

Advance Computer Architecture II

Course - Code: CS704

Reference by

Computer Architecture, A Quantitative Approach by Jhon L. Hennessy and David A. Patterson

Chapter	Subject	Lectures	Page
1	Fundamentals of Computer Design	01 - 03	02
(Appendix B)	Instruction Set Principles	04-06	27
(Appendix A)	Basic and Intermediate Concepts	07-11	55
2 - 3	Instruction-Level Parallelism and Its Exploitation	12-24	110
5	Memory Hierarchy Design	25-33	260
4	Multiprocessors and Thread-Level Parallelism	34-37	344
6	Storage Systems	38-45	388

Presented by:

Dr. M. Ashraf Chughtai

Ph.D, University of Manchester, UK

Reference Book 1:

Computer Architecture, A Quantitative Approach by Jhon L. Hennessy and David A. Patterson

Download Link:

<http://www.vumultan.com/Books/CS704-Computer-Architecture-A-Quantitative-Approach.pdf>

Reference Book 2:

Computer Organization and Architecture: Designing for Performance by William Stalling

Download Link:

<http://www.vumultan.com/Books/CS704-Computer-Organization-and-Architecture.pdf>

A World Class Education at Your Door Step

Lecture	Subject	Page
01	Introduction	03
Quantitative Principles		
02	Detailed discussion on the computer Performance	13
03	Design for Performance	24
Instruction Set Principles		
04	Instruction Set Principles	27
05	Encoding instructions and MIPS Instruction format	37
06	ISA Performance Analysis, Fallacies and Pitfalls	48
Computer Hardware Design		
07	Single Cycle Data path and Control Design	55
08	Multi Cycle Datapath and Control Design	67
09	Multi Cycle and Pipeline - Datapath and Control Design	81
10	Pipeline Datapath and Control Design	88
11	Pipeline and Instruction Level Parallelism	95
Instruction Level Parallelism		
12	Introduction to multi cycle pipelined Datapath	110
13	Dynamic Scheduling - Scoreboard Approach	115
14	Dynamic Scheduling - Vsq ewyjs w/Ct t vsegl	132
15	Dynamic Branch Prediction	160
16	F}req r g"Dver gl "Rvi h rpxsr " ØEsr x h	170
17	High-performance Instructions delivery - Multiple Issue	180
18	Hardware-based speculations and exceptions	186
19	Limitations of ILP and Conclusion	198
20	Static Scheduling	206
21	Static Scheduling Multiple Issue Processor	216
22	Software pipelining and Trace Scheduling	228
23	Hardware Support at Compile Time	242
24	Concluding Instruction Level Parallelism	250
Memory Hierarchy Design		
25	Storage Technologies Trends and Caching	260
26	Concept of Caching and Principle of Locality	271
27	Cache Design Techniques	281
28	Cache Design and policies	293
29	Cache Performance Enhancement by: Reducing Cache Miss Penalty	301
30	Cache Performance Enhancement: Reducing Miss Rate	310
31	Cache Performance Enhancement by: Miss Penalty/Rate Parallelism and Hit time	317
32	Main and Virtual Memories	324
33	Main and Virtual Memories ØEsr x h	335
Multiprocessors		
34	Shared Memory Architectures	344
35	Cache Coherence Problem	356
36	Eegl i "Esl i vi r gi "Rvsf p q " "Esr x h	365
37	Performance and Synchronization	378
Input Output Systems		
38	Storage and I/O Systems	388
39	Bus Structures Connecting I/O Devices	401
40	RAID and I/O System Design	410
Networks and Clusters		
41	Networks: Interconnection and Topology	422
42	Pi x{ svow"Vst spsk}"er h"K xi vi x{ svomk""ØEsr x h	433
43	Internetworks and Clusters	445
44	Putting It All Together: Case Studies	454
45	Putting It All Together: Review: Lecture 1 - 43	469

Lecture 1

Introduction

Welcome to the course of Advanced Computer Architecture.

This course of Advanced Computer Architecture has been developed with the assumption that the students have basic knowledge of: digital logic design, computer organization and design, programming model of microprocessor; memory and input/output interfacing with the microprocessor and fundamentals of Computer Architecture

O} "r eq i "m"Oyl eq q eh"Cw vej"El ykl xentCjx v'gsq t p xmk"q } "Degl i psv w'er h"Oewi w" degrees from the University of Engineering and Technology, Lahore in 1974 and 1978, respectively I completed my Ph.D. from the University of Manchester (UMIST) UK in 1986. After spending 32 years at UET Lahore, as lecturer through professor, presently I am associated with COMSATS.

Computer System

Computer system is a collection of Central Processing Unit, memory system and peripheral devices, all interconnected by groups of conductors called buses

Computer Architecture Verses Organization

Architecture refers to those attributes of a computer visible to a programmer or compiler writer such as: instruction set, addressing techniques, I/O mechanisms etc. The members of each family (say Intel x86 family and IBM family) share the same basic architecture and have code compatibility,

Organization refers to how the features of a computer are implemented; i.e., control signal generation as FSM or microprogramming, memory technology-SRAM, DRAM etc, hardware or software based realization of operation-multiplication by hardware or algorithmically. Moreover, the organization of same architecture may differ between different versions; i.e., different versions of Intel x86 family may have different organizations

Academic History:

In 1944, John von Neumann introduced the concept of stored-program computer, referred to as Electronic Discrete Variable Automatic Computer EDVAC.

1946. The first general-purpose computer, Electronic Numerical Integrator And Calculator- ENIAC, built at the Moore School of University of Pennsylvania and became operational in the World War-II. ENIAC used 18,000 vacuum tubes and could perform 1900 additions per second.

In 1946, Institute for Advanced Study-ISA at Princeton, introduced IAS machine which was roughly 10 times faster than ENIAC.

In 1948, a small prototype stored-program machine, Mark-I (the name adopted from first electromechanical computer built at Harvard), was built at the University of Manchester. At Harvard, the Mark-I was followed by the relay machine Mark-II; and a pair of vacuum tube computers - Mark-III and Mark-IV, terms as Harvard Architecture. These computers had separate memories for instructions and data.

1949. The second general-purpose computer, Electronic Delay Storage Automatic Calculator EDSAC.

In 1949, Eckert-Mauchly Computer Corporation built first machine BINAC. The company was acquired by Remington-Rand, where first general-purpose Universal Automatic Computer UNIVAC-I was produced in 1951, which was sold for \$1 million.

In 1952, IBM® produced its first stored-program computer IBM-701.

In 1963, Seymour Cray announced first supercomputer CDC 6600 from Control Data Corporation CDC; in 1964 announced a series of System/360; and introduced models 40, 50, 65 and 75. These models varied in clock rate (1.6-5.1 MHz), memory size (32KB-1 MB), cost \$(0.225M - 1.900M) and performance.

In 1965, Digital Equipment Corporation DEC introduced first low-cost computer, minicomputer PDP-8, for under \$0.02M.

In 1976, Cray Research Inc. (formed by Seymour Cray left in 1970 when he left CDC) introduced first supercomputer Cray-I.

Microprocessors 1971 - 2006:

In 1971, Intel® introduced first cheap microprocessor 4004.

In 1977, personal computer Apple-II was introduced by Steve Jobs and Steve Wozniak.

In 1981, Intel® introduced first microprocessor 80x86 and running Microsoft operating system.

1995 and onwards: With the introduction of personal computers, the era of supercomputer could not last for more than two decades, as:

In 1996, Cray Research was handed over to Silicon Graphics

In 1998, more than 350 million microprocessors with different instruction set architectures were in use.

In 2006, his number has risen to more than a billion

Course Focus: Quantitative principle of computer design

The primary aim of this course is to have an in-depth study of the concepts of computer architecture, which have already been introduced to you; and the concepts which are relevant for professional computer scientist, engineers and architects.

The emphasis will be given to expose the advances in the field through cost-performance-power trade-offs and good engineering design.

The course gives:

- quantitative principles of computer design
- instruction set architecture
- datapath and control: implementation and performance
- advanced pipelining concepts
- memory hierarchy design, Main memory, Cache, Hard drives
- multiprocessor architecture
- storage and input/output systems
- computer clusters

Textbook:

Hennessy J. L and Petterson D. A, Computer Architecture: A quantitative approach, 3rd Edition, Morgan Kaufmann publishers, 2003

Fundamentals of Computer Design (Review)	03 Lectures	Ch. 1
Instruction Set Principles	02 Lectures	Ch. 2
Pipelining Basics	05 Lectures	(App. A)
Instruction Level Parallelism	15 Lectures	Ch. 3 & 4
Memory Hierarchy Design	06 Lectures	Ch. 5
Multiprocessors	04 Lectures	Ch. 6
I/O and Storage System	04 Lectures	Ch. 7
Interconnection Networks	6 Lectures	Ch. 8

References:

Hennessy J. L and Petterson D. A, Computer Organization and Design: The hardware/software interface, 2nd Edition, Morgan Kaufmann publishers, 1998

Stalling W, Computer Organization and Architecture, 6th Edition, Prentice Hall, 2003

Relevant research papers on Computer Architecture from conferences, transactions and journals of IEEE and ACM etc.

Course Style: Research in the small

We want you to succeed, but you need to show initiative

Y i "{ n t szrh " }sy" st t swr n " s" hs" v i w evl " m" x i vq ep " s" l i p " q eoi "
transition from good student to research colleague

Research Project:

pick topic for 3 weeks work full time

meet 3 times with faculty/TA

give online presentation

Submit written report like conference paper

Four Perspective of Computer Architecture

Computer Architecture can be considered from four perspectives

Processor Design

Memory Hierarchy

Input/output and storages

Multiprocessor and Network interconnection

Processor Design:

Instruction set architecture: Addressing modes and protections

Datapath designs: Multiple cycles and pipelines, ILP, Hazards etc.

L1 Caches: design, blocks and associativity

Memory Hierarchy:

L2 Caches: Coherence, bandwidth and latency

Main memory: DRAM, interfacing and protocols

Virtual memory: Interfacing and protocols

I/O and Storage:

Disks and taps

Redundant Array of Inexpensive Disks-RAID

Multiprocessor and Networks:

Shared memory
Interconnecting networks
Network switches
Routing, bandwidth and latency

Computer Design cycle

The computer design and developments have been under the influence of:

Technology
Performance
Cost

The new trends in technology have been used to reduce the cost and enhance the performance

The computer design evaluated for bottlenecks using certain benchmarks to achieve the optimum performance.

Time is the key measurement of performance: a desktop user may define the performance of his/her machine in terms of time taken by the machine to execute a program; whereas a computer center manager running a large server system may define the performance in terms of the number of jobs completed in a specified time.

The desktop user is interested in reducing the response time or execution time time between the start and completion of an event; while a data processing center in increasing the throughput.

Benchmarks are the programs which try to average frequency of operation and operands of a large set of programs.

Standardized benchmarks tools are available from Standard Performance Evaluation Corporation SPEC at www.spec.org

Hardware: Cost, delay, area, power estimation

Simulation at different levels of design abstraction, such as: ISA, RT, Gate, Circuit

Queuing Theory: to calculate the response time and throughput of entire I/O system

Rules of Thumb

Higher performance with lower cost

The new designs are simulated to evaluate the performance for different levels of workloads.

Simulation helps in keeping the result verification cost minimum.

The cost-performance is optimized for workloads

1: Technology Trends: Computer Generations

Vacuum tube	1946-1957 1 st Gen.
Transistor -	1958-1964 2 nd Gen.
Small scale integration	1965-1968
✓ Up to 100 devices/chip	
Medium scale integration	1969-1971 3 rd Gen.
✓ 100-3,000 devices/chip	
Large scale integration	1972-1977
✓ 3,000 - 100,000 devices/chip	
Very large scale integration	1978 on.. 4 th Gen.
✓ 100,000 - 100,000,000 devices/chip	
Ultra large scale integration	
✓ Over 100,000,000 devices/chip	

The technology trends at different times in the history of the computer development have played major role in the journey of computers from uniprocessor to multiprocessor mainframe systems; and from super structure to desktop computing.

Usually identified by dominant implementation technologies, the length of first three generations spreads over a period of 8 to 10 years.

The first generation, 1950-59, is the era of commercial electronic computers employing vacuum tube technology.

The second generation, 1960-68, is the period of cheaper computers made using transistor or discrete component technology.

The third generation, 1969-77, is the age of minicomputers developed employing integrated circuit technology.

The fourth generation, 1978 to date, the era of personal computers and workstations, is the result of VLSI and ULSI technology.

The gateway to the fifth and higher generations is not clear as no revolutionary technology has been proclaimed.

The present transistor technology has developed to an extent where up to 15 millions transistors are integrated on one VLSI chip to make PowerPC and Pentium Processors

In 1980, the performance of supercomputers was 50 times that of microprocessor, 20 times of minicomputers and 10 time that of mainframe computers.

The performance of microprocessors was equivalent to that of supercomputer in 1992 and it was much higher in 1995 and onwards

3: Cost

The systems are implemented using the latest technology to obtain cost effective high performance solution

Implementation complexities are given due consideration

Price Verses Cost

The relationship between cost and price is complex one
 The cost is the total amount spends to produce a product
 The price is the amount for which a finished good is sold.
 The cost passes through different stages before it becomes price.
 A small change in cost may have a big impact on price

Component Costs

Direct Costs (add 25% to 40%) recurring costs: labor, purchasing, scrap, warranty
 Gross Margin (add 82% to 186%) nonrecurring costs: R&D, marketing, sales, equipment maintenance, rental, financing cost, pretax profits, taxes
 Average Discount to get List Price (add 33% to 66%): volume discounts and/or retailer markup

Summary:

1: A Computer systems development is viewed with reference to computer architecture and computer system organization.

Architecture refers to those attributes of a computer visible to a programmer or compiler writer; e.g. instruction set, addressing techniques, I/O mechanisms etc. Organization refers to how the features of a computer are implemented? i.e., control signals are generated using the principles of finite state machine (FSM) or microprogramming

The architecture of the members of a processor family are same whereas organization of same architecture may differ between different members of the family

2: Computer development has been discussed with academic and commercial perspectives.

Academically, modern computer developments have their infancy in 1944-49, when John von Neumann introduced the concept of stored-program computer, referred to as Electronic Discrete Variable Automatic Computer EDVAC and Maurice Wilkes of Cambridge University, built the first small scale, operational stored program computer, Electronic Delay Storage Automatic Calculator EDSAC.

Commercially, the first machine BINAC was built by Eckert-Mauchly Computer Corporation in 1949; and Universal Automatic Computer UNIVAC-I by Remington-Rand, in 1951, which was sold for \$1 million. In 1976, Cray Research Inc. (formed by Seymour Cray announced the first high speed, operational stored program computer, Electronic Delay Storage Automatic Calculator EDSAC.

Microprocessor and PCs: In 1971, Intel introduced first cheap microprocessor 4004 and then 80 x 86 series, which was used by IBM in personal computers.

In 1998, more than 350 million microprocessors with different instruction set architectures were in use; this number has risen to more than a billion in 2006

3: Decisive Factors for rapid changes in the computer development are performance enhancements price reduction and functional improvements.

4: Technological developments, from vacuum tubes to VLSI circuits, dynamic memory and network technology gave birth to four different generations of computers. However, at present no revolutionary technology has been proclaimed, hence the gateway to the fifth and higher generations is not clear.

5: Computer Architecture have been viewed from four perspectives

Processor Design

Memory Hierarchy

Input/output and storages

Multiprocessor and Network interconnection

6: Course Focus and Topics Coverage: The course is focused to have in-depth study of the concepts of computer architecture. The topics covered give emphasis to expose the advances in the field through cost-performance-power trade-offs and good engineering design.

7: Computer Design Cycle: Computer design and development has been under the influence of technology, performance and cost; and the new trends in technology are used to enhance the performance and reduce the cost

Lecture 2

Quantitative Principles

Detailed discussion on the computer Performance

The key to quantitative design and analysis

]

After a quick review of the previous lecture we will continue with the discussion on the growth in processor performance.

An introduction to the computer hardware performance will be given because it is the key to the quantitative analysis in determining the effectiveness of an entire computing system the hardware and software. Today we will talk about:

- Price-performance design
- CPU performance metrics
- CPU benchmark suites

1: Architecture refers to those attributes of a computer visible to a programmer or compiler writer; e.g. instruction set, addressing techniques, I/O mechanisms etc.

Organization refers to how the features of a computer are implemented? i.e., control signals are generated using the principles of finite state machine (FSM) or microprogramming

2: Computer development was discussed with academic and commercial perspectives.

Academically, modern computer developments have their infancy in 1944-49, when John von Neumann introduced the concept of stored-program computer, referred to as Electronic Discrete Variable Automatic Computer EDVAC and Maurice Wilkes of Cambridge University, built the first small scale, operational stored program computer, Electronic Delay Storage Automatic Calculator EDSAC.

Commercially, the first machine BINAC was built by Eckert-Mauchly Computer Corporation in 1949; and Universal Automatic Computer UNIVAC-I by Remington-Rand, in 1951, which was sold for \$1 million. In 1976, Cray Research Inc. (formed by Seymour Cray announced the first super computer, Cray-1. Technological developments, from vacuum tubes to VLSI circuits, dynamic memory and network technology gave birth to four different generations of computers. However, at present no revolutionary technology has been proclaimed, hence the gateway to the fifth and higher generations is not clear.

Microprocessor and PCs: In 1971, Intel introduced first cheap microprocessor 4004 and then 80 x 86 series, which was used by IBM in personal computers.

In 1998, more than 350 million microprocessors with different instruction set architectures were in use; this number has risen to more than a billion in 2006

3: Computer Architecture was viewed from four perspectives

- Processor Design
- Memory Hierarchy
- Input/output and storages
- Multiprocessor and Network interconnection

4: Computer Design Cycle: We concluded our discussion last time with the computer design cycle. We found that the computer design and development has been under the influence of technology, performance and cost; the decisive factors for rapid changes in the computer development have been the performance enhancements, price reduction and functional improvements. The new trends in technology are used to enhance the performance and reduce the cost.

Growth in Processor Performance:

The supercomputers and mainframes, costing millions of dollars and occupying excessively large space, prevailing form of computing in 1960s were replaced with relatively low-cost and smaller-sized minicomputers in 1970s

In 1980s, very low-cost microprocessor-based desktop computing machines in the form of personal computer (PC) and workstation were introduced.

The growth in processor performance since mid-1980s has been substantially high than in earlier years

Prior to the mid-1980s microprocessor performance growth was averaged about 35% per year

By 2001 the growth raised to about 1.58 per year

Price-Performance Design:

Technology improvements are used to lower the cost and increase performance.

The relationship between cost and price is complex one

The cost is the total amount spends to produce a product

The price is the amount for which a finished good is sold.

The cost passes through different stages before it becomes price.

A small change in cost may have a big impact on price

Price vs. Cost:

Manufacturing Costs: Total amount spent to produce a component

- ✓ Component Cost: Cost at which the components are available to the designer. It ranges from 40% to 50% of the list price of the product.
- ✓ Recurring costs: Labor, purchasing scrap, warranty 4% - 16 % of list price
- ✓ Gross margin Non-recurring cost: R&D, marketing, sales, equipment, rental, maintenance, financing cost, pre-tax profits, taxes

List Price:

- ✓ Amount for which the finished good is sold;
- ✓ it includes Average Discount of 15% to 35% of the as volume discounts and/or retailer markup

Cost-effective IC Design: Price-Performance Design

Yield: Percentage of manufactured components surviving testing

Volume: increases manufacturing hence decreases the list price and improves the purchasing efficiency

Feature Size: the minimum size of a transistor or wire in either x or y direction

Reduction in feature size from 10 microns in 1971 and 0.18 in 2001 has resulted in:

- Quadratic rise in transistor count
- Linear increase in performance
- 4-bit to 64-bit microprocessor
- Desktops have replaced time-sharing machines

Cost of Integrated Circuits

Manufacturing Stages: The Integrated circuit manufacturing passes through many stage:

- ✓ Wafer growth and testing
- ✓ Wafer chopping it into dies
- ✓ Packaging the dies to chips
- ✓ Testing a chip.

