

Superscalar Architecture

A superscalar machine has following typical features

- It has one or more IUs (integer units) , FPU (floating point units), and BPU (branch prediction units)
- It divides instructions into three classes
 - Integer
 - Floating point
 - Branch prediction

The general operation of a superscalar processor is as follows

- Fetch multiple instructions
- Decode some of their portion to determine the class
- Dispatch them to the corresponding functional unit

As stated earlier the superscalar design uses multiple pipelines to implement instruction level parallelism.

Operation of Branch Prediction Unit

- BPU calculates the branch target address ahead of time to save CPU cycles
- Branch instructions are routed from the queue to the BPU where target address is calculated and supplied when required without any stalls
- BPU also starts executing branch instructions by speculating and discards the results if the prediction turns out to be wrong

Superscalar Design

The philosophy behind a superscalar design is

- to prefetch and decode as many instructions as possible before execution
- and to start several branch instruction streams speculatively on the basis of this decoding
- and finally, discarding all but the correct stream of execution

The superscalar architecture uses multiple instruction issues and uses techniques such as branch prediction and speculative instruction execution, i.e. it speculates on whether a particular branch will be taken or not and then continues to execute it and the following instructions. The results are not written back to the registers until the branch decision is confirmed. Most superscalar architectures contain a reorder buffer. The reorder buffer acts like an intermediary between the processor and the register file. All results are written onto the reorder buffer and when the speculated course of action is confirmed, the reorder buffer is committed to the register file.

Superscalar Processors

Examples of superscalar processors

- PowerPC 601
- Intel P6
- DEC Alpha 21164

VLIW Architecture

VLIW stands for “Very Long Instruction Word” typically 64 or 128 bits wide. The longer instruction word carries information to route data to register files and execution units. The execution-order decisions are made at the compile time unlike the superscalar design where decisions are made at run time. Branch instructions are not handled very efficiently in this architecture. VLIW compiler makes use of techniques such as loop unrolling and code reordering to minimize dependencies and the occurrence of branch instructions.

Lecture No. 22**Microprogramming****Reading Material**

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

Chapter 5
5.3

Summary

- Microprogramming
- Working of a General Microcoded Controller
- Microprogram Memory
- Generating Microcode for Some Sample Instructions
- Horizontal and Vertical Microcode Schemes
- Microcoded 1-bus SRC Design
- The SRC Microcontroller

Microprogramming

In the previous lectures, we have discussed how to implement logic circuitry for a control unit based on logic gates. Such an implementation is called a hardwired control unit. In a microprogrammed control unit, control signals which need to be generated at a certain time are stored together in a control word. This control word is called a microinstruction. A collection of microinstructions is called a microprogram. These microprograms generate the sequence of necessary control signals required to process an instruction. These microprograms are stored in a memory called the control store.

As described above microprogramming or microcoding is an alternative way to design the control unit. The microcoded control unit is itself a small stored program computer consisting of

- Micro-PC
 - Microprogram memory
 - Microinstruction word

Comparison of hardwired and microcoded control unit

Hardwired Control Unit	Microcoded Control Unit
The relationship between control inputs and control outputs is a series of Boolean functions.	The control signals here are stored as words in a microcode memory.
Hardwired control units are generally faster.	Microcode units simplify the computer logic but it is comparatively slower.

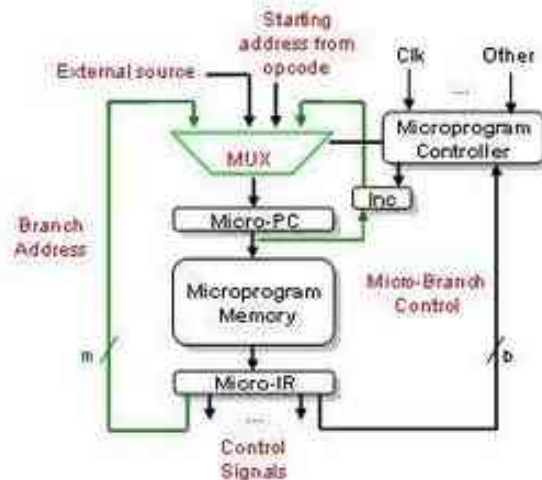
Working of a general microcoded controller

A microcoded controller works in the same way as a small general purpose computer.

1. Fetch a micro-instruction and increment micro-PC.
2. Execute the instruction present in micro-IR.
3. Fetch the next instruction and so on...

The microcoded control unit is like a small computer in itself. It consists of a microprogram memory, which is read using a micro program counter. The micro PC is controlled by the microprogram controller. Values of the micro PC depends on a 4 to 1 MUX. The source may be the incremented micro PC value, or a calculated branch value, or a value derived by decoding an opcode for an instruction. The microprogram memory writes the control word at the chosen address into the micro

instruction register. This control word is basically the set of all the control signals needed to execute the instruction at that particular instant.



Fields in the micro instruction

C Bits

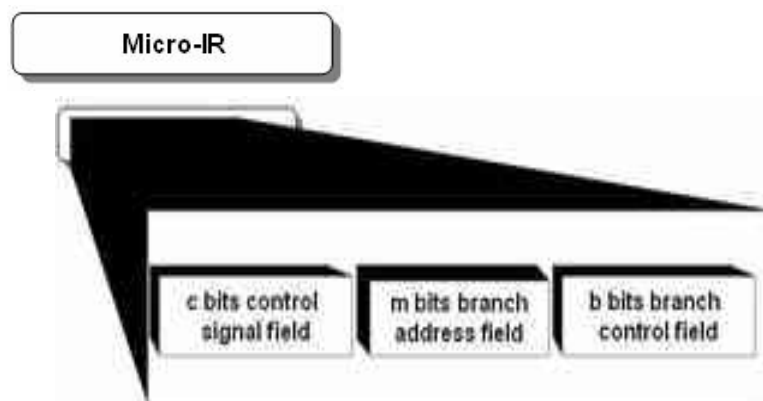
These form the control signal field

M Bits

These form the branch address field

B Bits

These form the branch control field.



Loading the micro-PC

The micro-PC can be loaded from one of the four possible sources

- **Simple increment** Steps sequentially from microinstruction to microinstruction
- **Lookup table** A lookup table maps the opcode field to the starting address of the microcode routine that generates control signals.
- **External source** Initializes micro-PC to begin an operation e.g. interrupts service, reset etc.
- **Branch addresses** Jumps anywhere in the microprogram memory on the basis of conditional or unconditional branch.

Microprogram Memory

- This small memory contains micro routines for all the instructions in the ISA
- The micro-PC supplies the address and it returns the control word stored at that address
- It is much faster and smaller than a typical main memory

Layout of a typical microprogram memory

Micro-Address	Memory Contents
0	Microcode for instruction fetch
...	Microcode for load instruction
...	Microcode for add instruction
...	Microcode for br instruction
2^n-1	Microcode for reset instruction

Generating Microcode for Some Sample Instructions

- The control word for an instruction is used to generate the equivalent microcode sequence
- Each step in RTL corresponds to a microinstruction executed to generate the control signals.

Each bit in the control words in the microprogram memory represents a control signal. The value of that bit decides whether the signal is to be activated or not.

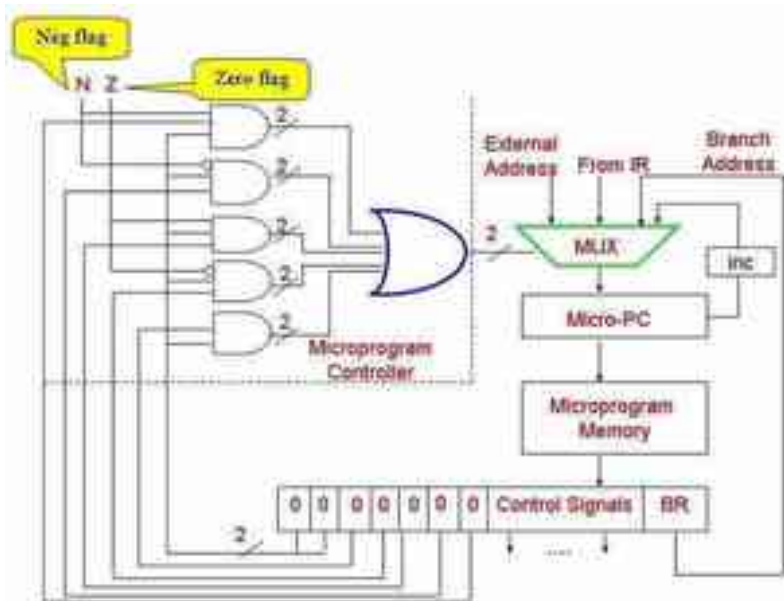
Example: Control Signals for the sub Instruction

The first three addresses from 100 to 102 represent microcode for instruction fetch and the last three addresses from 203 to 205 represent microcode for sub instruction. In the first cycle at address 100, the control signal PCout, LMAR, LC, and INC4 are activated and all other signals are deactivated. All these control signals are for the SRC processor. So, if the micro-PC contains 100, the contents of microprogram memory are copied into the micro IR. This corresponds to the structural RTL description of the T0 clock during instruction fetch phase. In the same way, the content of address 101 corresponds to T1, and the content of address 102 corresponds to T2.

Address	Branch Control	PC-out	Out	Cond	R2Bus	LMAR	LC	LPC	LR	LA	BoCR	INCA	Read	LMBR	MarOut	SUB	RAE	RBE	RCE	END
100	---	1	0	0	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0
101	---	0	1	0	0	0	0	1	0	0	0	0	1	1	1	0	0	0	0	0
102	---	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
203	---	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0
204	---	0	0	0	1	0	1	0	0	0	0	0	0	0	0	1	0	0	1	0
205	---	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1

Microprogram Controller functions: Branching and looping

- **Microprogram** controller controls the sequence of the flow of microinstructions.
- The inputs to the microcontroller are from the branch control fields specified in the microcode word.
- Its output controls the 4 to 1 multiplexer inside the microcoded control unit.
- It implements conditional execution and both conditional and unconditional branch



If a branch instruction is encountered within the microprogram hardwired logic selects the branch address as the source of micro-PC using 4 to 1 mux. This hardwired logic caters for all branch instructions including branch if zero.

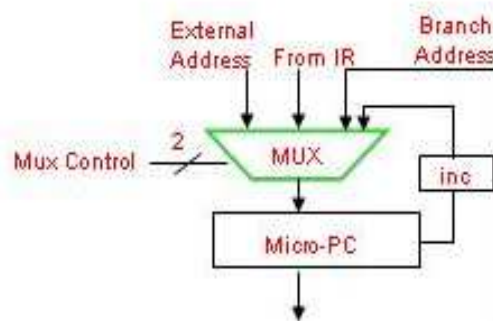
4-1 Multiplexer

The multiplexer supplies one of the four possible values to the micro-PC
The incremented value of the micro-PC is used when dealing with the normal flow of microinstructions.

Advance Computer Architecture – CS501

The opcode from the instruction is used to set the micro-PC when a microroutine is initially being loaded.

Mux Control	Select
00	Increment micro-PC
01	Opcode from IR
10	External address
11	Branch address



External address is used when it is required to reset the microprogram controller.
Branch address is set into the micro-PC when a branch microinstruction is encountered.

How to form a branch

- A branch can be implemented by choosing one alternative from each of the following two lists.
- This scheme provides flexibility in choosing branches as we can form any combination of conditions and addresses.

Condition
unconditional
not zero
zero
positive
negative

Address
From IR
External Address
Branch Address

• Microcode Branching Examples

Following is an example of branch instructions in microcode

Address	Max. Control	Branch	brmf	brz	brp	brn	Control Signals	Branch Address	Branching Action	Equivalent Construct
400	00	0	0	0	0	0	...	xxx	No branch goto next address in sequence-401	
401	01	1	0	0	0	0	...	xxx	To the address supplied by opcode	goto initial address: [...]
402	10	0	0	1	0	0	...	xxx	To external address if Z flag is set	goto Ext. Add. [...]; if Z then
403	11	0	0	0	0	1	...	200	To 200 if N flag is set, else to 404	goto Label [...]; if N then
404	11	0	0	0	1	0	000	406	To 406 if N is false, else to 405	While (N) [...]
405	11	1	0	0	0	0	...	404	Branch to 404	While count...

Similarity between microcode and high level programs

- Any high level construct such as if-else, while, repeat etc. can be implemented using microcode
- A variety of microcode compilers similar to the high level compilers are available that allow easier programming in microcode
- This similarity between high level language and microcode simplifies the task of controller design.

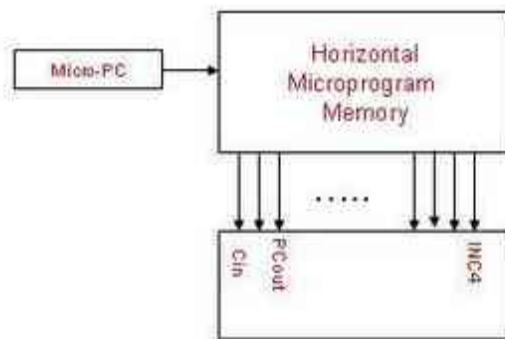
Horizontal and vertical microcode schemes

In horizontal microcode schemes, there are no intermediate decoders and the control word bits are directly connected to their destination i.e. each bit in the control word is directly connected to some control signal and the total number of bits in the control word is equal to the total number of control signals in the CPU.

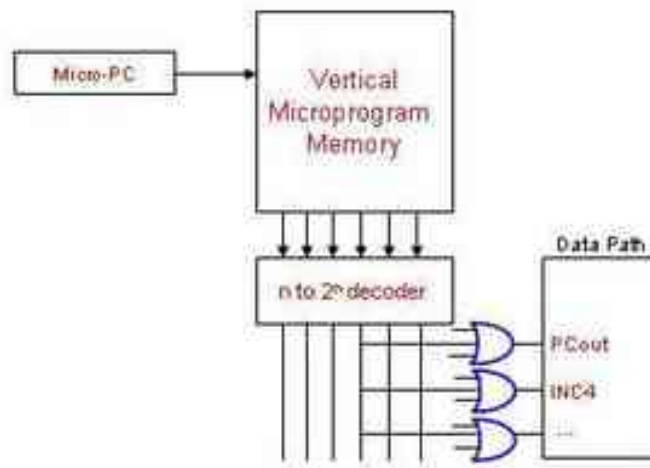
Vertical microcode schemes employ an extra level of decoding to reduce the control word width. From an n bit control word we may have 2^n bit signal values.

However, a completely vertical scheme is not feasible because of the high degree of fan out.

Horizontal Microcode Scheme



Vertical Microcode Scheme

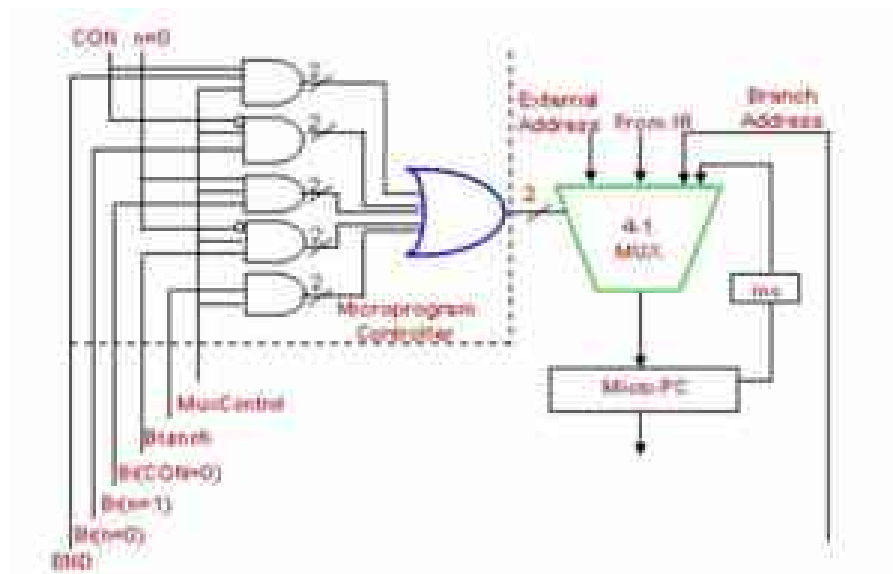
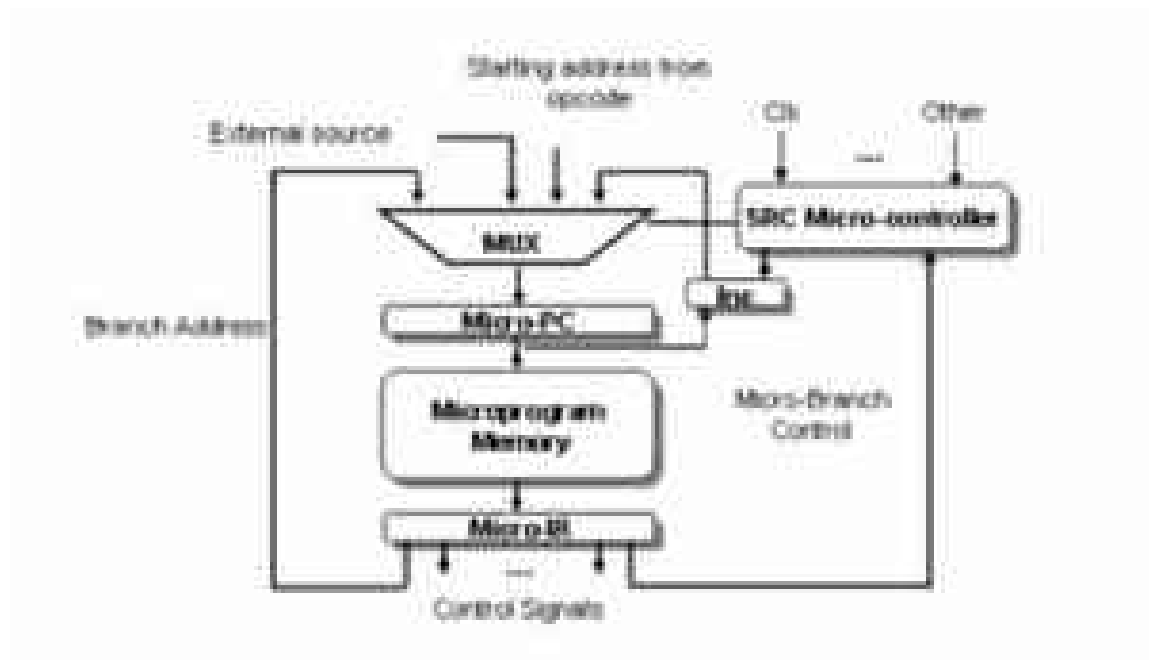


Microcoded 1-bus SRC design

In the SRC the bits from the opcode in the instruction register are decoded to fetch the address of the suitable microroutine from the microprogram memory. The microprogram controller for the SRC microcoded control unit employs the logic for handling exceptions and reset process. Since the SRC does not have any condition codes, we use the CON and n signals instead of N and Z flags to control branches in case of branch if equal to zero or branch if less than instructions.

The SRC Microprogram Controller

- The microprogram controller for the SRC microcoded control unit employs the logic for handling exceptions and reset process
- Since the SRC does not have any condition codes, we use the CON and n signals instead of N and Z flags to control branches



Microcode for some SRC instructions

Address	MicroControl	Branch	Bit CDS(0)	Bit m(1)	Bit m(0)	Find	PCout	LMAR	Control Signals	Branch Address	RTL
300	00	0	0	0	0	0	1	1	---	xxx	$MAR \leftarrow PC; C \leftarrow PC + 4;$
301	00	0	0	0	0	0	0	0	---	xxx	$MBR \leftarrow M[MAR]; PC \leftarrow C;$
302	01	1	0	0	0	0	0	0	---	xxx	$IR_{Micro} \leftarrow PC \leftarrow MBR(31..27);$
400	00	0	0	0	0	0	0	0	---	xxx	$A \leftarrow R[rb];$
401	00	0	0	0	0	0	0	0	---	xxx	$C \leftarrow A + R[re];$
402	11	1	0	0	0	1	0	0	---	100	$R[re] \leftarrow C; Micro-PC \leftarrow 400;$

Assume the first control word at address 300. The RTL of this instruction is MAR PC combined with C PC+4. To facilitate these actions the PCout signal bit and the LMAR signal bit are set to one, so that the value of the PC may be written to the internal processor bus and written onto the MAR. The instructions at 300, 301 and 302 form the microcode for instructions fetch. If we examine the RTL we can see all the functionality of the fetch instruction. The value of PC is incremented, the old value of PC is sent to memory, the instruction from the sent address is loaded into memory buffer register. Then the opcode of the fetched instruction is used to invoke the appropriate microroutine.

Alternative approaches to microcoding

- Bit ORing
- Nanocoding
- Writable Microprogram Memory
- Subroutines in Microprogramming

Lecture No. 23

I/O Subsystems

Reading Material

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

Chapter 8
8.1, 8.2

Summary

- Introduction to I/O Subsystems
- Major Components of an I/O Subsystems
- Computer Interface
- Memory Mapped I/O versus Isolated I/O
- Considerations during I/O Subsystem Design
- Serial and Parallel Transfers
- I/O Buses

Introduction to I/O Subsystems

This module is about the computer's input and output. As we have seen in the case of memory subsystems, that when we use the terms “read” and “write”, then these terms are from the CPU's point of view. Similarly, when we use the terms “input” and “output” then these are also from the CPU's point of view. It means that when we are talking about an input cycle, then the CPU is receiving data from a peripheral device and the peripheral device is providing data. Similarly, when we talk about an output cycle then the CPU is sending data to a peripheral device and the peripheral device is receiving data. I/O Subsystems are similar to memory subsystems in many aspects. For example, both exchange bits or bytes. This transfer is usually controlled by the CPU. The CPU sends address information to the memory and the I/O subsystems. Then these subsystems decode the address and decide which device should be involved in the transfer. Finally the appropriate data is exchanged between the CPU and the memory or the I/O device.

Memory and I/O subsystems differ in the following ways:

1. Wider range of data transfer speed:

I/O devices can be very slow such as a keyboard in which case the interval between two successive bytes (or keystrokes) can be in seconds. On the other extreme, I/O devices can be very fast such as a disk drive sending data to the CPU or a stream of packets arriving over a network, in which case the interval between two successive bytes can be in microseconds or even nanoseconds. While I/O devices can have such a wide range of data transfer speed compared to the CPU's speed, the case of memory devices is not so. Even if a memory device is slow compared to the CPU, the CPU's speed can be made compatible by inserting wait states in the bus cycle.

2. Asynchronous activity:

Memory subsystems are almost always synchronous. This means that most memory transfers are governed by the CPU's clock. Generally this is not the case with I/O subsystems. Additional signals, called handshaking signals, are needed to take care of asynchronous I/O transfers.

3. Larger degradation in data quality:

Data transferred by I/O subsystems can carry more noise. As an example, telephone line noise can become part of the data transferred by a modem. Errors caused by media defects on hard drives can corrupt the data. This implies that effective error detection and correction techniques must be used with I/O subsystems.

4. Mechanical nature of many I/O devices:

Many I/O devices or a large portion of I/O devices use mechanical parts which inherently have a high failure rate. In case an I/O device fails, interruptions in data transfer will occur, reducing the throughput. As an example, if a printer runs out of paper, then additional bytes cannot be sent to it. The CPU's data should be buffered (or kept in a temporary place) till the paper is supplied to the printer, otherwise the CPU will not be able to do anything else during this time.

To deal with these differences, special software programs called device drivers are made a part of the operating system. In most cases, device drivers are written in assembly language.

You would recall that in case of memory subsystems, each location uses a unique address from the CPU's address space. This is generally not the case with I/O devices. In most cases, a group or block of contiguous addresses is assigned to an I/O device, and data is exchanged byte-by-byte. Internal buffers (memory) within the device store this data if needed.

In the past, people have paid a lot of attention to improve the CPU's performance, as a result of which the performance improvement of I/O subsystems was ignored. (I/O subsystems were even called the "orphans" of computer architecture by some people). Perhaps, many benchmark programs and metrics that were developed to evaluate computer systems focused on the CPU or the memory performance only. Performance of I/O subsystems is as important as that of the CPU or the memory, especially in today's world. For example, the transaction processing systems used in airline reservation systems or the automated teller machines in banks have a very heavy I/O traffic, requiring improved I/O performance. To illustrate this point, look at the following example.

Suppose that a certain program takes 200 seconds of elapsed time to execute. Out of these 200 seconds, 180 seconds is the CPU time and the rest is I/O time. If the CPU performance improves by 40% every year for the next seven years because of developments in technology, but the I/O performance stays the same, let us look at the following table, which shows the situation at the end of each year. Remember that Elapsed time = CPU time + I/O time. 

This gives us the I/O time = $200 - 180 = 20$ seconds at the beginning, which is 10 % of the elapsed time.

Year #	CPU Time	I/O Time	Elapsed Time	<u>I/O Time x100 %</u>
0	180	20	200	10 %
1	129	20	149	13.42 %
2	92	20	112	17.85 %
3	66	20	86	23.25 %
4	47	20	67	29.85 %
5	34	20	54	37.03 %
6	24	20	44	45.45 %
7	17	20	37	54.05 %

It can be easily seen that over seven years, the I/O time will become more than 50 % of the total time under these conditions. Therefore, the improvement of I/O performance is as important as the improvement of CPU performance. I/O performance will also be discussed in detail in a later section.

Major components of an I/O subsystem

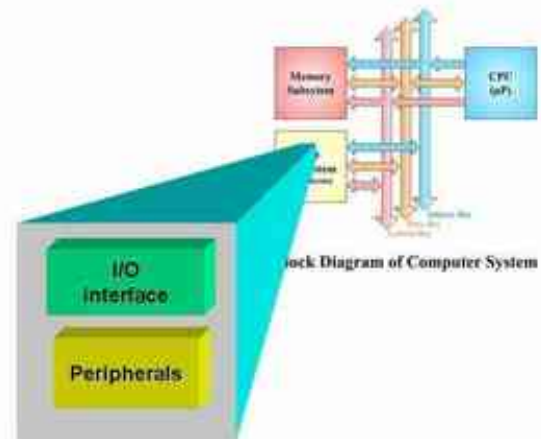
I/O subsystems have two major parts:

- The I/O interface, which is the electronic circuitry that connects the CPU to the I/O device.
- Peripherals, which are the devices used to communicate with the CPU, for example, the keyboard, the monitor, etc.

Computer Interface

A Computer Interface is a piece of hardware whose primary purpose is to connect together any of the following types of computer elements in such a way that the signal levels and the timing requirements of the elements are matched by the interface. Those elements are:

- The processor unit
- The memory subsystem(s)
- Peripheral (or I/O) devices
- The buses (also called "links")

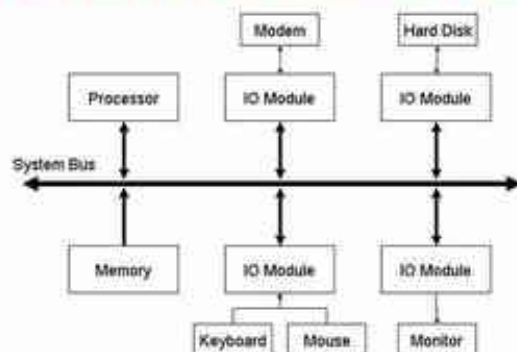


In other words, an interface is an electronic circuit that matches the requirements of the two subsystems between which it is connected. An interface that can be used to connect the microcomputer bus to peripheral devices is called an I/O Port. I/O ports serve the following three purposes:

- Buffering (i.e., holding temporarily) the data to and from the computer bus.
- Holding control information that dictates how a transfer is to be conducted.
- Holding status information so that the processor can monitor the activity of the interface and its associated I/O element.

This control information is usually provided by the CPU and is used to tell the device how to perform the transfer, e.g., if the CPU wants to tell a printer to start a new page, one of the control signals from the CPU can be used for a paper advance command, thereby telling the printer to start printing from the top of the next page. In the same way the CPU may send a control signal to a tape drive connected in the system asking it to activate the rewind mechanism so that the start of the tape is positioned for access by the CPU. Status information from various devices helps the CPU to know what is going on in the system. Once again, using the printer as an example, if the printer runs out of paper, this information should be

I/O Subsystem Block Diagram



sent to the CPU immediately. In the same way, if a hard drive in the system crashes, or if a sector is damaged and cannot be read, this information should also be conveyed to the CPU as soon as possible

The term “buffer” used in the above discussion also needs to be understood. In most cases, the word buffer refers to I/O registers in an interface where data, status or control information is temporarily stored. A block of memory locations within the main memory or within the peripheral devices is also called a buffer if it is used for temporary storage. Special circuits used in the interfaces for voltage/current matching, at the input and the output, are also called buffers.

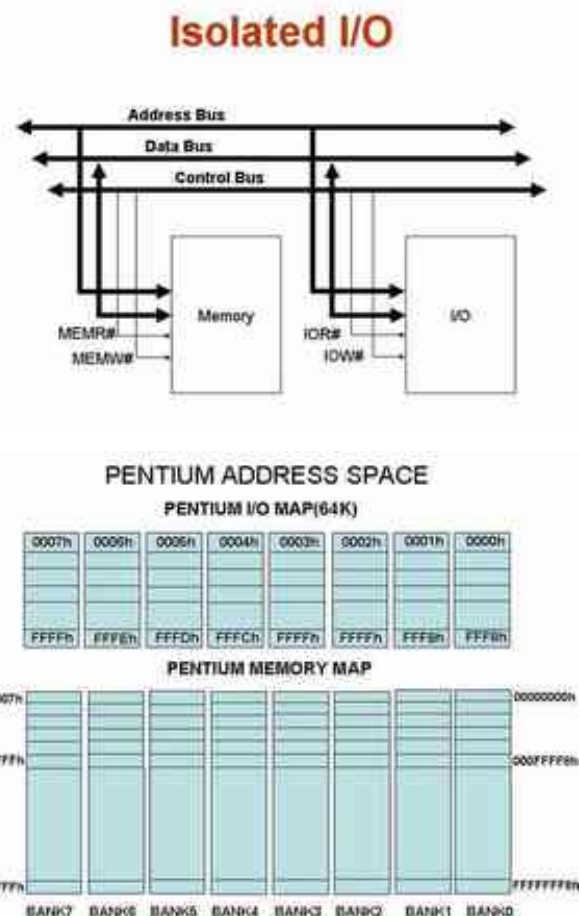
The given figure shows a block diagram of a typical I/O subsystem connected with the other components in a computer. The thick horizontal line is the system bus that serves as a back-bone in the entire computer system. It is used to connect the memory subsystems as well as the I/O subsystems together. The CPU also connects to this bus through a “bus interface unit”, which is not shown in this figure. Four I/O modules are shown in the figure. One module is used to connect a keyboard and a mouse to the system bus. A second module connects a monitor to the system bus. Another module is used with a hard disk and a fourth I/O module is used by a modem. All these modules are examples of I/O ports. A somewhat detailed view of these modules is shown in the next figure.

As we already know that the system bus actually consists of three buses, namely the address bus, the data bus and the control bus. These three buses are being applied to the I/O module in this figure. At the bottom, we see a set of data, status and control lines from each “device interface logic” block. Each of these sets connects to a peripheral device. I/O decoding logic is also shown in this figure.

Memory Mapped I/O versus Isolated I/O

Although this concept was explained earlier as well, it will be useful to review it again in this context. In isolated I/O, a separate address space of the CPU is reserved for I/O operations. This address space is totally different from the address space used for memory devices. In other words, a CPU has two distinct address spaces, one for memory and one for input/output. Unique CPU instructions are associated with the I/O space, which means that if those instructions are executing on the CPU, then the accessed address space will be the I/O space and hence the devices mapped on the I/O space.

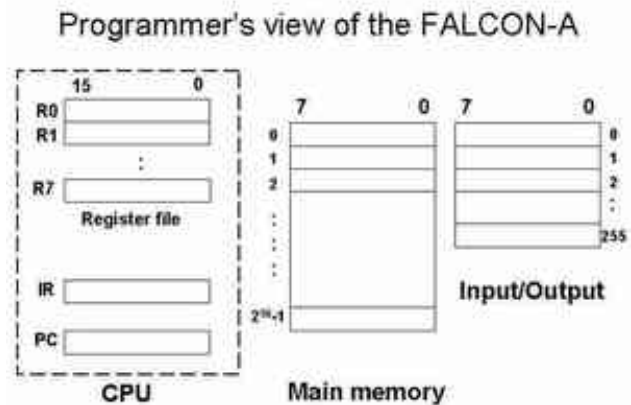
The x86 family with the **in** and the **out** instructions is a well-known example of this situation. Using the **in** instruction, the Pentium processor can receive information from a peripheral device, and using the **out** instruction, the Pentium processor can send information to a peripheral device. Thus, the I/O devices are mapped on the I/O space in case of the Pentium processor. In some processors, like the SRC, there is no separate I/O space. In this case, some address space out of the memory address space must be used to



map I/O devices. The benefit will be that all the instructions which access memory can be used for I/O devices. There is no need for including separate I/O instructions in the ISA of the processor. However, the disadvantage will be that the I/O interface will become complex. If partial decoding is used to reduce the complexity of the I/O interface, then a lot of memory addresses will be consumed. The given figure shows the memory address space as well as the I/O address space for the Pentium processor. The I/O space is of size 64 Kbytes, organized as eight banks of 8 Kbytes each.

A similar diagram for the FALCON-A was shown earlier and is repeated here for easy reference.

The next question to be answered is how the CPU will differentiate between these two address spaces. How will the system components know whether a particular transfer is meant for memory or an I/O device? The answer is simple: by using signals from the control bus, the CPU will indicate which address space is meant during a particular transfer. Once again, using the Pentium as an example, if the **in** instruction is executing on the processor, the IOR# signal will become active and the MEMR# signal will be deactivated. For a **mov** instruction, the control logic will activate the MEMR# signal instead of the IOR# signal.



Considerations during I/O Subsystem Design

Certain things must be taken care of during the design of an I/O subsystem.

Data location:

The designer must identify the device where the data to be accessed is available, the address of this device and how to collect the data from this device. For example, if a database needs to be searched for a record that is stored in the fourth sector of the second track of the third platter on a certain hard drive in the system, then this information is related to data location. The particular hard drive must be selected out of the possibly many hard drives in the system, and the address of this record in terms of platter number, track number and sector number must be given to this hard drive.

Data transfer:

This includes the direction of transfer of data; whether it is out of the CPU or into the CPU, whether the data is being sent to the monitor or the hard drive, or whether it is being received from the keyboard or the mouse. It also includes the amount of data to be transferred and the rate at which it should be transferred. If a single mouse click is to be transferred to the CPU, then the amount of data is just one bit; on the other hand, a block of data for the hard drive may be several kilo bytes. Similarly, the rate of the transfer of data to a printer is very different from the transfer rate needed for a hard drive.

Data synchronization:

This means that the CPU should input data from an input device only when the device is ready to provide data and send data to an output device only when it is ready to receive data.

There are three basic schemes which can be used for synchronization of an I/O data transmission:

- Synchronous transmission
- Semi-synchronous transmission
- Asynchronous transmission

Synchronous transmission:

This can be understood by looking at the waveforms shown in Figure A. M stands for the bus master and S stands for the slave device on the bus. The master and the slave are assumed to be permanently connected together, so that there is no need for the selection of the particular slave device out of the many devices that may be present in the system. It is also assumed that the slave device can perform the transfer at the speed of the master, so no handshaking signals are needed.

At the start of the transfer operation, the master activates the Read signal, which indicates to the slave that it should respond with data. The data is provided by the slave, and the master uses the Enable signal to latch it. All activity takes place synchronously with the system clock (not shown in the figure). A familiar example of synchronous transfer is a register-to-register transfer within a CPU.

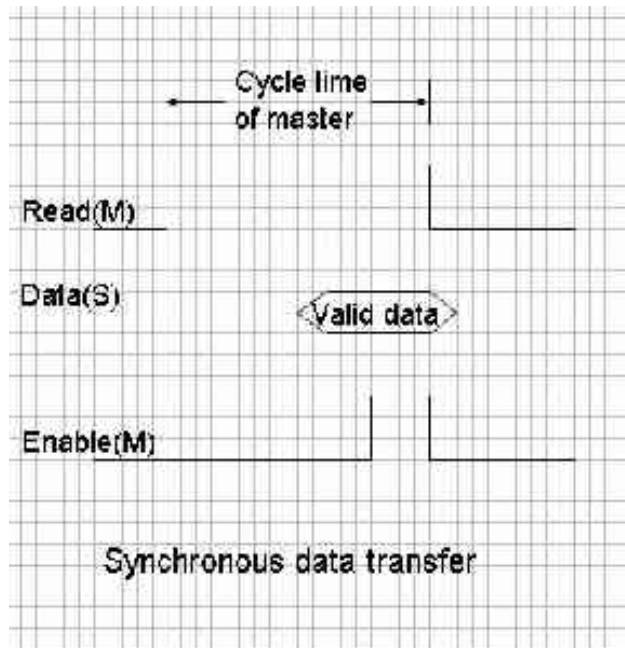


Figure A

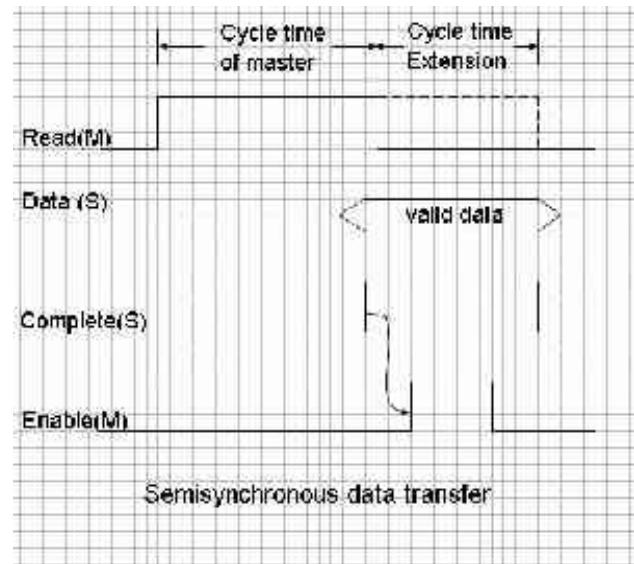


Figure B

Semi-synchronous transmission:

Figure B explains this type of transfer. All activity is still synchronous with the system clock, but in some situations, the slave device may not be able to provide the data to the master within the allotted time. The additional time needed by the slave, can be provided by adding an integral number of clock periods to Figure A the master's cycle time.

The slave indicates its readiness by activating the complete signal. Upon receiving this signal, the master activates the Enable signal to latch the data provided by the slave. Transfers between the CPU and the main memory are examples of semi-synchronous transfer.

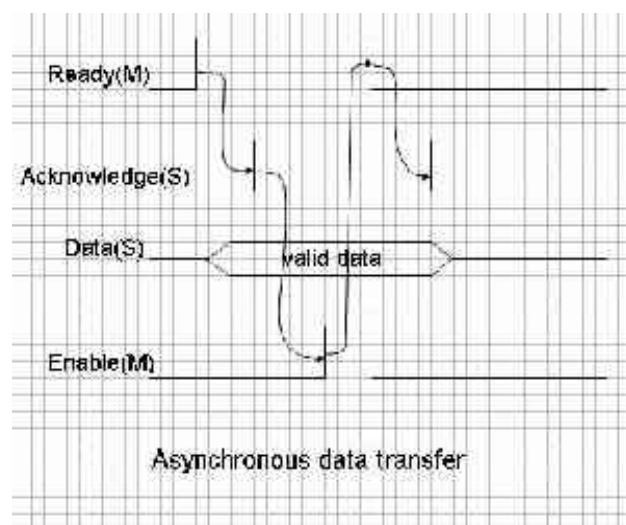


Figure C

Asynchronous transmission:

This type of transfer does not require a common clock. The master and the slave operate at different speeds. Handshaking signals are necessary in this case, and are used to coordinate the data transfer between the master and the slave as shown in the Figure C. When the master wants

to initiate a data transfer, it activates its Ready signal. The slave detects this signal, and if it can provide data to the master, it does so and also activates its Acknowledge signal. Upon receiving the Acknowledge signal, the master uses the Enable signal to latch the incoming data. The master then deactivates its Ready line, and in response to it, the slave removes its data and deactivates its Acknowledge line.

In all the three cases discussed above, the waveforms correspond to an “input” or a “read” operation. A similar explanation will apply to an “output” or a “write” operation. It should also be noted that the latching of the incoming data can be done by the master either by using the rising edge of the Enable signal or by using its falling-edge. This will depend on the way the intermediate circuitry between the master and the slave is designed.

Serial and Parallel Transfers

There are two ways in which data can be transferred between the CPU and an I/O device: serial and parallel.

Serial Transfer, or serial communication of data between the CPU and the I/O devices, refers to the situation when all the data bits in a "piece of information", (which is a byte or word mostly), are transferred one bit at a time, over a single pair of wires. Advantages:

- Easy to implement, especially by using UARTs⁷ or USARTs⁸.
- Low cost because of less wires.
- Longer distance between transmitter and receiver. Disadvantages:
- Slow by its very nature.
- Inefficient because of the associated overhead, as we will see when we discuss the serial wave forms.

Parallel Transfer, or parallel communication of data between the CPU and the I/O devices, refers to the situation when all the bits of data (8 or 16 usually), are transferred over separate lines simultaneously, or in parallel. Advantages:

- Fast (compared to serial communication) Disadvantages:
- High cost (because of more lines).
- Cost increases with distance.
- Possibility of interference (noise) increases with distance.

Remember that the terms "serial" and "parallel" are with respect to the computer I/O ports and not with respect to the CPU. The CPU always transfers data in parallel. 

Types of serial communication

There are two types of serial communication:

Asynchronous:

- Special bit patterns separate the characters.
- "Dead time" between characters can be of any length.
- Clocks at both ends need not have the same frequency (within permissible limits).

7

Universal Asynchronous Receiver Transmitter.

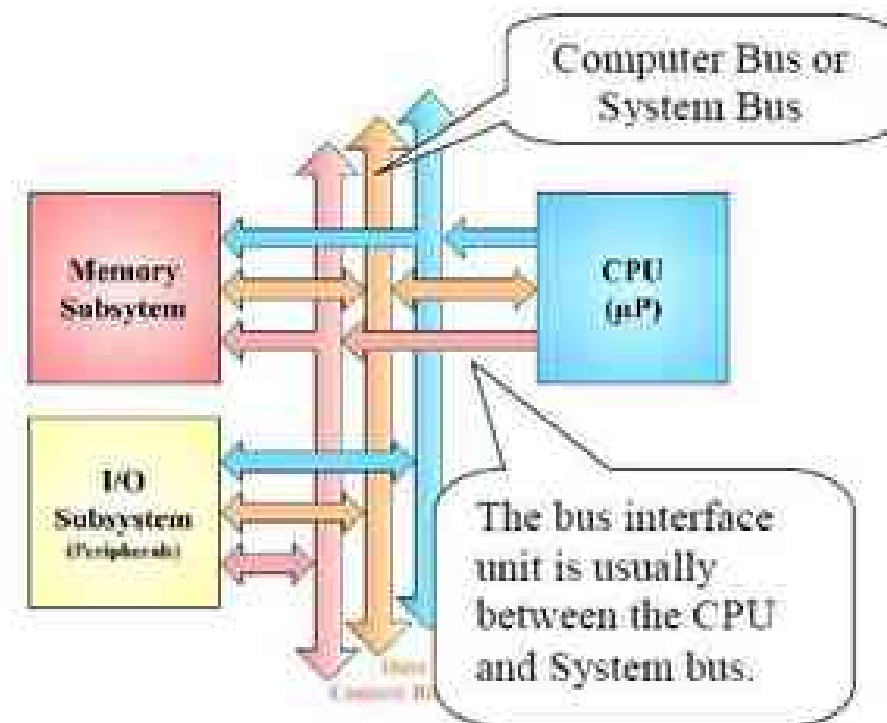
8

Universal Synchronous Asynchronous Receiver Transmitter.

Synchronous:

- Characters are sent back to back.
- Must include special "sync" characters at the beginning of each message.
- Must have special "idle" characters in the data stream to fill up the time when no information is being sent.
- Characters must be precisely spaced.
- Activity at both ends must be coordinated by a single clock. (This implies that the clock must be transmitted with data).

The "maximum information rate" of a synchronous line is higher than that of an asynchronous line with the same "bit rate", because the asynchronous transmission must use extra bits with each character. Different protocols are used for serial and parallel transfer. A protocol is a set of rules understood by both the sender and the receiver. In some cases, these protocols can be predefined for a certain system. As an alternate, some available standard protocols can be used.



Block Diagram of a Computer System

Figure 1

Error conditions related to serial communication

(Some related to synchronous transmission, some to asynchronous, and some to both).