Die: is the square area of the wafer containing the integrated circuit

- ✓ See that while fitting dies on the wafer the small wafer area around the periphery goes waist

Cost of a die: The cost of a die is determined from cost of a wafer; the number of dies fit on a wafer and the percentage of dies that work, i.e., the yield of the die.

The cost of integrated circuit can be determined as ratio of the total cost; i.e., the sum of the costs of die, cost of testing die, cost of packaging and the cost of final testing a chip; to the final test yield.

Cost of IC =

- ✓ die cost + die testing cost + packaging cost + final testing cost / final test yield

The cost of die is the ratio of the cost of the wafer to the product of the dies per wafer and die yield

- ✓ Cost of die = Cost of wafer / dies per wafer x die yield

The number of dies per wafer is determined by the dividing the wafer area (minus the waist wafer area near the round periphery) by the die area

- ✓ Dies per wafer =
$$\left[\frac{\pi (D/2)^2 - \pi (W/2)^2}{\text{die area}} \right] \left[\frac{\pi (D/2)^2 - \pi (W/2)^2}{\pi (d/2)^2} \right]$$

Example Calculating Number of Dies

For die of 0.7 Cm on a side, find the number of dies per wafer of 30 cm diameter

Answer:

$$\begin{aligned} & [\text{Wafer area} / \text{Die Area}] - \text{Wafer Waist area} \\ & = \left[\frac{\pi (30/2)^2 - \pi (5/2)^2}{0.49} \right] - \frac{\pi (5/2)^2}{0.49} \\ & = 1347 \text{ dies} \end{aligned}$$

Calculating Die Yield

Die yield is the fraction or percentage of good dies on a wafer number

Wafer yield accounts for completely bad wafers so need not be tested

Wafer yield corresponds to on defect density D which depends on number of masking levels

Good estimate for CMOS is 4.0 and

Die yield = Wafer yield $\times (1 + \text{defects per unit area} \times \text{die area})^{-1}$

Example: The yield of a die, 0.7cm on a side, with defect density of $0.6/\text{cm}^2$
 $= (1 + [0.6 \times 0.47] / 4.0)^{-4} = 0.75$

Price-Performance Design

Time to run the task:

Execution time, response time, latency

Throughput or bandwidth:

View of the system's performance

Time is the key measurement of performance. However, the throughput - number of tasks completed in specified time cannot be ignored.

For example, where the task in hand is to move 2400 employees of a company from Lahore to Islamabad in minimum overall time at minimum cost, the cost-performance of the train versus plane has to be evaluated.

Train takes 4:00 Hrs per trip and carries 2400 passengers/trip, i.e., 6:00 sec/person, while the airplane with a capacity of 300 passengers, flying 10 times faster than the train, takes 45 min/trip, will require 6:00 Hrs, i.e., 9.00 sec. per person to complete the task. Thus the throughput of the train and hence its performance is 50% more than the airplane. Moreover the traveling cost per person by train is 10 times less than the airplane, thus the cost-performance per person by train versus airplane is 1:15.

Example:

To carry 2400 passengers from Lahore to Islamabad

Train completes the task in 4:00 hrs while airplane completes the same task in 6.00 hrs.; i.e., 66.67% of the task in same time throughput and hence performance of train is 50% more than airplane

Vehicle	Time Lah to Isb	Passengers/ trip	Time to complete job	Execution time /person	Cost person /	Cost-performance
Train	4.0 hours	2400	4.0 hours	6.0 sec	300 Rs	$300 \times 6 = 1,800$ Rs-sec/person
Plane	45 min.	300	$45 \times 8 \text{ min.} = 6.0 \text{ Hr}$	9.0 sec	3000 Rs	$3000 \times 9 = 27,000$ Rs-sec/person

Plane 10 times faster but takes 50% more time to complete the job; i.e., lesser throughput thus performance of train is 50% better than plane

The time per person and cost per person of train is less than that of plane Thus the cost-performance of plane is 1:15

Metrics of Performance

The execution time or **latency** to complete a task includes everything - the processing activities, disk access, memory access, input/output activities, operating system overheads and etc. In other words, it is the total execution time which is also regarded as the wall-clock time, elapsed time or response time.

With the multiprogramming CPU may be working for another program while waiting for I/O activities of one program to complete; thus the CPU time is different from the response time.

The CPU time can further be divided into the user CPU time - the time spent in the program and system CPU time - the time spent in operating system.

UNIX command time reports four measurements of time as: 99.4u 28.1s 2:50 75%; i.e., the user time, system time, elapsed time and percent of the elapsed time that is CPU time

Aspects of CPU Performance

CPU time = Seconds / Program

$$= (\text{Instructions} / \text{Program}) \times (\text{Cycles} / \text{Instruction}) \times (\text{Seconds} / \text{Cycle})$$

	Inst Count	CPI	Clock Rate
Program			
Compiler			
Inst. Set.			
Organization			
Technology			

All computers run at a constant rate which is determined by the discrete time events called ticks, clock ticks, clock periods, clocks, cycles or clock cycles, define as time (sec.) or rate (Hz).

The CPU clocks of a program and the number of instructions executed or instruction count - IC is a useful measure to find the number of clock cycles per instruction

$CPI = \text{CPU clock cycles for program} / \text{Instruction count}$

The CPU time of a program is equal to the product of CPU clock cycles for program and clock cycle-time; where the cycle-time = $1/\text{clock rate}$. Hence the

$\text{CPU Time} = \text{CPU clock cycles for program} / \text{clock rate}$ or

$\text{CPU time} = \text{instruction count} \times \text{clock cycle time} \times \text{cycles per instruction sec.} / \text{program}$
 $= (\text{instructions/program}) \times (\text{clock cycles/instructions}) \times (\text{seconds/clock cycles})$

Cycles Per Instruction

Cycles per Instruction CPI

$= \text{CPU Clock Cycles for program} / \text{Instruction Count}$

$= (\text{CPU Time} \times \text{Clock Rate}) / \text{Instruction Count}$

Instruction Frequency For instruction mix, the relative frequency of occurrence of different types of instructions is given as:

$FIC_i = IC \text{ of } i^{\text{th}} \text{ instruction} / \text{Total Instruction count}$

Average Cycles per Instruction

$CPI = [1 / \sum_{i=1}^n FIC_i] \times \sum_{i=1}^n FIC_i \times CPI_i$

Example: Calculating average CPI

Base Machine (Reg / Reg)

Op	Freq	Cycles	CPI (i)	(% Time)
ALU	50%	1	0.5	(33%)
Load	20%	2	0.4	(27%)
Store	10%	2	0.2	(13%)
Branch	20%	2	0.4	(27%)
			1.5	

Cycles Per Instruction

The total execution time of n programs running on a machine may be computed as:

Arithmetic mean time: Time_i is the execution time of i^{th} program, and it is assumed that each program in the workload runs once (or for equal number of times).

$$\checkmark \quad 1/n \sum_{i=1}^n \text{Time}_i$$

Weighted arithmetic mean time: For unequal mix of programs in the workload, i.e., individual programs running for different number of times, a weighting factor w_i is assigned to each program that indicates the relative frequency of a program in the workload. The weighted arithmetic mean is determined by summing the product of weighting factor and program execution time as:

$$\checkmark \quad \sum_{i=1}^n w_i \times \text{Time}_i$$

Geometric mean time: is the second approach to unequal mix of programs in the workload. Here, the execution time is normalized to a reference machine and then average of the normalized execution time is taken, which is expressed by geometric mean as:

$$\checkmark \quad \sqrt[n]{\sum_{i=1}^n \text{Execution time ratio}_i}$$

Where execution time ratio_i is the execution time, normalized to the reference machine, for the ith program of total of n in the workload.

Summary: Price-Performance Design

Cost of a Computer: The total cost of manufacturing a computer is distributed among different parts of the system such as the cost of cabinet, processor board and I/O devices. The cabinet cost lies close to 6% of the total cost whereas the cost of other two parts is around 37% each. However, the major cost at the component level is of the processor, which is approximately 22% of the total cost.

Performance: Time is the key measurement of performance: a desktop user may define the performance of his/her machine in terms of time taken by the machine to execute a program; whereas a computer center manager running a large server system may define the performance in terms of the number of jobs completed in a specified time. The desktop user is interested in reducing the response time or execution time – time between the start and completion of an event; while a data processing center in increasing the throughput. The performance of two designs, say X and Y; is often compared by the factor n, which determines how much lower execution time machine Y takes as compared to X or how much faster is machine Y than X, i.e.,

$$n = \text{Execution time X} / \text{Execution time Y}$$

as performance is inverse of execution time, therefore

$$n = \text{Performance Y} / \text{Performance X}$$

Cost-Performance: The issue of cost-performance is complex one. At one extreme, high-performance computers designer may not give importance to the cost in achieving the performance goal. At the other end, low-cost designer may sacrifice performance to some extent. The price-performance design lies between these extremes where the designer balances cost and hence price versus performance. The PCs, workstations and servers market operates in this region.

Instruction Execution Rate MIPS

MIPS specify performance inversely to execution time; For a given program:

$$MIPS = (\text{instruction count}) / (\text{execution time} \times 10^6)$$

MIPS could not be calculated from the instruction mix

Time is defined based on some reference machine as:

$$RMIPS = [\text{Performance M} / \text{Performance}_{\text{reference}}] \times MIPS_{\text{reference}}$$

or

$$= [\text{Time}_{\text{reference}} / \text{Time}_M] \times MIPS_{\text{reference}}$$

MFLOPS defined for Floating-point-intensive programs as millions of floating-point operations per second

CPU Benchmark Suites

Performance Comparison: the execution time of the same workload running on two machines without running the actual programs

Benchmarks: the programs specifically chosen to measure the performance.

Five levels of programs: in the decreasing order of accuracy

- Real Applications
- Modified Applications
- Kernels
- Toy benchmarks
- Synthetic benchmarks

In order to compare the performance of two machines, a user can simply compare the execution time of the same workload running on both the machines.

In practice users want to know, without running their own programs, that how well the machine will perform on their workload.

This is accomplished by evaluating the machine using a set of benchmarks the programs specifically chosen to measure the performance.

The following five levels of programs, given in the decreasing order of accuracy of prediction, are used as benchmarks:

1. **Real Applications** scientific programs such as Mat Lab used by engineers, C language compiler used by program developer or Photoshop used by a graphic designer are the most suitable benchmarks to evaluate the performance of a machine as the users can select input, output and options according to their own needs.
2. **Modified Applications** the real applications are used as the building blocks with certain blocks modified to focus desired aspects of application, e.g., I/O may be restructured or removed to minimize its impact on execution time and to simulate real application
3. **Kernels** the small key pieces extracted from the real program, run exclusively to evaluate the performance by isolating the performance of individual features of a machine.
4. **Toy benchmarks** small codes, say 10 to 100 lines, which produce a result already known to the user, normally used as beginning programming assignments.
5. **Synthetic benchmarks** the small section of program created artificially to match the average frequency of operations and operands of a large set of programs. Synthetic benchmarks are rarely used because they are not even the pieces of a real user.

SPEC: System Performance Evaluation Cooperative

First Round 1989: 10 programs yielding a single number SPECmarks

Second Round 1992: SPECint92 (6 integer programs) and SPECfp92 (14 floating point programs)

Third Round 1995

new set of programs: SPECint95 (8 integer programs) and SPECfp95 (10 floating point)

f i r g l q evow'yi jyp'sv'5"}i ew

Single flag setting for all programs: SPECint_base95, SPECfp_base95

In late 1980s, System Performance and Evaluation Cooperation SPEC was founded to improve the state of benchmarking and to provide more valid basis of comparison.

First Round: In 1989, SPEC made its first release, SPEC89 to measure CPU performance. In 1990, concept of throughput, which provided benchmark for timeshared usage of uniprocessor or a multiprocessor, was included to define performance.

Second Round: In 1992, the SPEC92 provided separate means for integer (SPEC Int 92 6 Integer) and floating-point (SPECfp92 14 floating point) programs.

Third Round: The SPEC95 added new integer and floating point benchmarks and also changed the base machine for normalization to a Sun SPARC-station 10/40 as the operating versions of the original base machines were difficult to find. SPEC offers verity of benchmark versions; a few typical benchmarks are as follows:

Desktop benchmarks can be divided into two classes: CPU-intensive and Graphic-intensive benchmarks. The SPEC89, SPEC92, SPEC95 and SPEC2000 are CPU intensive benchmarks. The SPEC2000 has portability with minimum role of I/O in overall benchmark performance. SPECviewperf is used for benchmarking systems supporting OpenGL graphic library and SPECcapc contains application for graphic use.

Server benchmarks are of multiple classes: SPEC CPU2000 is a throughput benchmarks capable of measuring throughput and converting it into processing rate of multiprocessor, considering the I/O activities which is the significant feature of servers. The SPECsfs is a fileserver benchmark capable of measuring network file system (NFS) performance; it test the performance of I/O systems (disk and network). The SPECweb is a server benchmark that simulates multiple clients requesting both the static and dynamic pages from the server, as well as client posting data to the server.

Summary: Designing and performance comparison

Designing to Last through Trends

	Capacity	Speed
Logic	2x in 3 years	2x in 3 years
DRAM	4x in 3 years	2x in 10 years
Disk	4x in 3 years	2x in 10 years

6yrs to graduate => 16X CPU speed, DRAM/Disk size

Time to run the task

Execution time, response time, latency

Vevow't i v'he}. "l syv."{ i i o."w g."r w"

Throughput, bandwidth

$\frac{Performance(X)}{ExTime(X)} = \frac{Performance(Y)}{ExTime(Y)}$

$ExTime(Y) / ExTime(X) = Performance(X) / Performance(Y)$

CPI Law:

- ✓ $CPU_{TIME} = \text{Seconds} / \text{Program}$
 $= (\text{Instructions} / \text{Program}) \times (\text{Cycles} / \text{Instruction}) \times (\text{Seconds} / \text{Cycle})$

Execution time is the REAL measure of computer performance!

Good products created when have:

- ✓ Good benchmarks, good ways to summarize performance

Die Cost goes roughly with die area⁴

Good products created when have:

Good benchmarks

Good ways to summarize performance

Given sales is a function in part of performance relative to competition, investment in improving product as reported by performance summary

If benchmarks/summary inadequate, then choose between improving product for real programs vs. improving product to get more sales; Sales almost always wins!

Execution time is the measure of computer performance!

Lecture 3

Quantitative Principles

Design for Performance

] √pm ""] √..ω"

Recap

I/O performance

Laws and Principles

Performance enhancement

Concluding: quantitative principles

Home work

Summary

Computer I/O System

Producer-Server model

Producer: the device that generates request to be serviced

Queue: the area where the tasks accumulate waiting to be serviced

Server: the device performing the requested service

Response Time: the time a task takes from the moment it is placed in the buffer to the time server finishes the task

I/O Performance Parameters

Diversity: Which I/O device can connect to the CPU

Capacity: How many I/O devices can connect to the CPU

Latency: Overall response time to complete a task

Bandwidth: Number of task completed in specified time - throughput

I/O Transaction Time

The interaction time or transaction time of a computer is sum of three times:

Entry Time: the time for user to enter a command average 0.25 sec; from keyboard 4.0 sec.

System Response Time: time between when user enters the command and system responds

Think Time: the time from reception of the command until the user enters the next command

Throughput verses Response time: $R_i = \frac{1}{N_i} \times \frac{1}{T_i}$

Response time and throughput calculation

If the system is in steady state, then the number of tasks entering the system must be equal to the number of tasks leaving the system

Number of tasks in system $\{ N_i \}$

✓ Mean number of tasks in system = Mean response time x Arrival rate

Summary "A Little queuing theory"

Mean number of tasks in the system

✓ $N_i = \frac{\text{Time}_{\text{accumulated}}}{\text{Time}_{\text{observe}}}$

Mean response time

✓ $T_i = \frac{\text{Time}_{\text{accumulated}}}{\text{Number}_{\text{tasks}}}$

Arrival rate

✓ $R_i = \frac{\text{Number}_{\text{tasks}}}{\text{Time}_{\text{observe}}}$

The expression for mean number of task may be written as:

$\left(\frac{\text{Time}_{\text{accumulated}}}{\text{Time}_{\text{observe}}} \right) = \left(\frac{\text{Time}_{\text{accumulated}}}{\text{Number}_{\text{tasks}}} \right) \times \left(\frac{\text{Number}_{\text{tasks}}}{\text{Time}_{\text{observe}}} \right)$

Mean number of tasks = mean response time x Arrival rate

Amdahl's Law

Suppose that enhancement E accelerates a fraction F of the task by a factor S, and the remainder of the task is unaffected

Speedup due to enhancement E:

$$\begin{aligned} \checkmark \text{ Speedup (E)} &= \text{Ex Time without E} / \text{Ex Time with E} \\ &= \text{Performance with E} / \text{Performance without E} \end{aligned}$$

$$\text{Ex Time}_{\text{new}} = \text{Ex Time}_{\text{old}} \times [(1 - \text{Fraction}_{\text{enhanced}}) + \text{Fraction}_{\text{enhanced}} / \text{Speedup}_{\text{enhanced}}]$$

$$\begin{aligned} \text{Speedup}_{\text{overall}} &= \text{ExTime}_{\text{old}} / \text{ExTime}_{\text{new}} \\ &= 1 / [(1 - \text{Fraction}_{\text{enhanced}}) + \text{Fraction}_{\text{enhanced}} / \text{Speedup}_{\text{enhanced}}] \end{aligned}$$

Floating point instructions improved to run 2X; but only 10% of actual instructions are FP

$$\begin{aligned} \checkmark \text{ ExTime}_{\text{new}} &= \text{ExTime}_{\text{old}} \times (0.9 + .1/2) = 0.95 \times \text{ExTime}_{\text{old}} \\ \checkmark \text{ Speedup}_{\text{overall}} &= 1 / 0.95 = 1.053 \end{aligned}$$

Lecture 4

Instruction Set Principles

] √pm ""] √..ω"

Recap
ISA Taxonomy
Memory Addressing modes
Types of operands
Types of operations
Summary

Recap: Lec. 1-3 Chapter 1

In the **computer design cycle**, the decisive factors for rapid changes in the computer development have been the performance enhancements, price reduction and functional improvements

The processor **performance** of two designs is often compared by the factor n , which determines how much lower execution time one machine takes as compared to the other or how much faster the other machine is than first.

Price-Performance Design: The relationship between cost and price is complex one; and computer designers must understand this relationship as it effects the selling of their design.

✓ The **cost** is the total amount spends to produce a product and the **price** is the amount for which a finished good is sold and it is controlled by the die **yield** and **volume**.

Growth in Processor Performance: The supercomputers and mainframes, costing millions of dollars and occupying excessively large space, prevailing have been replaced with very low-cost microprocessor-based desktop computing machines in the form of personal computer (PC) and workstation.

Benchmark is a program developed to evaluate the performance of a computer. Good products created when have: proficient benchmarks and expert ways to summarize performance

An I/O system works on the principle of **producer-server model**, which comprises an area, called queue, where the tasks accumulate waiting to be serviced and the device performing the requested service, called server.

Producer creates tasks to be processed and place them in a FIFO buffer queue. The server takes the task form buffer and perform them

The response time is the time task takes from the moment it arrives in the buffer to the time the server finishes the task

Changing Definitions of Computer Architecture

Three Pillars of Computer Architecture

The three pillars of computer architecture are: hardware, instruction set and software. Hardware facilitates to run the software and instruction set is the interface between the hardware and software.

1950s to 1960s: The focus of the Computer Architecture Courses has been Computer Arithmetic

1970s to mid 1980s: The focus of Computer Architecture Course has been Instruction Set Design, the portion of the computer visible to programmer and compiler writer

1990s to date: The focus of the Computer Architecture Course is the Design of CPU, memory system, I/O system, Multiprocessors based on the quantitative principles to have price - performance design; i.e., maximum performance at minimum price

From the academic point of view, during the period 1950s - 1960s, the focus of computer architecture studies has been on the Computer arithmetic; i.e., the methodologies for the optimal solutions to arithmetic and logical problems.

Soon the researchers realized that the principles of instruction set, the portion visible to programmer and compiler writer, must be given importance to enhance the performance of the computer and reduce the complexity of hardware and hence optimize the price-performance. Thus, ISA design has been the focus of researchers during 1970s - 1980s.

The focus of the computer architecture studies, in 1990s is multi-dimensional and emphasis is given to CPU design, memory system, I/O system and multi-processor systems based on the quantitative principles to optimize the price-performance.

Instruction Set Architecture ISA

Our focus in couple of lectures will be the Instruction Set Architecture ISA which is the interface between the hardware-software

It plays a vital role in understanding the computer architecture from any of the above mentioned perspectives

It describes the instruction word format and identifies the memory addressing for data manipulation and control operations

Our focus in couple of lectures will be the Instruction Set Architecture ISA which is the interface between the hardware-software as it plays a vital role in understanding the computer architecture from any of the above mentioned perspectives

It describes the instruction word format and identifies the memory addressing for data manipulation and control operations

What is an interface?

A good interface:

- Lasts through many implementations (portability, compatibility)
- Is used in many different ways (generality)
- Provides convenient functionality to higher levels
- Permits an efficient implementation at lower levels

Taxonomy of Instruction Set

Major advances in computer architecture are typically associated with landmark instruction set designs stack, accumulator, general purpose register etc.

Design decisions must take into account:

- ✓ technology
- ✓ machine organization
- ✓ programming languages
- ✓ compiler technology
- ✓ operating systems

Basic Differentiator: The type of internal storage of the operand

Major Choices of ISA:

- ✓ Stack Architecture:
- ✓ Accumulator Architecture
- ✓ General Purpose Register Architecture
 - Register memory
 - Register Register (load/store)
 - Memory Memory Architecture (Obsolete)

Stack Architecture

Both the operands are implicitly on the TOS

Thus, it is also referred to as Zero-Address machine

The operand may be either an input (orange shade) or result from the ALU (yellow shade)

All operands are implicit (implied or inherited)

The first operand is removed from the stack and the second operand is replaced by the result

Example:

To execute: $C=A+B$

ADD instruction has implicit operands for the stack operands are written in the stack using PUSH instruction

PUSH A

PUSH B

ADD

POP C

Accumulator Architecture

An accumulator is a special register within the CPU that serves both as both the as the implicit source of one operand and as the result destination for arithmetic and logic operations.

Thus, it accumulates or collect data and hsi wr x' w' vzi " ew' er " ehv i ww' register at any time

Limited number of accumulators - usually only one are used

The second operand is in the memory, thus accumulator based machines are also called 1-address machines

They are useful when memory is expensive or when a limited number of addressing modes is to be used

To execute: $C=A+B$

ADD instruction has implicit operand A for the accumulator, written using LOAD instruction; and the second operand B is in memory at address B

Load A
ADD B
Store C

General Purpose Register Architecture

Many general purpose registers are available within CPU

Generally, CPU registers do not have dedicated functions and can be used for a variety of purposes address, data and control

A relatively small number of bits in the instruction is needed to identify the register

In addition to the GPRs, there are many dedicated or special-purpose registers as well, but many of these are not used frequently

GPR architecture has explicit operands either in register or memory thus there may exist:

- ✓ Register memory architecture
- ✓ Register Register (Load/Store) Architecture
- ✓ Memory Memory Architecture

One explicit operand is in a register and one in memory and the result goes into the register

The operand in memory is accessed directly
To execute: $C=A+B$

ADD instruction has explicit operand A loaded in a register and the operand B is in memory and the result is in register

Load R1, A
ADD R3, R1, B
Store R3, C

The explicit operands in memory are first loaded into registers temporarily and Are transferred to memory by Store instruction

To execute: $C=A+B$

ADD instruction has implicit operands A and B loaded in registers

```
Load R1, A
Load R2, B
ADD R3, R1, R2
Store R3, C
```

Both the explicit operands are not accessed from memory directly, i.e., Memory Architecture is obsolete

Comparison of three GPR Architectures

Register-Register

Advantages

- ✓ Simple, fixed-length instruction decoding
- ✓ Simple code generation
- ✓ Similar number of clock cycles / instruction

Disadvantages

- ✓ Higher Instruction count than memory reference
- ✓ Lower instruction density leads to larger programs

Register- Memory

Advantages

- ✓ Data can be accessed without separate Load first
- ✓ Instruction format is easy to encode

Disadvantages

- ✓ Operands are not equivalent since a source operand (in a register) is destroyed in operation
- ✓ Encoding a register number and memory address in each instruction may restrict the number of registers
- ✓ CPI vary by operand location

Memory- Memory

Advantages

- ✓ Most compact
- ✓ Fewer instructions

Disadvantages

- ✓ Large variation in instruction size
- ✓ Large variation in work per instruction
- ✓ Memory bottleneck by memory access

Evolution of Instruction Sets

Types and Size of Operands

Types of an Operand

- ✓ Integer
- ✓ Single-precision floating point
- ✓ Character

Size of Operand

- | | |
|-------------------------------|--------|
| ✓ Character | 8-bit |
| ✓ Half word | 16-bit |
| ✓ Single precision FP or Word | 32-bit |
| ✓ Double precision FP or | 64-bit |
| ✓ double word | |

Categories of Instruction Set Operations

All computer provide a full set of following operational instructions for:

Arithmetic and Logic

- ✓ Integer add, sub, and, or, multiply, divide

Data Transfer

- ✓ Load, store and
- ✓ Move instructions with memory addressing

Control

- ✓ Branch, Jump, procedure call and return

The following support instructions may be provided in computer with different levels

System

- ✓ operating system call, Virtual Memory Management

Floating point

- ✓ Add, multiply, divide and compare

Decimal

- ✓ BCD add, multiply and Decimal to Character Conversion

String

- ✓ String move, compare and search

Graphics

- ✓ Pixel and vertex operations, compression / de-compression operations

Operand Addressing Modes

Cr " i jji gxi "ehhvi ww "wx i binary bit pattern issued by the CPU to specify the location of operands in CPU (register) or the memory

Addressing modes are the ways of providing access paths to CPU registers and memory locations

Commonly used addressing modes are:

- ✓ Immediate
- ✓ Register
- ✓ Direct or Absolute
- ✓ Indirect

Data for the instruction is part of the instruction itself

- ✓ Immediate `ADD R4, # 24H` Ti k]T6d"" ""Ti k]T6d- "46"J "

Used to hold source operands only; cannot be used for storing results

- ✓ Register `ADD R4, R3` Ti k]T6d"" ""Ti k] R4] + Reg[R3]

Operand is contained in a CPU register

No memory access needed, therefore it is fast

- ✓ Direct (or absolute) `ADD R1,(1000)` Ti k]T3d"" ""Ti k]T3d- "Oi q]3222a

The address of the operand is specified as a constant, coded as part of the instruction

Limited address space ($2^{\text{operand field size}}$) locations

Indirect Addressing modes

The address of the memory location where the data is to be found is stored in the instruction as the operand, i.e., the operand is the address of an address

Large address space ($2^{\text{memory word size}}$) available

Two or more memory accesses are required

Types of Indirect addressing modes:

Register Indirect

Register Indirect Indexed

- ✓ Effective memory address is calculated by adding another register (index register) to the value in a CPU register (usually referred to as the base register)

- ✓ Useful for accessing 2-D arrays

Register Indirect plus displacement

- ✓ Useful for accessing an array element with respect to the first element. The address of the first element is stored in a register

Memory Indirect

Meanings of Indirect Addressing Modes

Register Indirect

- ✓ $ADD\ R4, (R1)$ $PC \leftarrow PC + 2$ $PC \leftarrow PC + 2$ $PC \leftarrow PC + 2$

Register Indirect Indexed

- ✓ $ADD\ R4, (R1+R2)$ $PC \leftarrow PC + 2$ $PC \leftarrow PC + 2$ $PC \leftarrow PC + 2$

Register Indirect plus displacement

- ✓ $ADD\ R4, 100(R1)$ $PC \leftarrow PC + 2$ $PC \leftarrow PC + 2$ $PC \leftarrow PC + 2$

Memory Indirect

- ✓ $ADD\ R4, @(R1)$ $PC \leftarrow PC + 2$ $PC \leftarrow PC + 2$ $PC \leftarrow PC + 2$

Special Addressing Modes

Used for stepping within loops; R2 points to the start of the array; each reference increments / decrements the value in R2 by a constant value.

- Auto-increment $ADD\ R1, (R2)+$
 $PC \leftarrow PC + 2$ $PC \leftarrow PC + 2$ $PC \leftarrow PC + 2$
 $PC \leftarrow PC + 2$ $PC \leftarrow PC + 2$ $PC \leftarrow PC + 2$
- Auto-decrement $ADD\ R1, (R2)-$
 $PC \leftarrow PC + 2$ $PC \leftarrow PC + 2$ $PC \leftarrow PC + 2$
 $PC \leftarrow PC + 2$ $PC \leftarrow PC + 2$ $PC \leftarrow PC + 2$

- Scaled

 $ADD\ R1, 100(R2)[R3]$
 $PC \leftarrow PC + 2$ $PC \leftarrow PC + 2$ $PC \leftarrow PC + 2$

Addressing Modes of Control Flow Instructions

Branch (conditional): a sort of displacement, in number of instructions, relative to PC

Jump (Unconditional): jump to an absolute address, independent of the position of PC

Procedure call / return: control transfer with some state and return address saving, some times in a special link register or in some GPRs

Summary

ISA Taxonomy

- ✓ Stack Architecture:
- ✓ Accumulator Architecture
 - General Purpose Register Architecture
 - Register memory
 - Register Register (load/store)
 - Memory Memory Architecture (Obsolete)

Memory Addressing modes

- ✓ Immediate
- ✓ Register
- ✓ Direct or Absolute
- ✓ Indirect
- ✓ Special
- ✓ Control Flow Instruction

Lecture 5

Instruction Set Principles (Encoding instructions and MIPS Instruction format)

] √pm ""] √...ω"

Recap Lecture 4
Instruction Set Encoding
MIPS Instruction Set
Summary

After a quick review of the previous discussion on quantitative principles of Computer Architecture we will have a detailed discussion on the instruction set principles and their examples. The Instruction set is an interface between the hardware and software design of computer

Recap: Lecture 4

Three pillars of Computer Architecture

- ✓ Hardware, Software and Instruction Set

Instruction Set

- ✓ Interface between hardware and software

Taxonomy of Instruction Set:

- ✓ Stack, Accumulator and General Purpose Register

Types and Size of Operands:

- ✓ Types: Integer, FP and Character
- ✓ Size: Half word, word, double word

Classification of operations

- ✓ Arithmetic, data transfer, control and support

Operand Addressing Modes

- ✓ Immediate, register, direct (absolute) and Indirect

Classification of Indirect Addressing

- ✓ Register, indexed, relative (i.e. with displacement) and memory

Special Addressing Modes

- ✓ Auto-increment, auto-decrement and scaled

Control Instruction Addressing modes

- ✓ Branch, jump and procedure call/return

Instruction set Encoding

Essential elements of computer instructions

1. Type of the operation to be performed
This information is encoded in the ~~st i~~ ^{first} 4 bits of the op-code, field of the machine language instruction
Examples: add, mov etc.
2. Place to find the source operand (s)
Possible locations are: CPU registers, memory cells, I/O locations, part of the instruction itself

3. Place to store the results
Possible locations are:
CPU registers, memory cells and I/O locations
4. Place to find the next instruction from
Address of the next instruction in sequence (this is the default case)
Address of the instruction at the branch target location

Instruction Word Types

Variable Length

- ✓ Operation is specified in one field, Op-code
- ✓ Can support any number of operands
- ✓ Each address specifier determines the addressing mode and length of the specifier for the operand
- ✓ Generally, it generates the smallest code representation as unused fields need not be included

Typical Examples: VAX, Intel 80x86

The decision regarding the length depends upon the range of addressing modes and degree of independence between op-code and mode

For example: immediate addressing requires one or two address field whereas indexed addressing requires 3 or 4 fields

The length of Intel 80x86 varies between 1 byte and 17 byte and is generally smaller than RICS architecture which uses fixed length format

Based on the previous discussion, we can classify instructions according to the format shown in this slide.

Strictly speaking, this classification is based on arithmetic and logic instructions in the instruction set. In this case, there are generally two operands and one result.

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. The discussion given in the book assumes that all operands are memory operands.

If a large number of CPU registers is available, a few bits in the instruction are used to encode one out of those registers

Fixed Length Format

Always has same number of operands
Addressing modes (if option exist) is specified in Op-code
It generally generates the largest code size
Fields may be used for different purposes, if necessary
Typical Examples: Alpha, MIPS, PowerPC, SPARC

Third Alternative: Hybrid Length

Multiple formats are specified by the op-code
One or two fields are added to specify addressing mode
Similarly, one or two fields specify operand address
It generally generates the optimum code size
Typical Examples: IBM 360/370, MIPS16, TI-TMS320c54x

Hybrid Length Taxonomy

Based on number of Address Fields

- ✓ 4-address instructions: Specifies the two source operands, the destination operand and the address of the next instruction
- ✓ 3-address instructions: Specifies addresses for both operands as well as the result

- ✓ **2-address instructions**
 - Overwrites one operand with the result
 - One field serves two purposes
- ✓ **1-address instructions:**
 - Dedicated CPU register, accumulator, to hold one operand and the result
 - The address of other operand is specified
- ✓ **0-address instructions:**
 - Uses a stack to hold both operands and the result
 - Operations are performed between the value on the top of the stack (TOS) and the second value on the stack (SOS) and the result is stored on the TOS

Example

Evaluate the expression: $F = (B + C) * D - E$, using 0- address through 3-address format

	0-Address	1-Address	2-Address	3-Address
	PUSH B	LDA B	LOAD F, B	ADD F, B, C
	PUSH C	ADD C	ADD F, C	MUL F, F, D
	ADD	MUL D	MUL F, D	SUB F, F, E
	PUSH D	SUB E	SUB F, E	
	MUL	STA F		
	PUSH E			
	SUB			
	POP F			
Number of Instructions:	8	5	4	3

Example: Using different instruction formats, write pseudo-code to evaluate the following expression: $Z = 4(A+B) - 16(C+58)$: Your code should not change the source operands

3-Address	2-Address	1-Address	0-Address
ADD x, A, B MUL y, x, 4 ADD r, C, 58 MUL s, r, 16 SUB Z, y, s	LOAD y, B MUL y, 4 LOAD s, C ADD s, 58 MUL s, 16 SUB y, s STORE Z, y	; order changed to reduce code size LDA C ADDA 58 MULA 16 STA S LDA A ADDA B MULA 4 SUBA s STA Z	PUSH C PUSH 58 ADD PUSH 16 MUL PUSH A PUSH B ADD PUSH 4 MUL SUB POP Z

Comparison of instruction formats

Assume that

A single byte is used for the op code

The size of the memory address space is 16 Mbytes

A single addressable memory unit is a byte

Size of operands is 24 bits

Data bus size is 8 bits

Use the following two parameters and compare the five instruction formats (O-4 address) mentioned earlier

- ✓ Code size: Its effect on the storage requirements
- ✓ Number of memory accesses: $k \times w_i \times j \times g \times s \times r \times i \times i \times g \times s \times r \times i \times i$

A single byte, or an 8-bit, op code can be used to encode up to 256 instructions.

A 16-Mbyte memory address space will require 24-bit memory addresses. We will assume a byte wide memory organization to make this example different from the example in the book.

The size of the address bus will be 24 bits and the size of the data bus will be 8-bits.

4-address instruction

Code size = $1+3+3+3+3 = 13$ bytes

of bytes accessed from memory

13 bytes for instruction fetch + 6 bytes for source operand fetch + 3 bytes for storing destination operand = 22 bytes

3-address instruction

Code size = $1+3+3+3 = 10$ bytes

of bytes accessed from memory

10 bytes for instruction fetch + 6 bytes for source operand fetch + 3 bytes for storing destination operand = 19 bytes

2-address instruction

Code size = $1+3+3 = 7$ bytes

of bytes accessed from memory

7 bytes for instruction fetch + 6 bytes for source operand fetch + 3 bytes for storing destination operand = 16 bytes

1-address instruction

Code size = $1+3 = 4$ bytes

of bytes accessed from memory

4 bytes for instruction fetch + 3 bytes for source operand fetch + 0 bytes for storing destination operand = 7 bytes

0-address instruction

Code size = 1 bytes

of bytes accessed from memory

1 bytes for instruction fetch + 6 bytes for source operand fetch + 3 bytes for storing destination operand = 10 bytes

Result Summary

A single byte is used for the op code, 16 MB memory address space, single addressable memory unit: byte, 24 bits operands is 24 bits and 8-bit data bus

Instruction Format	Code size	Number of memory bytes
4-address instruction	13	22
3-address instruction	10	19
2-address instruction	7	16
1-address instruction	4	7
0-address instruction	1	10

RISC and MIPS ISA

RISC and MIPS is a fixed length, 64-bit LOAD/STORE Architecture

Contains 32 GPR each of 32-bit

Supports:

- ✓ 3-address, reg-reg arithmetic instruction
- ✓ displacement instructions with address offset 12-16 bits
- ✓ immediate data 8-bit and 16-bit and
- ✓ register indirect
- ✓ data size 8-, 16-, 32- and 64-bit integer
- ✓ 64-bit IEEE 754 floating point

Instructions:

- ✓ Data Transfer: load, store, register-register move
- ✓ Simple Arithmetic: add, subtract, and shift
- ✓ Compare: equal, not-equal, less
- ✓ Branch: PC-relative, jump and call/return

Designed for pipelining efficiency

MIPS Instruction Word format

Recap: MIPS types and size of operands

Types of an Operand

- ✓ Integer
- ✓ Single-precision floating point
- ✓ Character

Size of Operand

- ✓ Character 8-bit
- ✓ Half word 16-bit
- ✓ Single precision FP or Word 32-bit
- ✓ Double precision FP or 64-bit
- ✓ double word

Register-Register (R-Type)

Op-code = 000000

Rs and Rt : source operand registers

Rd : Result carrying register

Sht: Number of bit-shift (left/right)

Func: ALU function to encode the data path operation

Execution: $Rd \leftarrow Rs \text{ func } Rt$

Example Encoding MIPS64

R-Type Arithmetic / Logical Instructions

Arithmetic Instructions

Instruction	Name	Meaning
DADD R1, R2, R3	Add word (signed)	$Reg[R1] \leftarrow Reg[R2] + Reg[R3]$
DADDU R1, R2, R3	Add unsigned	$Reg[R1] \leftarrow Reg[R2] + Reg[R3]$

Shift Instruction

DSLL R1, R2, # 30 Shift Left Logical $Reg[R1] \leftarrow Reg[R2] \ll 30$

Register-Immediate (I- Type)