- Framing Error: is said to occur when a 0 is received instead of a stop bit (which is always a 1). It means that after the detection of the beginning of a character with a start bit, the appropriate number of stop bits was not detected. [A]
- Parity Error: is said to occur when the parity* of the received data is not the same as it should be. [B] (PARITY is equivalent to the number of 1's; it is either EVEN or ODD. A PARITY BIT is an extra bit added to the data, for the purpose of error detection and correction. If even parity is used, the parity bit is set so that the total number of 1's, including the parity bit, is even. The same applies to odd parity.)

- **Overflow Error:** means that the prior character that was received, was not yet read from the USART's "receive data register" by the CPU, and is overwritten by the new received character. Thus the first character was lost, and should be retransmitted. [A]
- **Under-run Error:** If a character is not available at the beginning of an interval, an under-run is said to occur. The transmitter will insert an idle character till the end of the interval. [S]

I/O Buses

The block diagram of a general purpose computer system that has been referred to repeatedly in this course has three buses in addition to the three most important blocks. These three buses are collectively referred to as the system bus or the computer bus⁹. The block diagram is repeated here for an easy reference in Figure 1.

Another organization that is used in modern computers is shown in Figure 2. It has a memory bus for connecting the CPU to the memory subsystem. This bus is separate from the I/O bus that is used to connect peripherals and I/O devices to the system.

Examples of I/O buses include the PCI bus and the ISA bus. These I/O buses provide an “abstract interface” that can be used for interfacing a large variety of peripherals to the system with minimum hardware. It is also possible to standardize I/O buses, as done by several agencies, so that third party manufacturers can build add-on sub systems for existing architectures.

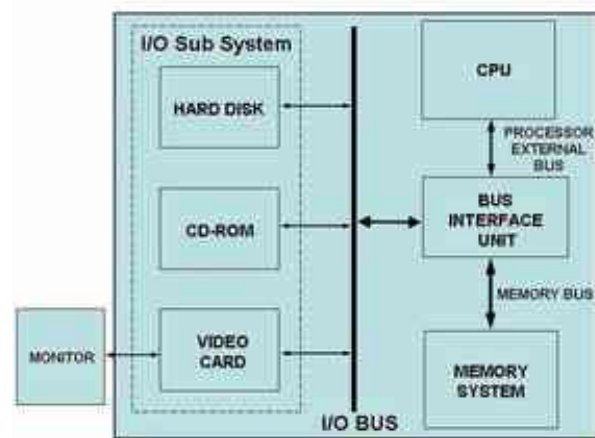


Figure 2

The location of these I/O buses may be different in different computers.

Earlier generation computers used a single bus over which the CPU could communicate with the memory as well as the I/O devices. This meant that the bandwidth of the bus was shared between the memory and I/O devices.

However, with the passage of time, FIGURE 1 computer architects drifted towards separate memory and I/O buses, thereby giving more flexibility to users wanting to upgrade their existing systems. A main disadvantage of I/O buses (and the buses in general) is that every bus has a fixed bandwidth which is shared by all devices on the bus. Additionally, electrical constraints like transmission line effects and bus length further reduce the bandwidth. As a result of this, the designer has to make a decision whether to sacrifice interface simplicity (by connecting more devices to the bus) at the cost of bandwidth, or connect fewer devices to the bus and keep things simple to get a better bandwidth. This can be explained with the help of an example.

Example # 1

Problem statement:

Consider an I/O bus that can transfer 4 bytes of data in one bus cycle. Suppose that a designer is considering to attach the following two components to this bus:

⁹

In some cases, the external CPU bus is the same as the dedicated systems. However, for most systems, there is a “bus interface unit” between the CPU and the system bus. The bus interface unit is not shown in the figure.

Hard drive, with a transfer rate of 40 Mbytes/sec Video card, with a transfer rate of 128 Mbytes/sec. What will be the implications?

Solution:

The maximum frequency of the bus is 30 MHz¹⁰. This means that the maximum bandwidth of this bus is $30 \times 4 = 120$ Mbytes/sec. Now, the demand for bandwidth from these two components will be $128 + 40 = 168$ Mbytes/sec which is more than the 120 Mbytes/sec that the bus can provide. Thus, if the designer uses these two components with this bus, one or both of these components will be operating at reduced bandwidth.

Bus arbitration:

Arbitration is another issue in the use of I/O buses. Most commercially available I/O buses have protocols defining a number of things, for example how many devices can access the bus, what will happen if multiple devices want to access the bus at the same time, etc. In such situations, an “arbitration scheme” must be established. As an example, in the SCSI¹¹ specifications, every device in the system is assigned an ID which identifies the device to the “bus arbiter”. If multiple devices send a request for the bus, the device with the highest priority will be given access to the bus first. Such a scheme is easy to implement because the arbiter can easily decide which device should be given access to the bus, but its disadvantage is that the device with a low priority will not be able to get access to the bus¹². An alternate scheme would be to give the highest priority to the device that has been waiting for the longest time for the bus. As a result of this arbitration, the access time, or the latency, of such buses will be further reduced. Details about the PCI and some other buses will be presented in a separate section.

Example # 2

Problem statement:

If a bus requires 10 nsec for bus requests, 10 nsec for arbitration and the average time to complete an operation is 15 nsec after the access to the bus has been granted, is it possible for such a bus to perform 50 million IOPS?

Solution:

For 50 million IOPS, the average time for each IOP is $1 / (50 \times 10^6) = 20$ nsec. Given the information about the bus, the sum of the three times is $10 + 10 + 15 = 35$ nsec for a complete I/O operation. This means that the bus can perform a maximum of $1 / (35 \times 10^{-9}) = 28.6$ million IOPS.

Thus, it will not be able to perform 50 million IOPS.

¹⁰ than this. These numbers correspond to an I/O bus that is relatively old. Modern systems use much faster buses

¹¹ Small Computer System Interface.

¹² Such a situation is called “starvation”.

Lecture No. 24

Designing Parallel Input and Output Ports

Reading Material

Handouts

Slides

Summary

- Designing Parallel I/O Ports
- Practical Implementation of the SAD
- NUXI Problem
- Variation in the Implementation of the Address Decoder
- Estimating the Delay Interval

Designing Parallel I/O Ports

This section is about designing parallel input and output ports. As you already know from the previous discussion, an interface that is used to connect the computer bus with I/O devices is called an I/O port. This I/O port can be connected directly to the computer bus (also called the system bus) or through an intermediate bus called the I/O bus. This intermediate bus is also called the expansion bus or the peripheral bus. In any case, the following general information about I/O bus cycles on a typical CPU should be kept in mind: At the start of a particular bus cycle (which will be an I/O bus cycle in this case), the CPU places an address on its address bus. This address will identify the I/O device to be involved in the transfer. After some time the CPU will activate certain control signals, which will indicate whether the particular I/O bus cycle, is an I/O read or an I/O write cycle. Based on these control signals, in case of I/O read cycle, the CPU will be expecting data from the selected input device over the data bus, and for an I/O write cycle the CPU will provide data to the selected device over the data bus. At the end of this I/O bus cycle, the address (and data) information will be removed from the buses and the control signals will be reset. It can be easily understood from this discussion that we must match the timing requirements of the I/O ports to be designed with the timing parameters of the given CPU. Additionally, the voltage and current requirements of the I/O ports must be matched with the voltage and current specifications of the CPU. For simplicity, we ignore the voltage and current matching details in this discussion and only focus on the logic levels and timing aspects of the design. Voltage and current related discussions are the topic of an electronics course.

Thus, there are two important functions which should be built into I/O ports.

1. Address decoding
2. Data isolation for input ports or data capturing for output ports.

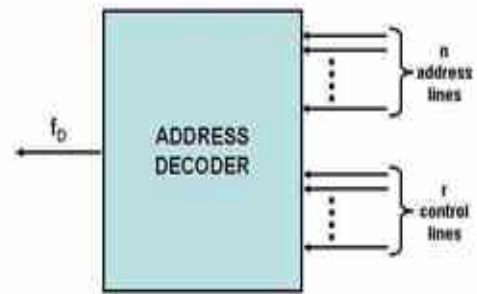
1. Address decoding: Since every I/O port has a unique identifier associated with it, (which is called its *address*, and no other port in the system should have the same address), by monitoring the system address bus, the I/O port knows when it is its turn to participate in a transfer. At this time, the address decoder within the I/O port generates an asserted output which can be applied to the enable input of tri-state buffers in input ports or the latch enable input of latches in output ports.

Our definition of an address decoder:

An "Address Decoder" is a combinational (logic) circuit with $n + r$ inputs and a single output, where n = the number of address lines into the decoder, and r = the number of control lines into the decoder.

The output f_D is active only when the corresponding address is present on the n address lines and the corresponding r control lines hold the "proper" (active or inactive) value. f_D is inactive for all other situations.

Block diagram of an address decoder



Suggestions for address decoder design:

1.1 Start by thinking of the address decoder as a "big AND gate". We will call this a "skeleton address decoder" or SAD. The output of the SAD will be active only when the correct address is present on the system address bus and the relevant control bus signals hold the proper values. At all other times, the output of the SAD should be deactivated.

1.2 Always write the port address of the port to be designed in binary. Associate the CPU's address lines with each bit. Those lines which are zero will be inverted before being fed into the "big AND gate"; other address lines will not be inverted.

1.3 List the relevant control signals for the system to which the port is to be attached. If the "proper" value of the signal is 0, it should be inverted before applying to the SAD, otherwise it is fed directly into the SAD.

1.4 Determine whether the decoder output should be active high or low. This will depend on the type of latch or buffer used in the design. If an active low decoder output is needed, invert the output from the "big AND gate".

1.5 Once the logic for the address decoder is established, the SAD can be implemented using any of the available methods of logic design. For example, HDL code in Verilog or VHDL can be generated and the address decoder can be implemented using PLDs. Alternately, the SAD can be implemented using SSI building blocks.

2. Data isolation or capturing: For input ports, the incoming data should be placed on the data bus only during the I/O read bus cycle. At all other times, this data should be isolated from the data bus otherwise it will cause "bus contention". Tri-state buffers are used for this purpose. Their input lines are connected to the peripheral device supplying data and their output lines are connected to the data bus. The common enable line of such buffers is driven with the output of the SAD. If this enable is active low, the output of the big AND gate in the SAD should be inverted, as described earlier.

For output ports, data is made available for the peripheral device at the data bus during the I/O write bus cycle. During other bus cycles, this data will be removed from the data bus by the processor. Latches (or registers) are used for this purpose. Their input lines are connected to the system data bus and their output lines are connected to the peripheral device receiving data. The common clock (or latch enable) line of such latches is driven with the output of the SAD. If this clock is active low, the output of the big AND gate in the SAD should be inverted.

Example # 1

Problem Statement:

Design a 16-bit parallel output port mapped on address DEh of the I/O space of the **FALCON-A CPU**.

Solution:

Using the guidelines mentioned above, we start with a “big AND gate” (SAD) and write the address to be decoded (DEh) in binary.

Thus, DEh → 1101 1110 b. Associating one CPU address line with each bit, we get A0 = 0, A1=1, etc as shown in the table below.

Because the I/O space on the FALCON-A is only 256 bytes, address lines A15 .. A8 are don't cares, and will not be used in this design.

1	1	0	1	1	1	1	0
A7	A6	A5	A4	A3	A2	A1	A0

Thus, A0 and A5 will be applied to the “big AND gate” after inversion. The remaining address lines will be connected directly to the inputs of the SAD.

Next, we look at the relevant control signals. The only signal which should be used in this case is IOW#. A logic 0 (zero) on this line indicates that it is active. Thus, it should be inverted before being applied to the input of the SAD.

We can easily see that our SAD intuitively conforms to the way we defined an address decoder. Its output is a 1 only when the address (xxxx xxxx 1101 1110 b) is present on the FALCON-A's address bus during an I/O write cycle (By the way, this will take place when the instruction **out reg, addr** with **addr=DEh or 222d** is executing on the FALCON-A). At all other times, its output will be inactive.

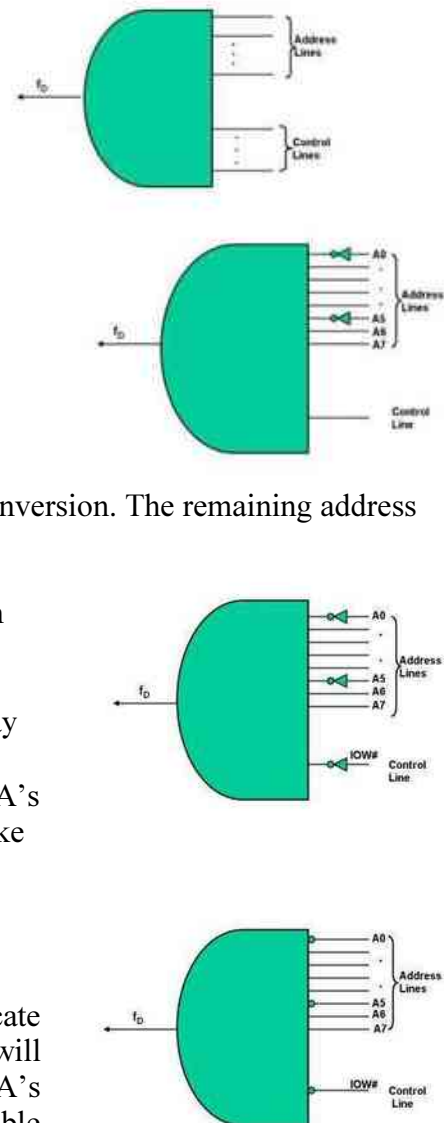
To make things simple, we use a circle (or a bubble) to indicate an inverter, as shown .Since this is a 16-bit output port, we will use two 8-bit registers to capture data from the FALCON-A's data bus. The output of the SAD will be connected to the enable inputs of the two registers. The D-inputs of the registers will be connected to the data bus and the Q outputs of the registers will be connected to the peripheral device.

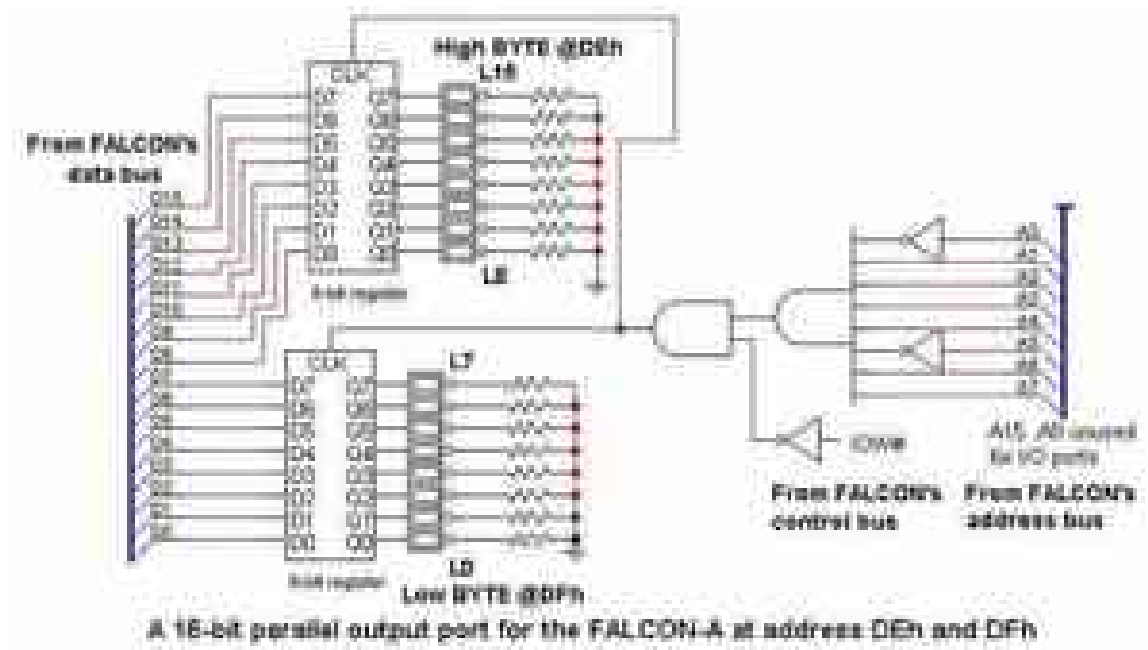
Practical implementation of the SAD

Our SAD in this design is an AND gate with 9 inputs. Using SSI chips, we can implement this SAD using an 8-input AND gate and a 2-input AND gate as shown in the figure shown below.

Displaying output data using LED branches:

An “LED branch” is a combination of a resistor and a light emitting diode (LED) in series. Sixteen LED branches can be used to display the output data captured by the registers as shown in the figure below.





Example # 2

Problem statement:

Given a 16-bit parallel output port attached with the FALCON-A CPU as shown in the figure. The port is mapped onto address DEh of the FALCON-A's I/O space. Sixteen LED branches are used to display the data being received from the FALCON-A's data bus. Every LED branch is wired in such a way that when a 1 appears on the particular data bus bit, it turns the LED on; a 0 turns it off. Which LEDs will be ON when the instruction

out r2, 222¹³

executes on the CPU? Assume r2 contains 1234h.

Solution:

Since r2 contains 1234h, the bit pattern corresponding to this value will be sent out to the output port at address 222 (or DEh). This is the address of the output port in this example. Writing the bit pattern in binary will help us determine the LEDs which will be ON.

Now 1234h gives us the following bit associations with the data bus

0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0
D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
MSB at address DEh								LSB at address DFh							

Note that the 8-bit register which uses lines D15 .. D8 of the FALCON-A's data bus is actually mapped onto address DEh of the I/O space.

¹³ Depending on the way the assembler is written, the syntax of the **out** instruction may allow only the decimal form of the port address, or only the hexadecimal form, or both. Our version of the assembler for the FALCON-A allows the decimal form only. It also requires that the port address be aligned on 16-bit "word boundaries", which means that every port address should be divisible by 2.

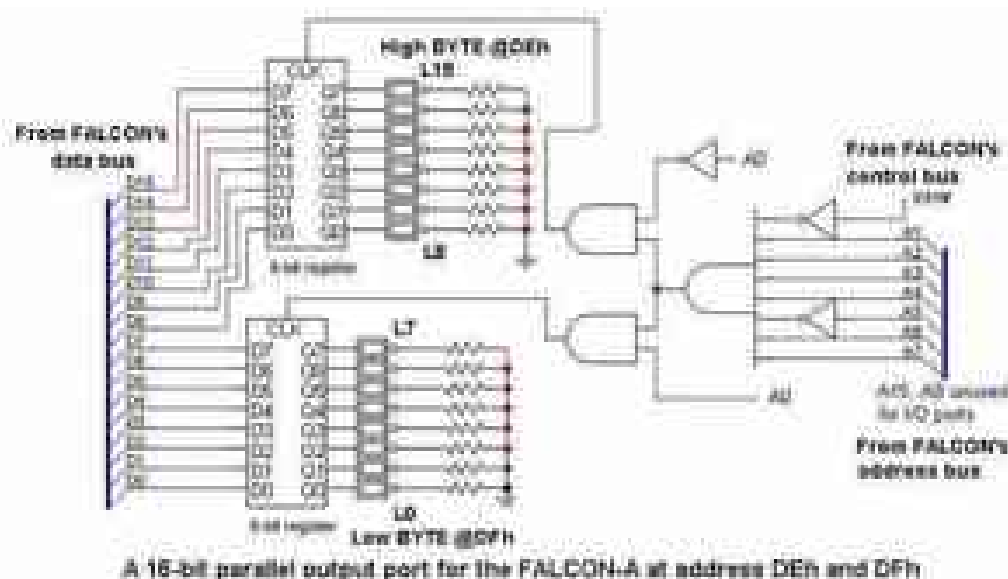
This is because the architect of the FALCON-A had chosen a “byte-wide” (i.e., x8) organization of the address space, a 16-bit data bus width, and the “big-endian” data format at the ISA design stage. Additionally, data bus lines D15...D8 will transfer the data byte of higher significance (MSB) using address DEh, and D7...D0 will transfer the data byte of lower significance (LSB) using address DFh. Thus the LEDs at L12, L9, L5, L4 and L2 will turn on.

The NUXI Problem

It can be easily understood from the previous example that the big-endian format results in the least significant byte being transferred over the most significant side of the data bus, and vice versa. The situation will be exactly opposite when the little-endian format is used. In this case, the least significant byte will be transferred over the least side of the data bus. Now imagine a computer using the little-endian format exchanging data with a computer using the big-endian format over a 16-bit parallel port. (this may be the case when we have a network of different types of computer, for example). The data transmitted by one will be received in a “swapped” form by the other, eg., the string “UN” will be received as “NU” and the string “IX” will be received as “XI”. So UNIX changes to NUXI --- hence the name NUXI problem. Special software is used to resolve this problem.

Variation in the Implementation of the Address Decoder

The implementation of the address decoder shown in Example #1(lec24) assumes that the FALCON-A does not allow the use of some part of its data bus during an I/O (or memory) transfer. Another restriction that was imposed by the assembler was that all port addresses should be divisible by 2. This implies that address line A0 will always be zero. If the FALCON-A architect had allowed the use some of part of its data bus (eg, 8-bits) during a transfer, the situation would be different.



The logic diagram shown in the next figure is a 16-bit parallel output port at the same address (DEh) for the FALCON-A assuming that part of its data bus (D15..D8) or (D7..D0) can be used independently during an I/O transfer. Note that the enable inputs of the two 8-bit registers are not connected together in this case. Moreover, since the 16-bit port uses two addresses, address line A0 will be at a logic 0 for address DEh, and at a logic 1 for address DFh. This means that it cannot be used at the input of the big AND gate. So, A0 has been used in a different position with the two 2-input AND gates. The 2-input AND gate where A0 is applied after inversion will generate a 1 at its output when A0 = 0. Thus, this output will enable the 8-bit register mapped on

the even address DEh. In case of the other AND gate, A0 is not inverted. So the corresponding 8-bit register will be mapped on the odd address DFh. The input that became available after removing A0 from its old position can be used for the IOW# control signal. The rest of the circuit is the same as it was in the previous figure.

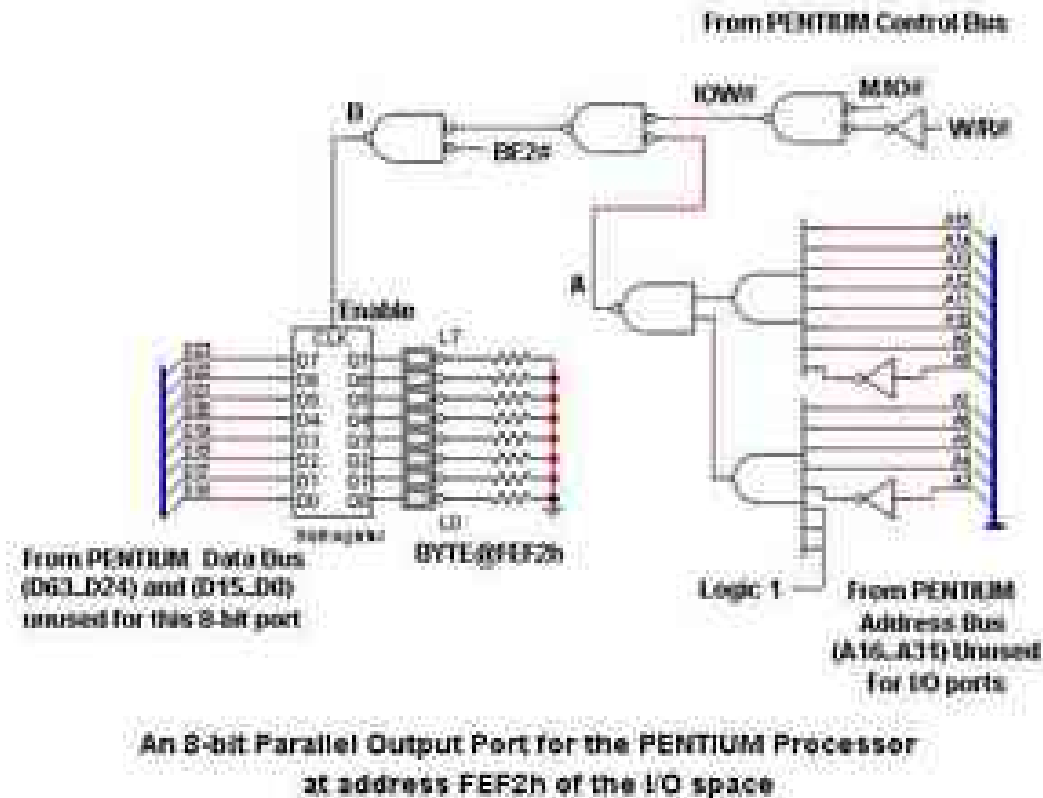
We can understand from the above discussion that the decisions made at the time of ISA design have a strong bearing on the implementation details and the working of the computer. Suppose we assume that the assembler developer had decided not to restrict the port addresses to even values, then what will be the implications?

As an example, consider the execution of the instruction **out r2, 223** assuming r2 contains 1234h. This is a 16-bit transfer at address 223 (DFh) and 224 (E0h).

For the output port (shown in the first figure) where the CPU does not allow the use of some part of its data bus in a transfer, none of the registers will be enabled as a result of this instruction because the output of the 8-input AND gate will be a zero for both addresses DFh and E0h. Thus, that output port cannot be used.

In the second figure, where the CPU has allowed to use a portion of its data bus in an I/O transfer, the register at the address DEh will not be enabled. The CPU will send the high data byte(12h) to the register at the address DFh (because it will be enabled at that time due to the address DFh) over data lines D7...D0. The fact that data lines D7...D0 should be used for the transfer of high byte, will be taken care of by the hardware, internal to the CPU.

Now the question is where the low data byte (i.e. 34h) present at D15...D8 data lines would be placed? If there exists an output port at address E0h in the system, then 34h will be placed there **(in the next bus cycle)**, otherwise it will be lost. Again, it is the CPU's responsibility to check whether the next address in the system exists or not and if exists then enable that port so that the low byte of data can be placed there.



A possible option for the architect in this case would be to revisit the design steps and allow the use of part of the CPU registers (or at least for some of them) for I/O transfers. The logic diagram shown below shows an 8-bit parallel output port at address FEF2h of the Pentium's I/O address space. Since the Pentium allows the use of some part of its data bus during a transfer, we

can use the BE2# signal in the address decoder to enable the 8-bit register. The following instructions will access this output port.

```
mov dx, 0FEF2h
mov al, 12h
out dx, al
```

The Pentium **does** allow the use of some part of its 32-bit accumulator register EAX. In case only 8-bits are to be transferred, register AL can be used, as shown in the program fragment above. The data byte 12h will be sent to the 8-bit register over lines D23..D16. Since 12h corresponds to 0001 0010 in binary, this will cause the LEDs L4 and L1 to turn on.

Example # 3

Problem statement:

Write an assembly language program to turn on the 16 LEDs one by one on the output port of Example #1(lec24). Each LED should stay on for a noticeable duration of time. Repeat from the first LED after the last LED is turned on.

Solution:

The solution is shown in the text box with a filename: Example_3.asmfa. The working of this program is explained below:

The first two instructions turn all the LEDs off by sending a 0 to each bit of the output port at address 222.

```
mov r1,0
out r1,222
```

Then a 1 is sent to L0 causing it to turn on, and the program enters a loop which executes 15 times to cause the other LEDs (L1 through L15) to turn on, one by one in sequence. Register r5 is being used as loop counter. The following three instructions introduce a delay between successive bit patterns sent to the output port, so that each LED stays on for a noticeable duration of time.

```
delay1: movi r2,0
again1: subi r2,r2,1
        jnz r2,[again1]
```

Starting with a value of 0 in r2¹⁴, this value is decremented to FFFFh when the again1 loop is entered. The jnz instruction will cause r2 to decrement again and again; thereby executing the loop 65,535 times. An estimate of the delay interval is presented at the end of this section. After this delay, all the LEDs are turned off, and a second delay loop executes. Finally, the next LED on the left, in sequence, is turned on by the following two instructions:

```
filename: Example_3.asmfa
ALL LEDs ARE turned OFF initially
mov r1,0
out r1,222

First LED will be turned on each time
start: movi r1,1
      out r1,222

      movi r5,15

DELAY LOOP
delay1: movi r2,0
again1: subi r2,r2,1
        jnz r2,[again1]

      movi r1,0 : TURN OFF ALL LEDs
      out r1,222

      delay1: movi r2,0
      again1: subi r2,r2,1
              jnz r2,[again1]

      shiftl r1,r1,1 : next LED ON
      out r1,222
      subi r5,r5,1
      jnz r5,[delay1]
      jump [start]
      halt
```

¹⁴ This is necessary because the immediate operand with the **movi** instruction of the FALCON-A has a range of 0h to FFh. This will not give us the large loop counter that we need here. So we use the above software trick. An alternate way would be to use nested loops, but that will tie up additional CPU registers.

```
shifl r1,r1,1
out r1, 222
```

After the left most LED is turned on, the process starts all over again because of the last jump instruction. The outermost loop executes indefinitely.

Estimating the Delay Interval

To make things simple, assume that the FALCON-A is operating at a clock frequency of 1 MHz. Also, assume that the **subi** and the **jnz** instructions take 3 and 4 clock periods, respectively, to execute. Since these two instructions execute 65,535 times each, we can use the following formula to compute the execution time of this loop:

$$ET = CPI \times IC \times T = CPI \times IC / f$$

where

CPI = clocks per instruction

IC = instruction count

T = time period of the clock,

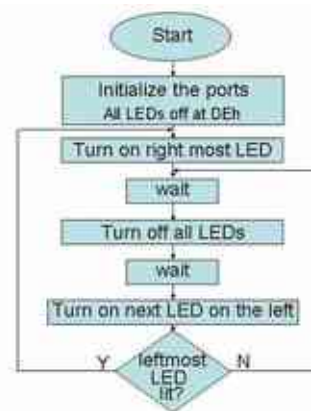
And

f = frequency of the clock.

Using the assumed values, we get

$$ET = (3+4) \times 65535 / (1 \times 10^6) = 0.459 \text{ sec}$$

Since the **movi r2, 0** instruction executes only once, the time it takes to execute is negligible and has been ignored in this calculation.



Lecture No. 25

Input Output Interface

Reading Material

Handouts

Slides

Summary

- Designing a Parallel Input Port
- Memory Mapped I/O Ports
- Partial Decoding and the “wrap around” Effect
- Data Bus Multiplexing
- A generic I/O Interface
- The Centronics Parallel Printer Interface

Designing a parallel input port

The following example illustrates a number of important concepts.

Example # 1

Problem statement:

Design an 16-bit parallel input port mapped on address 7Eh of the I/O space of the FALCON-A CPU.

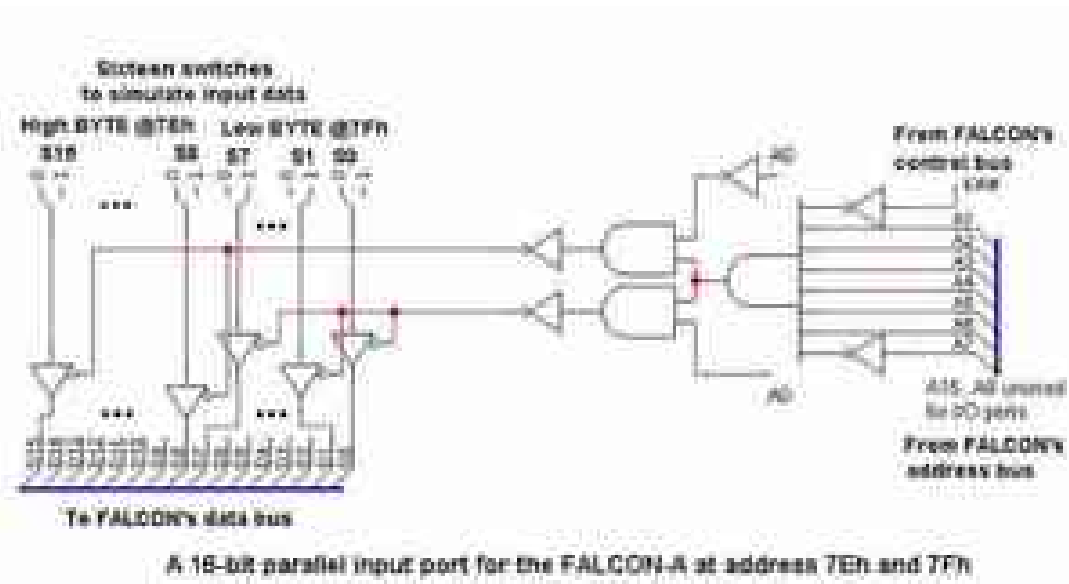
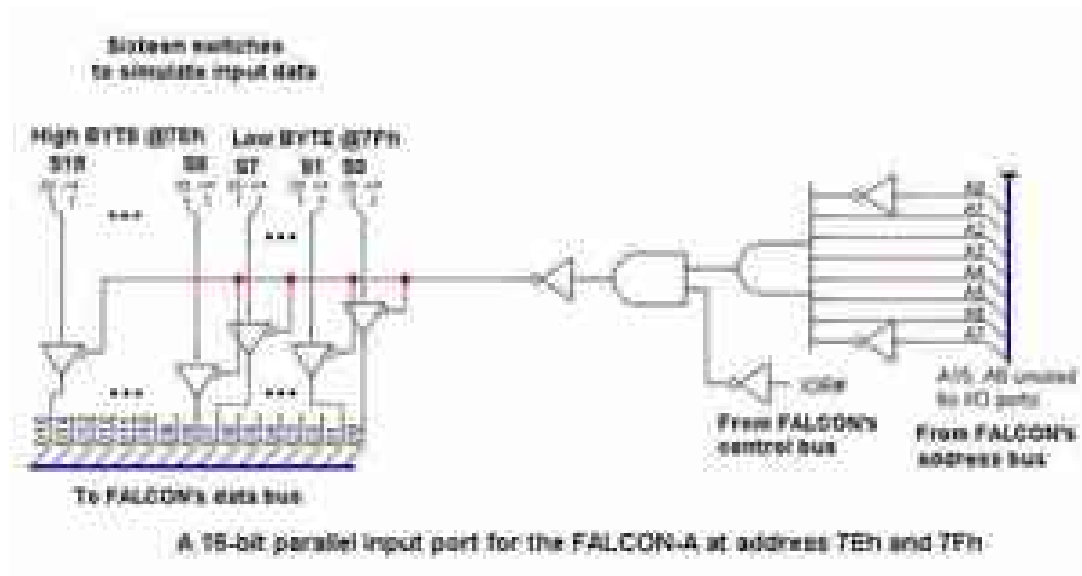
Solution:

The process of designing a parallel input port is very similar to the design of a parallel output port except for the following differences:

1. The address in this case is 7Eh, which is different from the previous value. Hence, the address decoder will have the inputs A7 and A0 inverted, while the other address lines at its input will not be inverted.
2. Control bus signal IOR# will be used instead of the signal IOW#.
3. A set of sixteen tri-state buffers will be used for data isolation. Their common enable line will be connected to the output of the big AND gate (in the figure, $\overline{f_D}$ is being inverted because Enable is active low). The input of these buffers can be connected to the input device and the output is connected to the FALCON-A's data bus.

In this example, switches S15...S0 are used to simulate the input data. The complete logic circuit is shown in the next two figures.

In the second figure, the CPU is assumed to allow the use of some part of its data bus during a transfer, while in the first figure it is not allowed.



Example # 2

Problem statement:

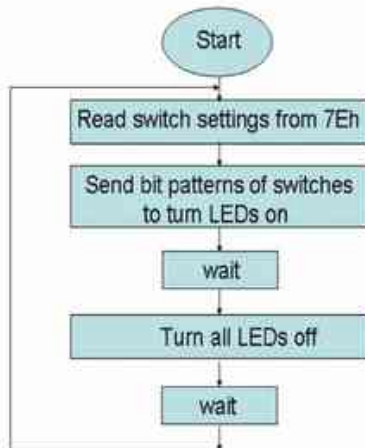
Given a FALCON-A processor with a 16-bit parallel input port at address 7Eh and a 16-bit parallel output port at address DEh. Sixteen LED branches are used to display the data at the output port and sixteen switches are used to send data through the input port. Write an assembly language program to continuously monitor the input port and blink the LED or LED(s) corresponding to the switch (es) set to logic 1. For example, if S0 and S2 are set to 1, then only the LEDs L0 and L2 should blink. If S7 is also set to logic 1 later, then L7 should also start blinking.

Solution:

The program is shown in the text box with filename: Example_2. It works as explained below;

The first two instructions read the input port at address 7Eh and send this bit pattern to the output port at address DEh. This will cause the LEDs corresponding to the switches that are set to a 1 to turn on. Next, the program waits for a suitable amount of time, and then turns all LEDs off and waits again.

After the second wait, the program reads the input port again. The LEDs that will be turn on at the output port will now be according to the new switch settings at the input port. The process repeats indefinitely. Please see the flowchart also.

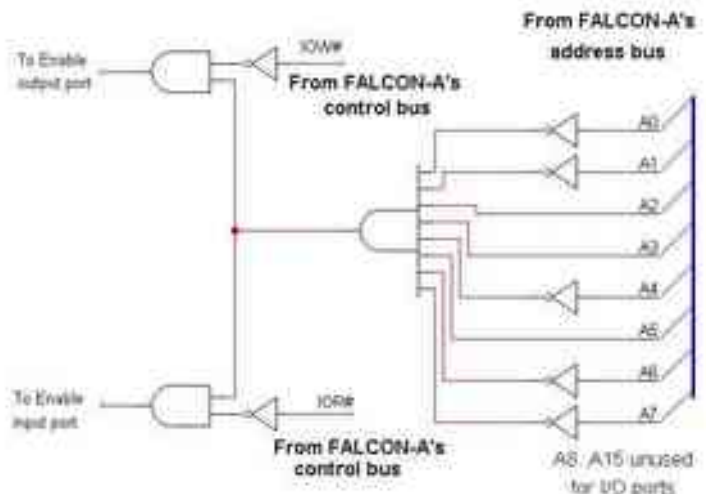


```

filename: Example_2.asmfa
Notes:
:   r1 is used as an I/O register
:   r2 is used as a delay counter
:
start:  in r1, 126      : 126d = 7Eh
        out r1, 222     : 222d = DEh
:
:   movi r2, 0
delay1: subi r2, r2, 1
        jnz r2, [delay1]
:
:   movi r1, 0         : all LEDs off
        out r1, 222
:
:   movi r2, 0
delay2: subi r2, r2, 1
        jnz r2, [delay2]
:
        jump [start]
:
halt
  
```

It is also possible to use a single address for both the input and the output port. The following diagram shows an address decoder for a 16-bit parallel input/output port at address 2Ch of the FALCON-A's

I/O space. Note that the control bus lines IOW# and IOR# will differentiate between the register and the tri-state buffer.

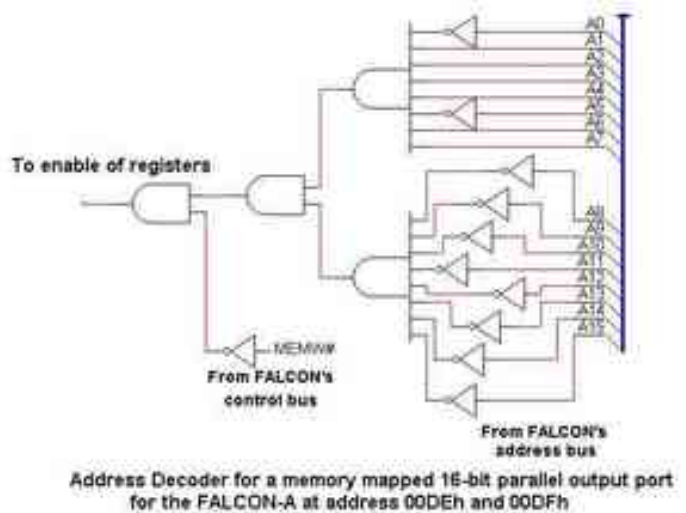


FALCON-A's Address Decoder for an I/O Port at the Address 2Ch

Memory mapped I/O ports

If it is desired to map the 16-bit output port of Example #1(lec24) on the memory space of the FALCON-A, the following changes would be needed.

1. Replace the IOW# signal with the MEMW# signal.
2. Use the entire CPU address bus at the input of the address decoder, as shown in the next figure. This address decoder uses the addresses 00DEh and 00DFh of the FALCON-A's memory space.
3. Use the **store** instruction instead of the **out** instruction for sending data to the output port (for memory mapped input ports, use the **load** instruction instead of the **in** instruction).



The program for Example #2 (lec25) is rewritten for the case of a memory mapped output port, and is shown in the attached text box. The advantage will be that more than 256 ports are available, but the disadvantage is that the address decoder will become more complex, resulting in increased hardware costs.

To avoid the increase in hardware complexity, many architects use what is called “partial decoding”. This is explained in the next section.

Partial decoding and the “wrap around” effect

Partial decoding is a technique in which some of the CPU's address lines forming an input to the address decoder are ignored. This reduces the complexity of the address decoder, and also lowers the cost. As an example, if the address lines A8...A15 from the FALCON-A are not used in the address decoder of the previous figure, this will save eight inverters and two AND gates. Partial decoding is an attractive choice in small systems, where the size of the address space is large but most of the memory is unimplemented. However, partial decoding has its price as well. Consider the memory map for the

FALCON-A, shown again in the next figure. With 16 address lines, the total address space is $2^{16} = 64$ Kbytes. When the upper eight address lines are unused, they become don't cares. The port shown in the previous figure will be accessed for address 00DEh. But, it will also be accessed for address 01DEh, 02DEh... FFDEh. In fact, the 64 Kbyte address space has been reduced to a 256 byte space. It “wrapped around” itself 256 times. If we only left 6 address lines, i.e., A15

```

filename: Example_1MM.asmfa
Notes:
For MEMORY MAPPED
output port at 00DEh.

r6 holds the output address
r7 holds the input address

movi r6, 111
add r6, r6, r6

movi r7, 128

r1 is used as an I/O register
r2 is used as a delay counter

start: load r1, [r7] ; 128d = 7Eh
       store r1, [r6] ; 222d = DEh

       movi r2, 0
delay1: subi r2, r2, 1
       jmp r2, [delay1]

       movi r1, 0 ; all LEDs off
       store r1, [r6]

       movi r2, 0
delay2: subi r2, r2, 1
       jmp r2, [delay2]

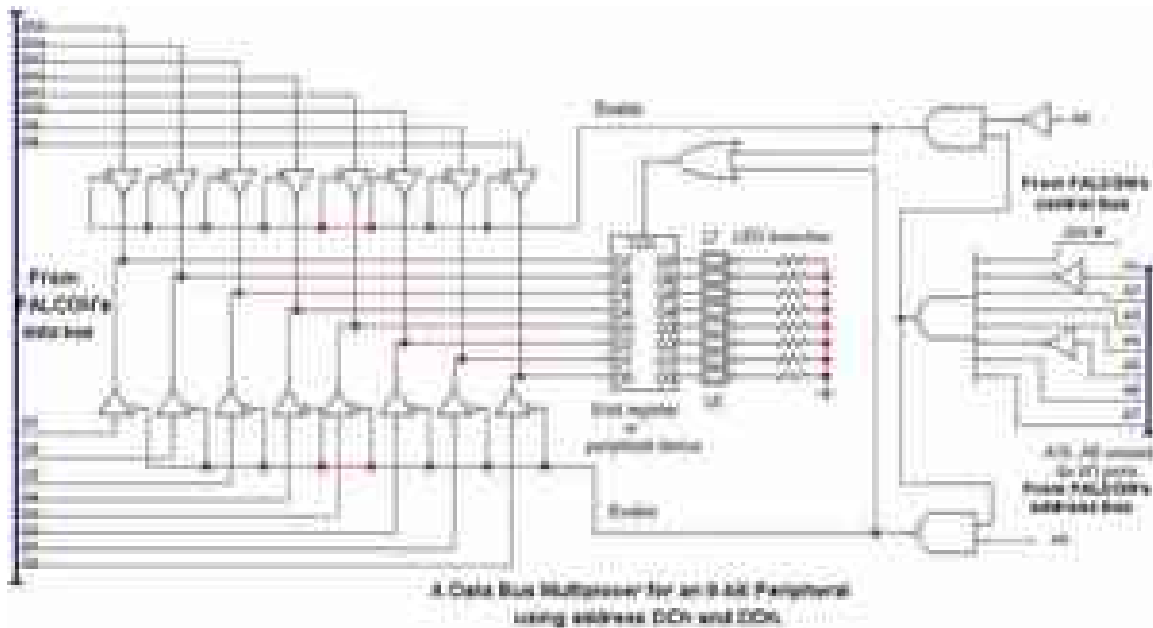
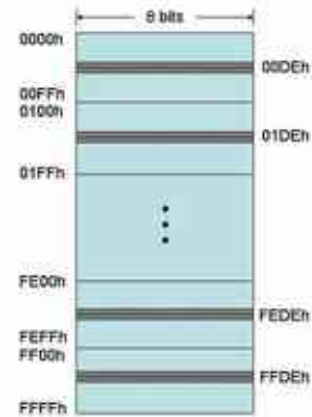
       jump [start]

halt
    
```

... A10, unconnected, then we will still have a “wrap around”, but of a different type. Now a 1 Kbyte ($= 2^{10}$) address area will wrap around itself 64 times ($= 2^6$).