Rt is the destination field for immediate data instructions

$Rt \leftarrow Rs \text{ op immediate}$; all immediate

Same format is used for Load/Store instructions

$Rt \leftarrow Mem [\text{immediate} + [Rs]]$; Load

$Mem [\text{immediate} + [Rs]] \leftarrow Rt$; Store

MIPS Operations

Immediate Arithmetic / Logical Instructions

Instruction	Name	Meaning
DADDIU R1, R2, # 30	Add unsigned Imm	Ti k]T3đ' "Ti k]T4đ- "Ti k]T5a

Load/Store Instructions

Instruction	Name	Meaning
LW R1, 30 (R2)	Load word	Ti k]T3đ' "Oi q "]52- Ti k] R2]
SW R1, 30(R2)	Store word	Oi q "]52- Ti k]T4đ' "Ti k]T3a

Branch /Jump Register

- i. Conditional Branch Instructions:
used after the compare or test BEQZ:
Rs is the register and Rt is unused;
Condition test the register for Zero or non-zero
- ii. Condition with the Branch; BNE
Rs and Rt are compared
- iii. Jump Register; Jump and Link Register
Rt=0, Rs = Destination and immediate = 0

MIPS Operations

Branch/Jump Register

Instruction	Name	Meaning
BEQZ R4, name	Branch equal zero	ķ"Ti k]T6đ?"2""ł i r "RE" "r eq i

Instruction	Name	Meaning
BNE R4, R3, name	Branch not equal zero	If $\text{Reg}[R4] \neq \text{Reg}[R3]$ then $\text{PC} \leftarrow \text{name}$

Instruction	Name	Meaning
JR R4	jump register	$\text{PC} \leftarrow \text{Reg}[R4]$
JALR R4	Jump and Link Register	$\text{PC} \leftarrow \text{Reg}[R4] + 4$; $\text{Reg}[R4] \leftarrow \text{PC}$

Jump / Call

Jump: uses 26-bit offset; shifted 2-bit then replace the lower 28 bits of PC [of the instruction following the jump

Jump and Link (Procedure Call) place the return address; the address of the next instruction in R31 used for Return from procedure

MIPS Operations

Jump/Call

Instruction	Name	Meaning
J	name Jump	$\text{PC} \leftarrow \text{PC} + \text{offset} \ll 2$
JAL R4	Jump and Link	$\text{PC} \leftarrow \text{PC} + \text{offset} \ll 2$; $\text{Reg}[R4] \leftarrow \text{PC}$

Summary

Instruction encoding

- ✓ Essential elements of computer instructions: type of operands, places of source and destinations and place of next instruction
- ✓ Instruction word length
 - Variable, fixed length and hybrid
- ✓ Hybrid length taxonomy
 - 4, 3, 2, 1 and 0 address format

- ✓ Comparison of hybrid instruction word format
Minimum number of memory bytes are required in case of 1 address (accumulator) format and maximum for 4-address format

MIPS Instruction word format

- ✓ RISC and MIPS is a fixed length, 64-bit LOAD/STORE Architecture
- ✓ It supports: Size of Operand
 - Character (8-bit)
 - Half word (16-bit)
 - Single precision FP or Word (32-bit)
 - Double precision FP or double word (64-bit)
- ✓ Instruction word formats
 - R-type, I-type and J-type

Lecture 6

Instruction Set Principles (ISA Performance Analysis, Fallacies and Pitfalls)

Welcome to the sixth lecture of the series on Advanced Computer Architecture.

Today we will conclude our discussion on the Instruction Set Architecture

We will be talking about the ISA performance, role of compiler writer, media and signal processing operations and fallacies and pitfalls

] √pm ""] √..ω"

Recap Lecture 5

DSP Media Operations

ISA Performance

Putting it all Together

Summary

Recap: Lecture 5

Instruction encoding

- ✓ Essential elements of computer instruction word:
 - Type of operands
 - Places of source and destinations
 - Place of next instruction
- ✓ Instruction word length
 - Variable Length
 - Fixed length
 - Hybrid variable fixed
- ✓ Categories of Hybrid length
 - 4, 3, 2, 1 and 0 address format
- ✓ Comparison of hybrid instruction word format
 - Minimum number of memory bytes are required in case of 1 address (accumulator) format
 - Maximum for 4-address format
- ✓ MIPS Instruction word format
 - RISC and MIPS a fixed length, 64-bit LOAD/STORE
 - Architecture
- ✓ It supports:
 - 8-, 16-, 32- and 64-bit operand
 - R-type, I-type and J-type
 - Arithmetic and logic operation
 - data transfer operations
 - Control flow operations

Media and Signal Processing Operands

Graphic applications deal with 2D and 3D images

3D data type is called vertex

Vertex structure has 4-components

- ✓ x- coordinate
- ✓ y- coordinate
- ✓ z- coordinate
- ✓ w-coordinate

The three vertices specify a graphic primitive, such as a triangle; and the fourth to help with color and hidden surfaces

Vertex values are usually 32-bit Floating point values

DSP adds fixed point to the data types binary point just to the right of the sign-bit

3D Data Type

A triangle is visible when it is depicted as filled with pixels

Pixels are typically 32-bits, usually consisting of four 8-bit channels

- ✓ R -red
- ✓ G-green
- ✓ B-blue
- ✓ A: Transparency of pixel when it is depicted

Media and Signal Processing Operations

Data for multimedia operations is usually much narrower than the 64-bit data word of modern processors

Thus, 64-bit may be partitioned in to four 16-bit data values so that the 64-bit ALU to perform four 16-bit operations (say add operation) in a single clock cycle

These operations are called Single-Instruction Multiple-Data (SIMD) or vector operations

Multimedia Operations

Most graphic multimedia applications use 32-bit floating point operations allowing a single instruction to launch two 32-bit operations on operands found side-by-side in double precision register

The table shown here summarizes SIMD instructions found in recent computers

You may note that there is very little common across the five architectures

All are fixed-width operation , performing multiple narrow operations on either 64-bit or 128-bit ALU

The narrow operation are shown as

- ✓ B-byte,
- ✓ H-half word
- ✓ W-word and
- ✓ 8B double word

Summary of SIMD instructions in recent computers

Insert Table given in Fig. 2.17 from page 110

Digital Signals Processing Issues

Saturating Add/Subtract

- ✓ Too Large Result and Overflow

Result Rounding

- ✓ Choose from IEEE 754 mode algorithms

Multiply Accumulate

- ✓ Vector and Matrix dot product operations

DSP Operations

Saturating Add/Sub

- ✓ DSP cannot ignore results of overflow otherwise it may miss an event, therefore, it uses saturating arithmetic.
- ✓ Here, if the result is too large to be presented it is set to the largest representable number, based on the sign of the number

Result Rounding

- ✓ IEEE 754 has several algorithms to round the wider accumulator into narrower one, DSPs select the appropriate mode to round the result

Multiply-Accumulate (MAC)

- ✓ MAC operations are the key to dot product operations of vector and matrix multiply which need to accumulate a series of product

ISA Performance

Role of Compiler

The interaction of compiler and high-level languages significantly effects how program uses an ISA

Optimizations performed by the compilers can be classified as follows:

- ✓ High-level optimization: is often done on the source with the output fed to the later optimization passes.
- ✓ Local Optimization: is done within a straight-line code fragment (basic block)
- ✓ Global Optimization: extends the optimization across branches
- ✓ Register Allocation: associate registers with operands
- ✓ Processor-dependent optimization: using the specific architecture

Impact of Compiler Technology

Interaction of compiler and high-level language affects how a program uses an ISA

Here, two important questions are:

1. How are variables allocated?
2. How many registers are needed to allocate variables appropriately?

These questions are addressed by using three areas in which high-level language allocates data

Three areas of data allocation

1. Local Variable area Stack
 - ✓ It is used to allocate local variable
 - ✓ it grows or shrinks on procedure call or return
 - ✓ Objects on stack are primarily scalar single variable rather than arrays and are addressed by stack-pointer
 - ✓ Register allocation is much more effective for stack-allocated objects
2. Global Data Area
 - ✓ It is used to allocate statically declared objects such as global variables and constants
 - ✓ These objects are mostly arrays and other aggregate data structures
 - ✓ Register allocation is relatively less effective for global variables
 - ✓ Global variables are aliased there are multiple way to address so make it illegal to put on registers
3. Dynamic Object Allocation: Heap
 - ✓ It is used to allocate the objects that do not adhere to stack
 - ✓ The objects in heap are accessed with pointer but are not scalars
 - ✓ Most heap variable are aliased so register allocation is almost impossible for heap

4. Floating-point Operations

MIPS Floating-point Operations

- ✓ The instructions manipulate the floating-point registers
- ✓ They indicate whether the operation is to be performed on single precision or double precision
 - MOV.S copies a single precision register to another of the same type
 - MOV.D copies a Double precision register to another of the same type

To get greater performance for graphic routines, MIPS64 offers Paired-Single Instructions

These instructions perform two 32-bit floating point operations on each half of the 64-bit floating point register

Examples:

- | | |
|----------|----------|
| ✓ ADD.PS | ✓ MUL.PS |
| ✓ SUB.PS | ✓ DIV.PS |

Putting it All Together

The earliest architectures were limited to instruction sets by the hardware technology of that time

In the 1960s, stack architecture became popular, viewed as being good match of high-level language

In the 1970s, the main concern of the architectures was to reduce the software cost, thus produced high-level architectures such as VAX machine

In the 1980s, return to simpler architecture took place due to sophisticated compiler technology

In the 1990s, new architectures were introduced; these include:

1990s Architectures

1. Address size doubles 32-bit to 64-bit
2. Optimization of conditional branches via conditional execution e.g.; conditional move
3. Optimization of Cache performance via pre-fetch that increased the role of memory hierarchy in performance of computers
4. Multimedia support
5. Faster Floating point instructions
6. Long Instruction Word

Concluding the Instruction set Principles

Three pillars of Computer Architecture

- ✓ Hardware, Software and Instruction Set

Instruction Set

- ✓ Interface between hardware and software

Taxonomy of Instruction Set:

- ✓ Stack, Accumulator and General Purpose Register

Types and Size of Operands:

- ✓ Types: Integer, FP and Character
- ✓ Size: Half word, word, double word

Classification of operations

- ✓ Arithmetic, data transfer, control and support

Operand Addressing Modes

- ✓ Immediate, register, direct (absolute) and Indirect

Classification of Indirect Addressing

- ✓ Register, indexed, relative (i.e. with displacement) and memory

Special Addressing Modes

- ✓ Auto-increment, auto-decrement and scaled

Control Instruction Addressing modes

- ✓ Branch, jump and procedure call/return

Explanation:

Hardware, software and Instruction set are the three pillars of Computer architecture

During the period 1950s - 1960s, the focus of computer architecture studies has been on the Computer arithmetic; i.e., the methodologies for the optimal solutions to arithmetic

and logical problems.

1970s to mid 1980s: The focus of Computer Architecture has been Instruction Set Design

Soon the researchers realized that the principles of instruction set, the portion visible to programmer and compiler writer, must be given importance to enhance the performance of the computer and reduce the complexity of hardware and hence optimize the price-performance. Thus, ISA design has been the focus of researchers during 1970s 1980s.

The focus of the computer architecture studies, in 1990s is multi-dimensional and emphasis is given to CPU design, memory system, I/O system and multi-processor systems based on the quantitative principles to optimize the price-performance.

Instruction encoding

Instruction encoding

- ✓ Essential elements of computer instructions:
 - type of operands, places of source and destinations and place of next instruction
- ✓ Instruction word length
 - Variable, fixed length and hybrid
- ✓ Hybrid length taxonomy
 - 4, 3, 2, 1 and 0 address format
- ✓ Comparison of hybrid instruction word format
 - Minimum number of memory bytes are required in case of 1 address (accumulator) format and maximum for 4-address format

MIPS Instruction word format

- ✓ RISC and MIPS a fixed length, 64-bit LOAD/STORE Architecture
- ✓ It supports:
 - 8-, 16-, 32- and 64-bit operand
 - R-type, I-type and J-type
 - Arithmetic and logic operation
 - data transfer operations
 - Control flow operations

Multimedia and Digital Signal Processing Operands

- ✓ Graphic applications deal with 2D and 3D images
- ✓ DSP adds fixed point to the data types binary point just to the right of the sign-bit

Multimedia and Digital Signal Processing operations

- ✓ All are fixed-width operation, performing multiple narrow operations on either 64-bit or 128-bit ALU
- ✓ The narrow operation B-byte, H-half word, W-word and 8B double word

Multimedia and Digital Signal Processing issues

- ✓ Saturating Add/Subtract
- ✓ Result Rounding
- ✓ Multiply Accumulate

ISA Performance

- ✓ Role of Compiler: The interaction of compiler and high-level languages significantly effects how program uses an ISA

Practice Problems: Quantitative Principles [Lecture 2-3]

- Computer hardware is designed using ISA having three types (Type A, B and C) of instructions. The clock cycles per instruction (CPI) for each type of instruction is as follows:

Type	A	2 CPI
Type	B	3 CPI
Type	C	4 CPI

A compiler writer has written two different code sequences with different instruction count to execute an expression as given below.

Code Sequence	Instruction count for instruction type		
	A	B	C
1	2	1	4
2	3	2	1

- What is the instruction count of each sequence?
- Which of the sequence is faster?
- What is the CPI (average) for each instruction?

Solution to Practice Problem 1

- The instruction count of

$$\text{Sequence 1} = 2 + 4 + 1 = 7$$

$$\text{Sequence 2} = 1 + 1 + 4 = 6$$

Result: Sequence 2 executes fewer instructions

- To find which sequence is faster, we have to find the CPU clock cycles for each sequence

$$\text{CPU Clock Cycles for sequence 1} = 2 \times 2 + 3 \times 1 + 4 \times 4 = 20 \text{ cycles}$$

$$\text{CPU Clock Cycles for sequence 2} = 2 \times 3 + 3 \times 2 + 4 \times 1 = 17 \text{ cycles}$$

Result: Sequence 1 is faster

- To find the CPI [CPU Cycles/Instruction Count] of each sequence

$$\text{CPI for sequence 1} = 20/7 = 2.85$$

$$\text{CPI for sequence 2} = 17/6 = 2.83$$

Result: Sequence 2 which has fewer instructions has higher CPI, thus is slower

Lecture 7

Computer Hardware Design (Single Cycle Data path and Control Design)

Welcome to the seventh lecture of the series on Advanced Computer Architecture.
Today we will start with the review discussion on the hardware design of computer

Topm ""]√..ω"

Recap: Instruction Set Principles

Basics of Computer Hardware Design (Review)

Single Cycle Design

- ✓ Data Path design
- ✓ Control Design

Summary

Recap: Instruction Set Principles

Three pillars of Computer Architecture

Instruction encoding

- ✓ Instruction word length: Fixed, variable and Hybrid length
- ✓ MIPS Instruction word format

Multimedia and Digital Signal Processor Operands and Operations

- ✓ Digital Signal Processing Issues
- ✓ Saturating Add/Subtract
- ✓ Result Rounding
- ✓ Multiply Accumulate

Instruction Set Performance

- ✓ Role of Compiler
- ✓ Impact of Compiler Technology
- ✓ Two ways the interaction of compiler and high-level language affects the use of ISA by a program
 1. How are variables allocated?
 2. How many registers are needed to allocate variables appropriately?

Three areas of data allocation

- ✓ Local Variable area Stack
- ✓ Global Data Area
- ✓ Dynamic Object Allocation: Heap

Basics of Hardware Design

We will be talking about!

Basic building blocks of a computer

Sub-systems of CPU

Processor design steps

Processor design parameters

Basic building blocks of a computer

Central Processing Unit

Subsystems:

Memory

Input / Output (Peripherals)

Buses

Sub-systems of Central Processing Unit

The Central Processing Unit (CPU) is composed of two main sub-systems:

- ✓ Datapath: the path that facilitates the transfer of information from one part (register/memory/ IO) to the other part of the system
- ✓ Control: the hardware that generates signals to control the sequence of steps and direct the flow of information through the datapath

Design Process

Design is a "creative process," not a simple method. Design finishes as Assembly

Design understood in terms of components and how they have been assembled

Top Down of complex functions (behaviors) into more primitive functions

Bottom-up composition of primitive building blocks into more complex assemblies

Processor Design Steps

Design the Instruction Set Architecture

Use RTL to describe the behavior of the processor

- ✓ Static as well as dynamic
- ✓ Includes the functional description of each instruction in the ISA

Select a suitable implementation (internal organization) of the data path

Map the behavioral RTL description of each instruction on to a set of structural RTL, based on the chosen implementation

- ✓ Implies the existence of suitable timing intervals provided by synchronous clocking signals

Develop logic circuits to generate the necessary control signals

Tie every thing together datapath and control signals

Other things which should be minimized

- ✓ Amount of control hardware
- ✓ Development time

Performing an Operation

Each instruction of a program is performed in two phases:

- ✓ Instruction Fetch
- ✓ Instruction Execute

Each phase is divided into number of steps, called Micro-operation

A micro-operation is completed in a fixed time interval

The number of micro-operations is determined by the datapath implementation

Datapath Implementations

The datapath is the arithmetic organ of the Von-Neumann architecture

Typically, the datapath may be implemented as:

- ✓ Unibus structure
- ✓ 2-bus structure
- ✓ 3-bus structure

It consists of registers, internal buses, arithmetic units and shifters

Each register in the register file has:

- ✓ A load control line that enables data load to register
- ✓ A set of tri-state buffers between its output and the bus
- ✓ A read control line that enables its buffer and place the register on the bus

A Typical Unibus Datapath Implementation

Typical Unibus Datapath Structure

It consists of a register file having 32 registers each of 32-bit and internal bus connecting the arithmetic and shifter unit to the register file

Other registers (PC, IR, MAR, MBR, A, C) have a load control line too

Registers PC and MBR also have a set of tri-state buffers between their output and the internal CPU bus

Additionally, registers MAR and MBR have other circuitry connecting them to the external CPU bus

RTL micro-operations of Unibus structure

Instruction Fetch:

Completed in the following three steps (time intervals):

T0	$MAR \leftarrow PC, C \leftarrow PC + 4;$
T1	$MBR \leftarrow M[MAR], PC \leftarrow C;$
T2	$IR \leftarrow MBR$

Instruction Execute:

Instructions of different classes are Completed in the different number of steps (time intervals):

Execution Phase micro-operations of Unibus

R-type Arithmetic/Logical Instructions (Add/Sub/And/OR ra, rb, rc) or immediate

T3 $A \leftarrow R[rb];$

T4 $C \leftarrow A \text{ op } R[\text{rc}]; \text{ or } T4 \quad C \leftarrow A \text{ op } \text{Const}(\text{sign extended})$

T5 $R[ra] \leftarrow C;$

R-type 2-address instructions (e.g. NOT ra, rb)

T3 $C \leftarrow \text{NOT}(R[\text{rb}]);$

T4 $R[ra] \leftarrow C;$

RTL micro-operations of Unibus structure

Load/store Instructions (ld/st ra, c2(rb))

T3 A ← **v f "?"2+"<2."*v f " "2+<"T]v f a=""

T4 $C \leftarrow A + (\text{sign extended and shifted } c2);$

T5 MAR $\leftarrow$ C;

T6 MBR $\leftarrow$ M[MAR]; (load) MBR $\leftarrow$ R[ra]; (store)

T7 $R[ra] \leftarrow MBR$; (load) $M[MAR] \leftarrow MBR$; (store)

Branch instructions (e.g. : brzr rb, rc)

```
T3    CON ← cond(R[rc]);
```

T4 CON: $PC \leftarrow R[rb];$

A 2-bus implementation

Typical 2-bus Datapath Structure

Registers and arithmetic and logic unit are identical to uni-bus structure

The structure contains two internal buses called the in-bus and out-bus

The in-bus carries data to be written into registers and out-bus carries data read out from the registers

The output of ALU is directly connected to the in-bus instead of through register C as in Uni-bus structure

Fetch/Execution Phase micro-operations of 2-bus

Three micro-operations (steps) of the Fetch Phase are identical to Uni-bus structure except $C \leftarrow PC+4$ in step T0

R-type Arithmetic/Logical Instructions are completed in two steps instead of three (Add/Sub/And/OR ra, rb, rc) or immediate

T3 $A \leftarrow R[rb];$

T4 $R[ra] \leftarrow A \text{ op } R[rc];$

R-type 2-address instructions (e.g. NOT ra, rb)

T3 $R[ra] \leftarrow \text{NOT}(R[rb]);$

Typical 3-bus Datapath Structure

Registers and arithmetic and logic unit are identical to uni-bus and 2-bus structure

The structure contains three internal buses called the A-bus, B-bus and C-bus

The register file contains two read ports connected to A-bus and B-bus and one write port connected to C-bus

The registers A and C are not provided as the A input and C output of ALU are connected to the bus A and C respectively

Fetch Phase is completed in two steps and Execute phase of R-type instructions in one step

Fetch and Execute of sub instruction

using the 3-bus data path implementation

Format: sub ra, rb, rc

Processor Design Parameters

Recall:

Execution time (ET) = IC x CPI x T

✓ Instruction Count = IC

✓ Clock Cycles per Instruction = CPI

✓ Clock cycle or time period = T

Note that Implementation affects CPI and T

Single and Multi cycle Datapaths

The datapath where an instruction is fetched and executed in one clock cycle, e.g.,

CPI = 1, is referred to as SINGLE CYCLE datapath

The datapath where different classes of instruction are fetched and executed in variable number of cycles is referred to as MULTI-CYCLE datapath

Single cycle Datapath

The instruction fetch and execute phases are completed in one clock cycle

A clock cycle is divided into number of steps to complete the operations

The cycle length is constant whereas number of steps (or micro operation) may be variable

The timing step generator returns to T0 on the completion of a cycle

Worst Case Timing (Load)

Single cycle Timing

This timing diagram shows the worst case timing of single cycle datapath (which occurs at the load instruction).