Data bus multiplexing

Data bus multiplexing refers to the situation when one part of the data bus is connected to the peripheral's data bus at one time and the second part of the data bus is connected to the peripheral's data bus at a different time in such a way that at one time, only one 8-bit portion of the data bus is connected to the peripheral.



Consider the situation where an 8-bit peripheral is to be interfaced with a CPU that has a 16-bit (or larger) data bus, but a byte-wide address space. Each byte transferred over the data bus will have a separate address associated with it. For such CPUs, data bus multiplexing can be used to attach 8-bit peripherals requiring a block of addresses. Tri-state buffers can be used for this

purpose as shown in the attached figure. The logic circuit shown is for an 8-bit parallel output port using addresses DCh and DDh of the FALCON's I/O address space. It is assumed that the CPU allows the use of a part of its data bus during a transfer, and that each 16-bit general purpose register can be used as two separate 8-bit registers, e.g., r1 can be split as r1L and r1H such that

r1L<7..0> := r1<7..0>, and

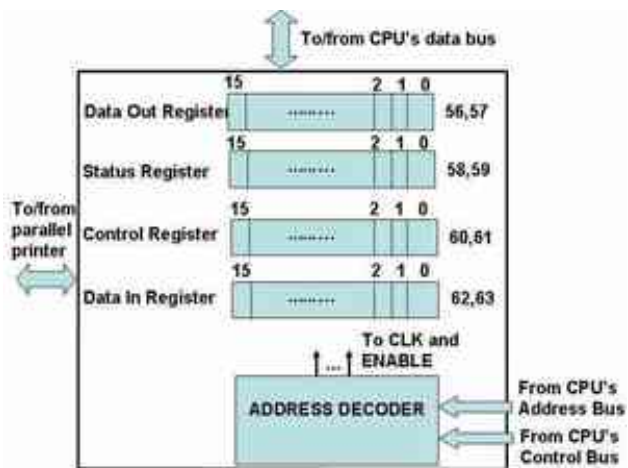
r1H<7..0> := r1<15..8>

The LED branches and the 8-bit register shown in the diagram serve as a place holder, and can be replaced by a peripheral device in actual practice. For an even address, A0=0, and the upper group of the tri-state buffers is enabled, thereby connecting D<15..8> of the CPU to the peripheral, while for an odd address from the CPU, A0=1, and the lower group of the tri-state buffers is enabled. This causes D<7..0> of the CPU to be connected with the peripheral device. In such systems the instruction **out r1H,220** will access the peripheral device using D<15..8>, while the instruction **out r1L,221** will access it using D<7..0>. The instruction **out r1,220** will

send r1H to the peripheral and the contents of r1L will be lost. Why? This is left as an exercise for the student. The advantage of data bus multiplexing is that all addresses are utilized and none of them is wasted, while the disadvantage is the increased complexity and cost of the interface.

A generic I/O interface

Most parallel I/O ports used with peripheral devices are mapped on a range of contiguous addresses. The following figure shows the block diagram of part of an interface that can be used with a typical parallel printer. It used eight consecutive addresses: address 56 to 63. A similar interface can be used with the FALCON-A. The registers shown within the interface are associated with some parallel device, and have some pre-defined functions. For example, the 16 bit register at addresses 56 and 57 can be used as a “data out” register for sending data bytes to the parallel device. In the same way, the register at addresses 60 and 61 can be used by the CPU to send control bits to the device. The double arrow shown at the top corresponds to the data bus connection of the interface with the CPU. The address decoder shown at the bottom receives address and control information from the CPU and generates enable signals for these registers. These abstract concepts are further explained in Example #3(lec25).



The Centronics Parallel Printer Interface

The Centronics Parallel Printer Interface is an example of a real, industry standard, set of signal specifications used by most printer manufacturers. It was originally developed for Centronics printers and can be used by devices having a uni-directional, byte-wide parallel interface. Table 1 shows the important signals and their functions as defined by the Centronics standard. Note that the direction of the signals is with respect to the printer and not with respect to the CPU.

Typically, the printer (or any other similar device) is connected to the CPU via a cable which has a 25-pin connector at the CPU side and a 36-pin connector at the printer side. Every data bit in the 8-bit data bus $D<7...0>$ uses a twisted pair for suppressing transmission-line effects, like radiation and noise. The return path of these pins should always be connected to signal ground. Additionally, the entire printer cable should be shielded, and connected to chassis ground on each side. The three signals STROBE#, BUSY and ACKNLG# form a set of handshaking signals. By using these signals, the CPU can communicate asynchronously with the printer, as shown in the accompanying timing waveforms. When the printer is ready for printing, the CPU starts data transfer to the printer by placing the 8-bit data (corresponding to the ASCII value of the character to be printed) on the printer's data bus (pin 2 through 9 on the 36-pin connector, as shown in Table 1). After this, a negative pulse of duration at least $0.5\mu s$ is applied to the STROBE# input (pin1) of the printer. The minimum set-up and hold times of the latches within the printer are specified as $0.5\mu s$ each, and these timing requirements must be observed by the CPU (the interface designer should make sure that these specifications are met). As soon as STROBE# goes low, the printer activates its BUSY line (pin 11) which is an indication to the CPU that additional bytes cannot be accepted. The CPU can monitor this status signal over an input port (a detailed assignment of these signals to I/O port bits is given in Table 2).

Table 1: The Centronics Parallel Printer Interface
(power and ground signals are not shown)

Signal Name	Direction	Function	Pin# (25-DB) CPU side 9,8,...,2	Pin# (36-DB) Printer side 9,8,...,2
D<7..0>	Input	8-bit data bus 1-bit control signal		
STROBE#	Input	High: default value. Low: read-in of data is performed. 1-bit status signal	1	1
ACKNLG#	Output	Low: data has been received and the printer is ready to accept new data. High: default value. 1-bit status signal	10	10
BUSY	Output	Low: default value High: see note#1 1-bit status signal	11	11
PE#	Output	High: the printer is out of paper. Low: default value. 1-bit control signal	12	12
INIT#	Input	Low: the printer controller is reset to its initial state and the print buffer is cleared. High: default value. 1-bit status signal	16	31
SLCT	Output	High: the printer is in selected state. 1-bit control signal	13	13
AUTO FEED XT#	Input	Low: paper is automatically fed after one line.	14	14

1-bit control signal

Low: data entry to the printer is possible.

SLCT IN#	Input	17	36
	High: data entry to printer is not Possible.		
	1-bit status signal		

ERROR#	Output	15	32
	Low: see note#2.		
	High: default value.		

Note#1

The printer cannot read data due to one of the following reasons:

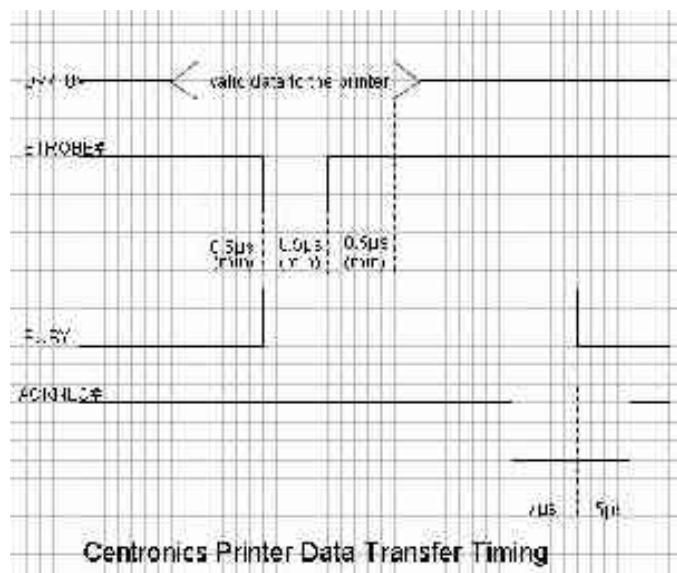
1. During data entry
2. During data printing
3. In offline state
4. During printer error status

Note#2

When the printer is in one of the following states:

1. Paper end state
2. Offline state
3. Error state

When this character is completely received, the ACKNLG# signal (pin 10) goes low, indicating that the transfer is complete. Soon after this, the BUSY signal returns to logic zero, indicating that a new transfer can be initiated. The BUSY signal is more suitable for level-triggered systems, while the ACKNLG# signal is better for edge-triggered systems.



The interface will typically use two eight bit parallel output ports of the CPU, one for the ASCII value of the character byte and the other for the control byte. It also specifies an 8-bit parallel input port for the printer's status information that can be checked by the CPU.

Table 2: Centronics Bit Assignment For I/O Ports

Logical Address	Description	7	6	5	4	3	2	1	0
0	8-bit output port for DATA	D<7>	D<6>	D<5>	D<4>	D<3>	D<2>	D<1>	D<0>
1	8-bit input port for	BUS Y	ACKNL G#	PE#	SLCT	ERROR #	Unuse d	Unuse d	Unused

	STATUS								
2	8-bit output port for CONTR OL	Unused	Unused	DIR ¹ ₅	IRQEN	SLCTIN#	INIT#	Auto Feed XT#	STROBE #

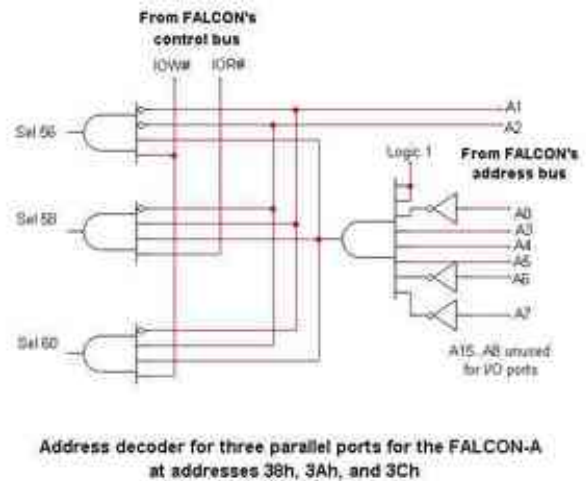
Example # 3:

Problem statement:

Design a Centronics parallel printer interface for the FALCON-A CPU. Map this interface starting at address 38h (56 decimal) of the FALCON-A's I/O address space. Solution:

The Centronics interface requires at least three I/O addresses. However, since the FALCON-A has a 16-bit data bus, and since we do not want to implement data bus multiplexing (to keep things simple), we will use three contiguous even addresses, i.e., 38h, 3Ah and 3Ch for the address decoder design. This arrangement also conforms to the requirements of our assembler.

Moreover, we will connect data bus lines D7...D0 of the FALCON-A to the 8-bit data bus of the printer (i.e. pins 9, 8, ... , 2 of the printer cable) and leave lines D15...D8 unconnected. Since the FALCON-A uses the big-endian format, this will make sure that the low byte of CPU registers will be transferred to the printer. (Recall that these bytes will actually be mapped on addresses 39h, 3Bh and 3Dh). The logic diagram of the address decoder for this interface is shown in the given figure.



¹⁵ This bit, when set, enables the bidirectional mode.

Lecture No. 26

Programmed I/O

Reading Material

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

Chapter 8
8.2.2

Summary

- The Centronic Parallel Printer Interface(Cont.)
- Programmed Input/Output
- Examples of Programmed I/O for FALCON-A and SRC
- Comparisons of FALCON-A, SRC examples

The Centronic Parallel Printer Interface (Cont.)

Table 1: The Centronics Parallel Printer Interface
(power and ground signals are not shown)
(The explanation of this table is provided in lecture 25 also)

Signal Name	Direction	Function	Pin# (25-DB) CPU side 9,8,...,2	Pin# (36-DB) Printer side 9,8,...,2
D<7..0>	Input	8-bit data bus 1-bit control signal		
STROBE#	Input	High: default value. Low: read-in of data is performed. 1-bit status signal	1	1
ACKNLG#	Output	Low: data has been received and the printer is ready to accept new data. High: default value. 1-bit status signal	10	10
BUSY	Output	Low: default value High: see note#1 1-bit status signal	11	11

PE#	Output	High: the printer is out of paper. Low: default value. 1-bit control signal	12	12
INIT#	Input	Low: the printer controller is reset to its initial state and the print buffer is cleared. High: default value. 1-bit status signal	16	31
SLCT	Output	High: the printer is in selected state. 1-bit control signal	13	13
AUTO FEED XT#	Input	Low: paper is automatically fed after one line. 1-bit control signal	14	14
SLCT IN#	Input	Low: data entry to the printer is possible. High: data entry to printer is not Possible. 1-bit status signal	17	36
ERROR#	Output	Low: see note#2. High: default value.	15	32

Note#1

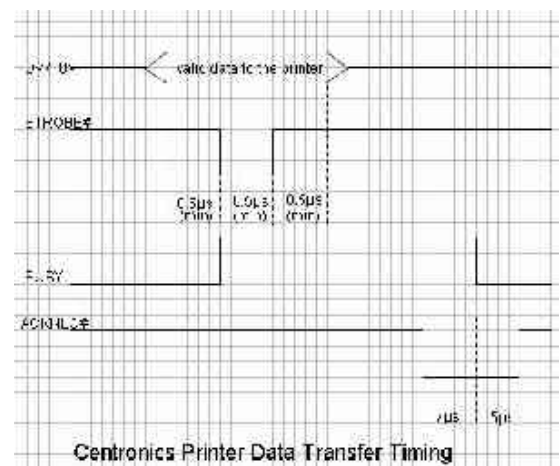
The printer cannot read data due to one of the following reasons:

5. During data entry
6. During data printing
7. In offline state
8. During printer error status

Note#2

When the printer is in one of the following states:

4. Paper end state
5. Offline state
6. Error state



When this character is completely received, the ACKNLG# signal (pin 10) goes low, indicating that the transfer is complete. Soon after this, the BUSY signal returns to logic zero, indicating that a new transfer can be initiated. The BUSY signal is more suitable for level-triggered systems, while the ACKNLG# signal is better for edge-triggered systems.

Advance Computer Architecture – CS501

The interface will typically use two eight bit parallel output ports of the CPU, one for the ASCII value of the character byte and the other for the control byte. It also specifies an 8-bit parallel input port for the printer's status information that can be checked by the CPU.

Table 2: Centronics Bit Assignment For I/O Ports

Logical Address	Description	7	6	5	4	3	2	1	0
0	8-bit output port for DATA	D<7>	D<6>	D<5>	D<4>	D<3>	D<2>	D<1>	D<0>
1	8-bit input port for STATUS	BUS Y	ACKNL G#	PE#	SLCT	ERROR #	Unuse d	Unuse d	Unused
2	8-bit output port for CONTR OL	Unus ed	Unused	DIR ¹ ₅	IRQE N	SLCT IN#	INIT#	Auto Feed XT#	STROB E #

Example # 1

Problem statement:

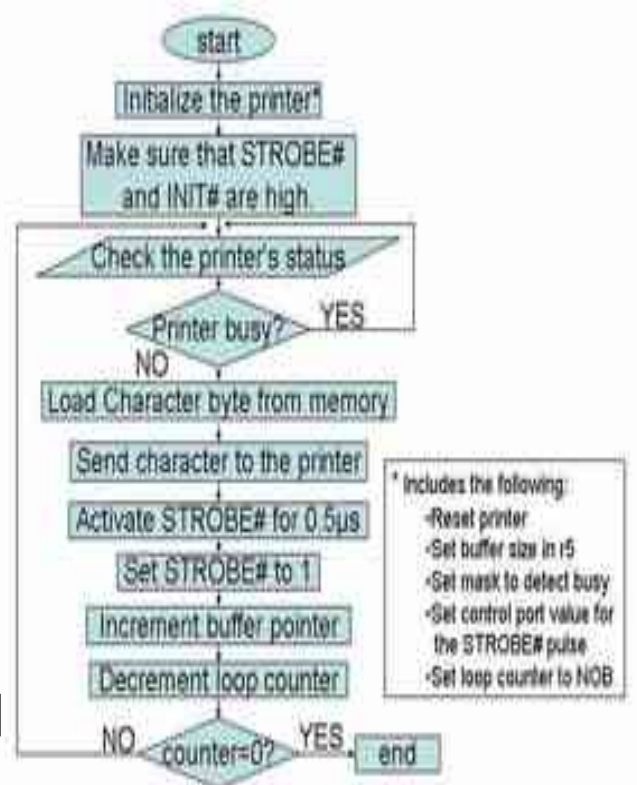
Assuming that a Centronics parallel printer is interfaced to the FALCON-A processor, as shown in example 3 of lecture 25, write an assembly language program to send an 80 character line to the printer. Assume that the line of characters is stored in the memory starting at address 1024.

Solution:

The flowchart for the solution is shown in given figure and the program listing is shown in the textbox with filename: Example_1.

The first thing that needs to be done is the **initialization of the printer**. This means that a “reset” command should be sent to the printer. Using the information from Table 1, this can be done by writing a 0 to bit 2 (i.e., INIT#) of the control register having logical address 2. In our example, this maps onto address 60 of the **FALCON-A**. (Remember to set this bit to logic 1 for normal operation of the printer). Then we make STROBE# high by

placing logic 1 in bit 0 of the control register. Bit 1 and bit 3 should be 0 because we want to activate auto line feed and keep the printer in selected mode. Additionally, bit 4 and bit 5 should be 0 so that interrupts are disabled and the bi-directional mode is not selected. The complete control word is 0000 0001 and this value has been assigned to the variable reset in the program. The following instruction pair performs the reset operation:




```
movi r1, reset  
out r1, controlp
```

As it is given that the starting address of the printer buffer is 1024¹⁷, so we place this address in r5. The mask to test the BUSY flag is placed in r3. The value for the mask is 80h. This corresponds to a logic 1 in bit 7 and logic zeros elsewhere for the status register having address 58 (logical address 1 in Table 1). Then the program enters a loop, called the polling loop, to test the status of the printer. If the printer is busy, the loop repeats. The following three instructions form the polling loop:

```
in r1, statusp  
and r1, r1, r3  
jnz r1, [again]
```

The status of the printer is placed in register r1, and bit 7 is tested for logic 0. If not so, the program repeats the status check operation.

When the printer is ready to accept a new character, it clears bit 7 (i.e., the BUSY bit) of the status register. At this time, the program picks the next character from the memory and sends it to the printer. The STROBE# line is activated and then it is deactivated to generate the necessary pulse on this input of the printer. Finally, the buffer pointer is advanced, the loop counter is decremented and the process repeats. When all the characters have been printed, the program halts.

A number of equates have been used in the program to make it flexible as well as easily readable. The program is shown on the next page.

¹⁷ The **mul** instruction is used for this purpose because the 8-bit immediate operand in the **movi** instruction can only be within the range –128 and +127. Using the **mul** instruction in this way overcomes the limitation of the FALCON-A. Similarly, the **shftl** instruction is used to bring 80h in register r3.

```

1 ////////////////////////////////////////////////////
2 // This program reads in 10 characters from
3 // a keyboard and prints them
4
5 #include <iostream>
6
7 using namespace std;
8
9 int main()
10 {
11     char ch;
12     for (int i = 0; i < 10; i++)
13     {
14         ch = getche();
15         cout << ch << " ";
16     }
17     cout << endl;
18     return 0;
19 }

```

I/O techniques:

There are three main techniques using which a CPU can exchange data with a peripheral device, namely

- Programmed I/O
- Interrupt driven I/O
- Direct Memory Access (DMA).

In this section, we present the first one.

Programmed Input/Output

Programmed I/O refers to the situation when all I/O operations are performed under the direct control of a program running on the CPU. This program, which usually consists of a “tight loop”, controls all I/O activity, including device status sensing, issuing read or write commands, and transferring the data¹⁸. A subsequent I/O operation cannot begin until the current I/O operation to a certain device is complete. This causes the CPU to wait, and thus makes the scheme extremely inefficient. The solution to Example # 3(lec24), Example #2(lec25), and Example #1(lec26) are examples of programmed input/output. We will analyze the program for Example #1(lec26) to explain a few things related to the programmed I/O technique.

Timing analysis of the program in Example # 1(lec26)

The main loop of the program given in the solution to Example #1(lec26) executes 80 times. This is equal to the number of characters to be printed on one line. This portion of the program is shown again with the execution time of each instruction listed in brackets with it. The numbers shown are for a uni-bus CPU implementation. A complete list of execution times for all the FALCON-A’s instructions is given in Appendix A. A number of things can be noted now.

1. Assuming that the output at the hardware pins changes at the end of the (I/O write) bus cycle, the STROBE# signal will go from logic1 to logic 0 at the end of the instruction pair.

```
movi r1, strb_L [2]
out r1, controlp [3]
```

```

movi r7, NOB [2]
;
again: in r1, statusp [3]
      and r1, r1, r3 [3]
      jnz r1, [again] [4]
;
      load r1, [r5] [5]
      out r1, datap [3]
      movi r1, strob_L [2]
      out r1, controlp [3]
      movi r1, s strob_H [2]
      out r1, controlp [3]
      addi r5, r5, 2 [3]
      subi r7, r7, 1 [3]
      jnz r7, [again] [4]
      halt
```

The execution time for these two instructions is $2+3 = 5$ clock periods. Therefore, STROBE# stays at logic1 for at least 5 clock periods i.e., during these two instructions. For a 10MHz FALCON-A CPU, this will correspond to $5 \times 100 = 500\text{nsec} = 0.5\mu\text{sec}$.

¹⁸ The I/O device has no direct access to the memory or the CPU, and transfer is generally done by using the CPU registers.

Since the data to the printer is being sent by the CPU using the two instructions (**load r1, [r5]** and **out r1, datap**) which are before the first **movi** instruction, the printer's data setup time requirement is satisfied as long as we do not increase the clock frequency beyond 10MHz.

After these two instructions, the next two instructions in the program cause STROBE# to go to logic 1 again.

```
movi r1, strb_H      [2]
out r1, controlp     [3]
```

These two instructions also take 5 clock periods, or 0.5 μ sec, to execute. Thus, the timing requirement of the STROBE# pulse width will also be satisfied as long as we do not increase the clock frequency beyond 10MHz. In case the frequency is greater than 10MHz, other instruction can be used in between these two pairs of instructions.

The printer's data hold time requirement is easily satisfied because there are a number of instructions after this **out** instruction which do not change the control port, and the character value is already present in the data register within the interface since the end of the **out r1, datap** instruction.

2. The three instructions given below:

```
again: in r1, statusp [3]
       and r1, r1, r3 [3]
       jnz r1, [again] [4]
```

form what is called a “polling loop”. The process of periodically checking the status of a device to see if it is ready for the next I/O operation is called “polling”. It is the simplest way for an I/O device to communicate with the CPU. The device indicates its readiness by setting certain bits in a status register, and the CPU can read these bits to get information about the device. Thus, the CPU does all the work and controls all the I/O activities. The polling loop given above takes 10 clock periods. For a 10MHz FALCON-A CPU, this is $10 \times 100 = 1\mu\text{sec}$. One pass of the main loop takes a total of $3+3+4+5+3+2+3+2+3+3+3+4 = 38$ clock periods which is $38 \times 100 = 3.8\mu\text{sec}$. This is the time that the CPU takes to send one character to the printer. If we assume that a 1000 character per second (cps) printer is connected to the CPU, then this printer has the capability to print one character in every 1msec or every 1000 μsec . So, after sending a character in 3.8 μsec to the printer, the CPU will wait for about 996 μsec before it can send the next character to the printer. This implies that the polling loop will be executed about 996 times for each character. This is indeed a very inefficient way of sending characters to the printer.

An improved way of doing this would be to include a memory of suitable size within the printer. This memory is also called a buffer, as explained earlier. The CPU can fill this buffer in a single “burst” at its own speed, and then do something else, while the printer picks up one character at a time from this buffer and prints it at its own speed. This is exactly the situation with today's printers. The task of generating the STROBE# pulse will also be done by the electronic circuits within the printer. In effect, a dedicated processor within the printer will do this job. However, if the buffer within the printer fills up, the CPU will still not be able to transfer additional data to it. A different handshaking scheme will then be needed to make the CPU to communicate asynchronously with the buffer in the printer, resulting in an inefficient operation again. This is explained below.

Assume that the printer has a FIFO type buffer of size 64 bytes that can be filled up without any delay at the time when the printer is not printing anything. When one or more character values are present in the buffer, the printer will pick up one value at a time and print it. Remember we have a 1000 cps printer, so it takes 1msec to print a character. The program for Example #1(lec26) is modified for this situation and is given below. All the assumptions are the same, unless otherwise mentioned.



```
again:   in r1, statusp      [3]
         and r1, r1, r3      [3]
         jnz r1, [again] [4]
         load r1, [r5]       [5]
         out r1, datap       [3]
         addi r5, r5, 2      [3]
         subi r7, r7, 1      [3]
         jnz r7, [again] [4]
```

Note that while the instructions for generating the STROBE# pulse have been eliminated, the polling loop is still there. This is necessary because the BUSY signal will still be present, although it will have a different meaning now. In this case, BUSY =1 will mean that the buffer within the printer is full and it cannot accept additional bytes.

The main loop shown in the program has an execution time of 28 clock periods, which is 2.8µsec for a 10MHz FALCON-A CPU. The polling loop still takes 10 clock periods or 1µsec. Assuming that this program starts when the buffer in the printer is empty, the outer loop will execute 64 times before the CPU encounters a BUSY=1 condition. After that the situation will be the same as in the previous case. The polling loop will execute for about 996 times before BUSY goes to logic 0. This situation will persist for the remaining 16 characters (remember we are sending an 80 character line to the printer).

One can argue that the problem can be solved by increasing the buffer size to more than 80 bytes. Well, first of all, memory is not free. So, a large buffer will increase the cost of the printer. Even if we are willing to pay more for an improved printer, the larger buffer will still fill up whenever the number of characters is more than the buffer size. When that happens, we will be back to square one again.

A careful analysis of the situation reveals that there is something wrong with the scheme that is being used to send data to the printer. This problem of having a larger overhead of polling was recognized long ago, and therefore, interrupts were invented as an alternate to programmed I/O. Interrupt driven I/O will be the topic of the next lecture.

Programmed I/O in SRC

In this section, we will discuss some more examples of programmed I/O with our example processor SRC which uses the memory mapped I/O technique.

Program for Character Output

To understand how programmed I/O works in SRC, we will discuss a program which outputs the character to the printer. The first instruction loads the branch target and the second instruction loads the character into lower 8 bits of register r2. The 2-instruction loop reads the status register and tests the ready signal by checking its sign bit. It executes until the ready signal becomes logic one. On exit from the loop, the character is written to the device data register by the store instruction.

```

        lar r3, wait
        ldr r2, char
wait:   ld r1, COSTAT
        brpl r3, r1
        st r2, COUT
    
```

A 10 MIPS, SRC would execute 10,000 instructions waiting for a 1,000 character/sec printer.

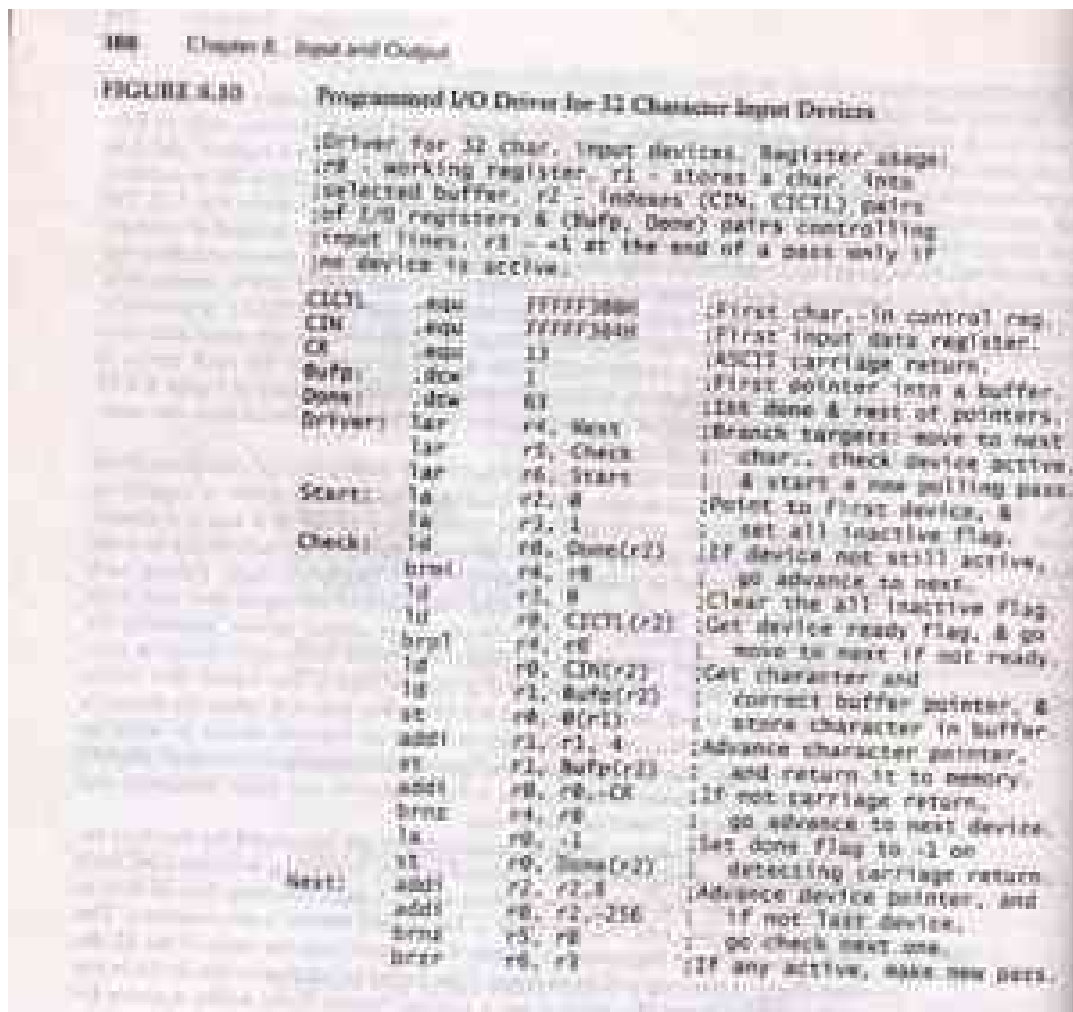
Program Fragment to Print 80-Character Line

The next example for the SRC is of a program which sends an 80-character line to a line printer with a command register. There are two nested loops starting at label wait. The two instruction inner loop, which waits for ready and the outer seven instruction loop which performs the following tasks.

- Outputs a character
- Advance the buffer pointer
- Decrement the register containing the number of characters left to print
- Repeat if there are more characters left to send.

The last two instructions issue the command to print the line.

The next example discussed from the book is of a driver program for 32-character input devices (Figure 8.10, Page 388).



Comparisons of the SRC and FALCON-A Examples

The FALCON-A and SRC programmed I/O examples discussed are similar with some differences. In the first example discussed for the SRC (i.e. Character output), the control signal responsible for data transfer by the CPU is the ready signal while for FALCON-A Busy (active low) signal is checked. In the second example for the SRC, the instruction set, address width and no. of lines on address is different.

Although different techniques have been used to increase the efficiency of the programmed I/O, overheads due to polling cannot be completely eliminated.

Lecture No. 27

Interrupt Driven I/O

Reading Material

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

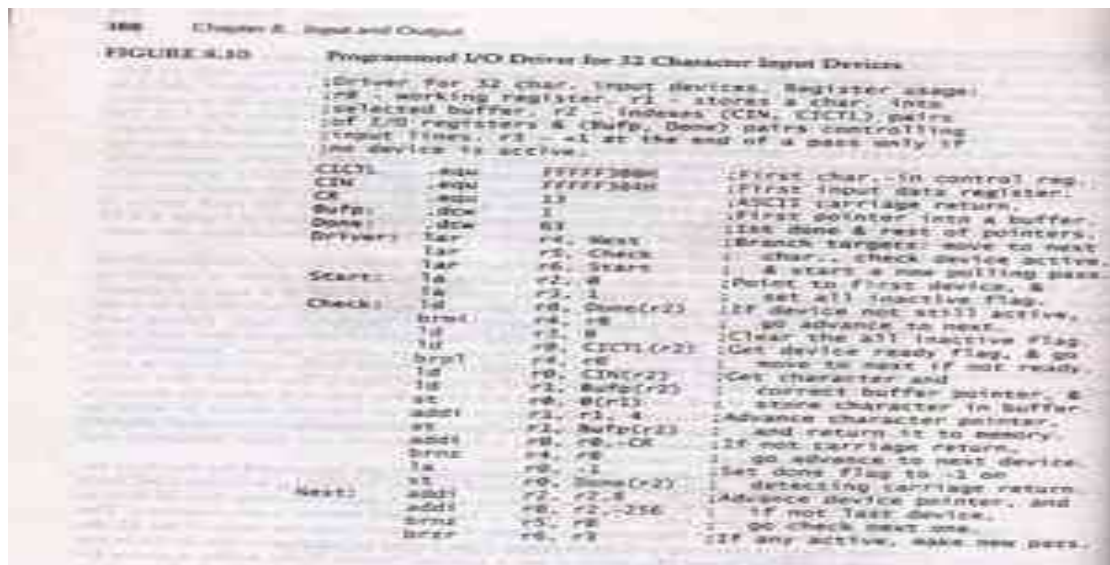
Chapter 8
8.2.2

Summary

- Programmed I/O Driver for SRC
- Interrupt Driven I/O

Programmed I/O Driver for SRC

Please refer to Figure 8.10 of the text and its associated explanation.

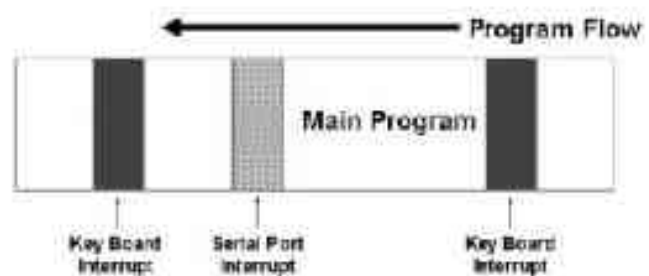


Interrupt Driven I/O:

Introduction:

An interrupt is a request to the CPU to suspend normal processing and temporarily divert the flow of control through a new program. This new program to which control is transferred is called an **Interrupt Service Routine** or **ISR**. Another name for an ISR is an **Interrupt Handler**.

Program Flow



- Interrupts are used to demand attention from the CPU.
- Interrupts are asynchronous breaks in program flow that occur as a result of events outside the running program.
- Interrupts are usually hardware related, stemming from events such as a key or button press, timer expiration, or completion of a data transfer.

The basic purpose of interrupts is to divert CPU processing only when it is required. As an example let us consider the example of a user typing a document on word-processing software running on a multi-tasking operating system. It is up to the software to display a character when the user presses a key on the keyboard. To fulfill this responsibility the processor can repeatedly poll the keyboard to check if the user has pressed a key. However, the average user can type at most 50 to 60 words in a minute. The rate of input is much slower than the speed of the processor. Hence, most of the polling messages that the processor sends to the keyboard will be wasted. A significant fraction of the processor's cycles will be wasted checking for user input on the keyboard. It should also be kept in mind that there are usually multiple peripheral devices such as mouse, camera, LAN card, modem, etc. If the processor would poll each and every one of these devices for input, it would be wasting a large amount of its time. To solve this problem, interrupts are integrated into the system. Whenever a peripheral device has data to be exchanged with the processor, it interrupts the processor; the processor saves its state and then executes an interrupt handler routine (which basically exchanges data with the device). After this exchange is completed, the processor resumes its task. Coming back to the keyboard example, if it takes the average user approximately 500 ms to press consecutive keys a modern processor like the Pentium can execute up to 300,000,000 instructions in these 500 Ms. Hence, interrupts are an efficient way to handle I/O compared to polling.

Advantages of interrupts:

- Useful for interfacing I/O devices with low data transfer rates.
- CPU is not tied up in a tight loop for polling the I/O device.

Program Flow for an interrupt driven interface:

The attached figure shows the program flow executing on a processor with interrupts enabled. As we can see, the program is interrupted in several locations to service various types of interrupts.

Types of Interrupts:

The general categories of interrupts are as follows:

- Internal Interrupts
- External Interrupts
- Hardware Interrupts
- Software Interrupts

Internal Interrupts:

- Internal interrupts are generated by the processor.
- These are used by processor to handle the exceptions generated during instruction execution.

Internal interrupts are generated to handle conditions such as stack overflow or a divide-by-zero exception. Internal interrupts are also referred to as traps. They are mostly used for exception handling. These types of interrupts are also called exceptions and were discussed previously.

External Interrupts:

External interrupts are generated by the devices other than the processor. They are of two types.

- Hardware interrupts are generated by the external hardware.
- Software interrupts are generated by the software using some interrupt instruction.

As the name implies, external interrupts are generated by devices external to the CPU, such as the click of a mouse or pressing a key on a keyboard. In most cases, input from external sources requires immediate attention. These events require a quick service by the software, e.g., a word processing software must quickly display on the monitor, the character typed by the user on the keyboard. A mouse click should produce immediate results. Data received from the LAN card or

the modem must be copied from the buffer immediately so that pending data is not lost because of buffer overflow, etc.

Hardware interrupts:

Hardware interrupts are generated by external events specific to peripheral devices. Most processors have at least one line dedicated to interrupt requests. When a device signals on this specific line, the processor halts its activity and executes an interrupt service routine. Such interrupts are always asynchronous with respect to instruction execution, and are not associated with any particular instruction. They do not prevent instruction completion as exceptions like an arithmetic overflows does. Thus, the control unit only needs to check for such interrupts at the start of every new instruction. Additionally, the CPU needs to know the identification and priority of the device sending the interrupt request.

There are two types of hardware interrupt:

Maskable Interrupts

Non-maskable Interrupts

Maskable Interrupts:

- These interrupts are applied to the INTR pin of the processor.
- These can be blocked by resetting the flag bit for the interrupts.

Non-maskable Interrupts:

- These interrupts are detected using the NMI pin of the processor.
- These can not be blocked or masked.
- Reserved for catastrophic event in the system.

Software interrupts:

Software interrupts are usually associated with the software. A simple output operation in a multitasking system requires software interrupts to be generated so that the processor may temporarily halt its activity and place the data on its data bus for the peripheral device. Output is usually handled by interrupts so that it appears interactive and asynchronous. Notification of other events, such as expiry of a software timer is also handled by software interrupts. Software



interrupts are also used with system calls. When the operating system switches from user mode to supervisor mode it does so through software interrupts. Let us consider an example where a user program must delete a file. The user program will be executing in the user mode. When it makes the specific system call to delete the file, a software interrupt will be generated, this will cause the processor to halt its current activity (which would be the user program) and switch to supervisor mode. Once in supervisor mode, the operating system will delete the file and then control will return to the user program. While in supervisor mode the operating system would need to decide if it could delete the specified file without harmful consequences to the systems integrity, hence it is important that the system switch to supervisor mode at each system call.

I/O Software System Layers:

The above diagram shows the various software layers related to I/O. At the bottom lies the actual hardware itself, i.e. the peripheral device. The peripheral device uses the hardware interrupts to communicate with the processor. The processor responds by executing the interrupt handler for that particular device. The device drivers form the bridge between the hardware and the software. The operating system uses the device drivers to communicate with the device in a hardware independent fashion, e.g., the operating system need not cater for a specific brand of CRT monitors, or keyboards, the specific device driver written for that monitor or keyboard will act as an intermediary between the operating system and the device. It would be clear from the previous statement that the operating system expects certain common functions from all brands of devices in a category. Actually implementing these functions for each particular brand or vendor is the responsibility of the device driver. The user programs run at top of the operating system.

Interrupt Service Routine (ISR):

- It is a routine which is executed when an interrupt occurs.
- Also known as an Interrupt Handler.
- Deals with low-level events in the hardware of a computer system, like a tick of a real-time clock.

As it was mentioned earlier, an interrupt once generated must be serviced through an interrupt service routine. These routines are stored in the system memory ready for execution. Once the interrupt is generated, the processor must branch to the location of the appropriate service routine to execute it. The branch address of the ISR is discussed next.

Branch Address of the ISR:

There are two ways used to choose the branch address of an Interrupt Service Routine.

Non-vectored Interrupts Vectored Interrupts

Non-vectored Interrupts:

In non-vectored interrupts, the branch address of the interrupt service routine is fixed. The code for the ISR is loaded at fixed memory location. Non-vectored interrupts are very easy to implement and not flexible at all. In this case, the number of peripheral devices is fixed and may not be increased. Once the interrupt is generated the processor queries each peripheral device to find out which device generated the interrupt. This approach is the least flexible for software interrupt handling.

Vectored Interrupts:

Interrupt vectors are used to specify the address of the interrupt service routine. The code for ISR can be loaded anywhere in the memory. This approach is much more flexible as the programmer may easily locate the interrupt vector and change its addresses to use custom interrupt servicing routines. Using vectored interrupts, multiple devices may share the same interrupt input line to the processor. A process called daisy chaining is then used to locate the interrupting device.

Interrupt Vector:

Interrupt vector is a fixed size structure that stores the address of the first instruction of the ISR.

Interrupt Vector Table:

- All of the interrupt vectors are stored in the memory in a special table called Interrupt Vector Table.
- Interrupt Vector Table is loaded at the memory location 0 for the 8086/8088.

Interrupts in Intel 8086/8088:

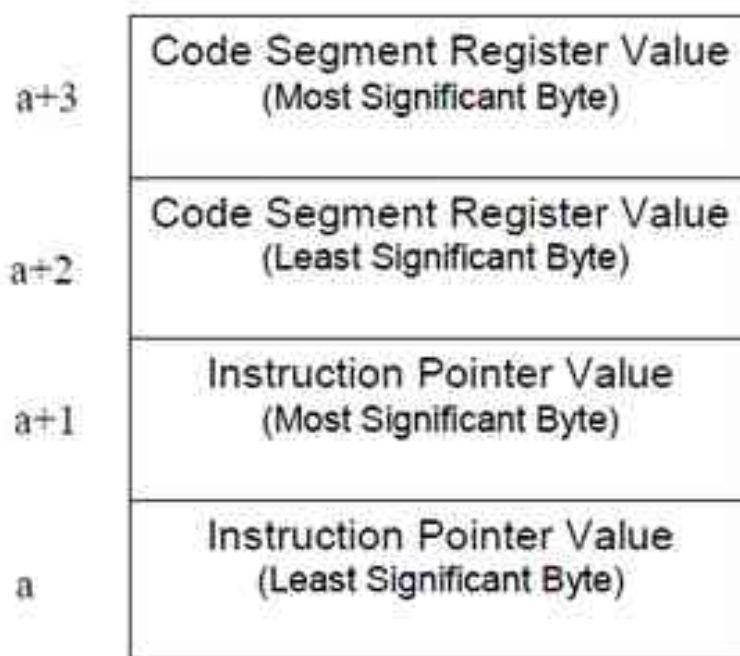
- Interrupts in 8086/8088 are vector interrupts.
- Interrupt vector is of 4 bytes to store IP and CS.
- Interrupt vector table is loaded at address 0 of main memory.
- There is provision of 256 interrupts.

Branch Address Calculation:

- The number of interrupt is the number of interrupt vector in the interrupt vector table.
- Since size of each vector is 4 bytes and interrupt vector starts from address 0, therefore, the address of interrupt vector can be calculated by simply multiplying the number by 4.

Interrupt Vector Example:

In 8086/8088 machines the size of interrupt vector is 4 bytes that holds IP and CS of ISR.



Returning from the ISR:

Every ISA should have an instruction, like the **IRET** instruction, which should be executed when the ISR terminates. This means that the **IRET** instruction should be the last instruction of every ISR. This is, in effect, a FAR RETURN in that it restores a number of registers, and flags to their value before the ISR was called. Thus the previous environment is restored after the servicing of the interrupt is completed.

Interrupt Handling:

The CPU responds to the interrupt request by completing the current instruction, and then storing the return address from PC into a memory stack. Then the CPU branches to the ISR that processes the requested operation of data transfer. In general, the following sequence takes place.

Hardware Interrupt Handling:

- Hardware issues interrupt signal to the CPU.
- CPU completes the execution of current instruction. CPU acknowledges interrupt.
- Hardware places the interrupt number on the data bus.
- CPU determines the address of ISR from the interrupt number available on the data bus.

- CPU pushes the program status word (flags) on the stack along with the current value of program counter.
- The CPU starts executing the ISR.
- After completion of the ISR, the environment is restored; control is transferred back to the main program.

Interrupt Latency:

Interrupt Latency is the time needed by the CPU to recognize (not service) an interrupt request. It consists of the time to perform the following:

- Finish executing the current instruction.
- Perform interrupt-acknowledge bus cycles.
- Temporarily save the current environment.
- Calculate the IVT address and transfer control to the ISR.

If wait states are inserted by either some memory module or the device supplying the interrupt type number, the interrupt latency will increase accordingly.

Interrupt Latency for external interrupts depends on how many clock periods remain in the execution of the current instruction.

On the average, the longest latency occurs when a multiplication, division or a variable-bit shift or rotate instruction is executing when the interrupt request arrives.

Response Deadline:

It is the maximum time that an interrupt handler can take between the time when interrupt was requested and when the device must be serviced.

Expanding Interrupt Structure:

When there is more than one device that can interrupt the CPU, an Interrupt Controller is used to handle the priority of requests generated by the devices simultaneously.

Interrupt Precedence:

Interrupts occurring at the same time i.e. within the same instruction are serviced according to a pre-defined priority.

- In general, all internal interrupts have priority over all external interrupts; the single-step interrupt is an exception.
- **NMI** has priority over **INTR** if both occur simultaneously.
- The above mentioned priority structure is applicable as far as the recognition of (simultaneous) interrupts is concerned. As far as servicing (execution of the related ISR) is concerned, the single-step interrupt always gets the highest priority, then the **NMI**, and finally those (hardware or software) interrupts that occur last. If **IF** is not 1, then **INTR** is ignored in any case. Moreover, since any ISR will clear **IF**, **INTR** has lower "service priority" compared to software interrupts, unless the ISR itself sets **IF**=1.

Simultaneous Hardware Interrupt Requests:

The priority of the devices requesting service at the same time is resolved by using two ways:
Daisy-Chained Interrupt Parallel Priority Interrupt

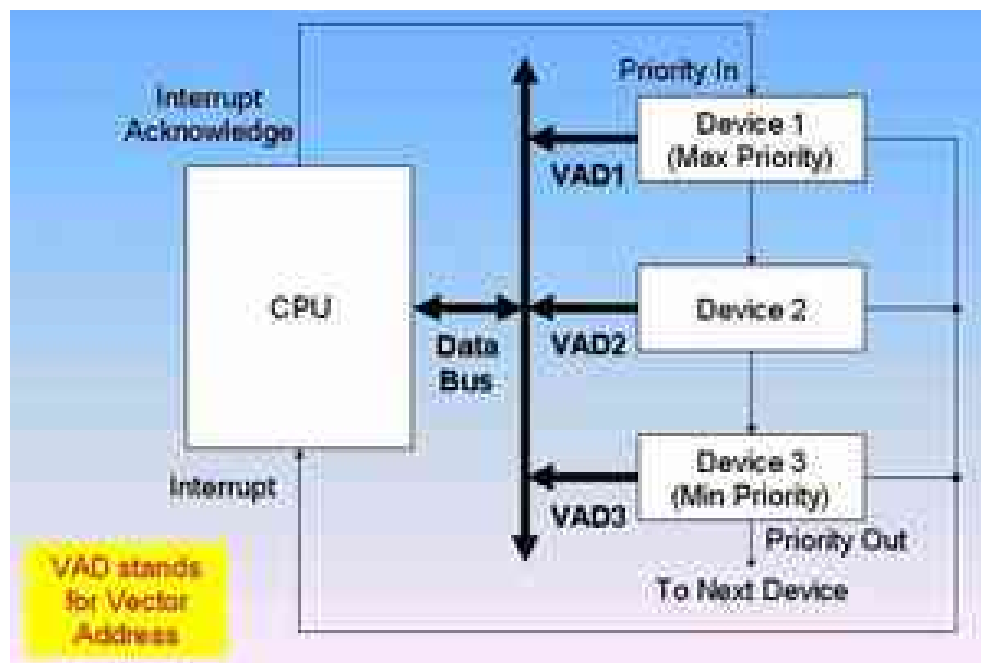
Daisy-Chaining Priority:

- The daisy-chaining method to resolve the priority consists of a series connection of the devices in order of their priority.
- Device with maximum priority is placed first and device with least priority is placed at the end.

Daisy-Chain Priority Interrupt

- The devices interrupt the CPU.
- The CPU sends acknowledgement to the maximum priority device.
- If the interrupt was generated by the device, the interrupt for the device is serviced.
- Otherwise the acknowledgement is passed to the next device.

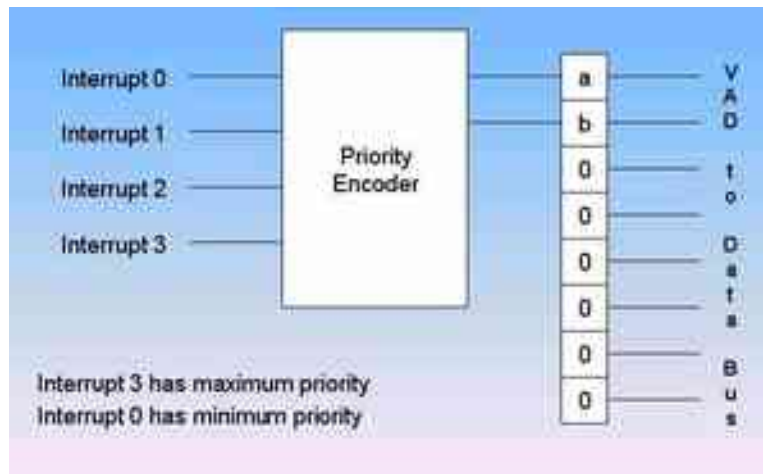
If the higher priority devices are going to interrupt continuously then the device with the lower priority is not serviced. So some additional circuitry is also needed to introduce fairness.



Parallel Priority:

- Parallel priority method for resolving the priority uses individual bits of a priority encoder.
- The priority of the device is determined by position of the input of the encoder used for the interrupt.

Parallel Priority Interrupt:



Lecture No. 28

Interrupt Hardware and Software

Reading Material

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

Chapter 8
8.3

Summary

- Comparison of Interrupt driven I/O and Polling
- Design Issues
- Interrupt Handler Software
- Interrupt Hardware
- Interrupt Software

Comparison of Interrupt driven I/O and Polling

Interrupt driven I/O is better than polling. In the case of polling a lot of time is wasted in questioning the peripheral device whether it is ready for delivering the data or not. In the case of interrupt driven I/O the CPU time in polling is saved.

Now the design issues involved in implementation of the interrupts are twofold. There would be a number of interrupts that could be initiated. Once the interrupt is there, how the CPU does know which particular device initiated this interrupt. So the first question is evaluation of the peripheral device or looking at which peripheral device has generated the interrupt. Now the second important question is that usually there would be a number of interrupts simultaneously available. So if there are a number of interrupts then there should be a mechanism by which we could just resolve that which particular interrupt should be serviced first. So there should be some priority mechanism.

Design Issues

There are two design issues:

1. Device Identification
2. Priority mechanism

Device Identification

In this issue different mechanisms could be used.

- Multiple interrupt lines
- Software Poll
- Daisy Chain

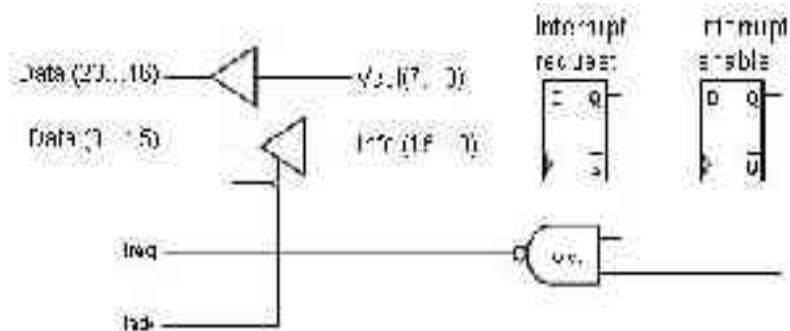
1. Multiple Interrupt Line

This is the most straight forward approach, and in this method, a number of interrupt lines are provided between the CPU and the I/O module. However, it is impractical to dedicate more than a few bus lines or CPU pins to interrupt lines. Consequently, even if multiple lines are used, it is

likely that each line will have multiple I/O modules attached to it. Thus on each line, one of the other technique would still be required.

2. Software Poll

CPU polls to identify the interrupting module and branches to an interrupt service routine on detecting an interrupt. This identification is done using special commands or reading the device status register. Special command may be a test I/O. In this case, CPU raises test I/O and places the address of a particular I/O module on the address line. If I/O module sets the interrupt then it responds positively. In the case of an addressable status register, the CPU reads the status register of each I/O module to identify the interrupting module. Once the correct module is identified, the CPU branches to a device service routine which is specific to that particular device.



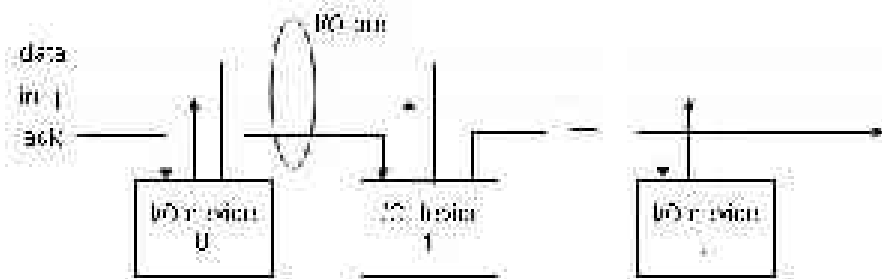
Simplified Interrupt Circuit for an I/O Interface

For above two techniques the implementation might require some hardware. The hardware would be specific to the processor which is being used. For example, for the case of SRC, simple hardware mechanism is indicated. Now the basic technique is handshaking and in this case of handshaking, the peripheral device would initiate an interrupt. This interrupt needs to be enabled. We will have a mechanism of ANDing the two signals. One is interrupt enable and other is interrupt request. Now these two requests would be passed on the CPU. The CPU passes on the acknowledge signal to the device. The acknowledge signal is shared and it goes on to different devices.

The information about interrupt vector is given in 8-bits, from bit 0 to 7, which is translated to bit 16 to 23 on the data bus. Now the other 16-bits, from 0 to 15 are mapped to the data lines from 0 to 15. Now both of these are available through the tri-state buffers, which would be enabled through interrupt acknowledge.

3. Daisy Chain

The wired or interrupt signal allows several devices to request interrupt simultaneously. However, for proper operation one and only one requesting device must receive an acknowledge signal, otherwise if we have more than one devices, we would have a data bus contention and the



interrupt information would not be resolved. The usual solution is called a daisy chain. Assuming that if we have j th devices requesting for interrupt then first device 0 would receive the acknowledge signal, so therefore, $iack_0 = iack$. The next device would only receive an acknowledge i.e., the j th device would receive an acknowledge if the previous device that means $j-1$ does not have an enabled interrupt request, that means interrupt was not initiated by the

previous device. Now the figure shows this concept in the form of a connection from device 0 to 1. From 0, we see the acknowledge is generated for device 1, device 1 generates acknowledge for device2 and so on. So this signal propagates from one device to other device. Logically we could write it in the form of equation:

$$iack_j = iack_{j-1} \wedge (req_{j-1} \wedge enb_{j-1})$$

As we said that the previous device should not have generated an interrupt, that means its interrupt was not enabled and therefore, it passes on the acknowledge signal from its output to the next device.

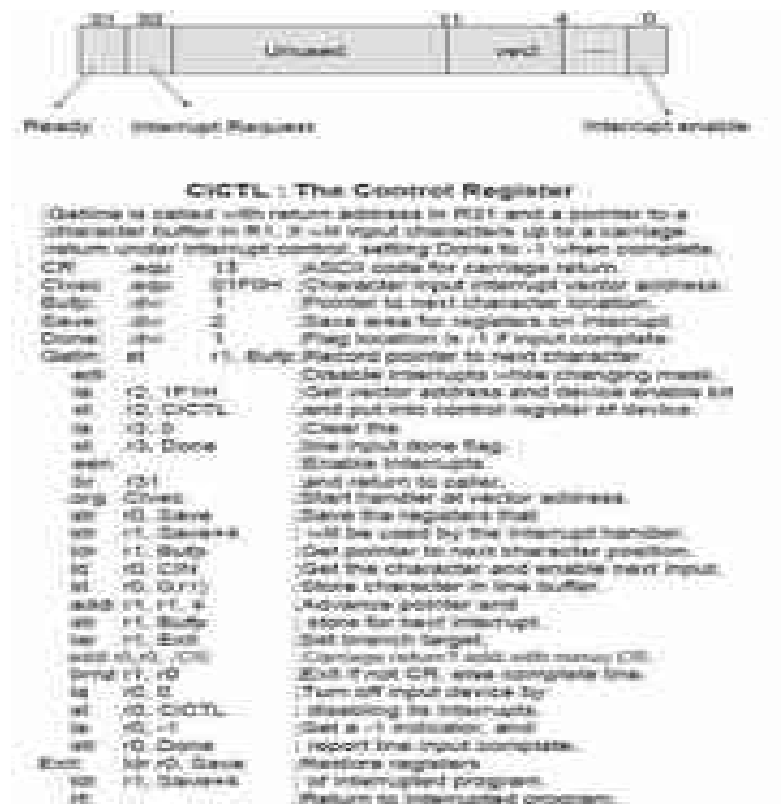
Disadvantages of Software Poll and Daisy Chain

The software poll has a disadvantage is that it consumes a lot of time, while the daisy chain is more efficient. The daisy chain has the disadvantage that the device nearest to the CPU would have highest priority. So, usually those devices which require higher priority would be connected nearer to the CPU. Now in order to get a fair chance for other devices, other mechanisms could be initiated or we could say that we could start instead of device 0 from that device where the CPU finishes the last interrupt and could have a cyclic provision to different devices.

Interrupt Handler Software

Example using SRC

(Read from Book, Jordan page395)



Example using FALCON-A

As an example of interrupt-driven I/O, consider an output device, such as a parallel printer connected to the FALCON-A CPU. Now suppose that we want to print a document while using an application program like a word processor or a spread sheet. In this section, we will explain the important aspects of hardware and software for implementing an interrupt driven parallel printer interface for the FALCON-A. During this discussion, we will also explain the differences and similarities between this interface and the one discussed earlier. To make things simple, we have made the assumption that only one interrupt pin is available on the FALCON-A, and only one interrupt is possible at a given time with this CPU. Implications of allowing only one interrupt at a time are that

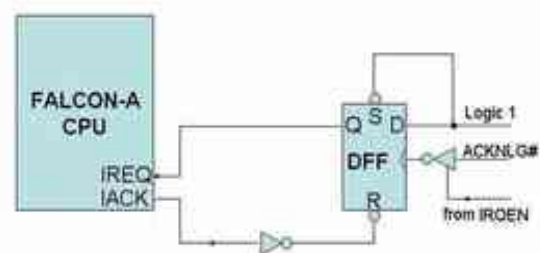
- No NMI is possible
- No nesting of interrupts is possible
- No priority structure needed for multiple devices
- No arbitration needed for simultaneous interrupts
- No need for vectored interrupts, therefore, no need of interrupt vectors and interrupt vector tables
- Effect of software initiated interrupts and internal interrupts (exceptions) has to be ignored in this discussion

Along with the previous assumption, the following assumptions have also been used:

- Hardware sets and clears the interrupt flag, in addition to handling other things like saving PC, etc.
- The address of the ISR is stored at absolute address 2 in memory.
- The ISR will set up a stack in the memory for saving the CPU's environment
- One ASCII character stored per 16-bit word in the FALCON-A's memory and one character transferred during a 16-bit transfer.
- The calling program will call the ISR for printing the first character through the printer driver.
- Printer will activate ACKNLG# only when not BUSY.

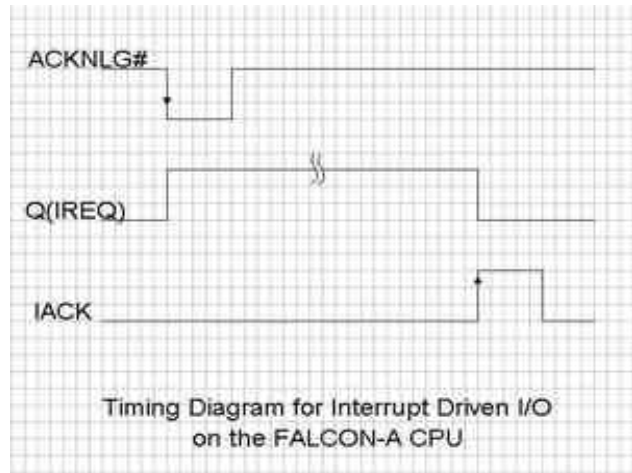
Interrupt Hardware:

The logic diagram for the interrupt hardware is shown in the Figure. The interrupt request is synchronized by handshaking signals, called IREQ and IACK. The timing diagram for the handshaking signals used in the interrupt driven I/O is shown in the next Figure. The printer will assert IREQ as soon as the ACKNLG# signal goes low (i.e. as soon as the printer is ready to accept new data) provided that IREQN=1. The processor will complete the current instruction and respond by executing the interrupt service routine. The inverting tri-state buffer at the clock input of the D flip flop is enabled by IRQEN. This will make sure that after the current print job is complete, additional requests on IREQ are disabled. This can happen as a result of the printer being available even though the user may not have requested a print operation. The IACK line from the CPU is connected to the asynchronous reset, R, of the D flip flop so that the same interrupt request from the printer is not presented again to the CPU. The asynchronous set input of the D flip flop, labeled S in the diagram, is permanently connected to logic 1.



Logic Diagram for Interrupt Driven I/O on the FALCON-A CPU

This will make sure that the flip flop will never be set asynchronously. The D input is also permanently connected to logic 1, as a result of which the flip flop will always be set synchronously in response to ACKNLG# provided IRQEN=1. Recall that IRQEN is bit 4 on the centronics control port at logical address 2, and this is mapped onto address 60 of the FALCON-A's I/O space. The rest of the hardware is case of the same as in the case of the programmed I/O example.

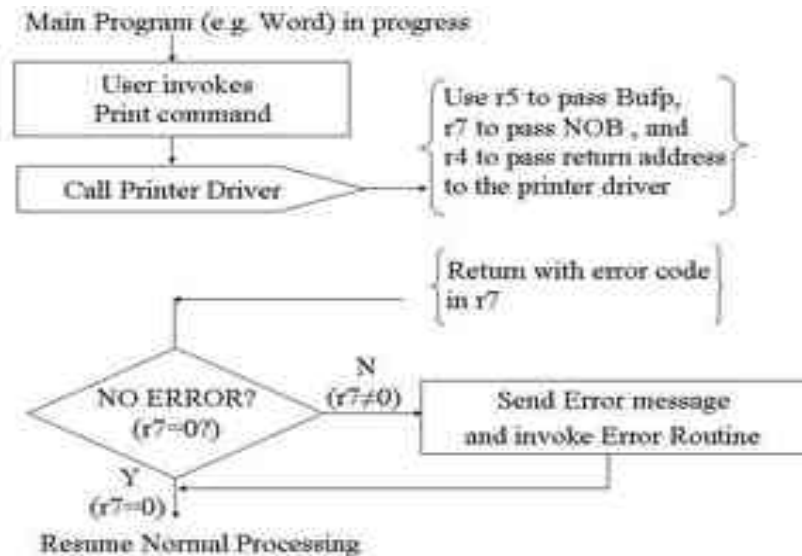


Interrupt Software:

Our software for the interrupt driven printer example consists of three parts:

- 1). Dummy calling program
- 2). Printer Driver
- 3). ISR

We are assuming that normal processing is taking place¹⁹ e.g., a word processor is executing. The user wants to print a document. This document is placed in a buffer by the word processor. This buffer is usually present somewhere else in the memory. The responsibility of the calling



program is to pass the number of bytes to be printed and the starting address of the buffer where these bytes are stored to the printer driver. The calling program can also be called the main program.

¹⁹ Since only one interrupt is possible, a question may arise about the way the print command is presented to the word processor. It can be assumed that polling is used for the input device in this case.

Printer: Example: Printer A prints
 This program sends a single character
 to a FALCON-A parallel printer
 using an interrupt-driven I/O interface

Notes:

1. 8-bit printer data bus connected to
 Dr 7, 6 of the FALCON-A connector (top-right).
 Thus, the printer actually uses addresses 77, 78 & 79.
2. one character per 16-bits of data shifted.

```

.org 0
main [main]
4400:    .w 1024
4401:    .w 40
Bufp:    .w 1
NOB:     .w 1
PB:      .w 1
temp:    .w 1
    
```

Dummy Calling Program: e.g., a word processor

```

.org 12
main:    load r4, [4400]    ; r4 holds address of printer driver
; over 16-bits print command here
    load r5, [Bufp]        ; Printer registers for printing
    load r7, [NOB]         ; information about print buffer
;
; call printer driver
    call r4, r5
; Handle error conditions, if any - open return
; Normal processing continues
    halt
    
```

Suppose that the total number of bytes to be printed are 40. (They are placed in a buffer having the starting address 1024.) When the user invokes the print command, the calling program calls the printer driver and passes these two parameters in r7 and r5 respectively. The return address of the calling program is stored in r4. A dummy calling program code is given below. Bufp, NOB, PB, and temp are the spaces reserved in memory for later use in the program. The first instruction is **jump [main]**. It is stored at absolute memory address 0 by using the **.org 0** directive. It will transfer control to the main program. The first instruction of the main program is

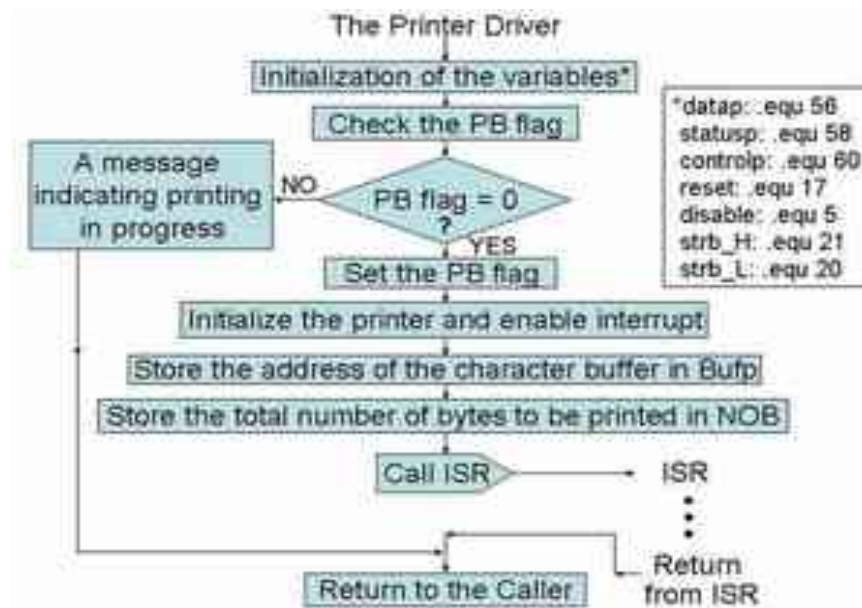
placed at address “**main**”, which is the entry point in this example. Note that the entry point is different in this case from the reset address, which is address 0 for the FALCON-A. Also note that the address of the first instruction in the printer driver is stored at address “**a4PD**” using the **.sw** directive. This value is then brought into r6. The main program calls the printer driver by using the instruction **call r4, r6**. In an actual program, after returning from the printer driver, the normal processing resumes and if there are any error conditions, they will be handled at this point. Next, consider the code for the printer driver, shown in the attached text box.

The printer driver is loaded at address 50. Initialization of the variables includes setting of port addresses, variables for the STROBE# pulse, initializing the printer and enabling its IRQEN. The variables can be defined anywhere in the program because they reserve no memory space. When the printer driver starts, the PB flag is tested to make sure that a previous print job is not in progress. If so, the ISR is not invoked and a message is returned to the main program indicating that printing is in progress. This may display a “printer busy” icon on the user’s screen, or cause some other appropriate action. If the printer is available, it is initialized by the driver.

The following activities are also performed by the driver (see the attached flow chart also).

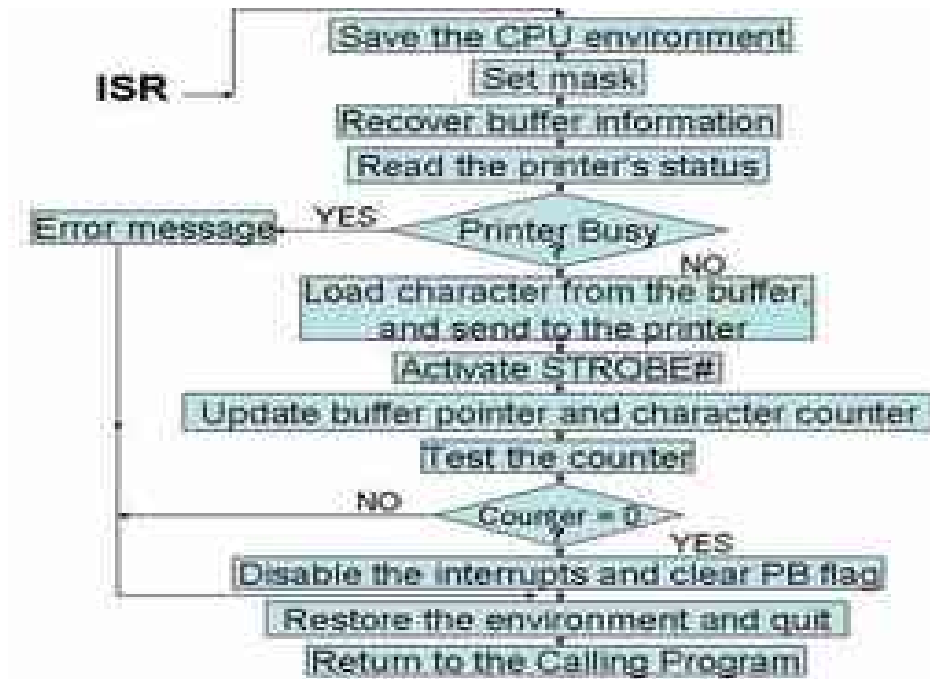
- Set port addresses
- Set up variables for the STROBE# puls
- Initialize printer and enable its IRQEN.
- Set up printer ISR by pointing to the buffer and initializing counter
- Make sure that the previous print job is not in progress
- Set PB flag to block further print jobs till current one is complete
- Invoke ISR for the first time
- Pass error message to main program if ISR reports an error
- Return to main program

The code and flow chart for the interrupt service routine (ISR) are discussed in the next few paragraphs.



; Printer driver		
	org 54	; starting address of Printer driver
;		
data:	org 54	
status:	org 58	
control:	org 60	
;		
start:	org 17	; at 17h
; read to set initialization, enable interrupts,		
; enable line feed, and enable high		
disable:	org 3	
;		
with_10:	org 23	; at 12h
with_11:	org 26	; at 14h
;		
; check PR flag here, if set,		
; return with message.		
;		
Printer: load r1, [PR]		
jnz r1, [message]		
mov r1, 1		
store r1, [PR]		; r1 is PR indicates Print in Progress
mov r1, start		; use r1 for data after
out r1, control		
store r1, [data]		
store r1, [NPR]		
;		
hlt		
;		
jump [finish]		
message: org		; in actual situation, put a message routine here
; to indicate print in progress		
finish: ret r1		

We have assumed that the address of the ISR is stored at absolute memory address 2 by the operating system. One way to do that is by using the **.sw** directive (as done in the dummy calling program). The symbol **sw** stands for “storage of word”. It enables the user to identify storage for a constant, or the value of a variable, an address or a label at a fixed memory location during the assembly process.



These values become part of the binary file and are then loaded into the memory when the binary file is loaded and executed. In response to a hardware interrupt or the software interrupt **int**, the control unit of the FALCON-A CPU will pick up the address of the first instruction in the ISR from memory location 2, and transfer control to it. This effectively means that the behavioral RTL of the **int** instruction will be as shown below:

int $IPC \leftarrow PC, PC \leftarrow M[2], IF \leftarrow 0$

The IPC register in the CPU is a holding place for the current value of the PC. It is invisible to the programmer. Since the **iret** instruction should always be the last instruction in every ISR, its behavior RTL will be as shown below:

iret $PC \leftarrow IPC, IF \leftarrow 1$

The saving and restoring of the other elements of the CPU environment like the general purpose registers should be done within the ISR. The five **store** instructions at the beginning are used to save these registers into the memory block starting at address **temp**, and the five **load** instructions at the end are used to restore these registers to their original values.

[illegible]

After setting the mask to 80h in r3, the current value of the buffer pointer and the number of bytes to be printed are brought from the memory into r5 and r7 respectively. After a byte is printed, these values are updated in the memory for use by the ISR when it is invoked again. The rest of the code in the ISR is the same as it was in case of the programmed I/O example. Note that we are testing the printer's BUSY flag within the ISR also. However, the difference here is

that this testing is being done for a different reason, and it is done only once for each call to the **ISR**.

Memory Map for our ISR	
0	Entry Point
2	Address of ISR
4	Data and Pointer Area
32	Main Program (Dummy Calling Program)
50	Printer Driver
100	ISR
	⋮
1024	Print Buffer
	⋮

The memory map for this program is as shown in the Figure. The point to be noted here is that the ISR can be loaded anywhere in the memory but its address will be present at memory location 2 i.e. $M[2]$.

Lecture No. 29

FALSIM

Reading Material

Handouts

Slides

Summary

- Introduction to FALSIM
- Preparing source files for FALSIM
- Using FALSIM
- FALCON-A assembly language techniques

Introduction to FALSIM:

FALSIM is the name of the software application which consists of the FALCON-A assembler and the FALCON-A simulator. It runs under Windows XP.

FALCON-A Assembler:

Figure 1 shows a snapshot of the graphical user interface (GUI) for the FALCON-A Assembler. This tool loads a FALCON-A assembly file with a (.asmfa) extension and parses it. It shows the parsed results in an error log, lets the user view the assembled file's contents in the file listing and also provides the features of printing the machine code, an Instruction Table and a Symbol Table to a FALCON-A listing file. It also allows the user to run the FALCON-A Simulator.

The FALCON-A Assembler source code has two main modules, the 1st-pass module and the 2nd-pass module. The 1st-pass module takes an assembly file with a (.asmfa) extension and processes the file contents. It then generates a Symbol Table which corresponds to the storage of all program variables, labels and data values in a data structure at the implementation level. The Symbol Table is used by the 2nd-pass module. Failures of the 1st-pass are handled by the assembler using its exception handling mechanism.

The 2nd-pass module sequentially processes the .asmfa file to interpret the instruction op-codes, register op-codes and constants using the Symbol Table. It then produces a list file with a .lstfa extension independent of successful or failed pass. If the pass is successful a binary file with a .binfa extension is produced which contains the machine code for the program contained in the assembly file.

FALCON-A Simulator:

Figure 6 shows a snapshot of the GUI for the FALCON-A Simulator. This tool loads a FALCON-A binary file with a (.binfa) extension and presents its contents into different areas of the simulator. It allows the user to execute the program to a specific point within a time frame or just executes it, line by line. It also allows the user to view the registers, I/O port values and memory contents as the instructions execute.

FALSIM Features:

The FALCON-A Assembler provides its user with the following features:

Select Assembly File: Labeled as “1” in Figure 1, this feature enables the user to choose a FALCON-A assembly file and open it for processing by the assembler. *Assembler Options:* Labeled as “2” in Figure 1.

- *Print Symbol Table*

This feature, if selected, writes the Symbol Table (produced after the execution of the 1st-pass of the assembler) to a FALCON-A list file with an extension of (.lstfa). The Symbol Table includes variables, addresses and labels with their respective values.

- **Print Instruction Table**

This feature, if selected, writes the FALCON-A instructions along with their op-codes at the end of the list file.

List File: Labeled as “3”, in Figure 1, the List File feature gives a detailed insight of the FALCON-A listing file, which is produced as a result of the execution of the 1st and 2nd-pass. It shows the Program Counter value in hexadecimal and decimal formats along with the machine code generated for every line of assembly code. These values are printed when the 2nd-pass is completed.

Error Log: The Error Log is labeled as “4” in Figure 1. It informs the user about the errors and their respective details, which occurs in any of the two passes of the assembler. The size of this window can be changed by dragging the boundary line up or down.

Highlight: This feature is labeled as “5” in Figure 1 and helps the user to search for a certain input with the options of searching with “**match whole**” and “**match any**” parts of the string. The search also has the option of checking with/without considering “**case-sensitivity**”. It searches the List File area and highlights the search results using the yellow color. It also indicates the total number of matches found.

Start Simulator: This feature is labeled as “6” in Figure 1. The FALCON-A Simulator is run using the FALCON-A Assembler’s “Start Simulator” option. Its features are detailed as follows:

Load Binary File: The button labeled as “11” in Figure 6, allows the user to choose and open a FALCON-A binary file with a (.binfa) extension. When a file is being loaded into the simulator all the register, constants (if any) and memory values are set.

Registers: The area labeled as “12” in Figure 6, enables, the user to see values present in different registers before, during and after execution.

Instruction: This area is labeled as “13” in Figure 6 and contains the value of PC, address of an instruction, its representation in Assembly, the Register Transfer Language, the op-code and the instruction type.

I/O Ports: I/O ports are labeled as “14” in Figure 6. These ports are available for the user to enter input operation values and visualize output operation values whenever an I/O operation takes place in the program. The input value for an input operation is given by the user before an instruction executes. The output values are visible in the I/O port area once the instruction has successfully executed.

Memory: The memory is divided into two areas and is labeled as “15” in Figure 6, to facilitate the view of data stored at different memory locations before, during and after program execution.

Processor’s State: Labeled as “16” in Figure 6, this area shows the current values of the Instruction Register and the Program Counter while the program executes.

Highlight: The highlight option for the FALCON-A simulator is labeled as “17” in Figure 6. This feature is similar to the way the highlight feature of the FALCON-A Assembler works. It offers to highlight the search string which is entered as an input, with the “All “ and “ Part “ option. The results of the search are highlighted using the yellow color. It also indicates the total number of matches.

The following is a description of the options available on the button panel labeled as “18” in Figure 6.

Single Step: “Single Step” lets the user execute the program, one instruction at a time. The next instruction is not executed unless the user does a “single step” again. By default, the instruction to be executed will be the one next in the sequence. It changes if the user specifies a different PC value using the Change PC option (explained below).

Change PC: This option lets the user change the value of PC (Program Counter). By changing the PC the user can execute the instruction to which the specified PC points. The value in the PC must be an even address.

Execute: By choosing this button, the user is able to execute the loaded program with the options of execution with/without breakpoint insertion. In case of breakpoint insertion, the user has the option to choose from a list of valid breakpoint values. It also has the option to set a limit on the time for execution. This “Max Execution Time” option restricts the program execution to a time frame specified by the user.

Change Register: Using the Change Register feature, the user can change the value present in a particular register.

Change Memory Word: This feature enables the user to change values present at a particular memory location.

Display Memory: Display Memory shows an updated memory area, after a particular memory location other than the pre-existing ones is specified by the user.

Change I/O: Allows the user to give an I/O port value if the instruction to be executed requires an I/O operation. Giving in the input in any one of the I/O ports areas before instruction execution, indicates that a particular I/O operation will be a part of the program and it will have an input from some source. The value given by the user indicates the input type and source.

Display I/O: Display I/O works in a manner similar to Display Memory. Here the user specifies the starting index of an I/O port. This features displays the I/O ports stating from the index specified.

2. Preparing Source Files for FALSIM:

In order to use the FALCON-A assembler and simulator, FALSIM, the source file containing assembly language statements and directives should be prepared according to the following guidelines:

- The source file should contain ASCII text only. Each line should be terminated by a carriage return. The extension **.asmfa** should be used with each file name. After assembly, a list file with the original filename and an extension **.lstfa**, and a binary file with an extension **.binfa** will be generated by FALSIM.
- Comments are indicated by a semicolon (;) and can be placed anywhere in the source file. The FALSIM assembler ignores any text after the semicolon.
- Names in the source file can be of one of the following types:
- Variables: These are defined using the **.equ** directive. A value must also be assigned to variables when they are defined.
- Addresses in the “data and pointer area” within the memory: These can be defined using the **.dw** or the **.sw** directive. The difference between these two directives is that when **.dw** is used, it is not possible to store any value in the memory. The integer after **.dw** identifies the number of memory words to be reserved starting at the current address. (The directive **.db** can be used to reserve bytes in memory.) Using the **.sw** directive, it is possible to store a constant or the value of a name in the memory. It is also possible to use pointers with this directive to specify addresses larger than 127. Data tables and jump tables can also be set up in the memory using this directive.
- Labels: An assembly language statement can have a unique label associated with it. Two assembly language statements cannot have the same name. Every label should have a colon (:) after it.
- Use the **.org 0** directive as the first line in the program. Although the use of this line is optional, its use will make sure that FALSIM will start simulation by picking up the first instruction stored at address 0 of the memory. (Address 0 is called the reset address of the processor). A **jump [first]** instruction can be placed at address 0, so that control is transferred to the first executable statement of the main program. Thus, the label **first**

serves as the identifier of the “entry point” in the source file. The **.org** directive can also be used anywhere in the source file to force code at a particular address in the memory.

- Address 2 in the memory is reserved for the pointer to the Interrupt Service Routine (ISR). The **.sw** directive can be used to store the address of the first instruction in the ISR at this location.
- Address 4 to 125 can be used for addresses of data and pointers²⁰. However, the main program must start at address 126 or less²¹, otherwise FALSIM will generate an error at the **jump [first]** instruction.
- The main program should be followed by any subprograms or procedures. Each procedure should be terminated with a **ret** instruction. The ISR, if any, should be placed after the procedures and should be terminated with the **iret** instruction.
- The last line in the source file should be the **.end** directive.
- The **.equ** directive can be used anywhere in the source file to assign values to variables.
- It is the responsibility of the programmer to make sure that code does not overwrite data when the assembly process is performed, or vice versa. As an example, this can happen if care is not exercised during the use of the **.org** directive in the source file.

3. Using FALSIM:

- To start FALSIM (the FALCON-A assembler and simulator), double click on the FALSIM icon. This will display the assembler window, as shown in the Figure 1.
- Select one or both assembler options shown on the top right corner of the assembler window labeled as “2”. If no option is selected, the symbol table and the instruction table will not be generated in the list (.lstfa) file.
- Click on the select assembly file button labeled as “1”. This will open the dialog box as shown in the Figure 2.
- Select the path and file containing the source program that is to be assembled.
- Click on the open button. FALSIM will assemble the program and generate two files with the same filename, but with different extensions. A list file will be generated with an extension .lstfa, and a binary (executable) file will be generated with an extension .binfa. FALSIM will also display the list file and any error messages in two separate panes, as shown in Figure 3.
- Double clicking on any error message highlights and displays the corresponding erroneous line in the program listing window pane for the user. This is shown in Figure 4. The highlight feature can also be used to display any text string, including statements with errors in them. If the assembler reported any errors in the source file, then these errors should be corrected and the program should be assembled again before simulation can be done. Additionally, if the source file had been assembled correctly at an earlier occasion, and a correct binary (.binfa) file exists, the simulator can be started directly without performing the assembly process.

²⁰ Any address between 4 and 14 can be used in place of the displacement field in load or store instructions. Recall that the displacement field is just 5 bits in the instruction word.

²¹ This restriction is because of the fact that the immediate operand in the **movi** instruction must fit an 8-bit field in the instruction word.

Advance Computer Architecture – CS501

- To start the simulator, click on the start simulation button labeled as “6”. This will open the dialog box shown in Figure 6.
- Select the binary file to be simulated, and click Open as shown in Figure 7. (It is also possible to open the file by double clicking on the file name in the “Open” window).
- This will open the simulation window with the executable program loaded in it as shown in Figure 8. The details of the different panes in this window were given in section 1 earlier. Notice that the first instruction at address 0 is ready for execution. All registers are initialized to 0. The memory contains the address of the ISR (i.e., 64h which is 100 decimal) at location 2 and the address of the printer driver at location 4. These two addresses are determined at assembly time in our case. In a real situation, these addresses will be determined at execution time by the operating system, and thus the ISR and the printer driver will be located in the memory by the operating system (called re-locatable code). Subsequent memory locations contain constants defined in the program.
- Click single step button labeled as “19”. FALSIM will execute the **jump [main]** instruction at address 0 and the PC will change to 20h (32 decimal), which is the address of the first instruction in the main program (i.e., the value of main).
- Although in a real situation, there will be many instructions in the main program, those instructions are not present in the dummy calling program. The first useful instruction is shown next. It loads the address of the printer driver in r6 from the pointer area in the memory. The registers r5 and r7 are also set up for passing the starting address of the print buffer and the number of bytes to be printed. In our dummy program, we bring these values in to these registers from the data area in the memory, and then pass these values to the printer driver using these two registers. Clicking on the single step button twice, executes these two instructions.
- The execution of the call instruction simulates the event of a print request by the user. This transfers control to the printer driver. Thus, when the **call r4, r6** instruction is single stepped, the PC changes to 32h (50 decimal) for executing the first instruction in the printer driver.
- Double click on memory location 000A, which is being used for holding the PB (printer busy) flag. Enter a 1 and click the change memory button. This will store a 0001 in this location, indicating that a previous print job is in progress. Now click single step and note that this value is brought from memory location 000E into register r1. Clicking single step again will cause the **jnz r1, [message]** instruction to execute, and control will transfer to the message routine at address 0046h. The **nop** instruction is used here as a place holder.
- Click again on the single step button. Note that when the **ret r4** instruction executes, the value in r4 (i.e., 28h) is brought into the PC. The blue highlight bar is placed on the next instruction after the **call r4, r6** instruction in the main program. In case of the dummy calling program, this is the **halt** instruction.
- Double click on the value of the PC labeled as “20”. This will open a dialog box shown below. Enter a value of the PC (i.e. 26h) corresponding to the **call r4, r6** instruction, so that it can be executed again. A “list” of possible PC values can also be pulled down using, and 0026h can be selected from there as well.
- Click single step again to enter the printer driver again.
- Change memory location 000A to a 0, and then single step the first instruction in the printer driver. This will bring a 0 in r1, so that when the next **jnz r1, [message]**

instruction is executed, the branch will not be taken and control will transfer to the next instruction after this instruction. This is **movi r1, 1** at address 0036h.

- Continue single stepping.
- Notice that a 1 has been stored in memory location 000A, and r1 contains 11h, which is then transferred to the output port at address 3Ch (60 decimal) when the **out r1, controlp** instruction executes. This can be verified by double clicking on the top left corner of the I/O port pane, and changing the address to 3Ch. Another way to display the value of an I/O port is to scroll the I/O window pane to the desired position.
- Continue single stepping till the **int** instruction and note the changes in different panes of the simulation window at each step
- When the **int** instruction executes, the PC changes to 64h, which is the address of the first instruction in the ISR. Clicking single step executes this instruction, and loads the address of **temp** (i.e., 0010h) which is a temporary memory area for storing the environment. The five **store** instructions in the ISR save the CPU environment (working registers) before the ISR change them.
- Single step through the ISR while noting the effects on various registers, memory locations, and I/O ports till the **iret** instruction executes. This will pass control back to the printer driver by changing the PC to the address of the **jump [finish]** instruction, which is the next instruction after the **int** instruction.
- Double click on the value of the PC. Change it to point to the **int** instruction and click single step to execute it again. Continue to single step till the **in r1, statusp** instruction is ready for execution.
- Change the I/O port at address 3Ah (which represents the status port at address 58) to 80 and then single step the **in r1, statusp** instruction. The value in r1 should be 0080.

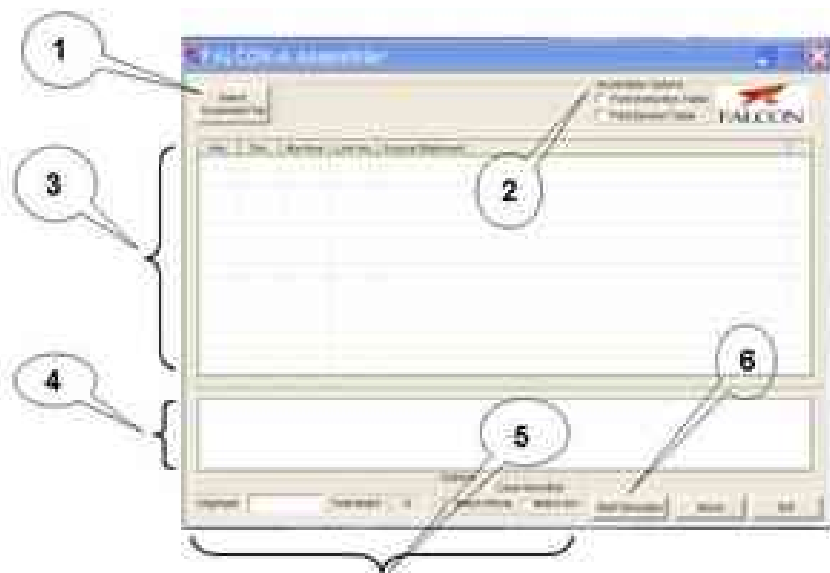


Figure 1

- Single step twice and notice that control is transferred to the **movi r7, FFFF**²² instruction, which stores an error code of -1 in r1.