Clock-to-Q time after the clock tick, PC will present its new value to the Instruction memory.

After a delay of instruction access time, the instruction bus (Rs, Rt, ...) becomes valid.

Then three things happen in parallel:

- First the Control generates the control signals (Delay through Control Logic).
- Secondly, the register file is accessed to put Rs onto busA.
- Thirdly, in case of memory reference or immediate data instructions, we have to sign extend the immediate field to get the second operand (busB)

Here we assume register file access takes longer time than doing the sign extension so we have to wait until busA valid before the ALU can start the address calculation (ALU delay).

With the address ready, we access the data memory and after a delay of the Data Memory Access time, busW will be valid.

And by this time, the control unit would have set the RegWr signal to one so at the next clock tick, we will write the new data coming from memory (busW) into the register file.

Single cycle Memory Structure

As clear from the timing diagram, the memory address (from PC) for instruction fetch; and from ALU for the data read/write; are available on the bus simultaneously thus gives rise to structural hazard

To overcome this problem memory unit is partitioned in to parts

- ✓ Instruction memory
- ✓ Data memory

Single Cycle Instruction Fetch Unit

Fetch the instruction from Instruction memory:

Instruction $\leftarrow$ Mem[PC]

- ✓ This is the same for all instructions

A Single Cycle Datapath

The Single Cycle Datapath during Add

WWW.VUMULTAN.COM

The Single Cycle Datapath during Add

This picture shows the activities at the main datapath during the execution of the Add or Subtract instructions.

The active parts of the datapath are shown in different color as well as thicker lines.

First of all, the Rs and Rt of the instructions are fed to the Ra and Rb address ports of the register file and cause the contents of registers specified by the Rs and Rt fields to be placed on busA and busB, respectively.

With the ALUctr signals set to either Add or Subtract, the ALU will perform the proper operation and with MemtoReg set to 0, the ALU output will be placed onto busW.

The control we are going to design will also set RegWr to 1 so that the result will be written to the register file at the end of the cycle.

In this case can either do a $PC \leftarrow PC + 4$ or $PC \leftarrow PC + PC_{old}$ --we are using busB.

The other control signals we need to worry about are:

- ✓ MemWr has to be set to zero because we do not want to write the memory.
- ✓ And Branch and Jump, we have to set to zero. Let me show you why.

Instruction Fetch Unit at the End of Add

$PC \leftarrow PC + 4$; This is the same for all instructions except: Branch and Jump

This picture shows the control signals setting for the Instruction Fetch Unit at the end of the Add or Subtract instruction.

Both the Branch and Jump signals are set to 0.

Consequently, the output of the first adder, which implements $PC + 1$, is selected through the two 2-to-1 mux and got placed into the input of the Program Counter register.

The Program Counter is updated to this new value at the next clock tick.

Notice that the Program Counter is updated at every cycle. Therefore it does not have a Write Enable signal to control the write.

Also, this picture is the same for or all instructions other than Branch and Jump.

Therefore I will only show this picture again for the Branch and Jump instructions and will not repeat this for all other instructions.

+2 = 17 min. (X:57)

Summary of Today's Lecture

Basic building blocks of a computer

Sub-systems of CPU

Processor design steps

Processor design parameters

Hardware design process

Timing signals

Uni-bus, 2-bus and 3-bus structures

3-bus based single cycles data path

Lecture 8

Computer Hardware Design (Multi Cycle Datapath and Control Design)

Welcome to the seventh lecture of the series on Advanced Computer Architecture.
Today we will start with the review discussion on the hardware design of computer

] √pm ""] √..ω"

Recap: Single cycle datapath and control

Example of Single Cycle Design

Multi Cycle Design - Datapath

Summary

Recap: Lecture 7

Basic building blocks of a computer:

CPU, Memory and I/O sub-systems and Buses

CPU sub-system: Datapath and control

Phases of instruction performing: Fetch and Execute

Datapath Designs: Uni-, 2- and 3-bus structures

Micro-operations of Fetch and execute phases:

Fetch: $MBR \leftarrow M[PC]$; $PC \leftarrow PC+4$; $IR \leftarrow MBR$

✓ Exe: ID, operand read; exe; mem; WB

✓ 3-bus based single cycles data path MIPS datapath

Control signals for single cycles data path Add Instruction

A critical review of single cycle datapath and control signals

Fetch Circuit

Control Signals for Add rd,rs,rt

$$R[rd] \leftarrow R[rs] + R[rt]$$

This picture shows the activities at the main datapath during the execution of the Add or Subtract instructions.

The active parts of the datapath are shown in different color as well as thicker lines.

First of all, the Rs and Rt of the instructions are fed to the Ra and Rb address ports of the register file and cause the contents of registers specified by the Rs and Rt fields to be placed on busA and busB, respectively.

With the ALUctr signals set to either Add or Subtract, the ALU will perform the proper

operation and with MemtoReg set to 0, the ALU output will be placed onto busW.

The control we are going to design will also set RegWr to 1 so that the result will be written to the register file at the end of the cycle.

Because ALUSrc will be equal to 0--we are using busB.

The other control signals we need to worry about are:

- MemWr has to be set to zero because we do not want to write the memory.
- And Branch and Jump, we have to set to zero. Let me show you why.

+3 = 15 min. (X:55)

Instruction Fetch Unit at the End of Add

PC $\leftarrow$ PC + 4; This is the same for all instructions except: Branch and Jump

This picture shows the control signals setting for the Instruction Fetch Unit at the end of the Add or Subtract instruction.

Both the Branch and Jump signals are set to 0.

Consequently, the output of the first adder, which implements PC plus 1, is selected through the two 2-to-1 mux and got placed into the input of the Program Counter register.

The Program Counter is updated to this new value at the next clock tick.

Notice that the Program Counter is updated at every cycle. Therefore it does not have a Write Enable signal to control the write.

Also, this picture is the same for or all instructions other than Branch and Jump.

Therefore I will only show this picture again for the Branch and Jump instructions and will not repeat this for all other instructions.

+2 = 17 min. (X:57)

The Single Cycle Datapath during Or Immediate

Consider the Or immediate instruction.

The OR immediate instruction ORs the content of the register specified by the Rs field to the Zero Extended Immediate field and write the result to the register specified in Rt.

This is how it works in the datapath. The Rs field is fed to the Ra address port to cause the contents of register Rs to be placed on busA.

The other operand for the ALU will come from the immediate field. In order to do this, the controller need to set ExtOp to 0 to instruct the extender to perform a Zero Extend operation.

Furthermore, ALUSrc must set to 1 such that the MUX will block off bus B from the register file and send the zero extended version of the immediate field to the ALU.

Of course, the ALUctr has to be set to OR so the ALU can perform an OR operation.

The rest of the control signals (MemWr, MemtoReg, Branch, and Jump) are the same as the Add and Subtract instructions.

One big difference is the RegDst signal. In this case, the destination register is specified by the Rt field, NOT the Rd field because we do not have a Rd field here.

The controller will set the RegWr signal to 1 to enable the register file write port.

Finally, in order to accomplish the register write, RegWr must be set to 1.

+3 = 20 min. (X:60)

The Single Cycle Datapath during OR Immediate

Ps{ "p x w"so"ex" i "gsr xspwr ep" setting for the OR immediate instruction.

The OR immediate instruction OR the content of the register specified by the Rs field to the Zero Extended Immediate field and write the result to the register specified in Rt.

This is how it works in the datapath. The Rs field is fed to the Ra address port to cause the contents of register Rs to be placed on busA.

The other operand for the ALU will come from the immediate field.

In order to do this, the controller need to set ExtOp to 0 to instruct the extender to perform a Zero Extend operation.

ALUSrc must set to 1 such that the MUX will block off bus B from the register file and send the zero extended version of the immediate field to the ALU.

The ALUctr has to be set to OR so the ALU can perform an OR operation.

The rest of the control signals (MemWr, MemtoReg, Branch, and Jump) are the same as the Add and Subtract instructions.

One big difference is the RegDst signal. In this case, the destination register is specified by Rt i "nwysr wTxjnp."PQV" i "Thfield because we do not have a Rd field in the instruction word

Consequently, RegDst must be set to 0 to place Tx'sr x" i "Ti kwi vHpi wT{ "ehhvi ww" port.

Finally, in order to accomplish the register write, RegWr must be set to 1.

The Single Cycle Datapath during Load

$R[rt] \leftarrow \text{Data Memory} \{R[rs] + \text{SignExt}[\text{imm16}]\}$

+3 = 28 min. (Y:08)

The Single Cycle Datapath during Store

The store instruction performs the inverse function of the load. Instead of loading data from memory, the store instruction sends the contents of register specified by Rt to data memory.

Similar to the load instruction, the store instruction needs to read the contents of register Rs (points to Ra port) and add it to the sign extended version of the immediate field (Imm16, ExtOp = 1, ALUSrc = 1) to form the data memory address (ALUctr = add).

However unlike the Load instruction where busB is not used, the store instruction will use busB to send the data to the Data memory.

Consequently, the Rt field of the instruction has to be fed to the Rb port of the register file.

In order to write the Data Memory properly, the MemWr signal has to be set to 1.

Notice that the store instruction does not update the register file. Therefore, RegWr = 0. The store instruction cares.

And once again we need to set the control signals Branch and Jump to zero to ensure proper Program Counter updating.

Well, by now, you are probably tired of these boring stuff where Branch and Jump are zero. The store instruction.

+3 = 31 min. (Y:11)

The Single Cycle Datapath during Branch

So how does the branch instruction work?

As far as the main datapath is concerned, it needs to calculate the branch condition. That is, it subtracts the register specified in the Rt field from the register specified in the Rs field and set the condition Zero accordingly.

In order to place the register values on busA and busB, we need to feed the Rs and Rt fields of the instruction to the Ra and Rb ports of the register file and set ALUSrc to 0.

Then we have to instruct the ALU to perform the subtract (ALUctr = sub) operation and set the Zero bit accordingly.

The Zero bit is sent to the Instruction Fetch Unit. I will show you the internal of the Instruction Fetch Unit in a second.

But before we leave this slide, I want you to notice that ExtOp, MemtoReg, and RegDst are set to 0 to prevent any write to occur.

And finally, the controller needs to set the Branch signal to 1 so the Instruction Fetch Unit can fetch the next instruction.

+2 = 33 min. (Y:13)

Instruction Fetch Unit at the End of Branch

```

if (Zero == 1) then PC = PC + 4 + SignExt[imm16]*4 ;
else PC = PC + 4

```

Well, if Zero is not asserted, we will have our boring case where PC + 1 is selected.

Anyway, with Branch = 1 and Zero = 1, the output of the second adder will be selected.

That is, we will add the sequential address, that is output of the first adder, to the sign extended version of the immediate field, to form the branch target address (output of 2nd adder).

With the control signal Jump set to zero, this branch target address will be written into the Program Counter register (PC) at the end of the clock cycle.

+2 = 35 min. (Y:15)

Step 4: Given Datapath: RTL -> Control

A Summary of the Control Signals

Here is a table summarizing the control signal setting for the seven (add, sub, ...) instructions we have looked at.

Instead of showing you the exact bit values for the ALU control (ALUctr), I have used the symbolic values here.

The first two columns (add and sub) are unique in the sense that they are R-type instructions; and in order to uniquely identify them, we need to look at BOTH the op field as well as the func field.

Ori, lw, sw, and branch on equal are I-type instructions and Jump is J-type. They all can be uniquely identified by looking at the op- code field alone.

Ps{ "p x w" eoi "e" q sv "gevi jyp" sso "ex" i first two columns. Notice that they are identical except the last row.

So we can combine these two columns here if { i "ger" h i p} "x i "ki r i v exs" sj "CNWgx" signals.

This lead yw"s" vsq i x mk "gep h" sgep hi gshmk0

The Concept of Local Decoding

+3 = 45 min. (Y:25)

The local decoding concept is where instead of asking the Main Control to generate the ALUctr signals directly ; the main control will generate a set of signals called ALUop.

For all I and J type instructions, ALUop will tell the ALU Control exactly what the ALU needs to do (Add, Subtract, ...) .

But whenever the Main Control sees a R-type instructions, it simply throws its hands up and says: "Y s{ ." K hsr x' or s{ " { l ex' x i " CNW' l ew' xs" hs" f yx' K or s{ x' w' e" T -type mwygxsr "er h" p x' x i " Ngep' Esr xs p' Dsgo. " CNW' Esr xs p' Dsgo. " take care of the rest.

Notice that this save us one column from the table we had on x i " ew' w' p' h i " Dyx' p x w' f i " honest, if one column is the ONLY thing we save, we probably will not do it.

But when you have to design for the entire MIPS instruction set, this column will be used for ALL R-type instructions, which is more than just Add and Subtract I showed you here.

Another advantage of this table over the last one, besides being smaller, is that we can uniquely identify each column by looking at the Op field only.

Putting it All Together: A Single Cycle Processor

A Single Cycle Processor

OK, now that we have the Main Control implemented, we have everything we needed for the single cycle processor and here it is.

The Instruction Fetch Unit gives us the instruction. The OP field is fed to the Main Control for decode and the Func field is fed to the ALU Control for local decoding.

The Rt, Rs, Rd, and Imm16 fields of the instruction are fed to the data path.

Based on the OP field of the instruction, the Main Control will set the control signals RegDst, ALUSrc, ... etc properly

Furthermore, the ALUctr uses the ALUop from the Main control and the func field of the instruction to generate the ALUctr signals to ask the ALU to do the right thing

How Effectively are we utilizing our hardware?

Example: memory is used twice, at different times

- ✓ Average mem access per inst = $1 + Flw + Fsw \sim 1.3$
- ✓ If CPI is 4.8, imem utilization = $1/4.8$, dmem = $0.3/4.8$

We could reduce HW without hurting performanc extra control

Alternative datapath: Multiple Cycle Datapath

Immunizes Hardware: 1 memory, 1 adder

Putting it all together, here it is: the multiple cycle datapath we set out to built.
+1 = 47 min. (Y:47)

Controller FSM Spec

Sequencer-based control unit

Two Types of Exceptions

Interrupts

- ✓ caused by external events
- ✓ asynchronous to program execution
- ✓ may be handled between instructions
- ✓ simply suspend and resume user program

Traps

- ✓ caused by internal events
 - exceptional conditions (overflow)
 - errors (parity)
 - faults (non-resident page)
- ✓ synchronous to program execution
- ✓ condition must be remedied by the handler
- ✓ instruction may be retried or simulated and program continued or program may be aborted

Precise Interrupts

Precise => state of the machine is preserved as if program executed upto the offending instruction

- ✓ Same system code will work on different implementations of the architecture
- ✓ Position clearly established by IBM
- ✓ Difficult in the presence of pipelining, out-of-order execution, ...
- ✓ MIPS takes this position

Imprecise => system software has to figure out what is where and put it all back together

Performance goals often lead designers to forsake precise interrupts

- ✓ system software developers, user, markets etc. usually wish they had not done this

Summary of Today's Lecture

3-bus based single cycles data path

Control signals generation for single cycles data path

Lecture 9

Computer Hardware Design

(Multi Cycle and Pipeline - Datapath and Control Design)

] √pm ""] √..ω"

Recap: multi cycle datapath and control

Features of Multi cycle design

Multi Cycle Control Design

Introduction to Pipeline datapath

Summary

Recap: Lecture 8

Information flow and Control signals for single cycles data path to execute:

- ✓ Add/Subtract Instruction
- ✓ Immediate Instruction
- ✓ Load/Store Instructions
- ✓ Control Instructions

Analysis of single cycle data path

How effectively are different sections used?

- ✓ Memory is used twice, at different times
(i.e., Instruction Fetch and Load or Store)
- ✓ Adders in IF section are used once for fraction of time (Fetch Phase)
- ✓ ALU is used for the execution of R-type instructions and memory address calculation

Conclusion: We can reduce H/W without hurting performance by using extra control

Multiple Cycle Approach

The single cycle operations are performed in five steps:

- ✓ Instruction Fetch
- ✓ Instruction Decode and Register Read
- ✓ Execute (R- I-type or address for Load/store/Branch)
- ✓ Memory (Read/write)
- ✓ Write (to register file)

Multiple Cycle Approach

In the Single Cycle implementation, the cycle time is set to accommodate the longest instruction, the Load instruction.

In the Multiple Cycles implementation, the cycle time is set to accomplish longest step, the memory read/write

Consequently, the cycle time for the Single Cycle implementation can be five times longer than the multiple cycle implementation.

As an example, if $T = 5 \mu \text{ Sec.}$ for single cycle then $T = 1 \mu \text{ Sec.}$ for multi cycle implementation

Single Cycle vs. Multiple Cycle

For different classes of instructions, Multi Cycle implementation may take 3, 4 or 5 cycles to fetch and execute an instruction

Now in order to compare the performance of single cycle and multi cycle implementations, let us consider a program segment comprising three instructions, given in the sequence: Load, Store, R-type (say Add)

The execution time for these three instructions using single cycle implementation with cycle length equals $5 \mu \text{ Sec}$ is: $T_{\text{exe}} = 3 \times 5 \mu \text{ Sec} = 15 \mu \text{ sec.}$

Note that here the cycle time is long enough for the load instruction, but it is too long for the Store and R-type instruction

So the last part of the cycle, in case of the store and 4th (memory) part in case of R-type instruction is wasted.

In Multi cycle implementation, Load is completed in 5 Cycles, and store and R-type each takes 4 cycles to complete.