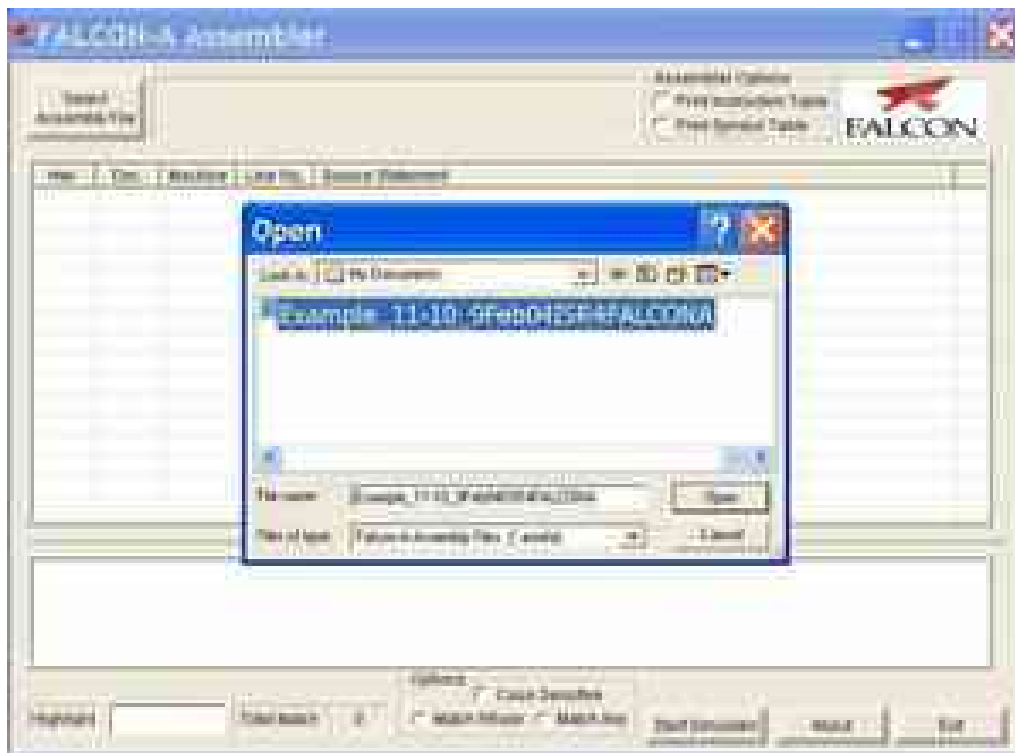


Figure 2

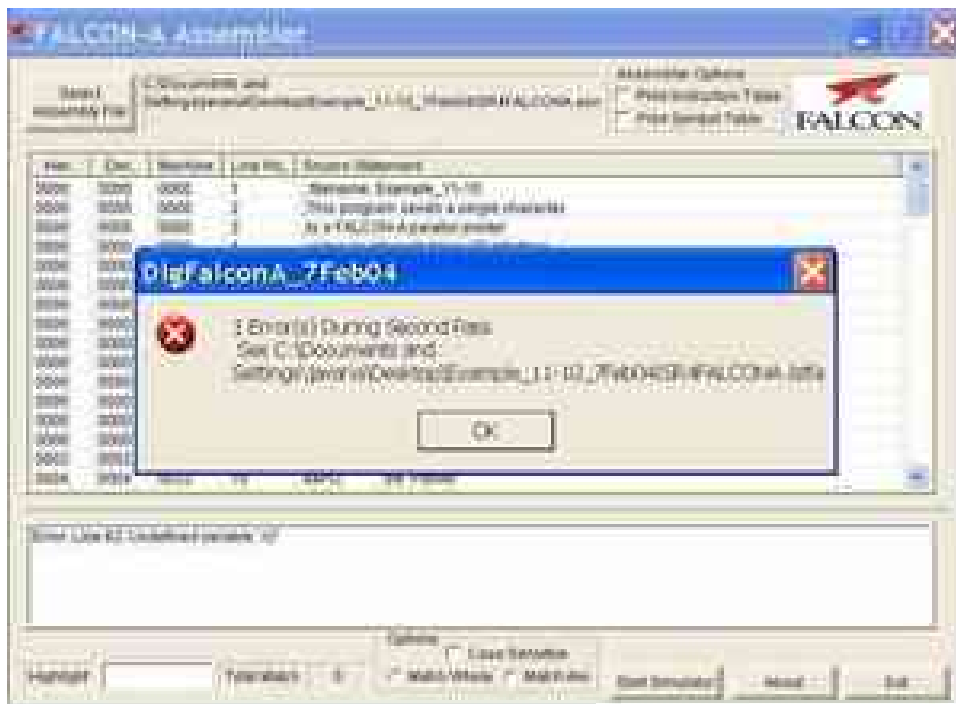


Figure 3

²²

The instruction was originally **movi r7, -1**. Since it was converted to machine language by the assembler, and then reverse assembled by the simulator, it became **movi r7, FFFF**. This is because the machine code stores the number in 16-bits after sign-extension. The result will be the same in both cases.

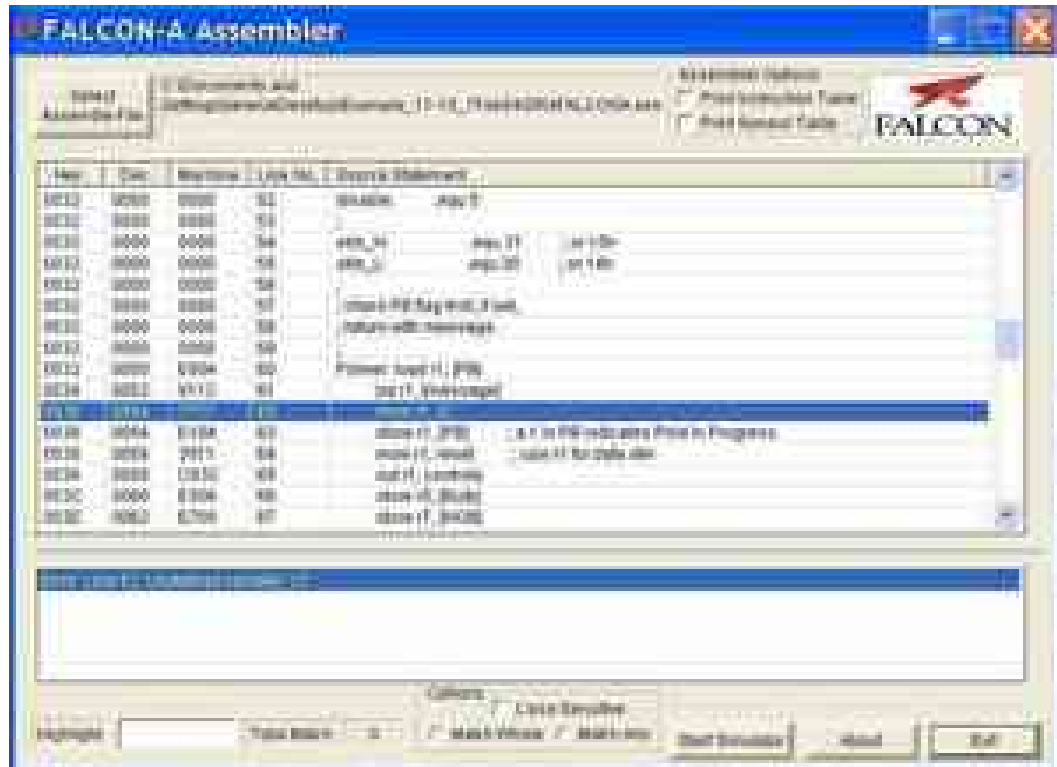


Figure 4

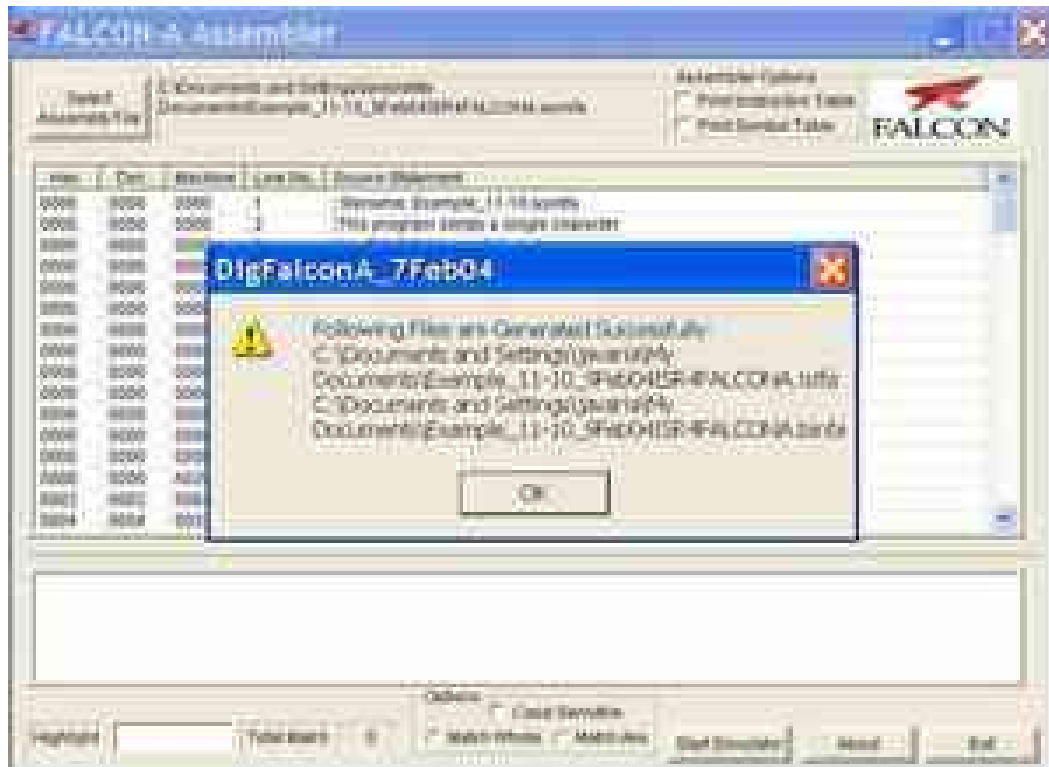


Figure 5

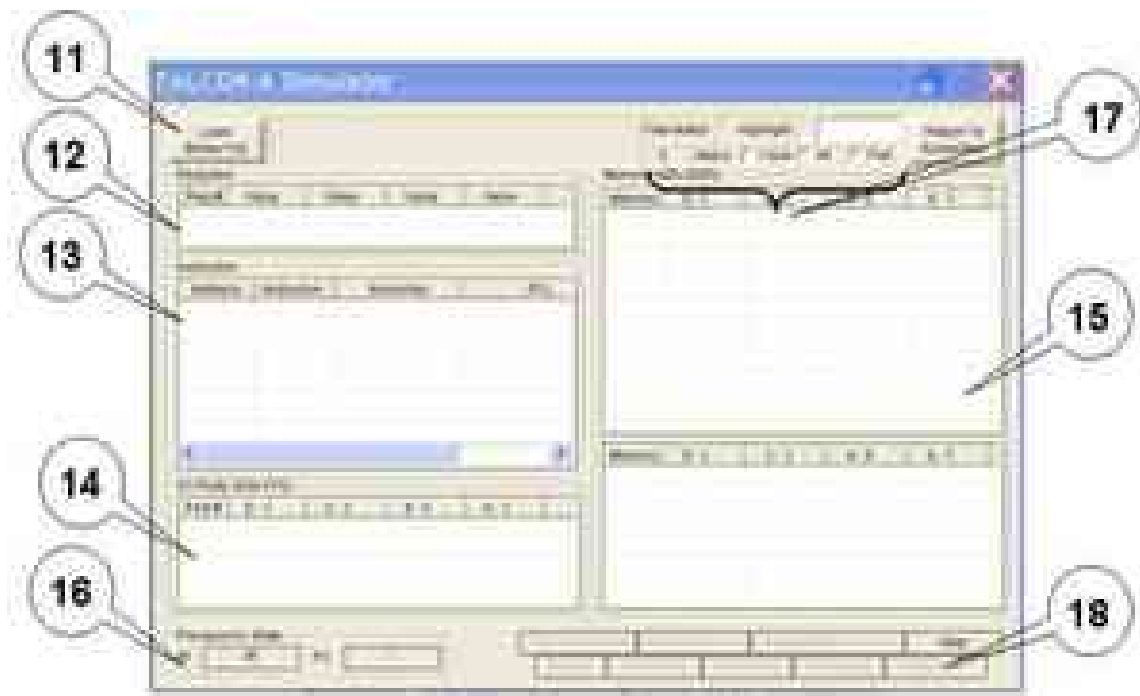


Figure 6

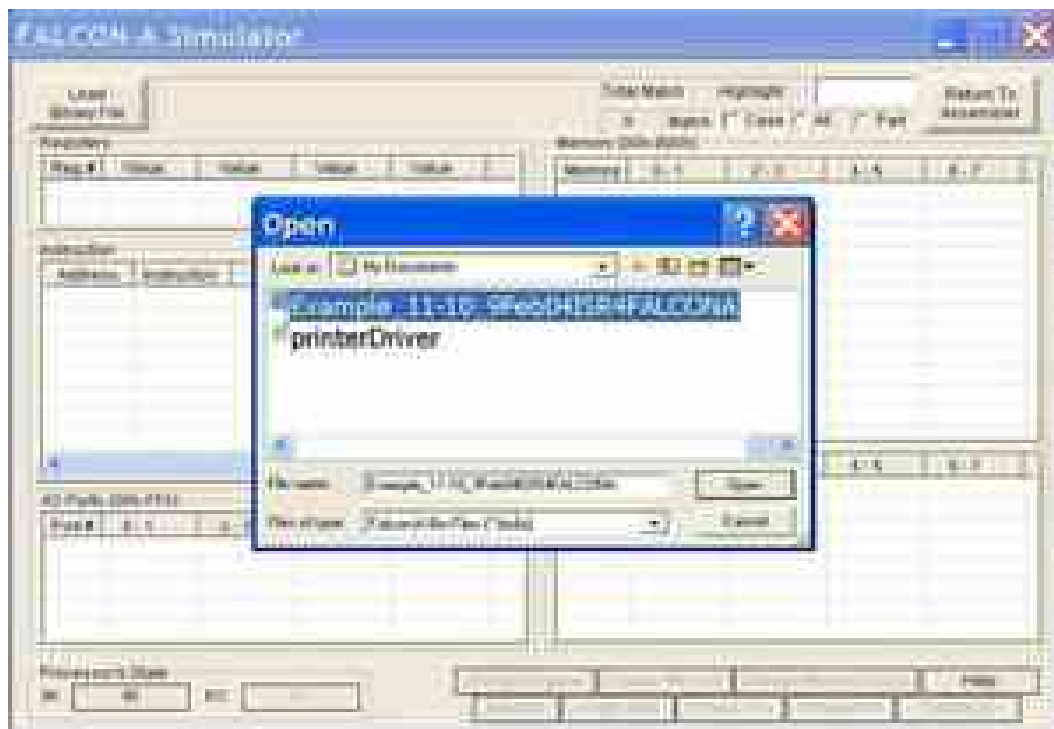


Figure 7

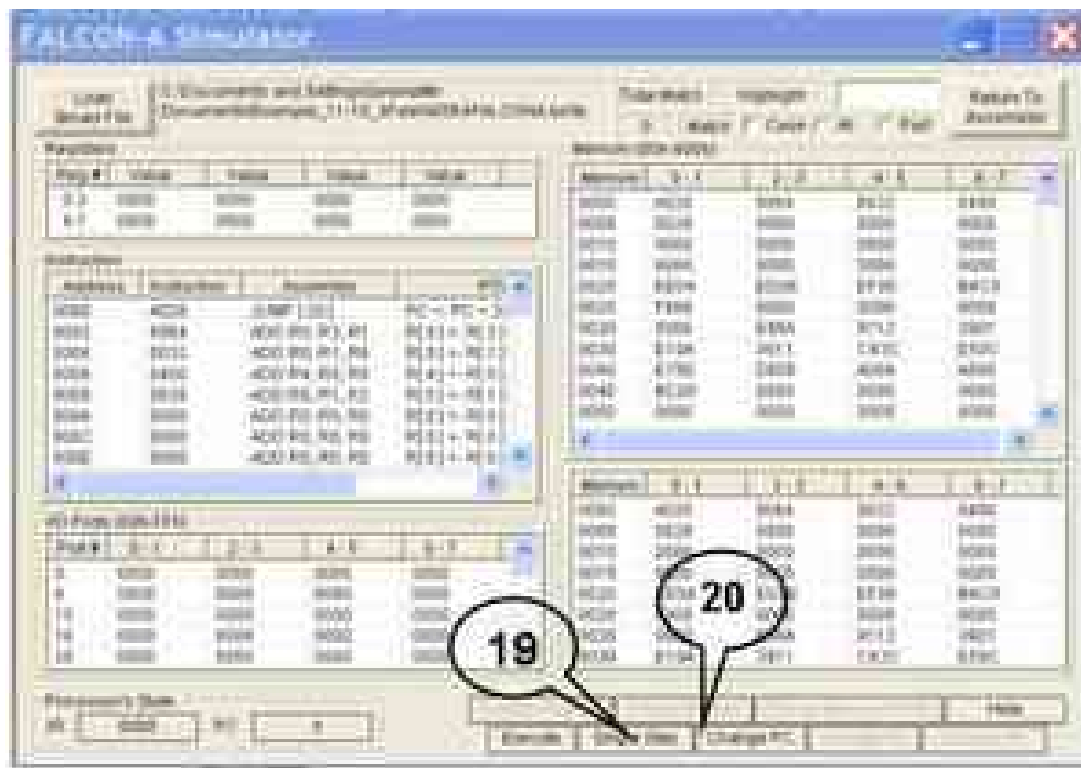


Figure 8

4. FALCON-A assembly language programming techniques:

- If a signed value, x, cannot fit in 5 bits (i.e., it is outside the range -16 to +15), FALSIM will report an error with a load r1, [x] or a store r1, [x] instruction. To overcome this problem, use movi r2, x followed by load r1, [r2].
- If a signed value, x, cannot fit in 8 bits (i.e., it is outside the range -128 to +127), even the previous scheme will not work. FALSIM will report an error with the movi r2, x instruction. The following instruction sequence should be used to overcome this limitation of the FALCON-A. First store the 16-bit address in the memory using the .sw directive. Then use two load instructions as shown below:

```
a:      .sw x load r2, [a]
        load r1, [r2]
```

This is essentially a “memory-register-indirect” addressing. It has been made possible by the .sw directive. The value of a should be less than 15.

- A similar technique can be used with immediate ALU instructions for large values of the immediate data, and with the transfer of control (call and jump) instructions for large values of the target address.
- Large values (16-bit values) can also be stored in registers using the mul instruction combined with the addi instruction. The following instructions bring a 201 in register r1.

```
movi r2, 10
movi r3, 20
mul r1, r2, r3      ; r1 contains 200 after this instruction
addi r1, r1, 1      ; r1 now contains 201
```

- Moving from one register to another can be done by using the instruction **addi r2, r1, 0**.
- Bit setting and clearing can be done using the logical (and, or, not, etc) instructions.
- Using shift instructions (shifl, asr, etc.) is faster than **mul** and **div**, if the multiplier or divisor is a power of 2.

Lecture No. 30

Interrupt Priority and Nested Interrupts

Reading Material

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

Chapter 8
8.3.3, 8.4

Summary

- Nested Interrupts
- Interrupt Mask
- DMA

Nested Interrupts

(Read from Book, Jordan Page 391)

Interrupt Mask

(Read from Book, Jordan Page 391)

Priority Mask

(Read from Book, Jordan Page 392)

Examples

Example # 1²³

Assume that three I/O devices are connected to a 32-bit, 10 MIPS CPU. The first device is a hard drive with a maximum transfer rate of 1MB/sec. It has a 32-bit bus. The second device is a floppy drive with a transfer rate of 25KB/sec over a 16-bit bus, and the third device is a keyboard that must be polled thirty times per second. Assuming that the polling operation requires 20 instructions for each I/O device, determine the percentage of CPU time required to poll each device.

Solution:

The hard drive can transfer 1MB/sec or 250 K 32-bit words every second. Thus, this hard drive should be polled using at least this rate.

Using $1K=2^{10}$, the number of CPU instructions required would be

$$250 \times 2^{10} \times 20 = 5120000 \text{ instructions per second.}$$

Percentage of CPU time required for polling is

$$(5.12 \times 10^6) / (10 \times 10^6) = 51.2\%$$

The floppy disk can transfer $25K/2 = 12.5 \times 2^{10}$ half-words per second. It should be polled with at least this rate. The number of CPU instructions required will be $12.5 \times 2^{10} \times 20 = 256,000$ instructions per second.

Therefore, the percentage of CPU time required for polling is

$$(0.256 \times 10^6) / (10 \times 10^6) = 2.56\%.$$

For the keyboard, the number of instructions required for polling is

$$30 \times 20 = 600 \text{ instructions per second.}$$

Therefore, the percentage of CPU time spent in polling is

$$600 / (10 \times 10^6) = 0.006\%$$

It is clear from this example that while it is acceptable to use polling for a keyboard or a floppy drive, it is very risky to use polling for the hard drive. In general, for devices with a high data rate, the use of polling is not adequate.

Example # 2²⁴

- What should be the polling frequency for an I/O device if the average delay between the time when the device wants to make a request and the time when it is polled, is to be at most 10 ms?
- If it takes 10,000 cycles to poll the I/O device, and the processor operates at 100MHz, what % of the CPU time is spent polling?
- What if the system wants to provide an average delay of 1msec?

Solution:

- Assuming that the I/O requests are distributed evenly in time, the average time that a device will have to wait for the processor to poll is half the time between polling attempts. Therefore, to provide an average delay of 10 ms, the processor will have to poll every 20 ms, or 50 times per second.
- If each polling attempt takes 10,000 cycles, then the processor will spend 500,000 cycles polling each second. The % of CPU time spent in polling is then $(0.5 \times 10^6) / (100 \times 10^6) = 0.5\%$
- To provide an average delay of 1ms, the polling frequency must be increased. The processor will have to poll every 2ms, or 500 times per second. This will consume 5,000,000 cycles for polling. The % of CPU time spent polling then becomes $5/100 = 5\%$.

Example # 3²⁵

What percentage of time will a 20MIPS processor spend in the busy wait loop of an 80-character line printer when it takes 1 msec to print a character and a total of 565 instructions need to be executed to print an 80 character line. Assume that two instructions are executed in the polling loop.

²⁴ Adopted from [Schaum]

²⁵ Adopted from [H&J]

Solution:

Out of the total 565 instructions executed to print a line, $80 \times 2 = 160$ are required for polling. For a 20MIPS processor, the execution of the remaining 405 instructions takes $405 / (20 \times 10^6) = 20.25 \mu\text{sec}$. Since the printing of 80 characters takes 80ms, $(80 - 0.02025) = 79.97\text{msec}$ is spent in the polling loop before the next 80 characters can be printed. This is $79.97/80 = 99.96\%$ of the total time.

Example # 4²⁶

Consider a 20 MIPS processor with several input devices attached to it, each running at 1000 characters per second. Assume that it takes 17 instructions to handle an interrupt. If the hardware interrupt response takes $1 \mu\text{sec}$, what is the maximum number of devices that can be handled simultaneously?

Solution:

A service for one character requires $17 / (20 \times 10^6) + 1 \mu\text{sec} = 1.85 \mu\text{sec}$. Since each device runs at 1000 characters per second, 1.85 ms of handling time is required by each device every second. Therefore the maximum number of devices that can be handled is $1 / (1.85 \times 10^{-3}) = 540$.

Example # 5²⁷

Assume that a floppy drive having a transfer rate of 25KB per second is attached to a 32 bit, 10MIPS CPU using an interrupt driven interface. The drive has a 16-bit data bus. Assume that the interrupt overhead is 20 instructions. Calculate the fraction of CPU time required to service this drive when it is active.

Solution:

Since the floppy drive has a 16-bit data bus, it can transfer two bytes at one time. Thus its transfer rate is $25/2 = 12.5\text{K}$ half-words (16-bits each) per second. This corresponds to an overhead of 20 instructions or $12.5\text{K} \times 20 = 12.5 \times 2^{10} \times 20 = 256000$ instructions per second.

Example # 6²⁸

A processor with a 500 MHz clock requires 1000 clock cycles to perform a context switch and start an ISR. Assume each interrupt takes 10,000 cycles to execute the ISR and the device makes 200 interrupt requests per second. Also, assume that the processor polls every 0.5msec during the time when there are no interrupts. Further assume that polling an I/O device requires 500 cycles. Compute the following:

- a. How many cycles per second does the processor spend handling I/O from the device if only interrupts are used?
- b. What fraction of the CPU time is used in interrupt handling for part (a)?
- c. How many cycles per second are spent on I/O if polling is also used with interrupts?
- d. How often should the processor poll so that polling incurs the same overhead as interrupts?

26 Adopted from [H&J]
 27 Adopted from [H&P org]
 28 Adopted from [Schaum]

Solution:

a. The device makes 200 interrupt requests per second, each of which takes $10,000 + 2 \times 1000$ (context switching to the ISR and back from it) = 12,000 cycles.

Thus, a total of $200 \times 12,000 = 2,400,000$ cycles per second are spent handling I/O using interrupts.

b. The percentage of the processor time used in interrupt handling is $2,400,000 / (500 \times 10^6)$ or 0.48%.

c. There are 200 interrupt requests per second, or one interrupt request every 5 ms. Every interrupt consumes a total of 12,000 cycles, as calculated in part (a). For a 500 MHz CPU, this is

$$12000 / (500 \times 10^6) = 24 \text{ microseconds.}$$

For 200 interrupts per second, this is 4.8 msec. This leaves $1000 - 4.8 = 995.2$ msec for polling.

Since the processor polls once every 0.5 msec during the time when there is no interrupt, this corresponds to

$$995 / 0.5 = 1990 \text{ times per second.}$$

The total number of cycles required for polling is

$$1990 \times 500 = 995,000 \text{ cycles per second.}$$

Thus, the total time spent on I/O when using polling with interrupts is
 $2,400,000 + 995,000 = 3,395,000$ cycles per second.

d. The interrupt overhead is 1000 cycles per second for a context switch to the ISR and 1000 cycles per second back from it. This is a total of 2×1000 cycles per second. With 200 interrupts per second, this is

$$200 \times 2000 = 400,000 \text{ cycles per second.}$$

The polling overhead is 500 cycles per second. Thus, for the same overhead as interrupts, the polling operation should be performed

$$400,000 / 500 = 800 \text{ times per second,}$$
$$\text{or } 1/800 = \text{every } 1.25 \text{ msec.}$$

Direct Memory Access (DMA)

Direct memory access is a technique, where by the CPU passes its control to the memory subsystem or one of its peripherals, so that a contiguous block of data could be transferred from peripheral device to memory subsystem or from memory subsystem to peripheral device or from one peripheral device to another peripheral device.

Advantage of DMA

The transfer rate is pretty fast and conceptually you could imagine that through disabling the tri-state buffers, the system bus is isolated and a direct connection is established between the I/O subsystem and the memory subsystem and then the CPU is free. It is idle at that time or it could do some other activity. Therefore, the DMA would be quite useful, if a large amount of data needs to be transferred, for example from a hard disk to a printer or we could fill up the buffer of a printer in a pretty short time.

As compared to interrupt driven I/O or the programmed I/O, DMA would be much faster.

What is the consequence? The consequence is that we need to have another chip, which is a **DMA controller**. “A DMA controller could be a CPU in itself and it could control the total activity and synchronize the transfer of data”. DMA could be considered as a technique of transferring data from I/O to memory and from memory to I/O without the intervention of the CPU. The CPU just sets up an I/O module or a memory subsystem, so that it passes control and the data could be passed on from I/O to memory or from memory to I/O or within the memory from one subsystem to another subsystem without interaction of the CPU. After this data transfer is complete, the control is passed from I/O back to the CPU.

Now we can illustrate further the advantage of DMA using following example.

Example of DMA

If we write instruction load as follows:

```
load [2], [9]
```

This instruction is illegal and not available in the SRC processor. The symbols [2] and [9] represent memory locations. If we want to have this transfer to be done then two steps would be required. The instruction would be:

```
load r1, [9]
```

```
store r1, [2]
```

Thus it is not possible to transfer from one memory location to another without involving the CPU. The same applies to transfer between memory and peripherals connected to I/O ports. For example we cannot have:

```
out [6], datap
```

It has to be done again in two steps:

```
load r1, [6]
```

```
out r1, datap
```

Similar comments apply to the “in” instruction. Thus the real cause of the limited transfer rate is the CPU itself. It acts as an unnecessary middle man. The example illustrates that in general, every data word travels over the system bus twice and this is not necessary, and therefore, the DMA in such cases is pretty useful.

DMA Approach

The DMA approach is to turn off i.e. through tri-state buffers and therefore, electrically disconnect from the system bus, the CPU and let a peripheral device or a memory subsystem or

any other module or another block of the same module communicate directly with the memory or with another peripheral device. This would have the advantage of having higher transfer rates which could approach that of limited by the memory itself.

Disadvantage of DMA

The disadvantage however, would be that an additional DMA controller would be required, that could make the system a bit more complex and expensive. Generally, the DMA requests have priority over all other bus activities including interrupts. No interrupts may be recognized during a DMA cycle.

Lecture No. 31

Direct Memory Access (DMA)

Reading Material

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

Chapter 8
8.4

Summary

- Direct Memory Access (DMA)

Direct Memory Access (DMA):

Introduction

Direct Memory Access is a technique which allows a peripheral to read from and/or write to memory without intervention by the CPU. It is a simple form of bus mastering where the I/O device is set up by the CPU to transfer one or more contiguous blocks of memory. After the transfer is complete, the I/O device gives control back to the CPU. The following DMA transfer combinations are possible:

- Memory to memory
- Memory to peripheral
- Peripheral to memory
- Peripheral to peripheral

The DMA approach is to "turn off" (i.e., tri-state and electrically disconnect from the system buses) the CPU and let a peripheral device (or memory - another module or another block of the same module) communicate directly with the memory (or another peripheral).

ADVANTAGE: Higher transfer rates (approaching that of the memory) can be achieved.
DISADVANTAGE: A DMA Controller, or a DMAC, is needed, making the system complex and expensive.

Generally, DMA requests have priority over all other bus activities, including interrupts. No interrupts may be recognized during a DMA cycle.

Reason for DMA:

The instruction **load [2], [9]** is illegal. The symbols [2] and [9] represent memory locations. This transfer has to be done in two steps:

- **load r1,[9]**
- **store r1,bx**

Thus, it is not possible to transfer from one memory location to another without involving the CPU. The same applies to transfer between memory and peripherals connected to I/O ports. e.g., we cannot have **out [6], datap**. It has to be done in two steps:

- **load r1,[6]**
- **out r1, datap**

Similar comments apply to the **in** instruction.

Thus, the real cause of the limited transfer rate is the CPU itself. It acts as an unnecessary "middleman". The above discussion also implies that, in general, every data word travels over the system bus twice.

Some Definitions:

- **MASTER COMPONENT:** A component connected to the system bus and having control of it during a particular bus cycle.
- **SLAVE COMPONENT:** A component connected to the system bus and with which the master component can communicate during a particular bus cycle. Normally the CPU with its bus control logic is the master component.
- **QUALIFICATIONS TO BECOME A MASTER:** A Master must have the capability to place addresses on the address bus and direct the bus activity during a bus cycle.
- **QUALIFIED COMPONENTS:**
 - Processors with their associated bus control logic.
 - DMA controllers.
- **CYCLE STEALING:** Taking control of the system bus for a few bus cycles.

Data Transfer using DMA:

Data transfer using DMA takes place in three steps.

1st Step:

In this step when the processor has to transfer data it issues a command to the DMA controller with the following information:

Operation to be performed i.e., read or write operation. Address of I/O device.

Address of memory block.

Size of data to be transferred.

After this, the processor becomes free and it may be able to perform other tasks.

2nd Step:

In this step the entire block of data is transferred directly to or from memory by the DMA controller.

3rd Step:

In this, at the end of the transfer, the DMA controller informs the processor by sending an interrupt signal.

See figure 8.18 on the page number 400 of text book.

The DMA Transfer Protocol:

Most processors have a separate line over which an external device can send a request for DMA. There are various names in use for such a line. HOLD, RQ, or Bus Request (BR), etc. are examples of these names.

The DMA cycle usually begins with the alternate bus master requesting the system bus by activating the associated Bus Request line and, of course, satisfying the setup and hold times. The CPU completes the current bus cycle, in the same way as it does in case of interrupts, and responds by floating the address, data and control lines. A Bus Grant pulse is then output by the CPU to the same device from where the request occurred. After receiving the Bus Grant pulse, and waiting for the "float delay" of the CPU, the requesting device may drive the system bus. This precaution prevents bus contention. To return control of the bus to the CPU, the alternate bus master relinquishes bus control and issues a release pulse on the same Bus Request line. The CPU may drive the system bus after detecting the release pulse. The alternate bus master should be tri-stated off the local bus and have other CPU interface circuits re-enabled within this time.

DMA has priority over Interrupt driven I/O:

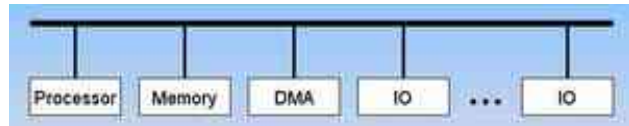
In interrupt driven I/O the I/O transfer depends upon the speed at which the processor tests and service a device. Also, many instructions are required for each I/O transfer. These factors become bottleneck when large blocks of data are to be transferred. While in the DMA technique the I/O transfers take place without the intervention by the CPU, rather CPU pauses for one bus cycle. So DMA technique is the more efficient technique for I/O transfers.

DMA Configurations:

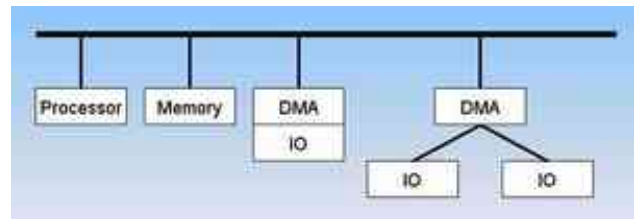
- Single Bus Detached DMA
- Single Bus Integrated DMA
- I/O Bus

Single Bus Detached DMA

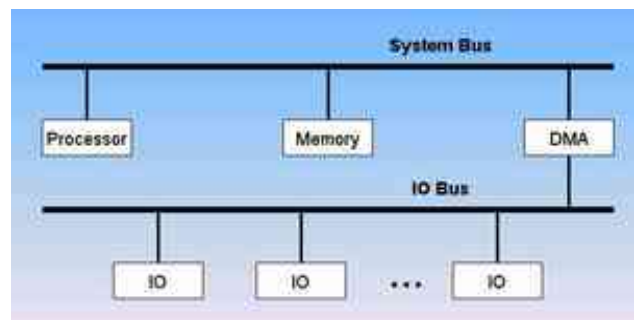
In the example provided by the above diagram, there is a single bidirectional bus connecting the processor, the memory, the DMA module and all the I/O modules. When a particular I/O module needs to read or write large amounts contiguous data it requests the processor for direct memory access. If permission is granted by the processor, the I/O module sends the read or write address and the size of data needed to be read or written to the DMA module. Once the DMA module acknowledges the request, the I/O module is free to read or write its contiguous block of data from or onto main memory. Even though in this situation the processor will not be able to execute while the transfer is going on (as there is a just a single bus to facilitate transfer of data), DMA transfer is much faster than having each word of memory being read by the processor and then being written to its location.

**Single Bus Integrated DMA**

In this configuration the DMA and one or more I/O modules are integrated without the inclusion of system bus functioning as the part of I/O module or may be as a separate module controlling the I/O module.

**I/O Bus**

In this configuration we integrate the DMA and I/O modules through an I/O bus. So it will cut the number of I/O interfaces required between DMA and I/O module.

**Example**

An I/O device transfers data at a rate of 10MB/s over a 100MB/s bus. The data is transferred in 4KB blocks. If the processor operates at 500MHz, and it takes a total of 5000 cycles to handle each DMA request, find the fraction of CPU time handling the data transfer with and without DMA.

Solution.

Without DMA

The processor here copies the data into memory as it is sent over the bus. Since the I/O device sends data at a rate of 10MB/s over the 100MB/s bus, 10 % of each second is spent transferring data. Thus 10% of the CPU time is spent copying data to memory. With DMA

Time required in handling each DMA request is 5000 cycles. Since 2500 DMA requests are issued (10MB/4KB) the total time taken is 12,500,000 cycles. As the CPU clock is 500MHZ, the fraction of CPU time spent is $12,500,000 / (500 \times 10^6)$ or 2.5%.

Example

A hard drive with a maximum transfer rate of 1Mbyte/sec is connected to a 32-bit, 10MIPS CPU operating at a clock frequency of 100 MHz. Assume that the I/O interface is DMA based and it takes 500 clock cycles for the CPU to set-up the DMA controller. Also assume that the interrupt handling process at the end of the DMA transfer takes an additional 300 CPU clock cycles. If the data transfer is done using 2 KB blocks, calculate the percentage of the CPU time consumed in handling the hard drive.

Solution

Since the hard drive transfers at 1MB/sec, and each block size is 2KB, there are

$$1000/2 = 500 \text{ blocks transferred/sec}$$

Every DMA transfer uses $500+300=800$ CPU cycles. This gives us

$$800 \times 500 = 400,000 = 400 \times 10^3 \text{ cycles/sec}$$

For the 100 MHz CPU, this corresponds to

$$(400 \times 10^3) / (100 \times 10^6) = 4 \times 10^{-3} = 0.4\%$$

This would be the case when the hard drive is transferring data all the time. In actual situation, the drive will not be active all the time, and this number will be much smaller than 0.4%.

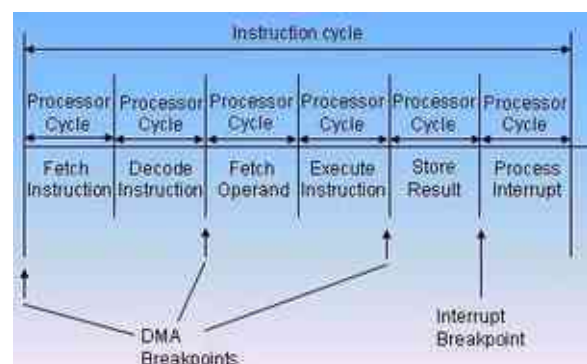
Another assumption that is implied in the previous example is that the DMA controller is the only device accessing the memory. If the CPU also tries to access memory, then either the DMAC or the CPU will have to wait while the other one is actively accessing the memory. If cache memory is also used, this can free up main memory for use by the DMAC.

Cycle Stealing

The DMA module takes control of the bus to transfer data to and from memory by forcing the CPU to temporarily suspend its operation. This approach is called Cycle Stealing because in this approach DMA steals a bus cycle.

DMA and Interrupt breakpoints during an instruction cycle

The figure shows that the CPU suspends or pauses for one bus cycle when it needs a bus cycle, transfers the data and then returns the control back to the CPU.



I/O processors

When I/O module has its own local memory to control a large number of I/O devices without the involvement of CPU is called I/O processor.

I/O Channels

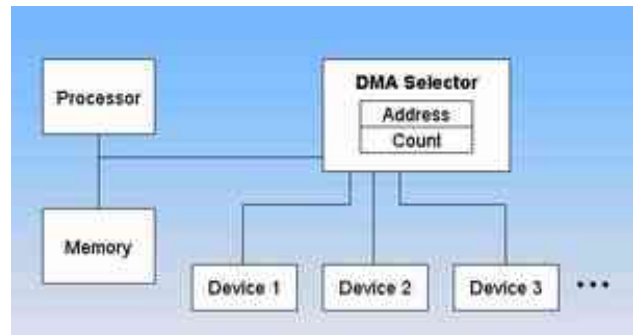
When an I/O module has a capability of executing a specific set of instructions for specific I/O devices in the memory without the involvement of CPU is called I/O channel.

I/O channel architecture:

Types of I/O channels:

Selector Channel

It is the DMA controller that can do block transfers for several devices but only one at a time.



Multiplexer Channel

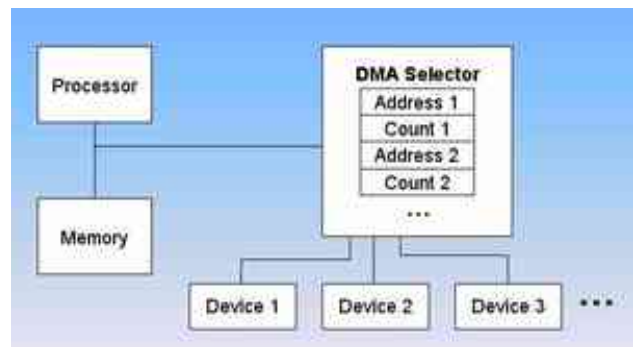
It is the DMA controller that can do block transfers for several devices at once.

Types of Multiplexer Channel

- Byte Multiplexer
- Block Multiplexer

Byte Multiplexer

- Byte multiplexer accepts or transmits characters.
- Interleaves bytes from several devices.
- Used for low speed devices.



Block Multiplexer

- Block multiplexer accepts or transmits block of characters.
- Interleaves blocks of bytes from several devices.
- Used for high speed devices.

Virtual Address:

Virtual address is generated by the logical by the memory management unit for translation.

Physical Address:

Physical address is the address in the memory.

DMA and memory system

DMA disturbs the relationship between the memory system and CPU.

Direct memory access and the memory system

Without DMA, all memory accesses are handled by the CPU, using address translation and cache mechanism. When DMA is implemented into an I/O system memory accesses can be made without intervening the CPU for address translation and cache access. The problems created by the DMA in virtual memory and cache systems can be solved using hardware and software techniques.

Hardware Software Interface

One solution to the problem is that all the I/O transfers are made through the cache to ensure that modified data are read and updated in the cache on the I/O write. This method can decrease the processor performance because of infrequent usage of the I/O data.

Another approach is that the cache is invalidated for an I/O read and for an I/O write, write-back (flushing) is forced by the operating system. This method is more efficient because flushing of large parts of cache data is only done on DMA block accesses.

Third technique is to flush the cache entries using a hardware mechanism, used in multiprogramming system to keep cache coherent.

SOME clarifications:

- The terms "serial" and "parallel" are with respect to the computer I/O ports --- not with respect to the CPU. The CPU always transfers data in parallel.
- The terms "programmed I/O", "interrupt driven I/O" and "DMA" are with respect to the CPU. Each of these terms refers to a way in which the CPU handles I/O, or the way data flow through the ports is controlled.
- The terms "simplex" and "duplex" are with respect to the transmission medium or the communication link.
- The terms "memory mapped I/O" and "independent I/O" are with respect to the mapping of the interface, i.e., they refer to the CPU control lines used in the interface.

Lecture No. 32

Magnetic Disk Drives

Reading Material

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

Chapter 9
9.1

Summary

- Hard Disk
- Static and Dynamic Properties
- Examples
- Mechanical Delays and Flash Memory
- Semiconductor Memory vs. Hard Disk

Hard Disk

Peripheral devices connect the outside world with the central processing unit through the I/O modules. One important feature of these peripheral devices is the variable data rate. Peripheral devices are important because of the function they perform.

A hard disk is the most frequently used peripheral device. It consists of a set of platters. Each platter is divided into tracks. The track is subdivided into sectors. To identify each sector, we need to have an address. So, before the actual data, there is a header and this header consisting of few bytes like 10 bytes. Along with header there is a trailer. Every sector has three parts: a header, data section and a trailer.

Static Properties

The storage capacity can be determined from the number of platters and the number of tracks. In order to keep the density same for the entire surface, the trend is to use more number of sectors for outer tracks and lesser number of sectors for inner tracks.

Dynamic Properties

When it is required to read data from a particular location of the disk, the head moves towards the selected track and this process is called seek. The disk is constantly rotating at a fixed speed. After a short time, the selected sector moved under the head. This interval is called the rotational delay. On the average, the data may be available after half a revolution. Therefore, the rotational latency is half revolution.

The time required to seek a particular track is defined by the manufacturer. Maximum, minimum and average seek times are specified. Seek time depends upon the present position of the head and the position of the required sector. For the sake of calculations, we will use the average value of the seek time.

- **Transfer rate**

When a particular sector is found, the data is transferred to an I/O module. This would depend on the transfer rate. It would typically be between 30 and 60 Mbytes/sec defined by the manufacturer.

- **Overhead time**

Up till now, we have assumed that when a request is made by the CPU to read data, then hard disk is available. But this may not be the case. In such situation we have to face a queuing delay. There is also another important factor: the hard disk controller, which is the electronics present in the form of a printed circuit board on the hard disk. So the time taken by this controller is called over head time.

The following examples will clarify some of these concepts.

Example 1

Find the average rotational latency if the disk rotates at 20,000 rpm.

Solution

The average latency to the desired data is halfway round the disk so
 Average rotational latency = $0.5 / (20,000 / 60)$
 = 1.5ms

Example 2

A magnetic disk has an average seek time of 5 ms. The transfer rate is 50 MB/sec. The disk rotates at 10,000 rpm and the controller overhead is 0.2 msec. Find the average time to read or write 1024 bytes.

Solution

Average Tseek=5ms
 Average Trot= $0.5 * 60 / 10,000 = 3$ ms
 Ttransfer=1KB/50MB=0.02ms
 Tcontroller=0.2ms
 The total time taken= Tseek +Trot+ Ttsfr +Tctr
 =5+3+0.02+0.2
 =8.22 ms

Example 3

A hard disk with 5 platters has 1024 tracks per platter, 512 sectors per track and 512 bytes/sector. What is the total capacity of the disk?

Solution

512 bytes x 512
 sectors=0.2MB/track
 0.2MB x 1024 tracks=0.2GB/platter
 Therefore the hard disk has the total capacity of $5 \times 0.2 = 1$ GB

Example 4

How many platters are required for a 40GB disk if there are 1024 bytes/sector, 2048 sectors per track and 4096 tracks per platter

Solution

The capacity of one platter
 = $1024 \times 2048 \times 4096$
 = 8GB

For a 40GB hard disk, we need $40/8$
 = 5 such platters.

Example 5

Consider a hard disk that rotates at 3000 rpm. The seek time to move the head between adjacent tracks is 1 ms. There are 64 sectors per track stored in linear order.

Assume that the read/write head is initially at the start of sector 1 on track 7.

- a. How long will it take to transfer sector 1 on track 7 to sector 1 on track 9?
- b. How long will it take to transfer all the sectors on track 12 to corresponding sectors on track 13?

Solution

Time for one revolution = $60/3000 = 20\text{ms}$

- a. Total transfer time = sector read time + head movement time + rotational delay + sector write time

Time to read or write on sector = $20 / 64 = 0.31\text{ms/sector}$

Head movement time from track 7 to track 9 = $1\text{ms} \times 2 = 2\text{ms}$

After reading sector 1 on track 7, which takes .31ms, an additional 19.7 ms rotational delay is needed for the head to line up with sector 1 again. The head movement time of 2 ms gets included in the 19.7 ms. transfer

Total time = $0.31\text{ms} + 19.7\text{ms} + 0.31\text{ms} = 20.3\text{ms}$

- b. The time to transfer all the sectors of track 12 to track 13 can be computed in the similar way.

Assume that the memory buffer can hold an entire track. So the time to read or write an entire track is simply the rotational delay for a track, which is 20 ms. The head movement time is 1ms, which is also the time for $1/0.3 = 3.3$ 4 sectors to pass under the head. Thus after reading a track and repositioning the head, it is now on track 13, at four sectors past the initial sector that was read on track 12. (Assuming track 13 is written starting at sector 5) therefore;

Total transfer time = $20 + 1 + 20 = 41\text{ms}$.

If writing of track 13 start at the first sector, an additional 19 ms should be added, giving a total transfer time = 60 ms

Example 6

Calculate time to read 64 KB (128 sectors) for the following disk parameters.

- 180 GB, 3.5 inch disk
- 12 platters, 24 surfaces
- 7,200 RPM; (4 ms avg. latency)
- 6 ms avg. seek (r/w)
- 64 to 35 MB/s (internal)
- 0.1 ms controller time

Solution

Disk latency = average seek time + average rotational delay + transfer time + controller overhead
 $= 6\text{ ms} + 0.5 \times 1/(7200\text{ RPM}) / (60000\text{ms/M}) + 64\text{ KB} / (64\text{ MB/s}) + 0.1\text{ ms}$
 $= 6 + 4.2 + 1.0 + 0.1\text{ ms} = 11.3\text{ ms}$

Mechanical Delay and Flash Memory

Mechanical movement is involved in data transfer and causes mechanical delays which are not desirable in embedded systems. To overcome this problem in embedded systems, flash memory is used. Flash memory can be thought of a type of electrically erasable PROM. Each cell consists of two MOSFET and in between these two transistors, we have a control gate and the presence/absence of charge tells us that it is a zero or one in that location of memory.

The basic idea is to reduce the control overheads, and for a FLASH chip, this control overhead is low. Furthermore flash memory has low power dissipation. For embedded devices, flash is a better choice as compared to hard disk. Another important feature is that read time is small for flash. However the write time may be significant. The reason is that we first have to erase the memory and then write it. However in embedded system, number of write operations is less so flash is still a good choice.

Example 7

Calculate the time to read 64 KB for the previous disk, this time using 1/3 of quoted seek time, 3/4 of internal outer track bandwidth

Solution

$$\begin{aligned}
 \text{Disk latency} &= \text{average seek time} + \text{average rotational delay} + \text{transfer time} + \text{controller overhead} \\
 &= (0.33 * 6 \text{ ms}) + 0.5 * 1/(7200 \text{ RPM}) + 64 \text{ KB} / (0.75 * 64 \text{ MB/s}) + 0.1 \text{ ms} \\
 &= 2 \text{ ms} + 0.5 / (7200 \text{ RPM} / (60000 \text{ ms/M})) + 64 \text{ KB} / (48 \text{ KB/ms}) + 0.1 \text{ ms} \\
 &= 2 + 4.2 + 1.3 + 0.1 \text{ ms} = 7.6 \text{ ms}
 \end{aligned}$$

Semiconductor Memory vs. Hard Disk

At one time developers thought that development of semiconductor memory would completely wipe out the hard disk. There are two important features that need to be kept in mind in this regard:

1. Cost

It is low for hard disk as compared to semi-conductor memory.

2. Latency

Typically latency of a hard disk is in milliseconds. For SRAM, it is 10^5 times lower as compared to hard disk.

Lecture No. 33

Error Control

Reading Material

William Stallings 6th edition
Computer Organization and Architecture

Summary

- Operating System Interface
- Error Control
- RAID

Operating System Interface

The Operating system interface plays an important role for disk operation. Operating system would define a logic block telling the controller about the track, sector, etc. There are different ways to define logic blocks. For example, we can define 5 bytes containing this information such that: the first 4 bits contain disk number(in case of a system having more than one disk), the next 4 bits contain the address of a particular track followed by a sector number and at the end, the number of bytes to transferred. So this defines a logical block transferred by the controller. Along this, we have additional information about control and status of the controller. The operating system essentially insulates the users from the hardware details of the disk.

Error Control

There are two main issues in error control:

1. Detection of Error
2. Correction of Error

For detection of error, we just need to know that there exists an error. When the error is detected then the next step is to ask the source to resend that information. This process is called automatic request for repeat. In some cases there is also possibility that redundancy is enough and we reconstruct and find out exactly which particular bits are in error. This is called error correction.

There are three schemes commonly used for error control.

1. Parity code
2. Hamming code
3. CRC mechanism

1. Parity code

Along with the information bits, we add up another bit, which is called the parity bit. The objective is the total number of 1's as even or odd. If the parity at the receiving end is different, an error is indicated. Once error is found, CPU may request to repeat that data. The concept of parity bit could be enhanced. In such a case, we would like to increase the distance between different code words. Consider a code word consists of four bits, 0000, and second code word consists of 1111. The distance between two codes is four. So the distance between the two codes would be the number of bits in which they differ from each other. So the concept of introducing redundancy is increase this distance. Larger the distance, higher will be the capacity of the code. For single parity, the distance is two, we can only detect the parity. But if the distance is three, we could also correct these single errors.

If D = minimum distance between two code words then $D-1$ errors could be detected and $D/2$ errors could be corrected.

2. Hamming code

Hamming code is an example of block code. We have an encoder which could be a program or a hardware device. We feed k inputs to it. These are k information input bits. We also feed some extra bits. Let r be the number of redundant bits. So at output we have $r+k = m$ bits. As an example, for parity bit, we have $k=7$ and $r=1$ and $m=8$. So for 7 bits we get eight output bits.

For any positive integer $m \leq 3$, a Hamming code with following parameters exists:

- Code Length: $n=2^m-1$
- Number of information symbols: $k = 2^m-1-m$
- Number of parity-check symbols: $n - k = m$

3. CRC

The basic principle for CRC is very simple. We divide a particular code word and make it divisible by a prime number, and if it is divisible by a prime number then it is a valid code word.

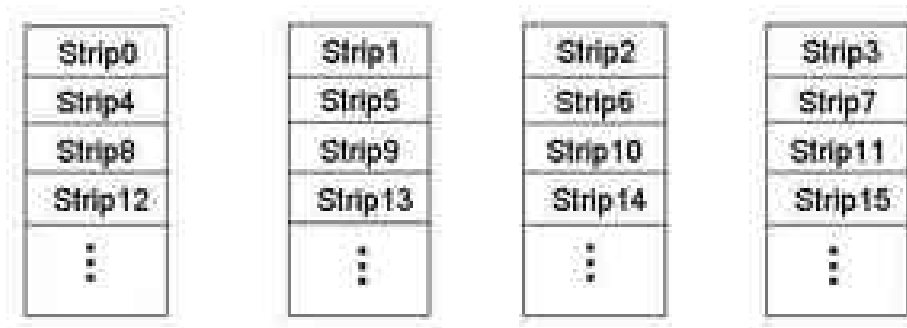
CRC does not support error correction but the CRC bits generated can be used to detect multi-bit errors. At the transmitter, we generate extra CRC bits, which are appended to the data word and sent along. The receiving entity can check for errors by re computing the CRC and comparing it with the one that was transmitted.

CRC has lesser overhead as compared to Hamming code. It is practically quite simple to implement and easy to use.

RAID

The main advantage of having an array of disks is that we could have a simultaneous I/O request. Latency could also be reduced...

RAID Level 0



- Not a true member of the RAID family.
- Does not include redundancy to improve performance.
- In few applications, capacity and performance are primary concerns than improved reliability. So RAID level 0 is used in such applications.
- The user and system data are distributed across all the disks in the array.
- Notable advantage over the use of a single large disk.
- Two requests can be issued in parallel, reducing the I/O queuing time.

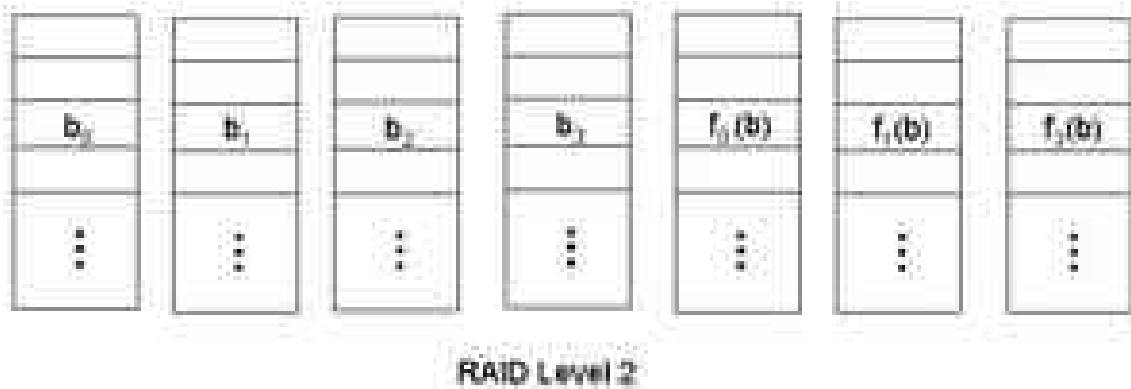
Performance of RAID Levels

Performance of RAID Levels depends upon two factors:

- Request pattern of the host system
- Layout of the data

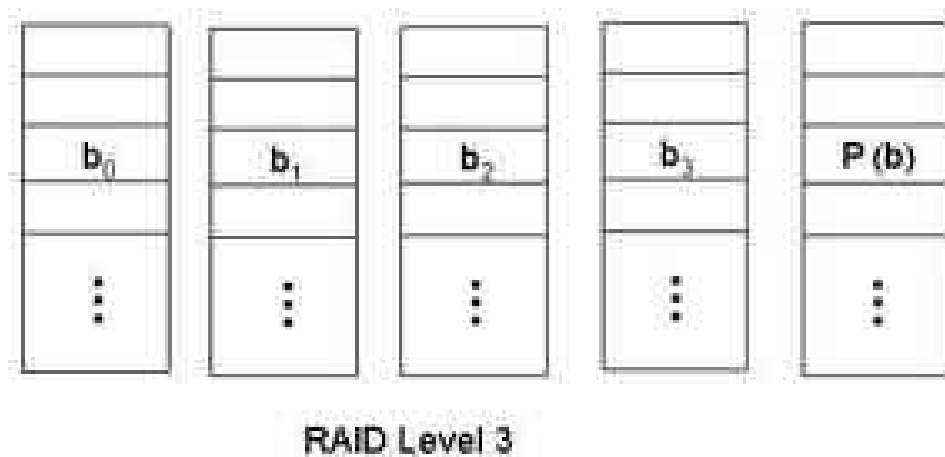
Similarities between RAID Levels 2 and 3

- Make use of parallel access techniques.
- All member disks participate in execution of every request.
- Spindles of the individual drives are synchronized
- Data striping is used.
- Strips are as small as a single byte or word.



Differences between RAID2 and RAID 3

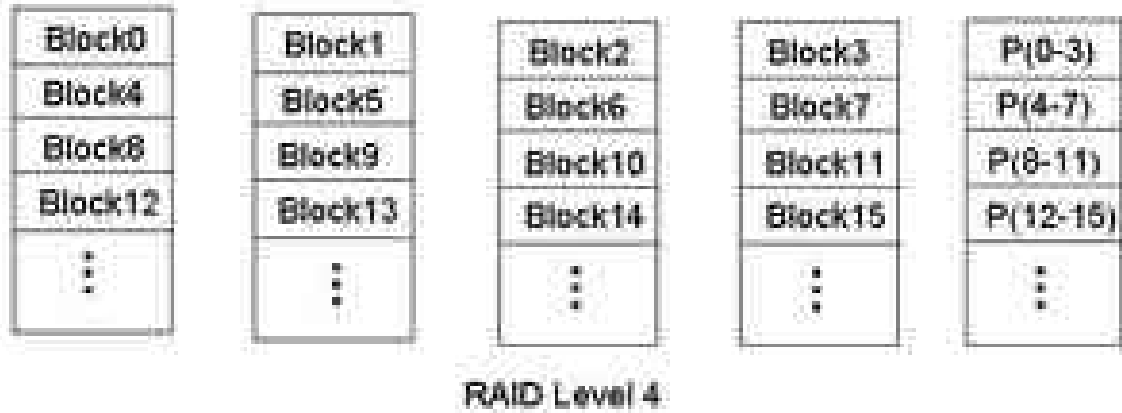
- In RAID 2, error-correcting code is calculated across corresponding bits on each data disk.
- RAID 3 requires only a single redundant disk.
- Instead of an error-correcting code, a simple parity bit is computed for the set of individual bits in RAID 3



RAID Level 4

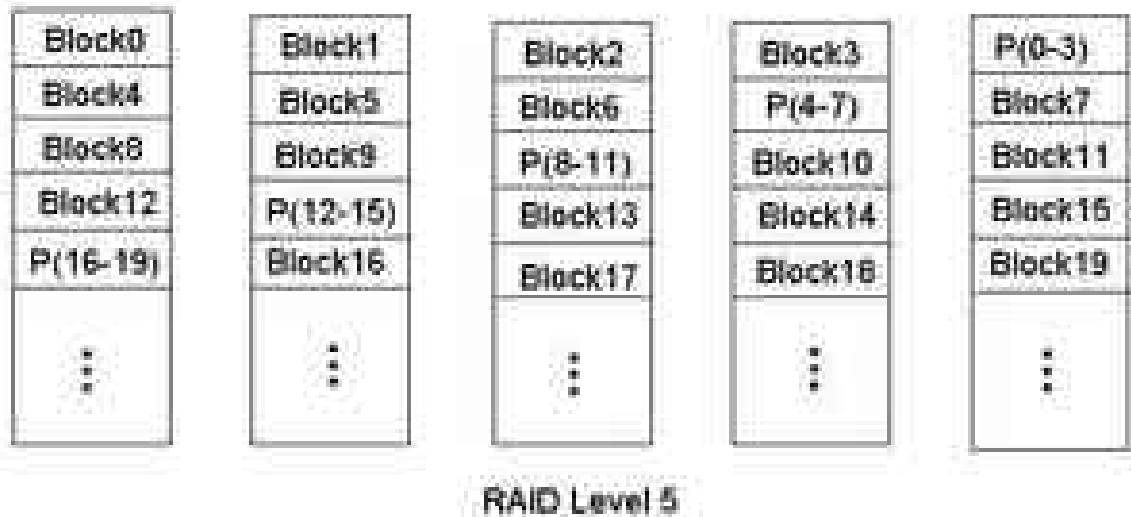
- Make use of independent access technique.

- Data striping is used.
- A bit-by-bit parity strip is calculated across corresponding strip on each data disk.
- Involves a write penalty when an I/O write request of small size is performed.
- To calculate the new parity, the array management software must read the old user parity strip.



RAID Level 5

- Organized in a similar fashion to RAID 4
- The only difference is that RAID 5 distributes the parity strips across all disks.



Lecture No. 34

Number Systems and Radix Conversion

Reading Material

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

Chapter 6
6.1, 6.2

Summary

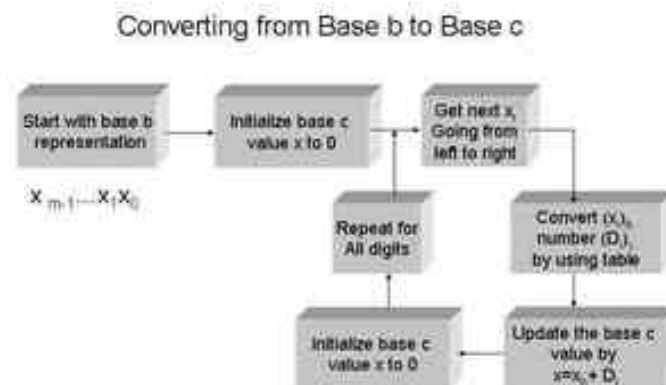
- Introduction to ALSU
- Radix Conversion
- Fixed Point Numbers
- Representation of Numbers
- Multiplication and Division using Shift Operation
- Unsigned Addition Operation

Introduction to ALSU²⁹

ALSU is a combinational circuit so inside an ALSU, we have AND, OR, NOT and other different gates combined together in different ways to perform addition, subtraction, and, or, not, etc. Up till now, we consider ALSU as a “black box” which takes two operands, a and b, at the input and has c at the output. Control signals whose values depend upon the opcode of an instruction were associated with this black box.

In order to understand the operation of the ALSU, we need to understand the basis of the representation of the numbers. For example, a designer needs to specify how many bits are required for the source operands and how many will be needed for the destination operand after an operation to avoid overflow and truncation.

Radix Conversion



Now we will consider the conversion of numbers from a representation in one base to another. As human works with base 10 and computers with base 2, this radix conversion operation is important to discuss here. We will use base c notion for decimal representation and base b for any other base. The following figure shows the algorithm of converting from base b to base c:

29

In our discussion we have used ALU and ALSU for the same thing. We use ALSU when the shift aspect also needs to be emphasized.

Example 1

Convert the hexadecimal number $B3_{16}$ to base 10.

Solution

According to the above algorithm,

$$X=0$$

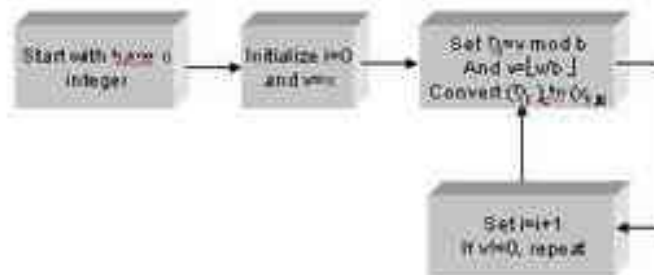
$$X = x + B (=11) = 11$$

$$X = 16 * 11 + 3 = 179$$

$$\text{Hence } B3_{16} = 179_{10}$$

The following figure shows the algorithm of converting from base c to base b:

Converting from Base c to Base b



Example 2

Convert 390_{10} to base 16.

Solution

According to the above algorithm

$$390/16 = 24 \text{ (rem=6), } x_0=6$$

$$24/16 = 1 \text{ (rem=8), } x_1=8, x_2=1$$

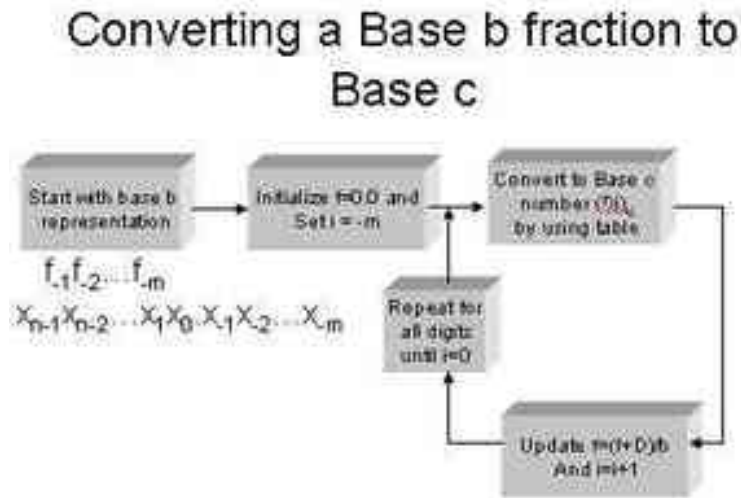
$$\text{Thus } 390_{10} = 186_{16}$$

Fixed Point Numbers

Suppose we have a number with a radix point. For example, in 16.12, there are two digits on the left side and two digits on the right of the decimal point. In this case, the radix point is a decimal point because we suppose that given number is a decimal number.

If we have an integer, then this decimal point will be on the right most position i.e. 1612.0 and if it is in fraction then decimal will be at the left most position i.e. 0.1612. There are situations when we shift the position of the radix point. Shifting of the radix point towards left or right is called scaling and we could have multiplication with a base or division by a base respectively.

The following figure shows the algorithm of converting a base b fraction to base c:



Example 3

Convert $(.4cd)_{16}$ to Base 10.

Solution

$$F = 0$$

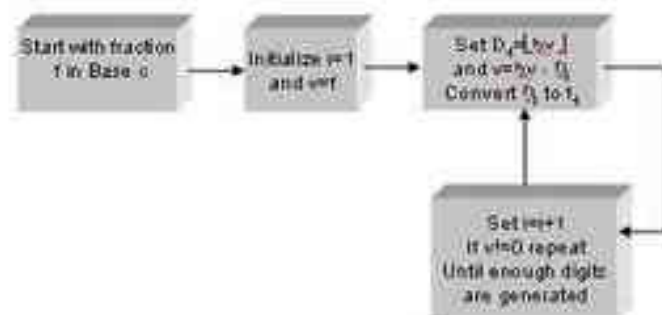
$$F = (0 + 13)/16 = 0.8125$$

$$F = (0.8125 + 12)/16 = 0.80078125$$

$$F = (0.80078125 + 4)/16 = (0.3000488)_{10}$$

The following figure shows the algorithm of converting fraction from base c to base b:

Converting a fraction from Base c to Base b



Example 4

Convert 0.24_{10} to base 2.

Solution

$$0.24 \times 2 = 0.48, f_1 = 0$$

$$0.48 \times 2 = 0.96, f_2 = 0$$

$$0.96 \times 2 = 1.92, f_3 = 1$$

$$0.92 \times 2 = 1.84, f_4 = 1$$

$$0.84 \times 2 = 1.68, f_5 = 1, \dots$$

Thus $0.24_{10} = (0.00111)_2$

Representation of Numbers

There are four possibilities to represent integers.

1. Sign magnitude form
2. Radix complement form
3. Diminished radix complement form
4. Biased representation

Sign magnitude form

- This is the simplest form for representing a signed number
- A symbol representing the sign of the number is appended to the left of the number
- This representation complicates the arithmetic operations

Radix complement form

- This is the most common representation.
- Given an m-digit base b number x, the radix complement of x is $x^c = (b^m - x) \bmod b^m$
- This representation makes the arithmetic operations much easier.

Diminished radix complement form

- The diminished radix complement of an m-digit number x is $x^c = b^m - 1 - x$
- This complement is easier to compute than the radix complement.
- The two complement operations are interconvertible, as
- $x^c = (x^{c'} + 1) \bmod b^m$

Table 6.1 of the text book shows the complement representation of negative numbers for radix complement and diminished radix complement form:

Table 6.2 of the text book shows the base 2 complement representation for 8-bit 2's and 1's complement numbers.

Example 5

The following table shows the decimal values in 2's complement, 1's complement, sign magnitude, 16's complement and in unsigned form:

Decimal	2's complement	1's complement	Sign-magnitude	16's complement	Unsigned
27	011011	011011	011011	1B	11011
.17	0.00101011	0.00101011	0.00101011	0.2B	0.00101011
-26	100110	100101	111010	E6	-
-0.57	1.01101110	1.01101101	1.10010010	F.5E	-

Multiplication and Division using Shift Operation

Shift left and shift right are two important operations used for various purposes. One typical example could be multiplication or division by base b . The following examples explain multiplication and division by using shift operation.

Example 6

- 6×4

$$00110_2 \times 4_{10} = 11000_2 = 24_{10}$$

Overflow would occur if we will use 4 bits instead of 5 bits here.

- $60/16$

$$0111100_2 / 16_{10} = 0000011_2 = 3_{10}$$

The fractional portion of the result is lost.

Example 7

- -6×4

$$-6 = (11010)_2$$

$$-6 \times 4 = (01000)_2 = 8 \text{ which is wrong!}$$

using less no. of bits might change sign

$$\text{So, } -6 = (111010)_2$$

$$-6 \times 4 = (101000)_2 = -24$$

Example 8

Multiplication and division of negative numbers

Solution

$$-24 \times 2$$

$$-24 = (101000)_2$$

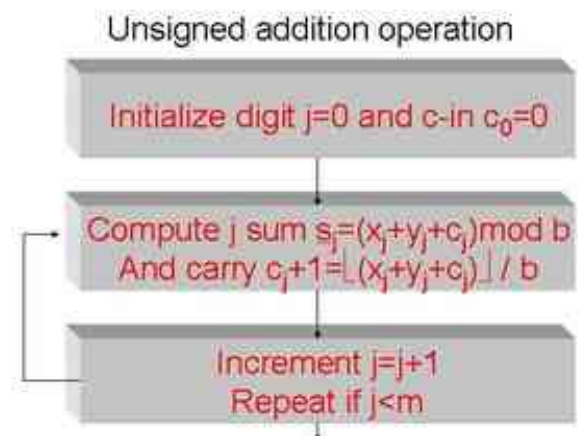
$$-24 \times 2 = (010100)_2 = 20 \quad -24 \times 2 = (110100)_2 = -12 \text{ Changing the size of the number,}$$

$$24 = 011000 \text{ (n=6) to } 00011000 \text{ (n=8)}$$

$$-24 = 101000 \text{ (n=6) to } 11101000 \text{ (n=8)}$$

Unsigned Addition Operation

The following diagram shows the digit wise procedure for adding m -digit base b numbers, x and y :



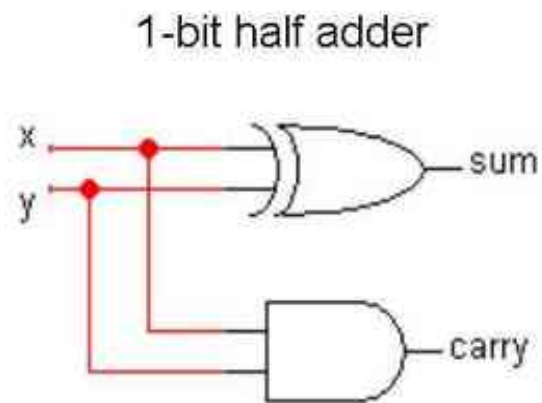
Example 9

Unsigned addition in base 2 and base16.

Solution

Base 16 addition	Base 2 addition
$ \begin{array}{r} AB42_{16} \\ + 31C1_{16} \\ \hline \text{carry } 0100 \\ \text{sum } DD03_{16} \end{array} $	$ \begin{array}{r} 100011_2 \\ + 011011_2 \\ \hline \text{carry } 000110 \\ \text{sum } 111110_2 \end{array} $

The following diagram shows the logic circuit for 1-bit half adder. It takes two 1-bit inputs x and y and as a result, we get a 1-bit sum and a 1-bit carry. This circuit is called a half adder because it does not take care of input carry. In order to take into account the effect of the input carry, a 1-bit full adder is used which is also shown in the figure. We can add two m -bit numbers by using a circuit which is made by cascading m 1-bit full adders.



The situation, when addition of unsigned m -bit numbers results in an $m+1$ bit number, is called overflow. Overflow is treated as exception in some processors and the overflow flag is used to record the status of the result.

Lecture No. 35

Multiplication and Division of Integers

Reading Material

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

Chapter 6
6.3, 6.4

Summary

- Overflow
- Different Implementations of the adder
- Unsigned and Signed Multiplication
- Integer and Fraction Division
- Branch Architecture

Overflow

When two m-bit numbers are added and the result exceeds the capacity of an m-bit destination, this situation is called an overflow. The following example describes this condition:

Example 1

Overflow in fixed point addition:

Base 2	Base 8	Base 16
$\begin{array}{r} 1011.1 \\ +1110.0 \\ \hline c\ 1100 \\ s\ 11001.1 \end{array}$	$\begin{array}{r} 7.25 \\ +6.76 \\ \hline c\ 11.1 \\ s\ 16.23 \end{array}$	$\begin{array}{r} C36.A \\ +72B.3 \\ \hline c\ 1010 \\ s\ 1361.D \end{array}$

In these three cases, the fifth position is not allowed so this results in an overflow.

Different Implementations of the Adder

For a binary adder, the sum bit is obtained by following equation:

$$s_j = x_j y_j c_j + x_j y_j c_j + x_j y_j c_j + x_j y_j c_j$$

and the equation for carry bit is

$$c_{j+1} = x_j y_j + x_j c_j + y_j c_j$$

where x and y are the input bits.

The sum can be computed by the two methods:

- Ripple Carry Adder

- Carry Look ahead Adder

Ripple Carry Adder

In this adder circuit, we feed carry out from the previous stage to the next stage and so on. For 64 bit addition, 126 logic levels are required between the input and output bits. The logic levels can be reduced by using a higher base (Base 16). This is a relatively slow process.

Complement Adder/Subtractor

We can perform subtraction using an unsigned adder by

- Complement the second input
- Supply overflow detection hardware

2's Complement Adder/Subtractor

A combined adder/subtractor can be built using a mux to select the second adder input. In this case, the mux also determines the carry-in to the adder. The equation for mux output is :

$$q_j = \bar{y}_j \bar{r} + y_j r$$

Carry Look ahead Adder

The basic idea in carry look ahead is to speed up the ripple carry by determining whether the carry is generated at the j position after addition, regardless of the carry-in at that stage or the carry is propagated from input to output in the digit.

This results in faster addition and lesser propagation delay of the carry bits. It divides the carry into two logical variables G_j (generate) and P_j (propagate). These variables are defined as:

$$G_j = x_j y_j$$

$$P_j = x_j + y_j$$

Hence the carry out will be

$$C_{j+1} = G_j + P_j c_j$$

Here the G and P each require one gate, and the sum bit needs two more gates in the full adder. This results in a less complexity i.e. $\log(m)$ which is much less as compare to ripple carry adder where complexity is m (m is the number of bits of a digit to be added). Ripple carry and look ahead schemes can be mixed by producing a carry-out at the left end of each look ahead module and using ripple carry to connect modules at any level of the look ahead tree.

Unsigned Multiplication

The general schema for unsigned multiplication in base b is shown in Figure 6.5 of the text book.

Parallel Array Multiplier

Figure 6.6 of the text book shows the structure of a fully parallel array multiplier for base b integers. All signal lines carry base b digits and each computational block consists of a full adder with an AND gate to form the product $x_i y_j$. In case of binary, m^2 full adders are required and the signals will have to pass through almost $4m$ gates.

Series parallel Multiplier

A combination of parallel and sequential hardware is used to build a multiplier. This results in a good speed of operation and also saves the hardware.

Signed Multiplication

The sign of a product is easily computed from the sign of the multiplier and the multiplicand. The product will be positive if both have same sign and negative if both have different sign. Also, when two unsigned digits having m and n bits respectively are multiplied, this results in a $(m+n)$ –bit product, and $(m+n+1)$ -bit product in case of sign digits. There are three methods for the multiplication of sign digits:

1. 2's complement multiplier
2. Booth recoding
3. Bit-Pair recoding

2's complement Multiplication

If numbers are represented in 2's complement form then the following three modifications are required:

1. Provision for sign extension
2. Overflow prevention
3. Subtraction as well as addition of the partial product

Booth Recoding

The Booth Algorithm makes multiplication simple to implement at hardware level and speed up the procedure. This procedure is as follows:

- Start with LSB and for each 0 of the original number, place a 0 in the recorded number until a 1 is indicated.
- Place a 1 for 1 in the recorded table and skip any succeeding 1's until a 0 is encountered.
- Place a 0 with 1 and repeat the procedure.

Example 2

Recode the integer 485 according to Booth procedure.

Solution

Original number:

$$00111100101 = 256 + 128 + 64 + 32 + 4 + 1 = 485$$

Recoded Number:

$$\overline{0}1\overline{0}00101111 = +512 - 32 + 8 - 4 + 2 - 1 = 485$$

Bit-Pair Recoding

Booth recoding may increase the number of additions due to the number of isolated 1s. To avoid this, bit-pair recoding is used. In bit-pair recoding, bits are encoded in pairs so there are only $n/2$ additions instead of n .

Division

There are two types of division:

- Integer division
- Fraction division

Integer division

The following steps are used for integer division:

1. Clear upper half of dividend register and put dividend in lower half. Initialize quotient counter bit to 0
2. Shift dividend register left 1 bit
3. If difference is +ve, put it into upper half of dividend and shift 1 into quotient. If – ve, shift 0 into quotient
4. If quotient bits < m, goto step 2
5. m-bit quotient is in quotient register and m-bit remainder is in upper half of dividend register

Example 3

Divide 47_{10} by 5_{10} .

Solution

D=000000 101111, d=000101			
D	000001 011110		
d	000101		
Diff(-)		q	0
D	000010 111100		
d	000101		
Diff(-)		q	00
D	000101 111000		
d	000101		
Diff(+)		q	001
D	000001 110000		
d	000101		
Diff(-)		q	0010
D	000011 100000		
d	000101		
Diff(-)		q	00100
D	000111 000000		
d	000101		
Diff(+)	000010	q	001001
Hence remainder = $(000010)_2 = 2_{10}$			
Quotient = $(001001)_2 = 9_{10}$			

Fraction Division

The following steps are used for fractional division:

1. Clear lower half of dividend register and put dividend in upper half. Initialize quotient counter bit to 0
2. If difference is +ve, report overflow
3. Shift dividend register left 1 bit
4. If difference is +ve, put it into upper half of dividend and shift 1 into quotient. If negative, shift 0 into quotient
5. If quotient bits < m, go to step 3
6. m-bit quotient has decimal at the left end and remainder is in upper half of dividend register

Branch Architecture

The next important function performed by the ALU is branch. Branch architecture of a machine is based on

1. Condition Codes
2. Conditional Branches

Condition Codes

Condition Codes are computed by the ALU and stored in processor status register. The 'comparison' and 'branching' are treated as two separate operations. This approach is not used in the SRC. Table 6.6 of the text book shows the condition codes after subtraction, for signed and unsigned x and y. Also see the SRC Approach from text book.

Usually implementation with flags is easier however it requires status registers. In case of branch instructions, decision is based on the branch itself.

Note: For more information on this topic, please see chapter 6 of the text book. 

Lecture No. 36

Floating-Point Arithmetic

Reading Material

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

Chapter 6
6.3.2, 6.4, 6.4.1, 6.4.2, 6.4.3

Summary

- NxN Crossbar Design for Barrel Rotator
- Barrel Shifter with Logarithmic Number of Stages
- ALU Design
- Floating-Point Representations
- IEEE Floating-Point Standard
- Floating-Point Addition and Subtraction
- Floating-Point Multiplication
- Floating-Point Division

NxN Crossbar Design for Barrel Rotator

Figure 6.11 of the text book

The figure shows an NxN crossbar design for barrel rotator. x indicates the input. So $x_0, x_1, \dots, x_{n-1}$ are applied to the rows. The vertical lines are indicated by $y_1, y_2, \dots, y_{n-1}$ where y shows the output. So this forms a cross of x and y and the number of cross points are NxN. There is also a connection between each input and output using a tri-state buffer. At the input, we have a decoder which is used to select the shift count. Each output from the decoder is connected diagonally to the tri-state buffers. This arrangement requires N^2 gates.

Barrel Shifter with Logarithmic Number of Stages

Another alternate to an NxN crossbar barrel rotator is a logarithmic barrel shifter. This design is time-space trade-off. In this case, the number of shifts required is eight, and then there will be three stages for this purpose. Now a word is passed as input to the shifter. There are two possibilities. First the input word is passed to the next stage without any shift. This process is called bypass and second option is shift. The word is passed to the next stage after shift.

For the first stage, we have 1-bit right shift, for second stage, 2-bit right shift and so on. There is also a shift count unit which controls the number of shifts. For example, if 1-bit shift is required then only s_0 will be one and other signals from shift count will be zero. If we want a 3-bit shift, then s_0 and s_1 will be 1 and all other signals will be zero.

The figure also shows one shift/bypass cell which is a combinational logic circuit. A shift/bypass signal decides whether the input word should be shifted or bypassed. This design requires only $O(N \log N)$ switches but propagation delay has increased i.e. from $O(1)$ to $O(\log N)$.

Figure 6.12 of the text book

ALU Design

ALU is a combination of arithmetic, logic and shifter unit along with some multiplexers and control unit. The idea is that based on the op-code of an instruction, appropriate control signals are activated to perform required ALU operation. Figure 6.13 of the text book

The diagram shows two inputs x and y and one output z. All these are of n-bits. The inputs x and y are simultaneously provided to arithmetic, logic and shifter unit. There is a control unit which accepts op-code as input. Based on the op-code, it provides control signals to arithmetic, logic and shifter unit. The control unit also provides control signals to the two multiplexers. One mux has three inputs; each from arithmetic, logic and shifter unit and its output is z. The second mux provides status output corresponding to condition codes.

Floating Point Representations

Example

$$-0.5 \times 10^{-3}$$

$$\text{Sign} = -1$$

$$\text{Significand} = 0.5$$

$$\text{Exponent} = -3$$

$$\text{Base} = 10 = \text{fixed for given type of representation}$$

Significant is also called mantissa.

In computers, floating-point representation uses binary numbers to encode significant, exponent and their sign in a single word.

The diagram on Page 293 of the text shows an m-bit floating point number where s represents the sign of the floating point number. If $s = 1$ then the floating-point number will be a positive number; if $s = 0$ then it will be a negative number. The e field shows the value of exponent. To represent the exponent, a biased representation is used. So we represent e^+ instead of e to show biased representation. In this technique, a number is added to the exponent so that the result is always positive. In general floating point numbers are of the form.

$$(-1)^s \times f \times 2^e$$

Normalization

A normalized, non-zero floating point number has a significand whose left-most digit is non-zero and is a single number.

Example

$$0.56 \times 10^{-3} \dots \dots \dots \text{(Not normalized)}$$

$$5.6 \times 10^{-4} \dots \dots \dots \text{(Normalized form)}$$

Same is the case for binary.

IEEE Floating-Point Standard

IEEE floating -point standard has the following features.

Single-Precision Binary Floating Point Representation

- 1-bit sign
- 8-bit exponent
- 23-bit fraction
- A bias of 127 is used.

Figure 6.15 of the text book

Double precision Binary Floating Point Representation

- 1-bit sign
- 11-bit exponent
- 52-bit fraction
- Exponent bias is 1023

Figure 6.16 of the text book.

Overflow

In table 6.7 of the text book, $e^{\wedge}=255$, denotes numbers with no numeric value including $+\infty$ and $-\infty$ and called Not-a-Number or NaN. In computers, a floating-point number ranges from $1.2 \times 10^{-38} \leq x \leq 3.4 \times 10^{38}$ can be represented. If a number does not lie in this range, then overflow can occur.

Overflow occurs when the exponent is too large and can not be represented in the exponent field.

Floating –Point Addition and Subtraction

The following are the steps for floating-point addition and subtraction.

- Unpack sign , exponent and fraction fields
- Shift the significant
- Perform addition
- Normalize the sum
- Round off the result
- Check for overflow

Figure 6.17 of the text book.

Example 1

Perform addition of the following floating-point numbers.

0.5_{10} , -0.4375_{10}

Binary:

$$0.5_{10} = 1/2_{10} = 0.1_2 = 1.000 \times 2^{-1}$$

$$-0.4375_{10} = -7/16_{10} = -7/24 = -0.0111_2 = -1.110 \times 2^{-2}$$

$$\text{Align: } -1.110 \times 2^{-2} \rightarrow -0.111 \times 2^{-1}$$

$$\text{Addition: } 1.000 \times 2^{-1} + (-0.111 \times 2^{-1}) = 0.001 \times 2^{-1}$$

Normalization of Sum:

$$\begin{aligned} 0.001_2 \times 2^{-1} &= 0.010_2 \times 2^{-2} \\ &= 1.000_2 \times 2^{-4} \end{aligned}$$

Hardware Structure for Floating-Point Add and Subtract Figure 6.17 of the text book.

Floating-Point Multiplication

The floating-point multiplication uses the following steps:

- Unpack sign, exponent and significands
- Apply exclusive-or operation to signs, add exponents and then multiply significands.
- Normalize, round and shift the result.

- Check the result for overflow.
- Pack the result and report exceptions.

Floating-Point Division

The floating-point division uses the following steps:

- Unpack sign, exponent and significands
- Apply exclusive-or operation to signs, subtract the exponents and then divide the significands.
- Normalize, round and shift the result.
- Check the result for overflow.
- Pack the result and report exceptions.

Lecture No. 37

Components of Memory Systems

Reading Material

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

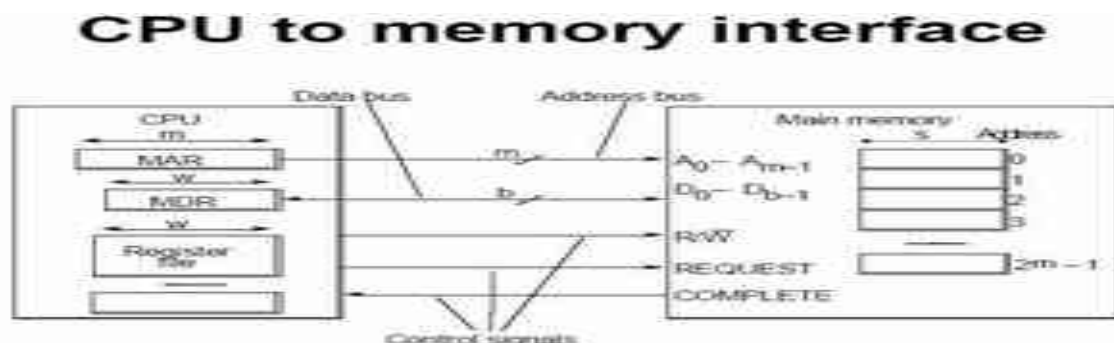
Chapter 7
7.1, 7.2

Summary

- CPU to Memory Interface
- Static RAM cell Organization and Operation
- One & two Dimensional Memory Cells
- Matrix and Tree Decoders
- Dynamic RAM

CPU to Memory Interface

The memory address register (MAR) is m -bits wide and contains memory address generated by the CPU directly connected to the m -bit wide address bus. The memory buffer register (MBR) is w -bit wide and contains a data word, directly connected to the data bus which is b -bit wide. The register file is a collection of 32, 32-bit wide registers used for data transfer between memory and the CPU. Memory address ranges from 0 to 2^m-1 . There also exist three control signals: $R/\overline{W}$, REQUEST, and COMPLETE. When $R/\overline{W}$ signal is high, this would correspond to a read operation equivalent to having an input data to the CPU and output from the memory. If this signal is low then it would be a write operation and data would come from the CPU as an output and it would be written into a portion in the memory. In this case, the REQUEST signal coming from the CPU telling the memory that some interaction is required between the CPU and memory. As a result of this request (either read/write), along with the signal on the control and the address on the address bus, we might have the corresponding data on the data bus for a read operation and after the operation is complete, the memory would issue a control signal which corresponds in this case to COMPLETE. Figure 7.1 of the text book.