Thus, these three instructions take $5+4+4 = 13$ cycles, if the cycle length is $1 \mu \text{ Sec}$ then the execution time for the three instructions is: $T_{\text{exe}} = 13 \times 1 \mu \text{ Sec} = 13 \mu \text{ sec.}$

Conclusion: The multi cycle is $15/13 = 1.24$ times faster

High Level View of Multiple Cycle Datapath

Here, a shared memory is used, as the instruction fetch and data read/write are performed in different cycles

The single ALU is shared among the instruction fetch, execute arithmetic and logic instructions and address calculation in different cycles

The use of shared function unit (ALU) requires additional multiplexers or widening of multiplexers

New temporary registers, Instruction register, Data memory, operand A and B and ALUout, are included to hold the information for use in later cycle

E.g.; Memory read in cycle 4 is written in cycle 5 (Load), operand registers A and B read in cycle 2 may be used in cycle 3 or 4, and so on

Multiple Cycle Datapath Design

Multiple Cycle Datapath Architecture

Immunized Hardware: 1 memory, 1 adder

Cycle 1 - [Instruction Fetch]:

firstly, MUX-1 select input $lorD = 0$ and the PC is connected to the Memory Read address input RAdr; instruction is fetched from the memory at Dout and is placed in the Instruction Register by inserting IRWr [Yellow Path]

Secondly, the select input ALUSelA to MUX-3, is made equal to 0,, ALUSelB to MUX-5 is made equal to 00 to add 4 to PC; then PCSrc of MUX-2 is made 0 and PCWr is asserted to load PC+4 to the PC as address of the next instruction

Cycle 2 [ID and Reg. Rd.]

Firstly the Instruction is decoded; the Rs, Rt, Rd and Imm16 fields are made available on respective lines (Shown in orange)

Secondly the registers at Rs and Rt are read at buses A and B, respectively

Cycle 3 - [Exe]

The select inputs ALUSelA and ALUSelB to the MUX-3 and MUX-5, respectively for the instruction in hand; available at ALUop input to the ALU Control Unit

- ✓ For R-type instructions:
ALUSelA = 1 and ALUSelB = 01 to connect bus A and bus B to ALU to perform the operation [Green Path]
- ✓ For I-type and Memory Instructions:
ALUSelA = 1 and ALUSelB = 11 to connect bus A and Sign Extended Imm16 to ALU to perform the operation on immediate data [Red Path]. The ALU output is kept in ALU OUT Register as result of ALU OP execution in case of I-type operation and as Memory address in case of memory instructions Load/store
- ✓ For J- type Instructions:
1: Condition Test: ALUSelA = 1 and ALUSelB = 01; ALUop=SUB If ALU output Zero =1 then assert PCWrCond and
2: $PC \leftarrow PC+4+$ [Sign Extend Imm16 and Shift left 2 bits] ALUSelA = 0; ALUSelB = 10 Assert BrWr ; and PCSrc of MUX-2 = 1 to pass the target address to PC [Blue Path]

Cycle 4 - [Memory Instruction Load/Store]

- ✓ Load instruction: $lorD=1$ to pass the ALUout Register as RAdr (Read Address) input to the memory to read data at the Dout [Dark Green Path]
- ✓ Store instruction: MemWr is asserted; as the ALUout Register output is wired to WrAdr (Write address input) [Dark Green Path] and bus B of the register file is wired to Din (Data In) [Dark blue] of the memory

Cycle 5 - [Write Back]

- ✓ R-type instruction: RegDest of MUX-4 = 1 to select Rd as the destination address; MemToReg = 0 to connect ALUout to Bus-W and RegWr is asserted memory

- ✓ I-type instruction: RegDest of MUX-4 = 0 to select Rt as the destination address; MemToReg = 0 to connect ALUout to Bus-W and RegWr is asserted memory
- ✓ Load instruction: RegDest of MUX-4 = 0 to select Rt as the destination address; MemToReg = 1 to connect Dout of the memory to Bus-W or the register file and RegWr is asserted

Multi Cycle Control design

Control may be designed in the following steps using the initial representation as:

Finite State Machine: Here, the sequence control is defined by explicit next state functions, logic is represented by logic equations and usually PLAs are used to implement the machine

Micro-program: Here, micro-program counter and a dispatch ROM defines the sequence control, logic is represented by truth table and control is implemented using ROM

Multi Cycle Controller FSM Specifications

Micro program Controller

T no“\uz””“no’u/z ‘P’q“...‘q’mu/z

Designing a Microinstruction Set

1. Start with list of control signals
2. I vsyt "wkr epv"ski x i v'x ex'q eoi "wi r wi ""zwOver hsq +<gepi h" j rnpw
3. Places fields in some logical order (e.g., ALU operation & ALU operands first and microinstruction sequencing last)
4. Create a symbolic legend for the microinstruction format, showing name of field values and how they set the control signals
 - a. Use computers to design computers
5. To minimize the width, encode operations that will never be used at the same time

Microprogramming

Specialize state-diagrams easily captured by micro sequencer

- ✓ datapath control fields

Control design reduces to Microprogramming

Microprogramming is a fundamental concept

- ✓ implement an instruction set by building a very simple processor and interpreting the instructions
- ✓ essential for very complex instructions and when few register transfers are possible
- ✓ overkill when ISA matches datapath 1-1

Microprogramming: inspiration for RISC

If you could ever

If same memory used for control memory could be used instead as cache for

Then why not skip instruction interpretation by a micro-program and simply compile

directly into lowest language of machine? (microprogramming is overkill when ISA

matches datapath 1-1)

Summary

Single cycle verses multi cycle datapath

Key components of multi cycle data path

Design and information flow in multi cycle data path

Multi cycle control unit design

Finite State Machine based control Unit

Micro program- based controller

Lecture 10

Computer Hardware Design (Pipeline Datapath and Control Design)

Recap: Lecture 9

- Single cycle verses multi cycle datapath
- Key components of multi cycle data path
- Design and information flow in multi cycle data path
- Multi cycle control unit design
- Finite State Machine based control Unit
- Microprogram-based controller

What is pipelining?

Pipelining is a fundamental concept
It utilizes capabilities of the Datapath by

Pipelining is Natural!

Laundry Example!

Four loads: A, B, C, D

Four laundry operations:

Wash, Dry, fold and place into drawers

Washer takes 30 minutes "

Dryer takes 30 minutes "

Hand folding and placing into drawers "

Putting clothes into drawers "

Sequential Laundry

Pipelined Laundry: Start work ASAP

Pipelined laundry takes 3.5 hours for 4 loads!

Features of Pipelined Processor

All the functional units operate independently

Multiple tasks operating simultaneously using different resources

Reduces the time to complete a workload

Potential speedup = Number pipe stages

Pipelining Lessons

Pipeline rate limited by:

- ✓ Slowest pipeline stage
- ✓ Variations in the time to complete a stage
- ✓ Unbalanced lengths of pipe stages reduces speedup
- ✓ If washer takes longer time than the dryer then dryer has to wait!

Stall for Dependences

Five Steps of Datapath

1. Ins. Fetch,
2. 1Dec/Reg,
3. Exec,
4. Mem,
5. Wr

Pipelined Processor Design

Pipeline Control

WWW.VUMULTAN.COM

Pipelined Registers Included

Five Steps as Stages of Pipeline

WWW.VUMULTAN.COM

Multiple Cycle verses Pipeline Pipeline enhances performance

3 Instructions program reconsidered

1. Load
2. Store
3. R-type (ADD)

Example 1: The cycle time of a single cycle machine is 45 ns, and of multi cycle and pipelined machines is 10 ns; and average CPI due to instruction mix on multi cycle machine is 4.6.

What is the execution time on each type of machine?

Solution:

Single Cycle Machine

$$✓ \quad 45 \text{ ns/cycle} \times 1 \text{ CPI} \times 100 \text{ inst} = 4500 \text{ ns}$$

Multi Cycle Machine

$$✓ \quad 10 \text{ ns/cycle} \times 4.6 \text{ CPI} \times 100 \text{ inst} = 4600 \text{ ns}$$

Pipelined machine

$$✓ \quad 10 \text{ ns/cycle} \times (1 \text{ CPI} \times 100 \text{ inst} + 4 \text{ cycle drain}) = 1040 \text{ ns}$$

Example 2: Consider a multicycle, uniplined processor requires 4 cycles for the ALU and Branch operations and 5 cycles for the memory operation. Assume the relative frequency of these operations is 40%, 25% and 35% respectively; and the clock cycle is of 1 n sec.

In pipelined implementation, due to clock skew and setup processor adds 0.2 n sec. to the clock

Solution:

Unpipelined Processor:

Average Execution Time / Instruction = Clock Cycle x Average CPI

$$= 1 \text{ n sec.} \times [(0.4 + 0.25) \times 4 + 0.35 \times 5]$$

$$= 1 \text{ n sec} \times (0.65 \times 4 + 0.35 \times 5)$$

$$= 1 \text{ n sec.} \times (2.60 + 1.75) = 4.35 \text{ n sec}$$

Pipelined Processor:

Average Execution Time/ Instruction = Clock cycle + overhead

$$= 1 \text{ n sec.} + 0.2 \text{ n. sec}$$

$$= 1.2 \text{ n sec}$$

$$\text{Speed up} = 4.35 / 1.2 = 3.62 \text{ times}$$

Pipelined Execution Representation

Conventional Representation

- ✓ Helps showing the program flow viz-a-viz time

Graphical Representation

Why Pipeline? Because the resources are there!

Can pipelining get us into trouble?

Structural hazards

- ✓ Data hazards
- ✓ Control hazards

How Stall degrades the performance?

The pipelined CPI with stalls =

- ✓ $\text{Ideal CPI} + \text{Stall clock cycles per instruction}$

How Stall degrades the performance?

1. Speedup w.r.t uniplined
$$= \text{CPI Uniplined} / 1 + \text{stall cycles per instruction}$$
2. Speedup w.r.t. pipeline depth
$$= \text{pipeline depth} / 1 + \text{stall cycles per instruction}$$

Summary

Multi cycle datapath verses pipeline datapath
Key components of pipeline data path
Performance enhancement due to pipeline
Hazards in pipelined datapath

Lecture 11

Computer Hardware Design (Pipeline and Instruction Level Parallelism)

Recap Lecture 10

- Structural Hazards
- Data Hazards
- Control Hazards

Recap: Lecture 10

- Multi cycle datapath verses pipeline datapath
- Key components of pipeline data path
- Performance enhancement due to pipeline
- Introduction to hazards in pipelined datapath

Structural Hazards

Attempt to use the same resource two different ways at the same time, e.g.,
Single memory port is accessed for instruction fetch and data read in the same clock cycle would be a structural hazard

Single Memory is a Structural Hazard

Two memory read operations in the 4th cycle:
The LOAD instruction accesses memory to read data and the 4th instruction fetched from the same memory

Insert stall (bubble) to avoid memory structural hazard

Structural hazard exists when

Single write port of register accessed for two WB operations in same clock cycle

This situation does not exist in 5-stage pipeline

But it may exist in 4 and 5 stage multi-cycle pipeline

Pipelining the Load Instruction

The five independent functional units in the pipeline datapath are: Inst. Fetch, Dec/Reg.

Th. "CNW]sv'G|i g."Fex"Oi q "er h"Ti kwi v'Hip w'Y vxi "t sv'X i "Y v'weki

Hevi . "{ i "l ezi "wi t evxi "vi kwi v w'vi eh"er h" { vxi "t sw'v'vi kwi w'vi eh"er h" { vxi "w' allowed at the same time

Each functional unit is used once

Explanation:

For the load instructions, the five independent functional units in the pipeline datapath are:

- Instruction Memory for the Ifetch stage.
- Ti kwi v'Hip w'Ti eh't sw'jsv'X i "Ti k Fi gshi "weki 0

- c) ALU for the Exec stage.
- d) Data memory for the Mem stage.
- e) Cr h"jmepp "Ti kwwi v"Hrp w{ vni "t sw"jsv"X i "Y vniDego"weki 0

Psxgi "X ex"Kl ezi "xi ex"Ti kwwi v"Hrp w"viad and write ports as separate functional units because the register file we have allows us to read and write at the same time.

Notice that as soon as the 1st load finishes its Ifetch stage, it no longer needs the Instruction Memory. Consequently, the 2nd load can start using the Instruction Memory (2nd Ifetch).

Furthermore, since each functional unit is only used ONCE per instruction, we will not have any conflict down the pipeline (Exec-Ifet, Mem-Exec, Wr-Mem) either.

I will show you the interaction between instructions in the pipelined datapath later. But for now, I want to point out the performance advantages of pipelining.

If these 3 load instructions are to be executed by the multiple cycle processor, it will take 15 cycles. But with pipelining, it only takes 7 cycles. This (7 cycles), however, is not the best way to look at the performance advantages of pipelining.

A better way to look at this is that we have one instruction enters the pipeline every cycle so we will have one instruction coming out of the pipeline (Wr stages) every cycle.

Esr w i uyi r x p . "X i " i jji g x zi " "s v' ezi vek i + "r y q f i v s j" g i w t i v' m w y g x s r "w r s { "Q P G" even though it takes a total of 5 cycles to complete each instruction.

+3 = 14 min. (X:54)

The Four Stages of R-type

R-type instruction does not access data memory, so it only takes 4 clocks, or say 4 stages to complete

Here, the ALU is used to operate on the register operands

The result is written in to the register during WB stage

Explanation:

Y i p p "w s" j e v "w s" k s s h 0 N x w a k e a look at the R-type instructions.

The R-type instruction does NOT access data memory so it only takes four clock cycles, or in our new pipeline terminology, four stages to complete.

The Ifetch and Reg/Dec stages are identical to the Load instructions. Well they have to be because at this point, we do not know we have a R-type instruction yet.

Instead of calculating the effective address during the Exec stage, the R-type instruction will use the ALU to operate on the register operands.

The result of this ALU operation is written back to the register file during the Wb back stage.

+1 = 15 min. (55)

Pipelining the R-type and Load Instruction

We have pipeline conflict or structural hazard:

- ✓ Two instructions try to write to the register file at the same time!
- ✓ Only one write port

What happened if we try to pipeline the R-type instructions with the Load instructions?

Well, we have a problem here!!!

We end up having two instructions trying to write to the register file at the same time!

Why do { i | ezi "x w't vsf p q "x i "{ vxi " f yf f p +A

Well, the reason for this problem is that there is something I have not yet told you.

+1 = 16 min. (X:56)

Important Observation

Each functional unit can only be used once per instruction

Each functional unit must be used at the same stage for all instructions:

- ✓ Nseh"ywi w'Ti kwwi v'Hrp w'Y vxi "Rsw'hyvink"nw7x 'xeki
- ✓ R-x'ti "ywi w'Ti kwwi v'Hrp w'Y vxi "Rsw'hyvink"nw6x 'xeki

I already told you that in order for pipeline to work perfectly, each functional unit can ONLY be used once per instruction.

What I have not told you is that this (1st bullet) is a necessary but NOT sufficient condition for pipeline to work.

The other condition to prevent pipeline hiccup is that each functional unit must be used at the same stage for all instructions.

Hsv'i | eq t p "l i v i . "x i "seh"nwgyxsr "ywi w'x i "Ti kwwi v'Hrp w'Y v't sw'hyvink"nw7x " stage but the R-x'ti "nwgyxsr vki x'r s { "{ nplywi "x i "Ti kwwi v'Hrp w't sw'hyvink"nw6x " stage.

This (5 versus 4) is what caused our problem. How do we solve it? We have 2 solutions.

[√xπ'u/z'5A2"q'°" I πmnxq 'lz'√"t q'Wu.qxz q

Kwi w'e" f yf f p "mns"x i "t rhi pni "s"t vi zi r x'4"fiw'exx i "weq i "g} gp

- ✓ The control logic can be complex.
- ✓ Lose instruction fetch and issue opportunity.

No instruction is started in Cycle 6!

Vi i "jmw'wspyxsr "m"x"mwi w'e" f yf f p "mns"x i rhi pni "CHVGT"x i "seh"mwygxsr "s" push back every instruction after the load that are already in the pipeline by one cycle.

At the same time, the bubble will delay the Instruction Fetch of the instruction that is about to enter the pipeline by one cycle.

Needless to say, the control logic to accomplish this can be complex.

Furthermore, this solution also has a negative impact on performance.

Psxgi "x ex'hyi "s"x i " i | xe "weki "Oiq +Nseh"mwygsr "l ew"i "i { npsx'l ezi "sri " instruction finishes every cycle (points to Cycle 5).

Consequently, a mix of load and R-type instruction will NOT have an average CPI of 1 because in effect, the Load instruction has an effective CPI of 2.

Us"x mw'r sx'l ex'l s'er "rhi e""N x w'x} "wsq i x mk"pi 0

+2 = 19 min. (X:59)

Solution 2: Delay R-' ..q "'c "ùq'h 'Vzq'J oxq

Delay R-xt i w'i kwi v{ vni "f } "sri "g} gp <

- ✓ Now R-xt i "mwygxsr w'eps"ywi "Ti k"Hp w{ vns port at Stage 5
- ✓ Mem stage is a NO-OP stage: nothing is being done.

Yip sri "x mk" { i "ger "hs" w "s" ehh "e" Pst "w eki "s" x T " -type instruction pipeline to delay its register file write by one cycle.

Now the R-type instruction ALSO uses thi "i k n w i v " j n p w " { n i v t s w " e x " n w 7 x " w eki " w s " { l' eliminate the write conflict with the load instruction.

This is a much simpler solution as far as the control logic is concerned. As far as performance is concerned, we also gets back to having one instruction completes per cycle.

This is kind of like promoting socialism: by making each individual R-type instruction takes 5 cycles instead of 4 cycles to finish, our overall performance is actually better off.

The reason for this higher performance is that we end up having a more efficient pipeline.

Eliminating Structural Hazards

Structural hazards can be eliminated or minimized by either using the stall operation or adding multiple functional units

Example: Dual-port vs. Single-port

Machine A: Dual ported memory

Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate

Ideal CPI = 1 for both

Loads are 40% of instructions executed

$\text{SpeedUp}_A = \text{Pipeline Depth} / (1 + 0) \times (\text{clock}_{\text{unpipe}} / \text{clock}_{\text{pipe}}) = \text{Pipeline Depth}$

$\begin{aligned} \text{SpeedUp}_B &= \text{Pipeline Depth} / (1 + 0.4 \times 1) \times (\text{clock}_{\text{unpipe}} / (\text{clock}_{\text{unpipe}} / 1.05)) \\ &= (\text{Pipeline Depth} / 1.4) \times 1.05 \\ &= 0.75 \times \text{Pipeline Depth} \end{aligned}$

$\text{SpeedUp}_A / \text{SpeedUp}_B = \text{Pipeline Depth} / (0.75 \times \text{Pipeline Depth}) = 1.33$

Machine A is 1.33 times faster

Stall degrades the performance

Here, is an example: Suppose data reference instructions constitute 40% of mix, and processor with structural hazard has clock rate 1.05 times higher than the processor without hazard

$\begin{aligned} \text{The Average Instruction time} &= \text{CPI} \times \text{Clock Cycle Time} \\ &= (1 + 0.4 \times 1) \times \text{clock cycle time}_{\text{Ideal}} / 1.05 \\ &= 1.4 / 1.05 \times \text{clock cycle time}_{\text{Ideal}} \\ &= 1.3 \times \text{clock cycle time}_{\text{Ideal}} \end{aligned}$

The processor without structural hazard is 1.3 times faster than with Structural hazard

Additional Functional Units increase cost

Memory structural hazard is removed by using two Cache memory units:

- ✓ Instruction memory
- ✓ Data Memory

Two write ports in register file allow 4-stage and 5-stage pipe mix

Data Hazards

Attempt to use item before it is ready; e.g.,

Qr i "wsgo"sj"t em"m hv}i v'er h"sri "m">{ ew i v"ger x"jsph"y xdk i x"wsgo"jvsq "{ ew i v' through dryer

Instruction depends on result of prior instruction still in the pipeline

Pipelining changes the relative timing of instruction by overlapping their execution

This overlap introduces the Data and Control Hazard

Data Hazard occurs when order of operand read/write is changed viz-z-viz sequential access to the operands, which gives rise to data dependency

Example Data Hazard on R1

Add R1, R2, R3

Sub R4, R1, R3

And R6, R1, R7

Or R8, R1, R9

Xor R10, R1, R11

Data Hazard due to Dependencies backwards in time are hazards

Data Hazard Solution #1 Stall

WWW.VUMULTAN.COM

XOR: No Data Hazard here, as register is read after being written

Data Hazard Solution Forwarding

WWW.VUMULTAN.COM

Forwarding (or Bypassing): What about Loads?

Dependencies backwards in time are hazards

Must delay/stall instruction dependent on loads

Control Hazards

Control hazard occurs when one attempt to take action before condition is evaluated

When Branch instructions is executed it may or may not change the PC to something other than PC+4

Branch Taken: Branch changes the PC+4 to new target

Branch Not Taken: Branch does not change the PC+4

The simple way to deal with the branch is:

Freeze the pipeline holding any instruction after the branch instruction and

Flush the pipeline to delete the instructions after the branch if condition is evaluated, and branch is to take

Example BEQ Taken

Explanation Branch Hazard

Here, If the BEQ is taken then

The next instruction address is determined after evaluating the branch condition in the EX stage;

But the next instruction (LOAD) is fetched in the ID stage, i.e., before the PC+4 is changed to new target address

This gives rise to Branch or Control Hazard

Dealing with Branches

Stall

Redo Fetch after branch

Delayed branching

Branch prediction

Multiple Streams

Solution#1 Stall

Result of the condition evaluation is available after the EXE stage and the target address is available in the next stage

Thus 3 stall cycles

Reducing number of Stall

Reducing number of Stall

Here, you can see that if we move up decision to the end of ID stage (2nd stage) by adding hardware to compare the registers being read. The number of stalls reduces to 2 clock cycles per branch instruction

It can further be reduced to 1 in case of BEQZ or BNEZ if zero register is tested after Instruction Fetch

Solution# 2 Redo Fetch after Branch

We know that once a branch has been detected during the Instruction decode /Register read stage, the next instruction fetch cycle should essentially be a stall, if we assume that branch is taken

However, the instruction fetched in this cycle never performs useful work, and is ignored. Therefore, re-fetch the Branch successor instruction will provide the correct instruction.

Indeed, the second fetch is not essential if branch is not taken

Impact: 1 clock cycle per branch instruction if branch is un-taken

Solution# 3 Delayed Branch S/W method

Redefine branch behavior to take place after the next instruction by introducing other instruction (may be No-OP) which is always executed

Impact: 2nd instruction is always executed (50% of time)

Solution#4 Prediction

This technique suggests that for a branch instruction we guess one direction of the branch, to begin, then back up if wrong

The two possible predictions are:

- ✓ Predict Branch not-taken
- ✓ Predict branch taken

Branch Prediction Flowchart

1. Predict Branch not taken

This scheme is implemented assuming every branch as branch Not-taken
So the processor continues to fetch branch as normal instructions

Sequence when branch is not-taken

Dver gl "K w" m	IF	ID	EX	MEM	WB				
Inst0 m3		IF	ID	EX	MEM	WB			
K w0 m4			IF	ID	EX	MEM	WB		
K w0 m4				IF	ID	EX	MEM	WB	

We the decision has been made, and the branch is taken, then fetch operations are turned into NO-OP and fetch is restarted at the target address

Sequence when branch is taken

Veoi r "Dver gl "K w" m	IF	ID	EX	MEM	WB				
K w0 m3		IF	Idle	Idle	Idle	Idle			
Branch target			IF	ID	EX	MEM	WB		
Dver gl "xvki x" 3				IF	ID	EX	MEM	WB	

2. Predict - Branch taken

An alternative way is to treat every branch as Branch taken

As soon as the target address is computed, we assume that the branch is to be taken and start fetching and executing at the target

In a five stage pipeline the target address and condition evaluation are available at the same time, so this technique is of no use.

Let us consider this example of a LOOP to explain the concept:

```

i = 2" -----
Loop: |
      0-----|
      i = i+1 |
      IF i "3223"VJ GP"Nsst cccc•
  
```

J i v i . "x i "f ver gl "w"veoi r "jsv"3222"xq i . "vs"x i "t v i h r x s r " Dver gl "Veoi r "jerp"3"m"3222." hence no stall for 1000 times

Further, the compiler can improve performance by organizing the code so that the most frequent path matches the hardware choice

Solution #5 Multiple Streams

Have two pipelines

Pre-fetch each branch into a separate pipeline

Use appropriate pipeline

Results

Leads to bus & register contention
 Multiple branches lead to further pipelines being needed
 Target of branch is pre-fetched in addition to instructions following branch
 Keep target until branch is executed
 Used by IBM 360/91

Summary

Type of hazards in pipelined datapath
 Structural hazards occur when same resource is accessed by more than one instructions
 One memory port or one register write port
 It can be removed by using either multiple resources or inserting stall
 Stall degrades the pipeline performance
 Data Hazards occur when attempt is made to read invalid data
 Data hazard can be removed by using stall and forwarding techniques
 Control hazards occur when an attempt is made to branch prior to the evaluation of the condition

Four ways to handle control hazards

1. Stall until branch direction is clear
2. Predict Branch Not Taken
 - Execute successor instructions in sequence
 - PC+4 already calculated, so use it to get next instruction
3. Predict Branch Taken
4. Delayed Branch
 - Define branch to take place AFTER a following instruction
 - 1 slot delay allows proper decision and branch target address in 5 stage pipeline

Lecture 12

Instruction Level Parallelism

(Introduction to multi cycle pipelined datapath)

] √pm ""] √..ω"

Recap: Pipelining Basics
 Longer Pipelines FP Instructions
 Loop Level Parallelism
 FP Loop Hazards
 Summary

Recap: Pipelined datapath and control

In the previous lecture we reviewed the pipelined datapath to understand the basics of ILP overlap among the instruction execution to enhance performance

Key components of pipeline data path

Performance enhancement due to pipeline:

- ✓ Pipelining helps instruction bandwidth but not latency

Pipeline Hazards

- ✓ Structural hazards
- ✓ Data Hazards

Three Generic Data Hazards

Read After Write (RAW): (dependence)

- ✓ $instr_j$ tries to read operand before $instr_i$ writes it;
- i: add r1,r2,r3
 j: sub r4,r1,r3

Write After Read (WAR): anti-dependence

- ✓ Also called Name dependence (renaming)
- i: sub r4,r1,r3
 j: add r1,r2,r3

Write After Write (WAW)

i: sub r1,r4,r3
 j: add r1,r2,r3

Pipeline Hazards

Control hazards

Stall

How to overcome Hazards?

How to remove Hazards?

Structural Hazard:

Multiple functional units

Data Hazard

Forwarding or bypassing

Control Hazards:

Predict, delay branch

Instruction Level Parallelism

clock speed

number of instructions that can execute in parallel, i.e., increasing ILP

How to achieve Instruction Level Parallelism?

A superscalar processor:

- ✓ Pre-fetch and decode
- ✓ Start several branch instruction streams
- ✓ Finally, discard all but the correct stream

MIPS Longer Pipelines FP Instructions

For example to ADD two FP minimum four steps are performed in the following sequence:

Flow diagram of MIPS FP Adder

Draw flow diagram of pp284

Steps for FP Addition

Step 1: Exponents of two numbers are compared, the smaller number is shifted to the right to till its exponent matches to the larger exponent

Step 2: Add the significands

Step 3: Normalize the sum shift right and increment or shift left and decrement

Step 4: If no overflow or underflow then round the significand to number of bits

Stop if further normalization is not required, otherwise go to step 3

Latency of Functional Unit:

The latency of functional unit is defined as: the number of cycles between the instructions that produces a result and the one that uses the result of the operation

The initiation or repeat interval is defined as: the number of cycles that must elapse between issuing two operations (repeat of an operation) of the same type

	Latency	Initiation (repeat) Interval
Integer ALU	= 0	1
Data Memory (Int / FP Load)	= 1	1
FP ADD	= 3	1
FP/ Integer Multiply	= 6	1
FP/Integer Divide	= 24	25

Typical MIPS FP Pipeline

Let us consider a typical MIPS FP pipeline with three un-pipelined FP functional units

MIPS FP Pipeline with Pipelined FUs

The previous FP pipeline can be extended by adding additional pipeline stages in the functional units

Working of extended FP Pipeline

Note that additional pipeline register have been inserted between intervening stage, e.g., C3C4."C4C5." Ø

Furthermore, ID/EX register must be expanded to connect ID to A1, M1, EX and DIV Function Units

Here, the FP divide FP is not pipelined but it requires 24 clock cycles to complete

Hazards in Longer Latency Pipeline

All the functional units are not fully pipelined. So structural hazard may occur.

Instructions have varying running time, so more than one register write may occur.

Instructions are no longer reaching WB stage in order so WAW data hazard may occur.

WAR hazards are not possible since registers are read in ID stage.

Stall for RAW data hazard may be more frequent because of longer latency of operations.

FP Pipeline Hazards RAW

Clock Cycle Number

INST	1	2	3	4	5	6	7	8	9
L.D F4, 0(R2)	IF	ID	EX	Me	WB				
MUL.D F0,F4,F6		IF	ID	st	M1	M2	M3	M4	M5
ADD.D F2,F0,F8			IF	st	ID	st	st	st	st
S.DF2, 0(R2)					IF	st	st	st	st

INST	10	11	12	13	14	15	16	17	
L.D F4, 0(R2)									
MUL.D F0,F4,F6	M6	M7	Me	WB					
ADD.D F2,F0,F8	st	st	A1	A2	A3	A4	Me	WB	
S.DF2, 0(R2)	st	st	ID	st	st	st	EX	Me	

FP Pipeline Structural Hazard

Clock Cycle Number

INST	1	2	3	4	5	6	7	8	9	10	11
MUL.D F0,F4,F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	Me	WB
Ø		IF	ID	Ex	Me	WB					
			IF	ID	Ex	Me	WB				
ADD.D F2,F0,F8				IF	ID	A1	A2	A3	A4	Me	WB
					IF	ID	Ex	Me	WB		
						IF	ID	Ex	Me	WB	
L.D F4, 0(R2)							IF	ID	EX	Me	WB

Conclusion about FP Pipeline

1. Structural Hazard wait until required functional unit is available
2. Check for RAW data hazard : wait until the source registers are not listed as pending destinations register that will not be available
3. El i go"jsv"Y CY <hi x q mi "ijter}"mwygxr "m"C3."C4." F".O3."O4"." 0I ew"veq i " destination as this instruction

Precise Exceptions: Out-of-order Completion!

In the program:

DIV.D F0,F2,F4

ADD.D F10,F10,F8

SUB.D F12,F12,F14

Overcoming the Data Hazard by Scheduling

Static Scheduling Compiler based

Dynamic Scheduling Hardware based

Dynamic Scheduling (Overcoming the Data Hazard)

Advantages:

Allows to handle cases where dependence is unknown at the compile time

Allows code compiled for one pipeline to run on other pipe line

In the program:

DIV.D F0,F2,F4

ADD.D F10,F0,F8

SUB.D F12,F8,F14

Problems of Out-of-order execution WAR and WAW

In the program:

DIV.D F0,F2,F4

ADD.D F6,F0,F8

SUB.D F8,F10,F14

MUL.D F6,F10,F8

Exception due to out of order execution

Already completed instructions

Not Yet completed instructions

Overcoming Exceptions

Split the ID pipe stage into two:

Issue:

- ✓ Decode instructions and check for structural hazard

Read Operand:

- ✓ Wait until no data hazards, then read

Summary: We have talked about longer FP pipelines

Lecture 13

Instruction Level Parallelism (Dynamic Scheduling - Scoreboard Approach)

Today's Topics

Recap - Lecture 11-12
 Out-of-Order Execution
 Problems of Out-of-order execution
 Dynamic Scheduling
 Scoreboard Technique
 Summary

Recap: Lecture 12

FP and Integer Multiplier

FP and Integer Divider

Here, we observed that :

- ✓ Only one instruction is issued on every clock cycle
- ✓ The integer ADD instructions go through the FP pipeline as they go through in standard pipeline as the integer ALU operations have ZERO latency
- ✓ The FP add and FP/integer multiply and divide instructions enter into loop when they reach EX-stage due to longer latencies of these operations thus increases the number of stalls before the instruction is issued to EX stage

RAW and WAR hazards may occur because the instructions are of varying length and may reach WB out-of-order

There are different ways to RAW hazard:

WAW hazard

- ✓ (The j^{th} instruction writes prior to the i^{th} instruction; the i^{th} instruction overwrites the result of j^{th} instruction)

Two ways to resolve WAW hazard

- ✓ Delay the issue of j^{th} instruction until the i^{th} instruction enters the MEM stage
- ✓ Stamp out the i^{th} instruction by detecting the hazard and changing the control (WB) so that the i^{th} instruction does not write.
- ✓ Hence, the j^{th} instruction can be issued right-away.

In-Order Execution

Simple Pipelined datapath facilitates only the In-order instruction execution, i.e., Instructions are fetched, decoded and issued in the sequence of the program and no later instruction can proceed if an instruction is stalled due to hazard structural or data dependence

For example: in the code

DIV.D F0, F2, F4

ADD.D F10, F0, F8

SUB.D F12, F8, F14

In-order Execution: MIPS 5-stage Pipeline

The MIPS 5-stage pipeline, both the structural and data hazards are checked during the Instruction Decode (ID) stage; and

The instruction is issued from ID stage, if it could execute properly

Here, the issue process, at ID stage, is separated into two parts:

Checking the structural hazard

Waiting for the absence of data hazard

Out-of-order Execution: MIPS 5-stage pipeline

DIV.D F0, F2, F4

ADD.D F10, F0, F8

SUB.D F12, F8, F14

Basic Problems of Out-of-order Execution

Consider the example FP code

DIV.D F0, F2, F4

ADD.D F6, F0, F8

SUB.D F8, F10, F14

MUL.D F6, F10, F8

Scheduling for out-of-order execution

Static Scheduling:

- ✓ Rearrangement of the instruction execution by the compiler

Dynamic Scheduling:

- ✓ Rearrangement of the instruction execution by the hardware

Dynamic Scheduling

Issue:

Read Operand:

Dynamic Scheduling: Score boarding Technique

CDC 6600 contains:

- ✓ 4 FP units
- ✓ 5 Memory Reference Units
- ✓ 7 integer operation units

MIPS Processor with Scoreboard

Features of Scoreboard

- ✓ The Scoreboard:

Components of Scoreboard

- ✓ Instruction Status
- ✓ Functional Unit Status
- ✓ Register Result Status

Instruction Status

These four stages are:

Issue: If a functional unit for instruction is free and no other active instruction has the same destination register, the score board issues the instruction to the functional unit and updates the internal data structure. Thus guarantees that WAW cannot be present.

Read Operand: The score board monitors availability of the source operand, i.e., checks if no earlier issued active instruction is going to write. Thus, it resolves RAW hazard.

Execute: The FU begins the execution and notify the scoreboard when it has completed the execution. The scoreboard then updates the data structure.

Write Result: Once the score board is aware that the FU has completed execution then checks for the WAR hazard, it stalls if necessary and writes the result.

Instruction Status Data structure

Instruction	Issue	Read	Execution	Write	Operands	complete	result
MUL.D							
ADD.D		"					

Functional Unit Status

Busy: A single bit field which indicates if the FU is busy or not

OP: 2 or 3 bit field specifying the operation being performed by FU (ADD or SUB etc.)

Registers: Fi Destination register Number

Fj, Fk Source registers number

Qj and Qk: VI i "HW" CFF."OWN" 0"t vshygnk"wsyvgi "i kwi v Fj and Fk

Rj, Rk: Flags indicating source registers Rj, Rk are ready and not yet read Set to NO when operand are read

Typical Functional Unit Status table

FU name	Busy	OP	Fi	Fj	Fk	Qj	Qk	Rj	Rk
MUL1	Y	Mul	F0	F2	F4	--	--	No	No
0									
DIVIDE	Y	Div	F10	F0	F6	MUL1	--	No	Yes

Register Result Status

Format of the Table:

F0	F2	F4	F6	F8	F10	H34"	"H52
FU	Mul1	Add			Divide		

Detailed Scoreboard Pipeline Control

Scoreboard Example

FP Add Latency = 2 clock, Multiply = 10, Divide = 40

Scoreboard Example 1

WWW.VUMULTAN.COM

Scoreboard Example Cycle 3

Issue MULT?

Note: Scoreboard Example Cycle 3

Issue MULT? No, stall on structural hazard

Scoreboard Example Cycle 4

Scoreboard Example Cycle 5

WWW.VUMULTAN.COM

Scoreboard Example Cycle 6

Scoreboard Example Cycle 7

Read multiply operands?

Scoreboard Example Cycle 8a
(First half clock cycle)

Scoreboard Example Cycle 8b
(Second half clock cycle)

WWW.VUMULTAN.COM

Scoreboard Example Cycle 9

Read Operands for MULT & SUBD? Issue ADDD?

Scoreboard Example Cycle 11

WWW.VUMULTAN.COM

Scoreboard Example Cycle 12

Read operands for DIVD?

Scoreboard Example Cycle 13

WWW.VUMULTAN.COM

Scoreboard Example Cycle 14

Scoreboard Example Cycle 15

WWW.VUMULTAN.COM

Scoreboard Example Cycle 16

Scoreboard Example Cycle 17

Write result of ADDD?

Scoreboard Example Cycle 18

Scoreboard Example Cycle 19

Scoreboard Example Cycle 20

Scoreboard Example Cycle 21

WWW.VUMULTAN.COM

Scoreboard Example Cycle 22

Scoreboard Example Cycle 61

Scoreboard Example Cycle 62

Review: Scoreboard Example Cycle 62

In-order issue; out-of-order execute & commit

Summary

- Problems of Out-of-order execution
- Dynamic Scheduling
- Scoreboard Technique
- Summary

Lecture 14

Instruction Level Parallelism

(Dynamic Scheduling)

Today's Topics

Recap - Lecture 13
 Dynamic Scheduling
 Vsq ewyps wCt t vsegl
 Summary

Recap: Summary

Instruction Level Parallelism in Hardware or Software
 SW parallelism dependencies defined by program result in hazards if HW cannot resolve
 HW exploiting ILP works when dependence cannot be determined at run time
 Key idea of Scoreboard
 Allow instructions behind stall to proceed
 It is accomplished by dividing the ID stage into two parts
 Issue the instruction in-order
 Read operand out-of-order
 Structural and data dependencies are checked at ID stage
 It facilitates out-of-order execution which results in out-of-order completion

Review: Three Parts of the Scoreboard

1. Instruction status which of 4 steps the instruction is in
2. Functional unit status indicates the state of the functional unit(FU). 9 fields for each functional unit
 - a. Busy Indicates whether the unit is busy or not
 - b. Op Operation to perform in the unit (e.g., + or -)
 - c. Fi Destination register
 - d. Fj, Fk Source-register numbers
 - e. Qj, Qk Functional units producing source registers Fj, Fk
 - f. Rj, Rk Flags indicating when Fj, Fk are ready
3. Register result status Indicates which functional unit will write each register, if one exists. Blank when no pending instructions will write that register.