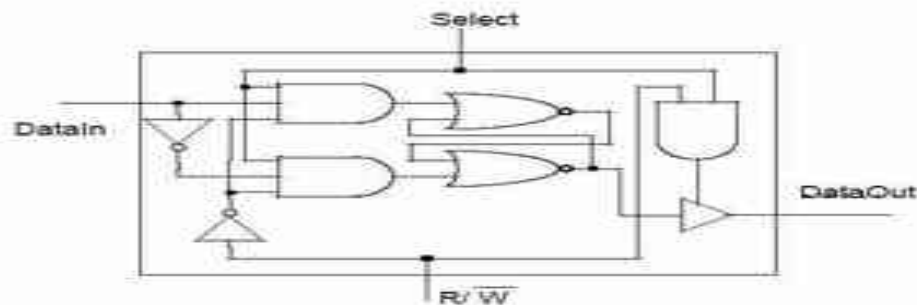
Static RAM Cell Organization and Operation

A Typical Memory Cell

A memory cell provides four functions: Select, DataIn, DataOut, and Read/Write. DataIn means input and DataOut means output. The select signal would be enabled to get an operation of Read/Write from this cell.

Figure 7.3 of the text book.

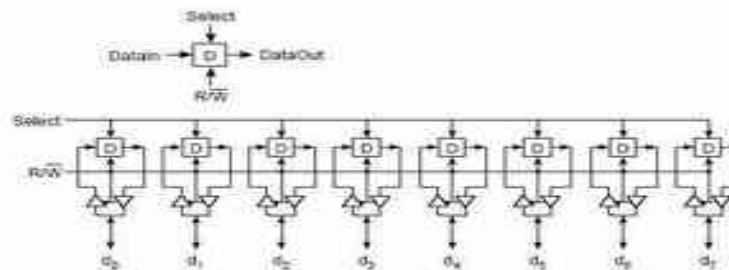
Memory cell internal diagram



1×8 Memory Cell Array (1D)

In this arrangement, each block is connected through a bi-directional data bus implemented with 2 tri-state buffers. $R/\overline{W}$ and Select signals are common to all these cells. This 1-dimensional memory array could not be very efficient, if we need to have a very large memory.

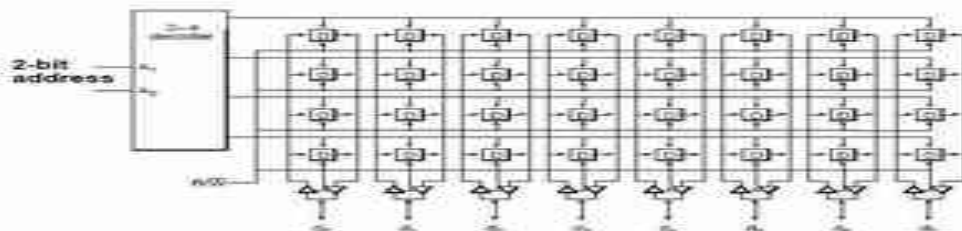
1x8 Memory Cell Array (1D)



4×8 Memory Cell Array (2D)

In this arrangement, 4×8 memory cell array is arranged in 2-dimensions. At the input, we have a 2×4 decoder. Two address bits at the input A0 and A1 would be decoded into 4 select lines. The decoder selects one of four rows of cells and then $R/\overline{W}$ signal specifies whether the row will be read or written.

4x8 Memory Cell Array (2D)

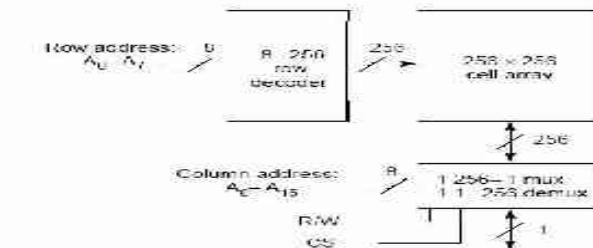


A 64k×1 Static RAM Chip

The cell array is indicated as 256 × 256. So, there would be 256 rows and 256 columns. A 64k × 1 cell array requires 16 address lines, a read/write line, $R/\overline{W}$, a chip select line, CS, and only a

single data line. The lower order 8-address lines select one of the 256 rows using an 8-to-256 line row decoder. Thus the selected row contains 256 bits. The higher order 8-address lines select one of those 256 bits. The 256 bits in the row selected flow through a 256-to-1 line multiplexer on a read. On a memory write, the incoming bit flows through a 1-to-256 line demultiplexer that selects the correct column of the 256 possible columns.

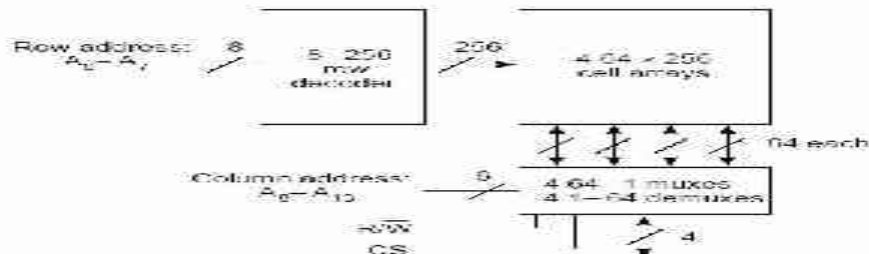
A 64 K x 1 Static RAM Chip



A 16k×4 Static RAM Chip

In this case, memory is arranged in the form of four 64×256 memory cells. Four bits can be read and written at a time. For this, we use one 8-256 row decoder, four 64-1 muxes and four 1-64 demuxes. The lower address lines (A_0-A_7) are decoded into 2^8 lines, 2^6 lines from these 2^8 are used to select row from one of the four 64×256 cell array and the remaining 2^2 lines are used to select one of the 64×256 cell array. Now the upper address lines (A_8-A_{13}) are input into the 4 muxes and their output is used to select the required column from the four 64×256 cell arrays. Control lines read/write, $R/\overline{W}$, chip select, CS, are just similar to previous arrangement.

A 16 K x 4 SRAM Chip



Matrix and Tree Decoders

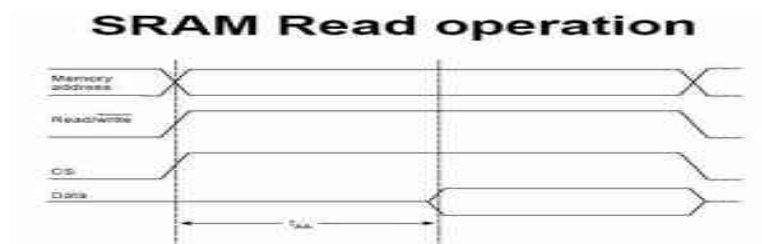
A typical one level decoder has n inputs and 2^n output, using one level of gates, each with a fan-in of n . Two level decoders are limited in size because of high gate fan-in. In order to reduce the gate fan-in to a value of 8 or 6, tree and matrix decoders are utilized.

Six Transistor SRAM Cell

In this arrangement, the cross connection is through inverters to make the latch, the basic storage cell. This implementation uses six transistor cells. One transistor is used to implement each of the two inverters, two transistors are used to control access to the inverters for reading and writing, and two are used as active loads.

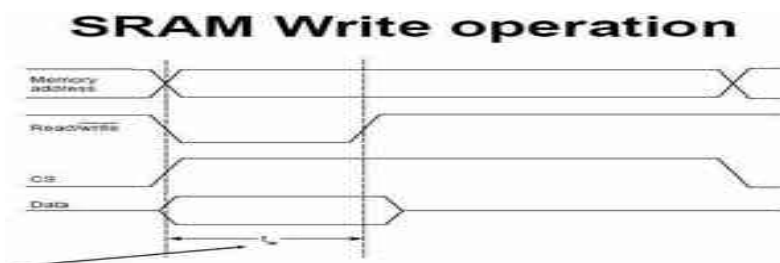
SRAM Read Operation

First of all, the CPU provides the address on the external address bus. The read/write signal becomes active high. After time " t_{AA} ", the data becomes available on the data bus. The chip retains this data on the data lines until the control signals are de asserted.



SRAM Write Operation

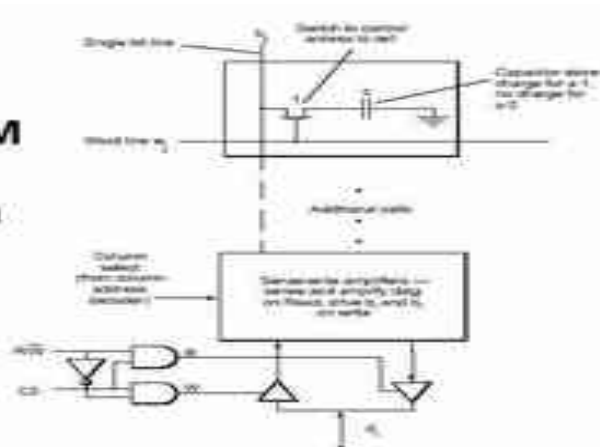
In the case of write cycle, the major difference is that along with the address the CPU has also provided the data on the data bus. The chip select, CS, is immediately provided and write signal is made low. The $\overline{R/\overline{W}}$ line must be held valid for a minimum time interval t_w , the write time, until data, address, and control information have been propagated to the cell and strobe into it. During this period the data lines must be driven with the data to be written.



Dynamic RAM

As an alternate to the SRAM cell, the data can be stored in the form of a charge on a capacitor (a charging/discharging transistor that can become a valid memory element), and this type of memory is called dynamic memory. The capacitor has to be refreshed and recharged to avoid data loss.

Dynamic RAM Cell organization and operation



Dynamic RAM Cell Operation

In a DRAM cell, the storage capacitor will discharge in around 4-15ms. Refreshing the capacitor by reading or sensing the value on bit line, amplifying it, and placing it back on to the bit line is required. The need to refresh the DRAM cell complicates the DRAM system design.

For details, refer to Chapter 7 of the text book.

Lecture No. 38

Memory Modules

Reading Material

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

Chapter 7
7.2.6, 7.3

Summary

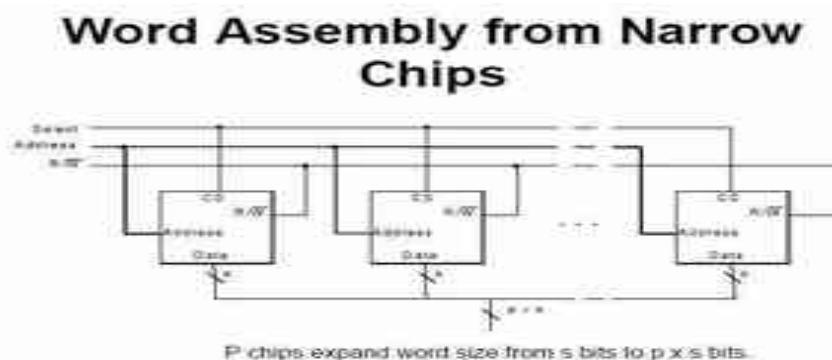
- Memory Modules
- Read Only Memory (ROM)
- Cache

Memory Module

Static RAM chips can be assembled into systems without changing the timing characteristics of a memory access. Dynamic RAM chips, however, have enough timing complexity that a memory module built from dynamic RAM chips will have complex control. The cause of timing complexity is the time-multiplexed row and column addresses, and the refresh operation.

Word Assembly from Narrow Chips

Chips can be combined to expand the memory word size while keeping the same number of words. Address, chip select, and R/W signals are connected in parallel to all the chips. Only the data signals are kept separate, with those from each chip supplying different bits of the wider word. For high capacity memory chips, narrow words are used. This is because adding a data pin to a chip with 2^m words of s bits increases the number of bits it can store by only a factor of $(s+1)/s$, while adding an address pin always doubles the capacity.



Dynamic RAM Module with Refresh Control

For Dynamic RAM chips the total address is divided into row and column address. Row address strobe signal RAS and a column strobe signal CAS are used to differentiate between these two signals.

Read Only Memory (ROM)

ROM is the read-only memory which contains permanent pattern of data that cannot be changed. ROM is nonvolatile i.e. it retains the information in it when power is removed from it. Different types of ROMs are discussed below.

PROM

The PROM stands for Programmable Read only Memory. It is also nonvolatile and may be written into only once. For PROM, the writing process is performed electrically in the field. PROMs provide flexibility and convenience.

EPROM

Erasable Programmable Read-only Memory or EPROM chips have quartz windows and by applying ultraviolet light erase the data can be erased from the EPROM. Data can be restored in an EPROM after erasure. EPROMs are more expensive than PROMs and are generally used for prototyping or small-quantity, special purpose work.

EEPROM

EEPROM stands for Electrically Erasable Programmable Read-only Memory. This is a read-mostly memory that can be written into at any time without erasing prior contents; only the byte or bytes addressed are updated. The write operation takes considerably longer than the read operation. It is more expensive than EPROM.

Flash Memory

An entire flash memory can be erased in one or a few seconds, which is much faster than EPROM. In addition, it is possible to erase just blocks of memory rather than an entire chip.

Cache

Cache by definition is a place for safe storage and provides the fastest possible storage after the registers. The cache contains a copy of portions of the main memory. When the CPU attempts to read a word from memory, a check is made to determine if the word is in the cache. If so, the word is delivered to the CPU. If not, a block of the main memory, consisting of some fixed number of words, is read into the cache and then the word is delivered to the CPU.

Spatial Locality

This would mean that in a part of a program, if we have a particular address being accessed then it is highly probable that the data available at the next address would be highly accessed.

Temporal Correlation

In this case, we say that at a particular time, if we have utilized a particular part of the memory then we might access the adjacent parts very soon.

Cache Hit and Miss

When the CPU needs some data, it communicates with the cache, and if the data is available in the cache, we say that a cache hit has occurred. If the data is not available in the cache then it interacts with the main memory and fetches an appropriate block of data. This is a cache miss.

Lecture No. 39

The Cache

Reading Material

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

Chapter 7
7.4, 7.5

Summary

- Cache Organization and Functions
- Cache Controller Logic
- Cache Strategies

Cache Organization and Functions:

The working of the cache is based on the principle of locality which has two aspects. **Spatial Locality**: refers to the fact when a given address has been referenced, the next address is highly probable to be accessed within a short period of time.

Temporal Locality refers to the fact that once a particular data item is accessed, it is likely that it will be referenced again within a short period of time.

To exploit these two concepts, the data is transferred in blocks between cache and the main memory. For a request for data, if the data is available in the cache it results in a cache hit. And if the requested data is not present in the cache, it is called a cache miss. In the given example program segment, spatial locality is shown by the array ALPHA, in which next variable to be accessed is adjacent to the one accessed previously. Temporal locality is shown by the reuse of the loop variable 100 times in For loop instruction.

Int ALPHA [100], SUM;

SUM=0;

For (i=0; i<100; i++)

{SUM= SUM+ALPHA[i];}

Cache Management

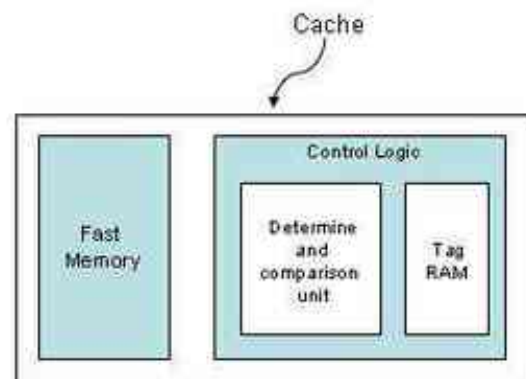
To manage the working of the cache, cache control unit is implemented in hardware, which performs all the logic operations on the cache. As data is exchanged in blocks between main memory and cache, four important cache functions need to be defined.

- Block Placement Strategy
- Block Identification
- Block Replacement
- Write Strategy

Block Diagram of a Cache System In the figure, the block diagram of a system using cache is shown. It consists of two components.

- Fast Memory
- Control Logic Unit

Control logic is further divided into two parts.



Determine and Comparison Unit: For determining and comparisons of the different parts of the address and to evaluate hit or miss.

Tag RAM: Second part consists of tag memory which stores the part of the memory address (called tag) of the information (block) placed in the data cache. It also contains additional bits used by the cache management logic.

Data Cache: is a block of fast memory which stores the copies of data and instructions frequently accessed by the CPU.

Cache Strategies

In the next section we will discuss various cache functions, and strategies used to implement these functions.

Block Placement

Block placement strategy needs to be defined to specify where blocks from main memory will be placed in the cache and how to place the blocks. Now various methods can be used to map main memory blocks onto the cache. One of these methods is the associative mapping explained below.

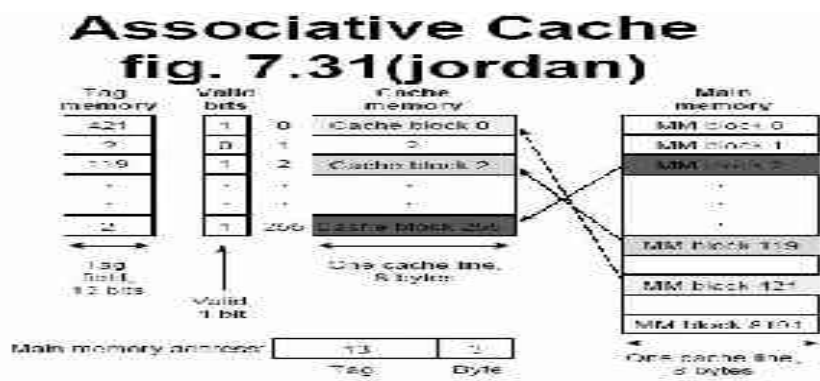
Associative Mapping:

In this technique, block of data from main memory can be placed at any location in the cache memory. A given block in cache is identified uniquely by its main memory block number, referred to as a tag, which is stored inside a separate tag memory in the cache. To check the validity of the cache blocks, a valid bit is stored for each cache entry, to verify whether the information in the corresponding block is valid or not. Main memory address references have two fields.

- The word field becomes a “cache address” which specifies where to find the word in the cache.
- The tag field which must be compared against every tag in the tag memory.

Associative Mapping Example

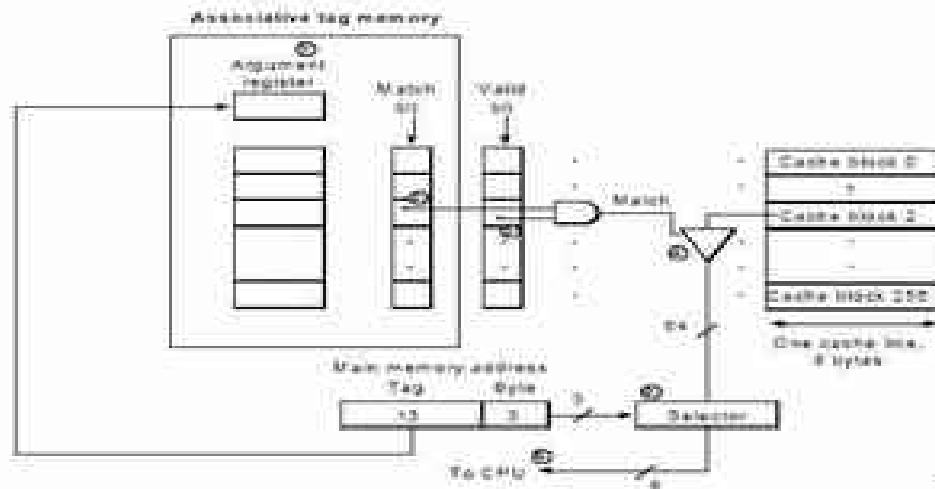
Refer to Book Ch.7 Section (7.5) Figure 7.31(page 350-351) for detailed explanation.



Mechanism of the Associative Cache Operation

For details refer to book Ch.7, Section 7.5, Figure 7.32 (Page 351-352).

fig. 7.32 (Jordan)



Direct Mapping

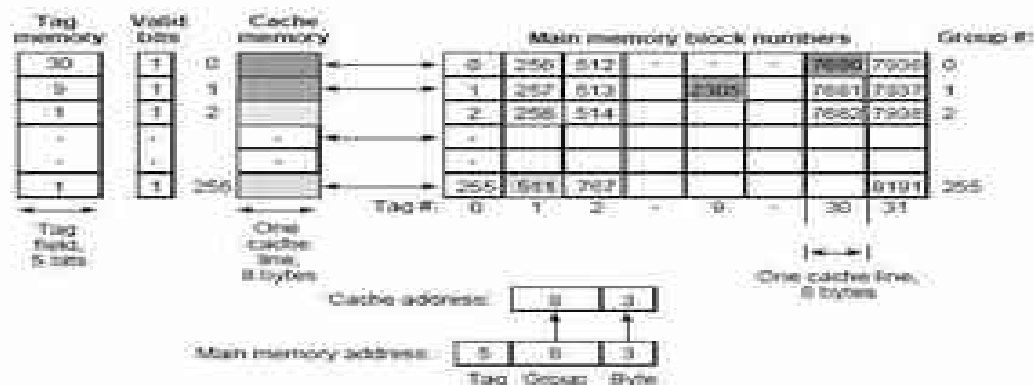
In this technique, a particular block of data from main memory can be placed in only one location into the cache memory. It relies on principle of locality. Cache address is composed of two fields:

- Group field
- Word field

Valid bit specifies that the information in the selected block is valid.

For a direct mapping example, refer to the book Ch.7, Section 7.5, Figure 7.33 (page 352 – 353).

Direct mapped cache fig. 7.33 (Jordan)



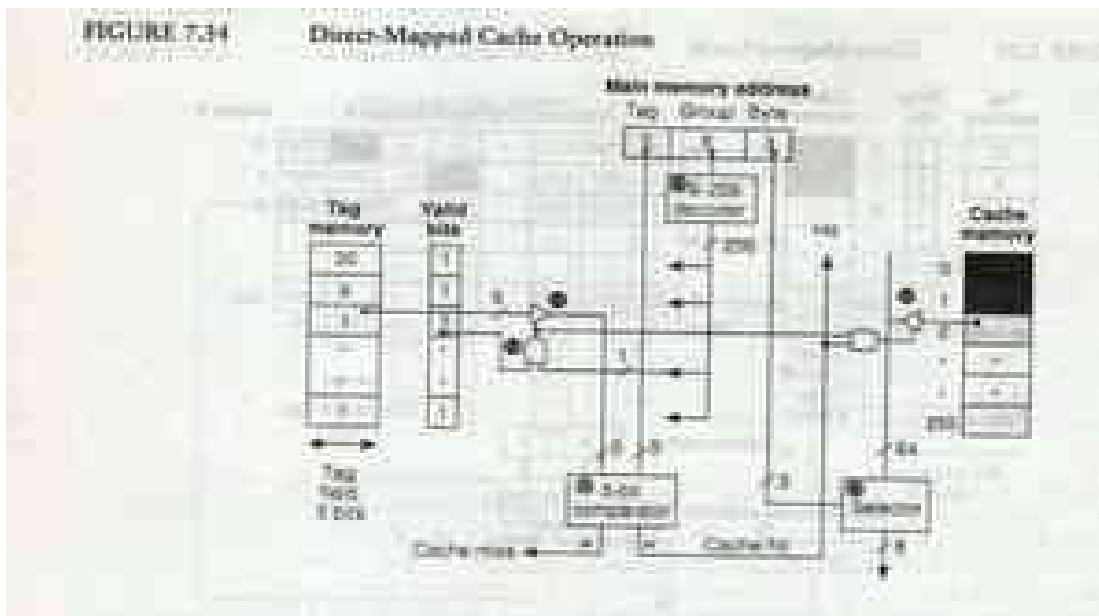
Logic Implementation of the Controller for Direct Mapping

Logic design for the direct mapping is simpler as compared to the associative mapping.

Only one tag entry needs to be compared with the part of the address called group field.

Tasks Required For Direct Mapping Cache:

For details refer to the book Ch. 7, Section 7.5, Figure 7.34 (Page 353-354).



Cache Design: Direct Mapped Cache

To understand the principles of cache design, we will discuss an example of a direct mapped cache.

The size of the main memory is 1 MB. Therefore 20 address bits needs to be specified. Assume that the block size is 8 bytes. Cache memory is assumed to be 8 KB organized as 1 K lines of cache memory. Cache memory addresses will range from 0 up to 1023. Now we have to specify the number of bits required for the tag memory. The least significant three bits will define the block. The next 10 bits will define the number of bits required for the cache. The remaining 7 bits will be the width of the tag memory.

Main memory is organized in rectangular form in rows and columns. Number of rows would be from 0 up to 1023 defined by 10 bits. Number of rows in the main memory will be the same as number of lines in the cache. Number of columns will correspond to 7 bits address of the tag memory. Total number of columns will be 128 starting from 0 up to 127.

With direct mapping, out of any particular row only one block could be mapped into the cache. Total number of cache entries will be 1024 each of 8 bytes.

Advantage:

Simplicity

Disadvantage:

Only a single block from a given group is present in cache at any time. Direct map Cache imposes a considerable amount of rigidity on cache organization.

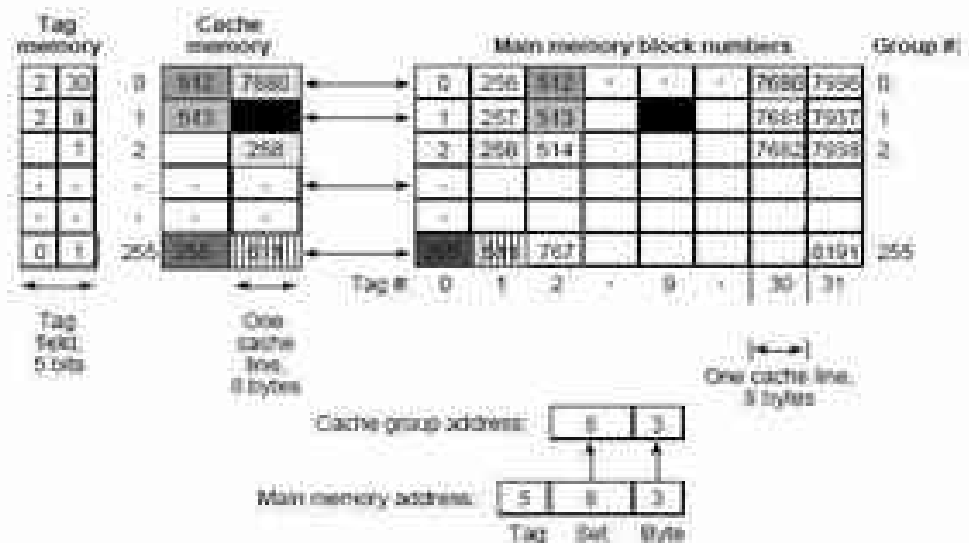
Set Associative Mapping

In this mapping scheme, a set consisting of more than one block can be placed in the cache memory.

The main memory address is divided into two fields. The Set field is decoded to select the correct group. After that the tags in the selected groups are searched. Two possible places in which a block can reside must be searched associatively. Cache group address is the same as that of the direct-mapped cache.

For details of the Set associative mapping example, refer to the book Ch.7, Section 7.5, Figure 7.35 (Page 354-355).

2-Way Set-Associative Cache fig. 7.35(Jordan)



Replacement Strategy

For a cache miss, we have to replace a cache block with the data coming from main memory. Different methods can be used to select a cache block for replacement. **Always Replacement:** For Direct Mapping on a miss, there is only one block which needs replacement called always replacement.

For associative mapping, there are no unique blocks which need replacement. In this case there are two options to decide which block is to be replaced.

- **Random Replacement:** To randomly select the block to be replaced
- **LFU:** Based on the statistical results, the block which has been least used in the recent past, is replaced with a new block.

Write Strategy

When a CPU command to write to a memory data will come into cache, the writing into the cache requires writing into the main memory also.

Write Through: As the data is written into the cache, it is also written into the main memory called Write Through. The advantages are:

- Read misses never result in writes to the lower level.
- Easy to implement than write back

Write Back: Data resides in the cache, till we need to replace a particular block then the data of that particular block will be written into the memory if that needs a write, called write back. The advantages are:

- Write occurs at the speed of the cache
- Multiple writes within the same block requires only one write to the lower memory.
- This strategy uses less memory bandwidth, since some writes do not go to the lower level; useful when using multi processors.

Cache Coherence

Multiple copies of the same data can exist in memory hierarchy simultaneously. The Cache needs updating mechanism to prevent old data values from being used. This is the problem of cache coherence. Write policy is the method used by the cache to deal with and keep the main memory updated.

Dirty bit is a status bit which indicates whether the block in cache is dirty (it has been modified) or clean (not modified). If a block is clean, it is not written on a miss, since lower level contains the same information as the cache. This reduces the frequency of writing back the blocks on replacement.

Writing the cache is not as easy as reading from it e.g., modifying a block can not begin until the tag has been checked, to see if the address is a hit. Since tag checking can not occur in parallel with the write as is the case in read, therefore write takes longer time.

Write Stalls: For write to complete in Write through, the CPU has to wait. This wait state is called write stall.

Write Buffer: reduces the write stall by permitting the processor to continue as soon as the data has been written into the buffer, thus allowing overlapping of the instruction execution with the memory update.

Write Strategy on a Cache Miss

On a cache miss, there are two options for writing.

Write Allocate: The block is loaded followed by the write. This action is similar to the read miss. It is used in write back caches, since subsequent writes to that particular block will be captured by the cache.

No Write Allocate: The block is modified in the lower level and not loaded into the cache. This method is generally used in write through caches, because subsequent writes to that block still have to go to the lower level.

Lecture No. 40

Virtual Memory

Reading Material

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

Chapter 7
7.6

Summary

- Virtual Memory Introduction
- Virtual Memory Organization

Virtual Memory

Introduction

Virtual memory acts as a cache between main memory and secondary memory. Data is fetched in advance from the secondary memory (hard disk) into the main memory so that data is already available in the main memory when needed. The benefit is that the large access delays in reading data from hard disk are avoided.

Pages are formulated in the secondary memory and brought into the main memory. This process is managed both in hardware (Memory Management Unit) and the software (The operating systems is responsible for managing the memory resources).

The block diagram shown (Book Ch.7, Section 7.6, and figure 7.37) specifies how the data interchange takes place between cache, main memory and the disk. The Memory Management unit (MMU) is located between the CPU and the physical memory. Each memory reference issued by the CPU is translated from the logical address space to the physical address space, guided by operating system controlled mapping tables. As address translation is done for each memory reference, it must be performed by the hardware to speed up the process. The operating system is invoked to update the associated mapping tables.

Memory Management and Address Translation

The CPU generates the logical address. During program execution, effective address is generated which is an input to the MMU, which generates the virtual address. The virtual address is divided into two fields. First field represents the page number and the second field is the word field. In the next step, the MMU translates the virtual address into the physical address which indicates the location in the physical memory.

Advantages of Virtual Memory

- Simplified addressing scheme: the programmer does not need to bother about the exact locations of variables/instructions in the physical memory. It is taken care of by the operating system.
- For a programmer, a large virtual memory will be available, even for a limited physical memory.
- Simplified access control.

Virtual Memory Organization

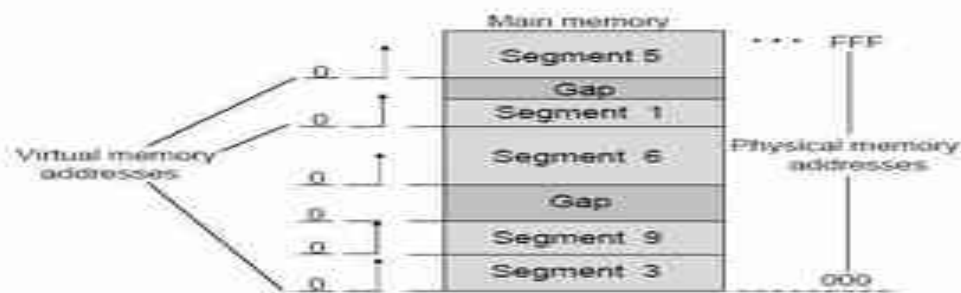
Virtual memory can be organized in different ways. This first scheme is segmentation.

Segmentation:

In segmentation, memory is divided into segments of variable sizes depending upon the requirements. Main memory segments identified by segments numbers, start at virtual address 0, regardless of where they are located in physical memory.

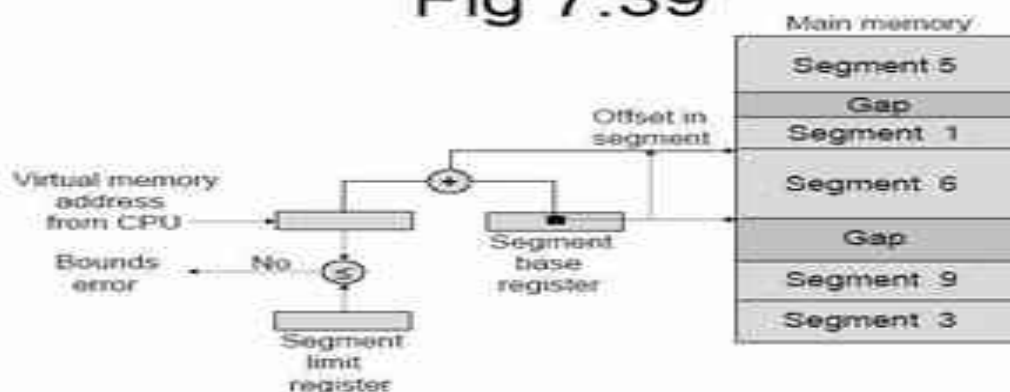
In pure segmented systems, segments are brought into the main memory from the secondary memory when needed. If segments are modified and not required any more, they are sent back to secondary memory. This invariably results in gap between segments, called external fragmentation i.e. less efficient use of memory. Also refer to Book Ch.7 , Section 7.6, Figure 7.38.

Memory Management by Segmentation
Fig 7.38

**Addressing of Segmented Memory**

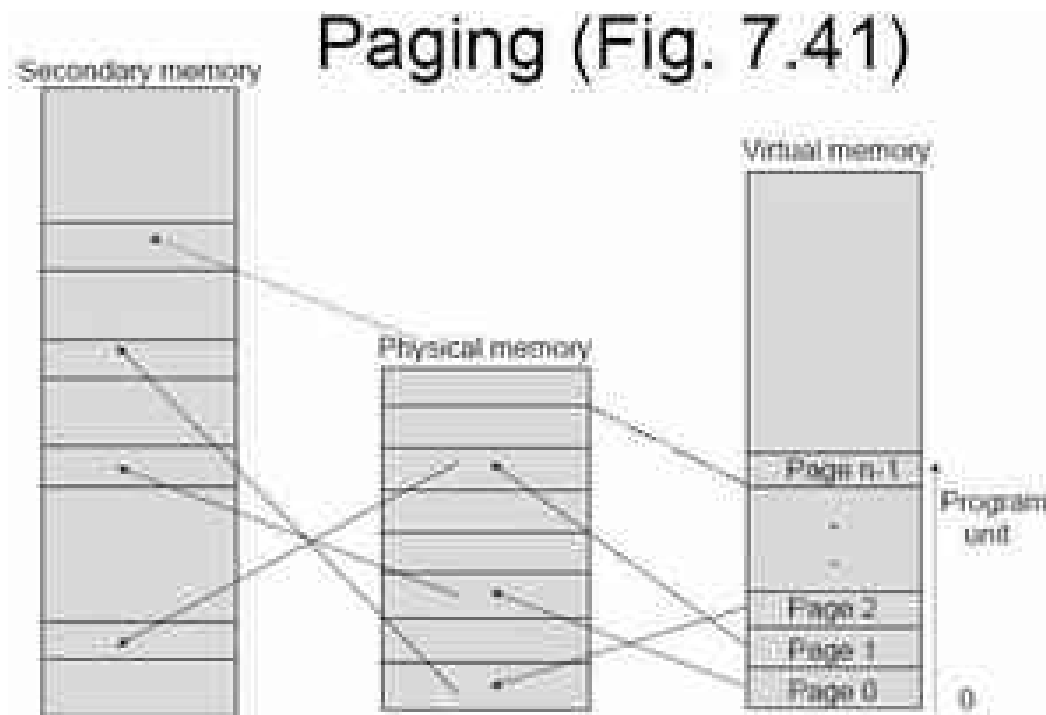
The physical address is formed by adding each virtual address issued by the CPU to the contents of the segment base register in the MMU. Virtual address may also be compared with the segment limit register to keep track and avoiding the references beyond the specified limit. By maintaining table of segment base and limit registers, operating system can switch processes by switching the contents of the segment base and limit register. This concept is used in multiprogramming. Refer to book Ch.7, Section 7.6, and Figure 7.39

Segmentation Mechanism
Fig 7.39

**Paging:**

In this scheme, we have pages of fixed size. In demand paging, pages are available in secondary memory and are brought into the main memory when needed.

Virtual addresses are formed by concatenating the page number with the word number. The MMU maps these pages to the pages in the physical memory and if not present in the physical memory, to the secondary memory. (Refer to Book Ch.7, Section 7.6, and Figure 7.41)



Page Size: A very large page size results in increased access time. If page size is small, it may result in a large number of accesses.

The main memory address is divided into 2 parts.

- Page number: For virtual address, it is called virtual page number.
- Word Field

Virtual Address Translation in a Paged MMU:

Virtual address composed of a page number and a word number, is applied to the MMU. The virtual page number is limit checked to verify its availability within the limits given in the table. If it is available, it is added to the page table base address which results in a page table entry. If there is a limit check fault, a bound exception is raised as an interrupt to the processor.

Page Table

The page table entry for each page has two fields.

Page field

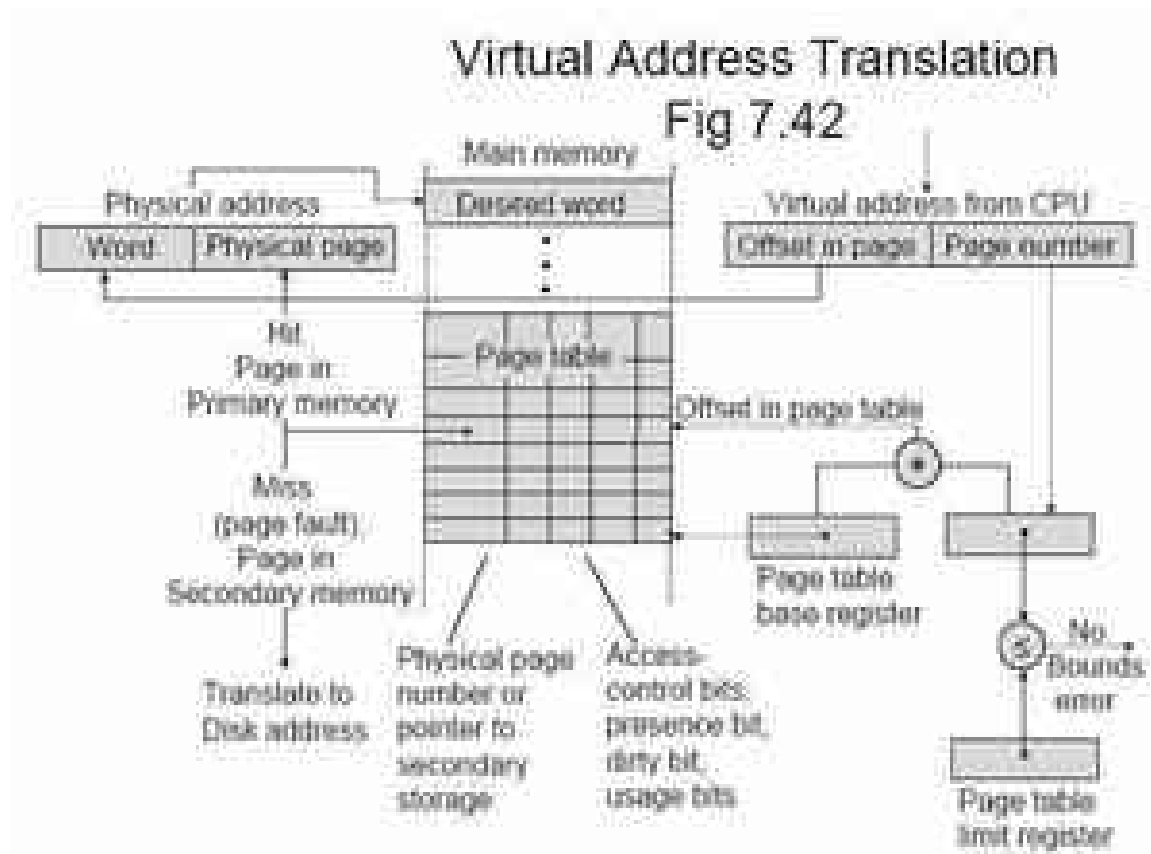
Control Field: This includes the following bits.

- Access control bits: These bits are used to specify read/write, and execute permissions.
- Presence bits: Indicates the availability of page in the main memory.
- Used bits: These bits are set upon a read/ write.

If the presence bit indicates a hit, then the page field of the page table entry contains the physical page number. It is concatenated with the word field of the virtual address to form a physical address.

Page fault occurs when a miss is indicated by the presence bit. In this case, the page field of the page table entry would contain the address of the page in the secondary memory. Page miss results in an interrupt to the processor. The requesting process is suspended until the page is brought in the main memory by the interrupt service routine.

Dirty bit is set on a write hit CPU operation. And a write miss CPU operation causes the MMU to begin a write allocate (previously discussed) process. (Refer to book Ch.7, Section 7.6, and Figure 7.42)



Fragmentation:

Paging scheme results in unavoidable internal fragmentations i.e. some pages (mostly last pages of each process) may not be fully used. This results in wastage of memory.

Processor Dispatch -Multiprogramming

Consider the case, when a number of tasks are waiting for the CPU attention in a multiprogramming, shared memory environment. And a page fault occurs. Servicing the page fault involves these steps.

1. Save the state of suspended process
2. Handle page fault
3. Resume normal execution

Scheduling: If there are a number of memory interactions between main memory and secondary memory, a lot of CPU time is wasted in controlling these transfers and number of interrupts may occur.

To avoid this situation, Direct Memory Access (DMA) is a frequently used technique. The Direct memory access scheme results in direct link between main memory and secondary memory, and direct data transfer without attention of the CPU. But use of DMA in virtual memory may cause coherence problem. Multiple copies of the same page may reside in main memory and secondary memory. The operating system has to ensure that multiple copies are consistent.

Page Replacement

On a page miss (page fault), the needed page must be brought in the main memory from the secondary memory. If all the pages in the main memory are being used, we need to replace one of them to bring in the needed page. Two methods can be used for page replacement.

Random Replacement: Randomly replacing any older page to bring in the desired page. **Least Frequently Used:** Maintain a log to see which particular page is least frequently used and to replace that page.

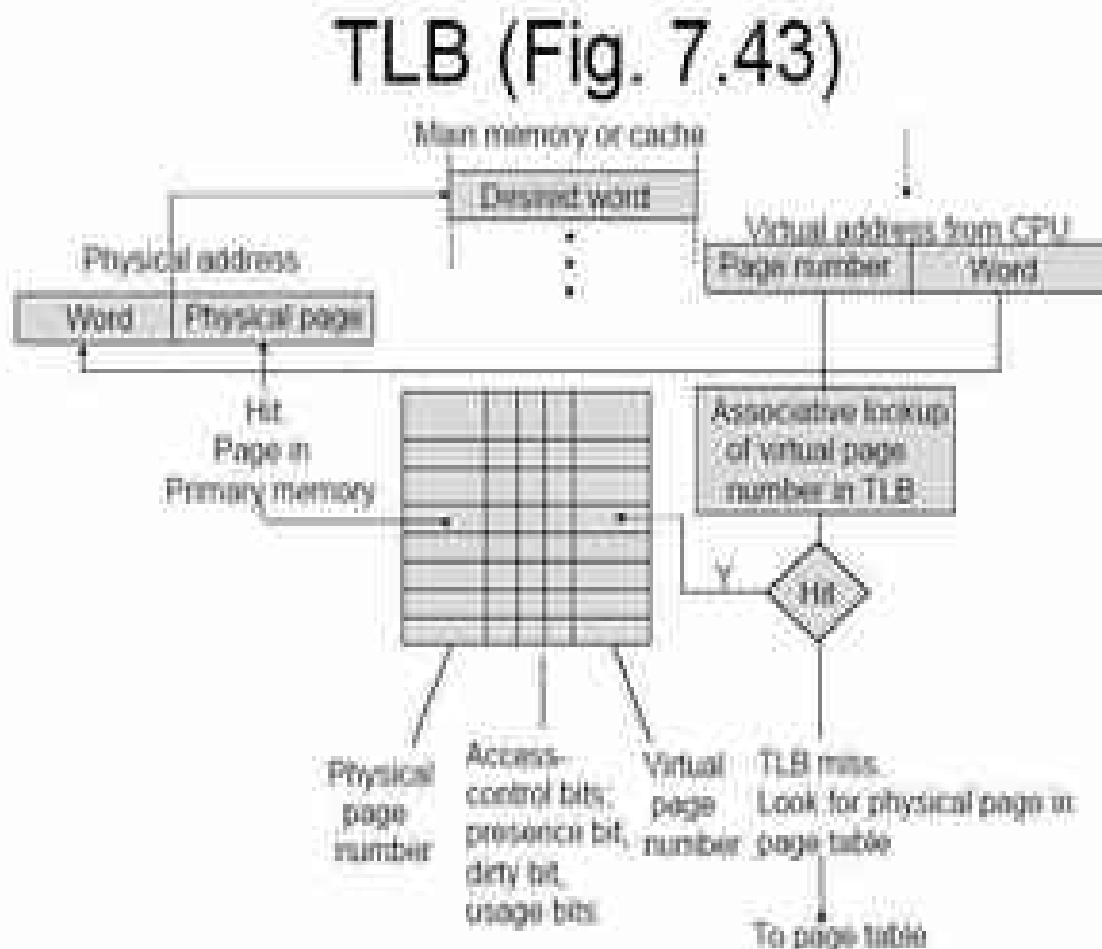
Translation Lookaside buffer

Identifying a particular page in the virtual memory requires page tables (might be very large) resulting in large memory space to implement these page tables. To speed up the process of virtual address translation, translation Lookaside buffer (TLB) is implemented as a small cache inside the CPU, which stores the most recent page table entry reference made in the MMU. It contents include

- A mapping from virtual to physical address
- Status bits i.e. valid bit, dirty bit, protection bit It may be implemented using a fully associative organization

Operation of TLB

For each virtual address reference, the TLB is searched associatively to find a match between the virtual page number of the memory reference and the virtual page number in the TLB. If a match is found (TLB hit) and if the corresponding valid bit and access control bits are set, then the physical page mapped to the virtual page is concatenated. (Refer to Book Ch.7, Section 7.6, and Figure 7.43)

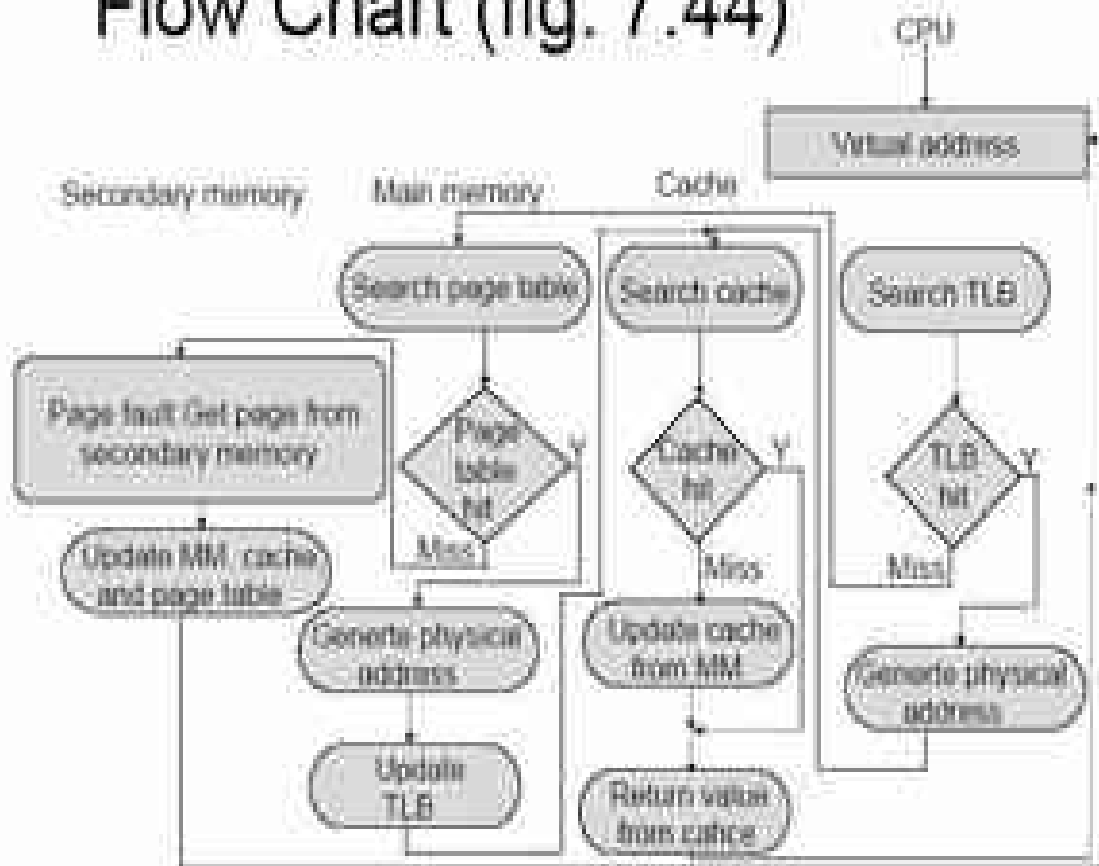


Working of Memory Sub System

When a virtual address is issued by the CPU, all components of the memory subsystem interact with each other. If the memory reference is a TLB hit, then the physical address is applied to the cache. On a cache hit, the data is accessed from the cache. Cache miss is processed as described previously. On a TLB miss (no match found) the page table is searched. On a page table hit, the physical address is generated, and TLB is updated and cache is searched. On a page table miss, desired page is accessed in the secondary memory, and main memory, cache and page table are updated. TLB is updated on the next access (cache access) to this virtual address. (Refer to Book Ch.7, Section 7.6, and Figure 7.44).

To reduce the work load on the CPU and to efficiently use the memory sub system, different methods can be used. One method is separate cache for data and instructions.

Flow Chart (fig. 7.44)



Instruction Cache: It can be implemented as a Translation Lookaside buffer.

Data Cache: In data cache, to access a particular table entry, it can be implemented as a TLB either in the main memory, cache or the CPU.

Lecture No. 41

Numerical Examples of DRAM and Cache

Reading Material

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

Summary

Numerical Examples related to

- DRAM
- Pipelining, Pre-charging and Parallelism
- Cache
- Hit Rate and Miss Rate
- Access Time

Example 1

If a DRAM has 512 rows and its refresh time is 9ms, what should be the frequency of row refresh operation on the average?

Solution

Refresh time= 9ms

Number of rows=512

Therefore we have to do 512 row refresh operations in a 9 ms interval, in other words one row refresh operation every $(9 \times 10^{-3})/512 = 1.76 \times 10^{-5}$ seconds.

Example 2

Consider a DRAM with 1024 rows and a refresh time of 10ms.

- Find the frequency of row refresh operations.
- What fraction of the DRAM's time is spent on refreshing if each refresh takes 100ns.

Solution

Total number of rows = 1024

Refresh period = 10ms

One row refresh takes place after every

$10\text{ms}/1024 = 9.7\text{micro seconds}$

Each row refresh takes 100ns, so fraction of the DRAM's time taken by row refreshes is,

$100\text{ns}/9.7\text{ micro sec} = 1.03\%$

Example 3

Consider a memory system having the following specifications. Find its total cost and cost per byte of memory.

Memory type	Total bytes	Cost per byte
SRAM	256 KB	30\$ per MB
DRAM	128 MB	1\$ per MB
Disk	1 GB	10\$ per GB

Solution

Total cost of system

256 KB($\frac{1}{4}$ MB) of SRAM costs = $30 \times \frac{1}{4} = \$7.5$

128 MB of DRAM costs= $1 \times 128 = \$128$

1 GB of disk space costs= $10 \times 1 = \$10$

Total cost of the memory system

= $7.5 + 128 + 10 = \$145.5$ Cost per byte

Total storage= 256 KB + 128 MB + 1 GB

= $256 \text{ KB} + 128 \times 1024 \text{ KB} + 1 \times 1024 \times 1024 \text{ KB} = 1,179,904 \text{ KB}$

Total cost = \$145.5

Cost per byte= $145.5 / (1,179,904 \times 1024)$

= $\$1.2 \times 10^{-7} \$/\text{B}$

Example 4

Find the average access time of a level of memory hierarchy if the hit rate is 80%. The memory access takes 12ns on a hit and 100ns on a miss.

Solution

Hit rate = 80%

Miss rate = 20%

$T_{\text{hit}} = 12 \text{ ns}$

$T_{\text{miss}} = 100 \text{ ns}$

Average $T_{\text{access}} = (\text{hit rate} \times T_{\text{hit}}) + (\text{miss rate} \times T_{\text{miss}})$

= $(0.8 \times 12 \text{ ns}) + (0.2 \times 100 \text{ ns})$

= 29.6ns

Example 5

Consider a memory system with a cache, a main memory and a virtual memory. The access times and hit rates are as shown in table. Find the average access time for the hierarchy.

	Main memory	cache	virtual memory
Hit rate	99%	80%	100%
Access time	100ns	5ns	8ms

Solution

Average access time for requests that reach the main memory

$$= (100\text{ns} \times 0.99) + (8\text{ms} \times 0.01)$$

$$= 80,099 \text{ ns}$$

$$\text{Average access time for requests that reach the cache} = (5\text{ns} \times 0.8) + (80,099\text{ns} \times 0.2) = 16,023.8\text{ns}$$

Example 6

Given the following memory hierarchy, find the average memory access time of the complete system

Memory type	Average access time	Hit rate
SRAM	5ns	80 %
DRAM	60ns	80%
Disk	10ms	100%

Solution

For each level, average access time = (hit rate x access time for that level) + ((1-hit rate) x average access time for next level)

Average access time for the complete system

$$\begin{aligned}
 &= (0.8 \times 5\text{ns}) + 0.2 \times ((0.8 \times 60\text{ns}) + (0.2)(1 \times 10\text{ms})) \\
 &= 4 + 0.2(48 + 2000000) \\
 &= 4 + 400009.6 \\
 &= 400013.6 \text{ ns}
 \end{aligned}$$

Example 7

Find the bandwidth of a memory system that has a latency of 25ns, a pre charge time of 5ns and transfers 2 bytes of data per access.

Solution

Time between two memory references

= latency + pre charge time

$$= 25 \text{ ns} + 5\text{ns}$$

$$= 30\text{ns}$$

Throughput = $1/30\text{ns}$

$$= 3.33 \times 10^7 \text{ operations/second}$$

$$\text{Bandwidth} = 2 \times 3.33 \times 10^7$$

$$= 6.66 \times 10^7 \text{ bytes/s}$$

Example 8

Consider a cache with 128 byte cache line or cache block size. How many cycles does it take to fetch a block from main memory if it takes 20 cycles to transfer two bytes of data?

Solution

The number of cycles required for the complete transfer of the block

$$= 20 \times 128/2$$

$$= 1280 \text{ cycles}$$

Using large cache lines decreases the miss rate but it increases the amount of time a program takes to execute as obvious from the number of clock cycles required to transfer a block of data into the cache.

Example 9

Find the number of cycles required to transfer the same 128 byte cache line if page-mode DRAM with a CAS-data delay of 8 cycles is used for main memory. Assume that the cache lines always lie within a single row of the DRAM, and each line lies in a different row than the last line fetched.

Solution

Memory requests to fetch each cache line = $128/2 = 64$

Only the first fetch require the complete 20 cycles, and the other 63 will take only 8 clock cycles.
Hence the no. of cycles required to fetch a cache line = $20 + 8 \times 63$

= 524

Example 10

Consider a 64KB direct-mapped cache with a line length of 32 bytes.

- Determine the number of bits in the address that refer to the byte within a cache line.
- Determine the number of bits in the address required to select the cache line.

Solution

Address breakdown

$n = \log_2$ of number of bytes in line

$m = \log_2$ of number of lines in cache

- For the given cache, the number of bits in the address to determine the byte within the line = $n = \log_2 32 = 5$
- There are $64K/32 = 2048$ lines in the given cache. The number of bits required to select the required line = $m = \log_2 2048 = 11$

Hence $n=5$ and $m=11$ for this example.

Example 11

Consider a 2-way set-associative cache with 64KB capacity and 16 byte lines.

- How many sets are there in the cache?
- How many bits of address are required to select a set in the cache?
- Repeat the above two calculations for a 4-way set-associative cache with same size.

Solution

- A 64KB cache with 16 byte lines contains 4096 lines of data. In a 2-way set associative cache, each set contains 2 lines, so there are 2048 sets in the cache.
- $\log_2 (2048) = 11$. Hence 11 bits of the address are required to select the set.
- The cache with 64KB capacity and 16 byte line has 4096 lines of data. For a 4-way set associative cache, each set contains 4 lines, so the number of sets in the cache would be 1024 and $\log_2 (1024) = 10$. Therefore 10 bits of the address are required to select a set in the cache.

Example 12

Consider a processor with clock cycle per instruction (CPI) = 1.0 when all memory accesses hit in the cache. The only data accesses are loads and stores, and these constitute 60% of all the

instructions. If the miss penalty is 30 clock cycles and the miss rate is 1.5%, how much faster would the processor be if all instructions were cache hits?

Solution

Without any misses, the computer performance is

CPU execution time = (CPU clock cycles + Memory stall cycles) x Clock cycle = (IC x CPI + 0) x

Clock cycle = IC x 1.0 x Clock cycle

Now for the computer with the real cache, first we compute the number of memory stall cycles:

$$\frac{\text{Memory accesses}}{\text{Memory stall cycles}} = \text{IC} \times \text{Instruction} \times \text{Miss Rate} \times \text{Miss Penalty}$$

= IC x (1 + 0.6) x 0.015 x 30

= IC x 0.72

Where the middle term (1 + 0.6) represents one instruction access and 0.6 data accesses per instruction. The total performance is thus

CPU execution time cache = (IC x 1.0 + IC x 0.72) x Clock cycle = 1.72 x IC x Clock cycles

The performance ratio is the inverse of the execution times

$$\frac{\text{CPU execution time cache}}{\text{CPU execution time}} = \frac{1.72 \times \text{IC} \times \text{clock cycle}}{1.0 \times \text{IC} \times \text{clock cycle}}$$

The computer with no cache misses is 1.72 times faster

Example 13

Consider the above example but this time assume a miss rate of 20 per 1000 instructions. What is memory stall time in terms of instruction count?

Solution

Re computing the memory stall cycles:

Memory stall cycles = Number of misses x Miss penalty

= IC * $\frac{\text{Misses}}{\text{Instruction}}$ * Miss penalty

= IC / 1000 * $\frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$

= IC / 1000 * 20 * 30

= IC / 1000 * 600 = IC * 0.6

Example 14

What happens on a write miss?

Solution

The two options to handle a write miss are as follows:

Write Allocate

The block is allocated on a write miss, followed by the write hit actions. This is just like read miss.

No-Write Allocate

Here write misses do not affect the cache. The block is modified only in the lower level memory.

Example 15

Assume a fully associative write-back cache with many cache entries that starts empty. Below is a sequence of five memory operations (the address is in square brackets):

Write Mem [300];
Write Mem [300];
Read Mem [400];
Write Mem [400];
Write Mem [300];

What is the number of hits and misses when using no-write allocate versus write allocate?

Solution

For no-write allocate, the address 300 is not in the cache, and there is *no* allocation on write, so the first two writes will result in misses. Address 400 is also not in the cache, so the read is also a miss. The subsequent write to address 400 is a hit. The last write to 300 is still a miss. The result for no-write allocate is four misses and one hit.

For write allocate, the first accesses to 300 and 400 are misses, and the rest are hits since 300 and 400 are both found in the cache. Thus, the result for write allocate is two misses and three hits.

Example 16

Which has the lower miss rate?

a 32 KB instruction cache with a 32 KB data cache or a 64 KB unified cache?

Use the following Miss per 1000 instructions.

Size	Instruction cache	Data cache	Unified cache
32 KB	1.5	40	42.2
64 KB	0.7	38.5	41.2

Assumptions

- The percentage of instruction references is about 75%.
- Assume 40% of the instructions are data transfer instructions.
- Assume a hit takes 1 clock cycle.
- The miss penalty is 100 clock cycles.
- A load or store hit takes 1 extra clock cycle on a unified cache if there is only one cache port to satisfy two simultaneous requests.
- Also the unified cache might lead to a structural hazard.
- Assume write-through caches with a write buffer and ignore stalls due to the write buffer.

What is the average memory access time in each case?

Solution

First let's convert misses per 1000 instructions into miss rates.

$$\text{Miss rate} = \frac{\frac{\text{Misses}}{1000 \text{ Instructions}}}{\frac{\text{Memory accesses}}{\text{Instruction}}}$$

Since every instruction access has exactly one memory access to fetch the instruction, the instruction miss rate is

$$\text{Miss rate}_{32 \text{ KB instruction}} = \frac{1.5/1000}{1.00} = 0.0015$$

Since 40% of the instructions are data transfers, the data miss rate is

$$\text{Miss Rate}_{32 \text{ kb data}} = \frac{40/1000}{0.4} = 0.1$$

The unified miss rate needs to account for instruction and data accesses:

$$\text{Miss Rate}_{64 \text{ kb unified}} = \frac{42.2/1000}{1.00 + 0.4} = 0.031$$

As stated above, about 75% of the memory accesses are instruction references. Thus, the overall miss rate for the split caches is

$$(75\% \times 0.0015) + (25\% \times 0.1) = 0.026125$$

Thus, a 64 KB unified cache has a slightly lower effective miss rate than two 16 KB caches. The average memory access time formula can be divided into instruction and data accesses:

Average memory access time

$$= \% \text{ instructions} \times (\text{Hit time} + \text{Instruction miss rate} \times \text{Miss Penalty}) + \% \text{ data} \times (\text{Hit time} + \text{Data miss rate} \times \text{Miss Penalty})$$

Therefore, the time for each organization is:

Average memory access time split

$$\begin{aligned} &= 75\% \times (1 + 0.0015 \times 100) + 25\% \times (1 + 0.1 \times 100) \\ &= (75\% \times 1.15) + (25\% \times 11) \\ &= 0.8625 + 2.75 = 3.61 \end{aligned}$$

$$\begin{aligned} &= 75\% \times (1 + 0.031 \times 100) + 25\% \times (1 + 1 + 0.031 \times 100) \\ &= (75\% \times 4.1) + (25\% \times 5.1) = 3.075 + 1.275 \\ &= 4.35 \end{aligned}$$

Hence split caches have a better average memory access time despite having a worse effective miss rate. Split cache also avoids the problem of structural hazard present in a unified cache.

Lecture No. 42

Performance of I/O Subsystems

Reading Material

Patterson, D.A. and Hennessy, J.L.
Computer Architecture -A Quantitative Approach

Chapter 8

Summary

- Introduction
- Performance of I/O Subsystems
- Loss System
- Single Server Model
- Little's Law
- Server Utilization
- Poisson distribution
- Benchmarks programs
- Asynchronous I/O and operating system

Introduction

Consider a producer-server model. A buffer (or queue) is present between them. Tasks are being received and when one task is finished (i.e. served) then the second task is taken up by the server. Now latency and the response time depend upon how many tasks are present in the queue and how quickly they are served. If there is no task, ahead in the queue the latency would be low and response time would be shorter.

Through put depends upon the average number of calls and the service time taken by a particular server.

Performance of I/O Subsystems

There are three methods to measure I/O subsystem performance:

- **Straight away calculations using execution time**
- **Simulation**
- **Queuing Theory**

Loss System

Loss system is a simple system having no buffer so it does not have any provision for the queuing. In a loss system, provision is time in term of how many switches we do need, then provide some redundancy how many individuals I/O controllers we do need, then how many CPUs are there. It is also called dimension of a loss system.

Delay System

This system provides additional facilities. If we find some call party busy, we can have provision of call waiting. If we have more than one call waiting, then once we finish the first call, we may receive the second call.

Single Server Model

Consider a black box. Suppose it represents an I/O controller. At the input, we have arrival of different tasks. As one task is done, we have a departure at the output. So in the black box, we have a server. Now if we expand and open-up the black box, we could see that incoming calls are coming into the buffer and the output of the buffer is connected to the server. This is an example of “single server model”.

Little’s Law

For a system with multiple independent requests for I/O service and input rate equal to output rate, we use Little’s law to find the mean number of tasks in the system and Time sys such that

Mean number of tasks = Arrival Rate x Mean Response time
 and
 $Time_{sys} = Time_q + Time_s$
 where
 $Time_s$ = Average time to serve task
 $Time_q$ = Average time per task in the queue
 $Time_{sys}$ = Aver time /task
 Arrival Rate = λ = Average number of arriving tasks
 $Length_s$ = Average number of task in service
 $Length_q$ = Average length of queue
 and
 $Length_{sys} = Length_q + Length_s$

Server Utilization

Server Utilization = Arrival Rate x $Time_q$

Server utilization is also called traffic intensity and its value must be between 0 and 1.

Server utilization depends upon two parameters:

1. Arrival Rate
2. Average time required to serve each task

So, we can say that it depends on the I/O bandwidth and arrival rate of calls into the system.

Example 1

Suppose an I/O system with a single disk gets (on average) 100 I/O requests/second. Assume that average time for a disk to service an I/O request is 5ms. What is the utilization of the I/O system?

Solution

Time for an I/O request = 5ms

=0.005sec

Server utilization = 100 x 0.005

= 0.5

Poisson distribution

In order to calculate the response time of an I/O system, we make the following assumptions:

1. Arrival is random

2. System is memory less. It means that incoming calls are not correlated.

For characterize random events, according to above two assumptions, we use Poisson distribution:

$$\text{Probability (k)} = (e^{-k} \times a^k) / k!$$

a = Rate of events $\times$ Elapsed time

= Arrival rate $\times t$ also

$$C^2 = \frac{\text{Variance}}{(\text{Arithmetic mean time})^2}$$

and

$$\text{Average Residual Service Time} = \frac{1}{2} \times \text{weighted mean time} \times (1 + C^2)$$

Example 2

For the system of previous example having server utilization of 0.5, what is the mean number of I/O requests in the queue?

Solution

$$\text{Length}_q = \frac{(\text{Server utilization})^2}{(1 - \text{Server utilization})}$$

$$\text{Length}_q = (0.5)^2 / (1 - 0.5) = 0.5$$

Assumptions about Queuing Model

1. Poisson distribution is assumed
2. The system is in equilibrium
3. The length of the queue is infinity
4. The system has only one server
5. The server will start the next task after finishing the previous one.

Example 3

Suppose a processor sends 10 disks I/O per second, these requests are exponentially distributed, and the average service time of an older disk is 10ms. Answer the following questions:

- What is the number of requests in the queue?
- What is the average time a spent in the queue?
- What is the average response time for a disk request?

Solution

Average number of arriving tasks/second = 20

Average disk time = 10ms = 0.01sec

Sever utilization = $20 \times 0.01 = 0.2$

$\text{Time}_q = 10\text{ms} \times 0.2 / (1 - 0.2) = 2.5\text{ms}$

Average response time = $2.5 + 10 = 22.5\text{ms}$

M/M/m model of queuing theory

A system which has multiple servers is called M/M/m model.

The following formulas are used for M/M/m model:

$$\text{Utilization} = \frac{\text{Arrival Rate} \times \text{Time}_s}{N_s}$$

$$\text{Length}_s = \text{Arrival Rate} \times \text{Time}_q$$

$$\text{Time}_q = \frac{(\text{Time}_s \times (P_{\text{tasks}} \geq N_s))}{N_s \times (1 - \text{utilization})}$$

$$P_{\text{tasks}} \geq N_s = \frac{N_s \times \text{utilization}}{N_s! \times (1 - \text{utilization})} \times P_{0\text{tasks}}$$

Example#4

Suppose instead of a new, faster disk, we add a second slow disk, and duplicate the data so that read can be serviced by either disk. Let's assume that the requests are all reads. Recalculate the answers to the earlier questions, this time using an M/M/m queue.

Solution

The average utilization of the two disks is given as;

$$\begin{aligned} \text{Server utilization} &= \frac{\text{Arrival rate} \times \text{Time}_s}{N_s} \\ &= \frac{(20 \times 0.01)}{2} \\ &= 0.1 \end{aligned}$$

$$P_{0\text{tasks}} = \left[1 + \frac{(2 \times \text{utilization})^2}{2! \times (1 - \text{utilization})} + \frac{(2 \times \text{utilization})^2}{2!} \right]^{-1}$$

$$\begin{aligned} P_{0\text{tasks}} &= \left[1 + \frac{(2 \times 0.1)^2}{2! \times (1 - 0.1)} + (2 \times 0.1)^2 \right]^{-1} \\ &= (1 + 0.22 + 0.2)^{-1} \\ &= 1.222^{-1} \end{aligned}$$

$$\begin{aligned}
 \text{Probability} = N_i &= \frac{(2 \times \text{utilization})^i}{2^i \times (1 - \text{utilization})} \times \text{Probability} \\
 &= \frac{(2 \times 0.1)^2}{2^2 \times (1 - 0.1)} \times 1.232^i \\
 &= 0.018 \\
 \text{Time}_s &= \text{Time}_e \times \frac{\text{Probability} = N_i}{N_i \times (1 - \text{utilization})} \\
 &= 0.01 \times 0.018 / (2 \times 0.9) \\
 &= 0.1 \text{msec} \\
 \text{Average response time} &= 10 \text{msec} + 0.1 \text{msec} \\
 &= 10.1 \text{msec}
 \end{aligned}$$

Benchmarks programs

In order to measure the performance of real systems and to collect the values of parameters needed for prediction, Benchmark programs are used.

Types of Benchmark programs

Two types of benchmark programs are used:

TPC-C
SPEC

Asynchronous I/O and operating system

In order to improve the I/O performance, parallelism is used.

For this, two approaches are available:

- Synchronous I/O
- Asynchronous I/O

Synchronous I/O

In this approach, operating system requests data and switches to another process. Until the desired data arrived. Then the operating system switches back to the requesting process.

Asynchronous I/O

This model is of the process to continue after making a request and it is not blocked until it tries to read requested data.

Bus versus switches

Consider a LAN, using bus topology. If we replace the bus with a switch, the speed of the data transfer will be improved to a great extent.

Lecture No. 43

Networks

Reading Material

Vincent P. Heuring & Harry F. Jordan
Computer Systems Design and Architecture

Patterson, D.A. and Hennessy, J.L.
Computer Architecture - A Quantitative Approach

Chapter 8

Summary

- Introduction to computer network
- Difference between distributed computing and computer networks
- Classification of networks
- Interconnectivity in WAN
- Performance Issues
- Effective bandwidth versus Message size
- Physical Media

Introduction to Computer Networks

A computer architect should know about computer networks because of the two main reasons:

Connectivity

Connection of components within a single computer follows the same principles used for the connection of different computers. It is important for the computer architect to know about connectivity for better sharing of bandwidth

Sharing of resources

Consider a lab with 50 computers and 2 printers using a network, all these 50 computers can share these 2 printers.

Protocol

A set of rules followed by different components in a network. These rules may be defined for hardware and software.

Host

It is a computer with a modem, LAN card and other network interfaces. Hosts are also called nodes or end points. Each node is a combination of hardware and software and all nodes are interconnected by means of some physical media.

Difference between Distributed Computing and Computer Networks

In distributed computing, all elements which are interconnected operate under one operating system. To a user, it appears as a virtual uni-processor system.

In a computer network, the user has to specify and log in on a specific machine. Each machine on the network has a specific address. Different machines communicate by using the network which exists among them.

Classification of Networks

We can classify a network based on the following two parameters:

- The number and type of machines to be interconnected
- The distance between these machines

Based on these two parameters, we have the following type of networks:

SAN (System/Storage Area Network)

It refers to a cluster of machines where large disk arrays are present. Typical distances could be tens of meters.

LAN (Local Area Network)

It refers to the interconnection of machines in a building or a campus. Distances could be in Kilometers.

WAN (Wide Area Network)

It refers to the interconnection between LANs.

Interconnectivity in WAN

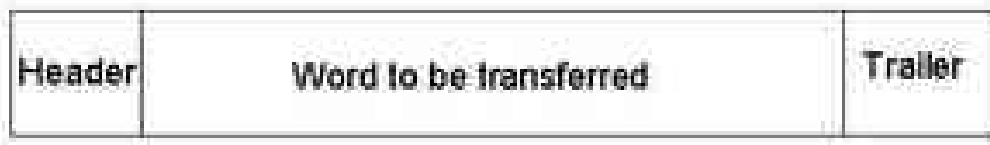
Two methods are used to interconnect WANs:

1. Circuit switching

It is normally used in a telephone exchange. It is not an efficient way.

2. Packet switching

A block (an appropriate number of bits) of data is called a packet. Transfer of data in the form packets through different paths in a network is called packet switching. Additional bits are usually associated with each packet. These bits contain information about the packet. These additional bits are of two types: header and trailer. As an example, a packet may have the form shown below:



If we use a 1-bit header, we may have the following protocol:

Header = 0, it means it is a request

Header = 0, Reply

By reading these header bits, a machine becomes able to receive data or supply data.

To transfer data by using packets through hardware is very difficult. So all the transfer is done by using software. By using more number of bits, in a header, we can send more messages. For example if n bits are used as header then 2^n is the number of messages that can be transmitted over a network by using a single header. For a 2 bit header: we may have 4 types of messages:

00= Request

01= Reply

10= Acknowledge request

11= Acknowledge reply

Error detection

The trailer can be used for error detection. In the above example, a 4 bit checksum can be used to detect any error in the packet. The errors in the message could be due to the long distance

transmission. If the error is found in some message, then this message will be repeated. For a reliable data transmission, bit error rate should be minimum.

Software steps for sending a message:

- Copy data to the operating system buffer.
- Calculate the checksum, include in trailer and start timer.
- Send data to the hardware for transmission.

Software steps for message reception:

- Copy data to the operating system buffer.
- Calculate the checksum; if same, send acknowledge and copy data to the user area otherwise discard the message.

Response of the sender to acknowledgment:

- If acknowledgment arrives, release copy of message from the system buffer.
- When timer expires, resend data and restart the time.

Performance Issues

1. Bandwidth

It is the maximum rate at which data could be transmitted through networks. It is measured in bits/sec.

2. Latency

In a LAN, latency (or delay) is very low, but in a WAN, it is significant and this is due to the switches, routers and other components in the network

3. Time of flight

It is the time for first bit of the message to arrive at the receiver including delays. Time of the flight increases as the distance between the two machines increases.

4. Transmission time

The time for the message to pass through the network, not including the time of flight.

5. Transport latency

Transport latency = time of flight + transmission time

6. Sender overhead

It is the time for the processor to inject message in to the network.

7. Receiver overhead

It is the time for the processor to pull the message from the network.

8. Total latency

Total latency = Sender overhead + Time of flight + Message size/Bandwidth + Receiver overhead

9. Effective bandwidth

Effective bandwidth = Message size/Actual Bandwidth Actual bandwidth may be larger than the effective bandwidth.

Example#1

Assume a network with a bandwidth of 1500Mbps/sec. It has a sending overhead of 100μsec and a receiving overhead of 120μsec. Assume two machines connected together. It is required to

send a 15,000 byte message from one machine to the other (including header), and the message format allows 15,00 bytes in a single message. Calculate the total latency to send the message from one machine to another assuming they are 20m apart (as in a SAN). Next, perform the same calculation but assume the machines are 700m apart (as in a LAN). Finally, assume they are 1000Km apart (as in a WAN). Assume that signals propagate at 66% of the speed of light in a conductor, and that the speed of light is 300,000Km/sec.

Solution

By using the assumption, we get:

$$\text{Time of flight} = \frac{\text{Distance between two machines in Km}}{2/3 \times 300,000 \text{ Km/sec}}$$

$$\text{Total Latency} = \text{Sender overhead} + \text{Time of flight} + \text{Message size/bandwidth} + \text{Receiver overhead}$$

For SAN:

$$\begin{aligned} \text{Total latency} &= 100 \mu\text{sec} \\ &+ (0.020 \text{ Km} / (2/3 \times 300,000 \text{ Km/sec})) \\ &+ 15,000 \text{ bytes} / 1500 \text{ Mbits/sec} \\ &+ 120 \mu\text{sec} \\ &= 100 \mu\text{sec} + 0.1 \mu\text{sec} + 80 \mu\text{sec} + 120 \mu\text{sec} \\ &= 300.1 \mu\text{sec} \end{aligned}$$

For WAN

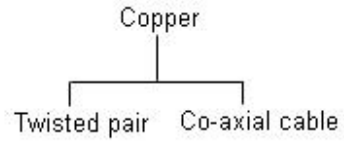
$$\begin{aligned} \text{Total latency} &= 100 \mu\text{sec} \\ &+ (1000 \text{ Km} / (2/3 \times 300,000 \text{ Km/sec})) \\ &+ 15,000 \text{ bytes} / 1500 \text{ Mbits/sec} \\ &+ 120 \mu\text{sec} \\ &= 100 \mu\text{sec} + 5000 \mu\text{sec} + 80 \mu\text{sec} + 120 \mu\text{sec} \\ &= 5300 \mu\text{sec} \end{aligned}$$

Effective bandwidth versus Message size

Effective bandwidth is always less than the raw bandwidth. If the effective bandwidth is closer to the raw bandwidth, the size of the message will be larger. If the message size is larger then network will be more effective.

If large number of the messages are present then a queue will be formed, and the user has to face delay. To minimize the delay, it is better to use packets of small size.

Physical Media



Twisted pair does not provide good quality of transmission and has less bandwidth. To get high performance and larger bandwidth, we use co-axial cable. For increased performance, better performance, we use fiber optic cables, which are usually made of glass. Data transmits through the fiber in the form of light pulses. Photo diodes and sensors are used to produce and receive electronic pulses.

Lecture No. 44

Communication Medium and Network Topologies

Reading Material

Patterson, D.A. and Hennessy, J.L.
Computer Architecture - A Quantitative Approach

Chapter 8

Summary

- Physical Media (Continued)
- Shared Medium
- Switched Medium
- Connection Oriented vs. Connectionless Communication
- Network Topologies
- Seven-layer OSI Model
- Internet and Packet Switching
- Fragmentation
- Routing

Modem

To interconnect different computers by using twisted pair copper wire, an interface is used which is called modem. Modem stands for modulation/demodulation. Modems are very useful to utilize the telephone network (i.e. 4 KHz bandwidth) for data and voice transmission.

Quality of Telephone Line

Data transfer rate depends upon the quality of telephone line. If telephone line is of fine quality, then data transfer rate will be sufficiently high. If the phone line is noisy then data transfer rate will be decreased.

Classification of Fiber Optic Cables

Fiber optic cables can be classified into the following types.

Multimode fiber

This fiber has large diameter. When light is injected, it disperses, so the effective data rate decreases.

Mono mode Fiber

Its diameter is very small. So dispersion is small and data rate is very high.

Wavelength –Division Multiplexing (WDM)

Waves of different wavelengths are simultaneously sent through fiber. So as a result, throughput increases.

Wireless Transmission

This is another effective medium for data transfer. Data is transferred in the form of electromagnetic waves. It has the following features:

- Data rate is in Mbits/Sec.
- Very effective because of flexibility.
- Band width is much less than fiber.

Example 1

Suppose we have 20 magnetic tapes, each containing 40GB. Assume that there are enough tape readers to keep any network busy. How long will it take to transmit the data over a distance of 5Km? The choices are category 5 twisted-pair wires at 100Mbits/sec, multimode fiber at 1500Mbits/sec and single-mode fiber at 3000Mbits/sec. (Adapted from CA3: H&P)

Solution

The total amount of data
 = total no. of mag. tapes x capacity of each tape
 = 20 x 40GB
 = 800GB

The time for each medium:

Twisted pair = $800\text{GB}/100\text{Mbits/sec}$
 = 65536 sec = 18.2 hr

Multimode Fiber = $800\text{GB}/1500\text{Mbits/sec}$
 = 4369.06sec = 1.213 hr

Single mode Fiber = $800\text{GB}/3000\text{Mbits/sec}$
 = 2184.55sec
 = 0.66hr

Car = time to load car + transport time + time to unload car
 = 250sec + 5Km/30Kph + 250sec
 = 500.16 sec = 0.13hr

Shared/Switched Medium

Shared Medium

If a number of computers are connected with a single physical medium (i.e. coaxial or fiber), this situation is called shared medium. Because of many computers, collision takes place and affects the data transfer rate. As the number of machines on a physical medium increases, the data transfer rate decreases.

Switched Medium

To increase the throughput, a switched medium is used.

Example 2

Compare 20 nodes connected in three different ways: a single 100Mbits/sec shared medium; a switch connected via cat5, each segment running at 100Mbits/sec; and a switch connected via optical fiber, each segment running at 1500Mbits/sec. The shared medium is 700m long, and the

average length of each segment to a switch is 55m. Both switches can support full bandwidth. Assume each switch adds 6μsec to the latency, and the average message size is 200bytes. Ignore the overhead of sending or receiving a message and contention for the network.

Solution

First we will calculate the aggregate bandwidth:

For shared medium

Aggregate bandwidth = 100Mbits/sec

For switched twisted pair

Aggregate bandwidth = 20 x 100Mbits/sec
= 2000Mbits/sec

For switched optical fiber

Aggregate bandwidth = 20 x 1500Mbit/sec
= 30,000Mbits/sec

Transport time = Time of flight + (message size/BW)

$$\begin{aligned} \text{Transport time shared} &= \frac{(700/1000)\text{Km}}{(2/3 \times 300,000)\text{Km}} \times 10^6 \mu\text{sec} + (200 \times 8\text{bits} / 100\text{Mbits/sec}) \\ &= 3.5\mu\text{sec} + 16\mu\text{sec} = 19.5\mu\text{sec} \end{aligned}$$

For the switches, the distance is twice the average segment. We must also add latency for the switch.

$$\begin{aligned} \text{Transport time switch} &= 2 \times \frac{(55/1000)\text{Km}}{(2/3 \times 300,000)\text{Km}} \times 10^6 \mu\text{s} \\ &\quad + 6\mu\text{sec} + (200 \times 8\text{bits} / 100\text{Mbits/sec}) \\ &= 0.55\mu\text{sec} + 6\mu\text{sec} + 16\mu\text{sec} \\ &= 22.55\mu\text{sec} \end{aligned}$$

$$\begin{aligned} \text{Transport time fiber} &= 2 \times \frac{(55/1000)\text{Km}}{(2/3 \times 300,000)\text{Km}} \times 10^6 \mu\text{s} \\ &\quad + 6\mu\text{sec} + (200 \times 8\text{bits} / 1500\text{Mbits/sec}) \\ &= 0.55\mu\text{sec} + 6\mu\text{sec} + 1.06\mu\text{sec} \\ &= 7.61\mu\text{sec} \end{aligned}$$

Although the bandwidth of the switch is many times that of the shared medium, the latency for unloaded networks is comparable.

Connection Oriented vs. Connection less Communication

Connection Oriented Communication

- In this method, same path is always taken for the transfer of messages.
- It reserves the bandwidth until the transfer is complete. So no other server could use that path until it becomes free.
- Telephone exchange and circuit switching is the example of connection oriented communication.

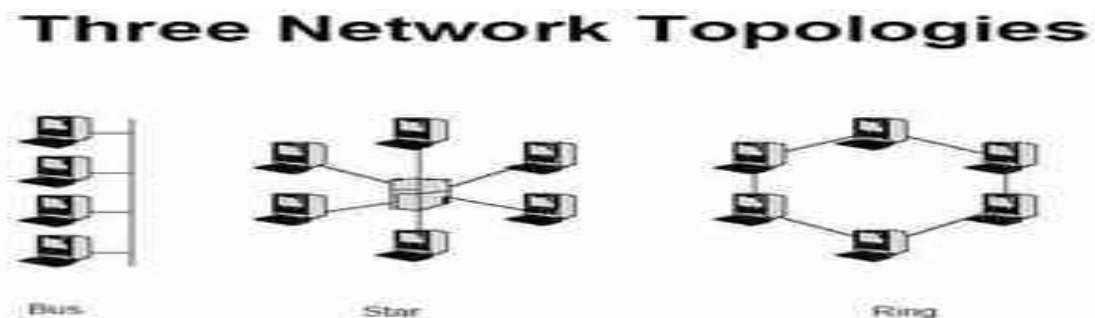
Connection less Communication

- Here message is divided into packets with each packet having destination address.
- Each packet can take different path and reach the destination from any route by looking at its address.
- Postal system and packet switching are examples of connection less communication.

Network Topologies

Computers in a network can be connected together in different ways. The following three topologies are commonly used:

- Bus topology
- Star topology
- Ring topology



Bus Topology

In this arrangement, computers are connected via a single shared physical medium.

Star topology

Computers are connected through a hub. All messages are broad cast because the hub is not an intelligent device.

Ring Topology

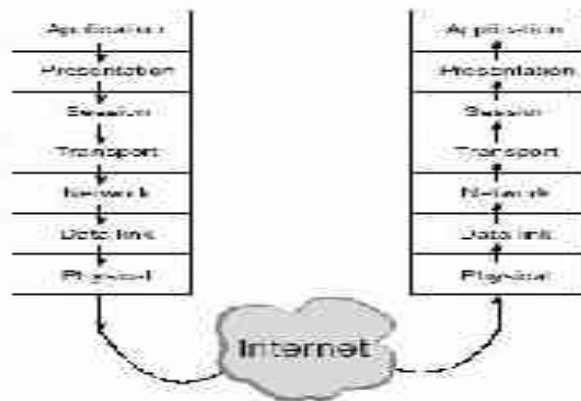
All computers are connected through a ring. Only one computer can transmit data at one time, having a pass called “Token”.

Seven Layer OSI Model

There are seven layers in this model.

1. Physical Layer
2. Data Layer
3. Network Layer
4. Transport Layer
5. Session Layer
6. Presentation Layer
7. Application Layer

7-Layer OSI Model



OSI Model Characteristics

- An interface is present between any two layers.
- A layer may use the data present in another layer.
- Each layer is abstracted from other layers.
- The service provided by one layer can be used by the other layer.
- Two layers can provide same service e.g. Check Sum calculated at different layers.
- On two machines, six layers are logically connected except the physical layer. The physical layers of two machines are physically connected.

Internet and Packet Switching

Internet works on the concept of packet switching. Application layer passes data to the lower layer and that lower layer passes data to the next lower layer and on so on. In this data passing process through different layers, different headers are attached with the data which shows the source and destination addresses, number of data bytes in packet, type of message etc. At physical layer, this packet is transmitted into the network. At reception, reverse procedure is adopted.

Fragmentation

When a packet is lost in the network, it is re-transmitted. If the size of the packet is large then retransmission of packet is wastage of resources and it also increases the delay in the network. To minimize this delay, a large packet is divided into small fragments. Each fragment contains a separate header having destination address and fragment number. This fragmentation effectively reduces the queuing delay. At destination, these fragments are re-assembled and data is sent to the application layer.

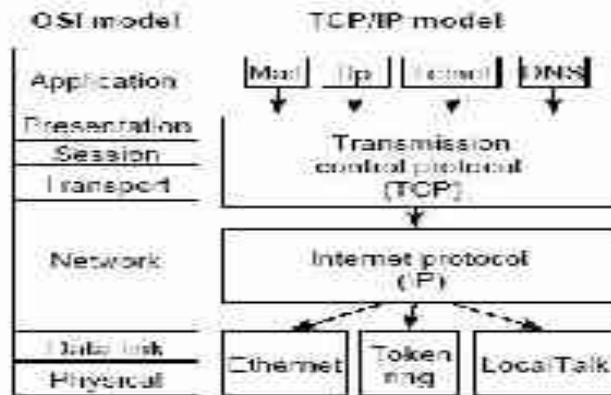
Routing

Routing works on store-and-forward policy. There are three methods used for routing:

- Source-based routing
- Virtual Circuit
- Destination-based routing

TCP/IP

Internet uses TCP/IP protocol. In the TCP/IP model, session and presentation layers are not present, so Store-Forward routing is used.

TCP/IP vs. OSI Model

Lecture No. 45

Review

Reading Material

Handouts

Slides

Review of all lectures (i.e. Lecture # 1 to 44)