Review: Scoreboard Example Cycle 3

Issue MULT? No, stall in structural hazard

Review: Scoreboard Example Cycle 9

Read operands for MULT & SUBD? Issue ADDD?

Review: Scoreboard Example Cycle 17

Write result of ADD? No, WAR hazard

Review: Scoreboard Example Cycle 62

In-order issue; out-of-order execute & commit

Review: Scoreboard Summary

Speedup 1.7 from compiler; 2.5 by hand

- ✓ But slow memory (no cache)

Limitations of 6600 scoreboard

- ✓ No forwarding(First write register then read it)
- ✓ Limited to instructions in basic block(small window)
- ✓ Number of functional units(structural hazards)
- ✓ Wait for WAR hazards
- ✓ Prevent WAW hazards

Another Dynamic Scheduling Approach: Tomasulo's Algorithm

Introduced by Tomasulo for IBM 360/91 about 3 years after CDC 6600 (1966)

Goal: High Performance without special compilers

Tomasulo's Algorithm Vs. Scoreboard

Differences between IBM 360 & CDC 6600 ISA is:

- ✓ IBM has only 2 register specifiers / instr vs. 3 in CDC 6600
- ✓ IBM has 4 FP registers vs. 8 in CDC 6600

Tomasulo's Organization For Dynamic Scheduling

Components of Tomasulo's Structure

FP Operation Queue: Instruction are sent from instruction unit into the instruction Queue in FIFO order

FP Adder Reservation Station

FP Multiplier Reservation Station

The reservation stations include

- ✓ Operations, actual operands and information to resolve hazards

Load Buffers have three functions:

- ✓ Hold components of effective address until it is computed
- ✓ Track outstanding Loads waiting on memory
- ✓ Hold the result of completed load waiting for CDB

Store Buffers also have three functions:

- ✓ Hold components of effective address until it is computed
- ✓ Hold the destination memory address of outstanding store instructions
- ✓ Hold the address and value of store until the memory unit is available

Common Data Bus: The difference between CDB and Normal bus is

Normal data bus: VI i "hex"er h"hi w"mexsr "*" ks"x "f yw+

Common data bus: VI i "hex"- "wsyvgi "*" gsq i "jvsq "f yw+

- ✓ Does the broadcast
- ✓ 64 bits of data + 4 bits of Functional Unit source address

Sequence of operations

All the results from the FP units or the Load unit are placed on the Common Data Bus, which goes to the FP register file as well as to the RS and store buffers

Tomasulo's Algorithm Vs. Scoreboard

Control & buffers distributed with Function Units (FU) in Tomasulo vs. centralized in scoreboard

FU buffers called vi wi vzexsr "wexsr w "l ezi "t i r hmk"st i ver hw

Registers in instructions are replaced by values or pointers to reservation stations(RS)

This is called register renaming

- ✓ avoids WAR, WAW hazards

Osvi "vi wi vzexsr "wexsr w"l er "i kwi w"ws"ger "hs"st xq m exsr w"l I rgl "gsq t n w"ger x

Tomasulo: Results to FU from RS, over Common Data Bus that broadcasts results to all FUs

Scoreboard: Result to FU through registers

Components of Reservation Station

Op Operation to perform in the unit *i k0"ehh."wyf ." +

Busy Indicates reservation station or FU is busy

Vj, Vk Value of Source operands

- ✓ Store buffers has V field to store the result

Qj, Qk Reservation stations producing source registers value to be written

Note: No ready flags as in Scoreboard; Qj,Qk=0 => ready

Qi Store buffers only have Qi for RS producing result

Register result status indicates which functional unit will write each register, if one exists.

- ✓ Blank when no pending instructions that will write that register.

] t "qq['nsq""r] √y m'nx/ "'Hs√"ut y

1. Issue get instruction from FP Op Queue
 - ✓ If reservation station is free, i.e., no structural hazard then control issues instruction and send operands to the RS (renames registers).
2. Execution operate on operands (EX)
 - ✓ When both operands ready then execute; if not ready, watch Common Data Bus for result
3. Write result finish execution (WB)
 - ✓ Write on Common Data Bus to all awaiting units;
 - ✓ mark reservation station available

Tomasulo Example Cycle 0

Tomasulo Example Cycle 1

Tomasulo Example Cycle 2

Note: Unlike 6600, can have multiple loads outstanding

Tomasulo Example Cycle 3

vs. scoreboards

Load1 completing: What is waiting for Load1?

Tomasulo Example Cycle 4

Load2 completing: what is waiting for it?

Tomasulo Example Cycle 5

Tomasulo Example Cycle 6

Issue ADDD here vs. scoreboard?

Tomasulo Example Cycle 7

Add1 completing; what is waiting for it?

Tomasulo Example Cycle 9

Tomasulo Example Cycle 10

Add2 completing; what is waiting for it?

Tomasulo Example Cycle 11

Write result of ADDD here vs. scoreboard?

Tomasulo Example Cycle 12

Note: all quick instructions complete already

Tomasulo Example Cycle 13

Tomasulo Example Cycle 14

WWW.VUMULTAN.COM

Tomasulo Example Cycle 14

Mult1 completing; what is waiting for it?

Tomasulo Example Cycle 16

Note: Just waiting for divide

Tomasulo Example Cycle 55

Tomasulo Example Cycle 56

Mult 2 completing; what is waiting for it?

Tomasulo Example Cycle 57

Again, in-order issue, out-of-order execution, completion

Compare to Scoreboard Cycle 62

Why takes longer on scoreboard/6600?

Another Example: Loop Unrolling

Let us consider another example to understand WAW and WAR are eliminated by register renaming

Here, is an example of multiplying an array (F0) by a scalar F2

Another Example: Loop Unrolling

Here, we predict the branches are taken, thus the RSs allow multiple executions of the loop by dynamic unrolling

The unrolling can be shown by two active iterations, based on prediction branch taken

Loop Example Cycle 0

Loop Example Cycle 1

Loop Example Cycle 2

Loop Example Cycle 3

Note: MULT1 has no registers names in RS

Loop Example Cycle 4

Loop Example Cycle 5

WWW.VUMULTAN.COM

Loop Example Cycle 6

Note: F0 never sees Load1 result

Loop Example Cycle 7

Note: MULT2 has no registers names in RS

Loop Example Cycle 8

Loop Example Cycle 9

Load1 completing; what is waiting for it?

Loop Example Cycle 10

Load2 completing; what is waiting for it?

Loop Example Cycle 11

WWW.VUMULTAN.COM

Loop Example Cycle 12

Loop Example Cycle 13

Loop Example Cycle 14

MULT1 completing; what is waiting for it?

Loop Example Cycle 15

MULT2 completing; what is waiting for it?

Loop Example Cycle 16

Loop Example Cycle 17

Loop Example Cycle 18

Loop Example Cycle 19

Loop Example Cycle 20

Loop Example Cycle 21

WWW.VUMULTAN.COM

WWW.VUMULTAN.COM

Lecture 15

Instruction Level Parallelism (Dynamic Branch Prediction)

Today's Topics

- Recap - Lecture 14
- Dynamic Branch Prediction
- Branch Prediction Buffer
- Examples of Branch Predictor
- Summary

Recap: Lecture 14

Tomasulo 's Approach for IBM 360/91 to achieve high Performance without special compilers

Here, the control and buffers are distributed with Function Units (FU)

Registers in instructions are replaced by values or pointers to reservation stations(RS) ; i.e., the registers are renamed

Unlike Scoreboard, Tomasulo can have multiple loads outstanding

These two properties allow to issue an instruction having name dependence ; e.g., MULT is issued which has name dependence of register F2

Tomasulo eliminates the WAR hazard as in this example ADD.D writes the result in Cycle 11 even if the DIV.D will start execution in Cycle 16

Tomasulo issues in-order and may execute out-of-order

Here, the integer instructions SUBI and BNEZ are executed out-of-order to evaluate the condition

The perdition Branch-Taken is implemented by repeating the loop instruction as shown

WWW.VUMULTAN.COM

Introduction to Dynamic Branch Prediction

In the last lecture, we considered a loop-few h"i | eq t p ."x" hmgw" x i "Vsq ewys w' approach to overcome the WAW and WAR hazards

Here, we observed that dynamically scheduled pipeline can yield high performance provided branches are predicted accurately

Branch History Table

If the prediction is wrong, then invert prediction-bit

1-bit Dynamic Branch Prediction

Problem:

- ✓ In a loop, 1-bit BHT will cause two mispredictions in a row
- ✓ 1-bit predictor mispredict at twice the rate that the branch is not-taken
- ✓ Let us consider an example of loop-branch (For i=1 to 10); i.e., the branch is taken 9 times and not-taken once

As the Performance = $\frac{\text{Actual Branches}}{\text{Predicted Branches}}$

The accuracy of the predictor is expected to match the taken-branch frequency, which in the previous example is 9 out of 10 (90%)

But the 1-bit prediction has 8 out of 10 (80%)

2-bit Dynamic Branch Prediction

2 bits are used to encode 4-states in the system (counter) Say:

- ✓ States 00 and 01 for Predict Not-Taken
- ✓ States 10 and 11 for Predict Taken

In a saturating counter implementation: 2-bit counter saturates at:

- ✓ 00 (Predict Taken) or
- ✓ 11 (Predict Not taken)

The counter is incremented when a branch is taken and decremented when it is not taken; e.g.,

- ✓ 00 to 01 for Taken when predicted not taken
- ✓ 10 to 11 for Taken when predicted taken

Here, when the counter is greater than or equal to $\frac{1}{2}$ of its maximum value (≥ 10 ; i.e., state 01 and 11) branch is predicted as taken;

Otherwise (i.e., < 10 : state 10 and 00) the branch is predicted as untaken

Let us try the example of loop For $i=1,10$

Let us try the example of loop For $i=1,10$

Iteration	P.S.	Branch	NS	Prediction
0	--	not Taken	11	Taken
1	11	Taken	11	Taken
2	11	Taken	11	Taken
:				
9	11	Taken	11	Taken
10	11	Not taken	10	Taken

Prediction fails once only

Branch Prediction Buffer (BPB) or BHT Implementation

If Prediction is wrong

Then prediction bits are changed

In case

Predicted Taken:

State changes 11 $\rightarrow$ 10)

Predicted not taken:

State changes 00 $\rightarrow$ 01

Branch History Table Accuracy

For example Place Fig. 3.8 pp 200 here

Here, for SPEC89 benchmark

A branch prediction buffer with 4096 entries results in:

- ✓ Prediction accuracy ranging from: 99% to 82 % or
- ✓ Mispredictions rate of 1% - 18%

Branch History Table Accuracy wrt size

Insert Fig. 3.9 pp 201

Impact of size on accuracy of BHT

As we try to exploit more ILP, the accuracy of the Branch Predictor becomes critical
Here, the accuracy of the predictor is shown by increasing the size of the buffer as

- ✓ 4096 Entries 2-bit BHT
- ✓ Unlimited Entries 2-bit BHT

Simply increasing the number of bits per predictor without changing the predictor structure has little impact so we have to look at other methods to increase the accuracy of the predictors

Correlating Branches

The 2-bit predictor scheme uses only the recent behavior of the single branch to predict the future behavior of branch

In practice, the behavior of other branches, rather than only a single branch, we are trying to predict, may also influence the prediction accuracy

Let us consider the worst case of SPEC92 benchmark for 2-bit predictor

SPEC92 benchmark for 2-bit predictor example:

Assume aa is assigned R1 and bb the register R2

```
IF (aa==2)          DSUBUI R3, R1, #2
aa=0;               BNEZ  R3, L1          ; branch b1 (aa!=2)
                   DADD   R1, R0, R0      ; aa=0 Not Branch

IF (bb==2)   L1     DSUBUI R3, R2, #2
bb=0;       BNEZ  R3, L2          ; branch b2 (bb!=2)
                   DADD   R2, R0, R0      ; bb=0 Not Branch

IF (aa!=bb)   L2     DSUBU  R3, R1,R2
{             BEQZ   R3, L3          ; branch b3 (aa=bb)
```

Here, the behavior of b3 (L2) is correlated with the behavior of b1 and b2

Here, if b1 and b2 are both not-taken (aa=0; bb=0) then b3 is taken

A predictor that uses the behavior of a single branch to predict the behavior of that branch cannot capture this behavior

So we need a correlating branch predictor

Correlating Branch Predictors

Hypothesis: Recent branches are correlated; that is, behavior of recently executed branches affects prediction of current branch

In general, (m,n) predictor means record last m branches to select between 2^m history tables each with n-bit counters

- ✓ Old 2-bit BHT is then a (0,2) predictor

Example

Let us consider an illustrative code:

(d is assigned to R1)

```
IF (d==0)      BNEZ   R1, L1      ; branch b1 (d!=0)
                d=1;      DADDIU R1,R0,#1    ; branch not taken, d=1
IF (d==1)      L1:   DADDIU R3, R1, #-1
                BNEZ   R3, L2      ; branch b2 (d!=1)
```

The working of correlating predictor is as follows

Initial d	d==0?	b1	d before b2	d==1?	b2
0	yes	NT	1	yes	NT
1	No	T	1	yes	NT
2	No	T	2	no	T

Here, if b1 is not taken b2 will not be taken

We write the pair of prediction bits as: Prediction if last branch in the program is not-taken/

Prediction if the last branch is taken

Therefore, the 4 possible combinations are:

Prediction bits	New Prediction if last branch Not Taken	New Prediction if last Branch Taken
NT / NT	NT	NT
NT/T	NT	T
T/NT	T	NT
T/T	T	T

The action of the 1-bit predictor with 1-bit of correlation, written as (1,1) for the above

In this case the only misprediction is on the first iteration, when d=2 as this is not correlated with the previous prediction

Correlating Branches

(2,2) branch prediction buffer uses 2-bit global history to choose from among 4 predictors for each branch address

Then behavior of recent branches selects between, say, four predictions of next branch, updating just that prediction

Accuracy of Different Schemes

Branch History Table or Branch Target Buffer

Dynamic Branch Prediction Summary

Branch History Table: 2 bits for loop accuracy

Correlation: Recently executed branches correlated with next branch

Branch Target Buffer: include branch address & prediction

Predicated Execution can reduce number of branches, number of mispredicted branches

Lecture 16

Instruction Level Parallelism

,K zmy ω'l 'mz ot 'W'qpω'u/z" 2J√z' p-

Today's Topics

Recap
 Correlating Branch Predictors
 Tournament Predictor
 High Performance Instruction Delivery Branch Target Buffer
 Summary

Recap: Dynamic Scheduling and Branch Prediction

Static: rely on the software (compiler)
 Dynamic: hardware intensive approaches

Important questions: Branch-Prediction Buffer

Q1: What is the impact of increasing the size of branch-prediction buffer on two branches in a program?

A single predictor predicting a single branch is generally more accurate than is that same predictor serving more than one instructions; and
 It is less likely that two branches in a program share a single predictor
 Therefore, increasing the size of predictor buffer does not have significant effect on two branches in a program

Q2: How sharing a predictor effects the misprediction rate?

This is explained with the help of following example: Consider two sequences of branch-taken and not-taken , sharing 1-bit predictor; and identify the sequence that

- ✓ Reduces the misprediction rate
- ✓ Increases the misprediction rate

Example: Sequence 1

Here, the columns B1 and B2 show the branches B1 and B2
 B1 is always TAKEN
 B2 is always Not-TAKEN

=====

Example: Sequence 2

=====

Example: Conclusion

Why sharing of predictor increases misprediction rate?

It is clear from the above example that:

if a predictor is shared by a set of branch instructions

then the members of the set of branch instruction may change, over the course of execution of long program

Hence, the branch action history changes and predictor is likely to mispredict more often

Correlating Branch Predictors Re-visited

We have observed that in program segment

IF (d==0) Branch b1 for d!=0

 d=1;

IF (d==1) Branch b2 for b!=1

 d=2;

This problem may be resolved in Correlating-Branch Predictor by recording m most recently executed branches as taken or not taken (in 2^m branch-history tables for 1-, 2-, "s"r -bit predictor), and using branch-pattern to select the proper branch history table for the current branch

In general, (m, n) predictor means record last m branches to select between 2^m history tables each with n-bit counters (2^m n-bit predictor)

A 2-bit BHT is regarded as (0,2) correlating predictor;

Example

1-bit predictor with 1-bit correlation is written as (1,1) predictor

Here, we have two (2^1) separate prediction bits (i.e., two 1-bit BHTs)

- ✓ One prediction bit is used if the last branch executed was not-taken
 - ✓ Other prediction bit is used if the last branch executed was taken
- And is denoted as: (New prediction when last NT / New prediction when last T)
- ✓ E.g., T/NT stands for: New prediction is TAKEN if previous was NOT-TAKEN and is NOT-TAKEN if previous was TAKEN

In an (m,n) predictor, the global history of most recent m branches is recorder in an m-bit shift register

Here, each bit records whether the branch was taken or not taken

The branch-prediction buffers is indexed using concatenation of low-order bits from branch-address with m-bit global history

(2, 2) Correlating Branches Predictor

Here, the buffer is drawn as 2-dimensional object, each buffer is 2 bit wide, in reality they are arranged linearly

(2, 2) branch prediction buffer uses 2-bit global history to choose from among 4 predictors, for each branch address of 4-bit (among the 16 entries in each of the 4 predictors)

Behavior of recent branches selects between, say, four predictions of next branch, updating just that prediction

Indexing is done by concatenation of 4 lower-order address bits of the branch (word address) and 2 global bits to form 6-bit address to select 2-bit prediction from 64 entries in 4 buffers each of 16 entries

Comparison of (0,2) and (2,2) predictors

Multilevel Branch Predictors (Tournament Predictors)

Multilevel branch prediction (Nested Branches) involve information at local and global levels to predict correctly

Several levels of Branch-Prediction Tables and

An algorithm to choose among different predictors

State Transition Diagram of Tournament Predictor

Modify as Fig. 3.16 pp 204

Multilevel Branch Predictors (Tournament Predictors)

The transition for predicted predictor is specified by:

- ✓ Correct = 1
- ✓ Incorrect = 0

The state transition diagram shows that from the saturating state

- ✓ The counter is incremented whenever predicted predictor is correct and other is incorrect (i.e., for 1/0) and
- ✓ The counter is decremented in the reverse direction (i.e., for 0/1)

The counter does not change for all other predictions for non-saturating present state

For the saturating state 00 (Use predictor 1)

- ✓ It increments to the state 01 (use predictor 1) for 1/0
- ✓ and decrements to state 11 (use predictor 2) for 0/1

For the saturating state 11 (Use predictor 2)

- ✓ It increments to the state 00 (use predictor 1) for 1/0
- ✓ and decrements to state 10 (use predictor 2) for 0/1

High Performance Instruction Delivery

In MIPS 5-stage pipeline, we need to know the address of the next-instruction-fetch at the end of current IF cycle

That is, for ZERO branch penalty, we need to know whether the as-yet un-decoded instruction is branch; and

if yes then what is the next-PC?

Branch Target Buffer

This is accomplished by introducing a Cache that contains the address of the next instruction if branch is taken as well as not-taken

This cache is known as the Branch-Target Cache or Branch-Target Buffer (BTB)

Branch-Prediction Buffer vs. Branch-Target Buffer

Recall from our discussion last time that branch-prediction buffer is accessed during the ID stage, after the instruction decode, i.e.,

We know the branch-target address at the end of ID stage to fetch the next predicted instruction

Branch Target Buffer

Branch Target Buffer has three fields:

- ✓ Lookup: addresses of the known branch instructions (predicted as taken)
- ✓ Predicted PC: PC of the fetched instruction predicted taken-branch
- ✓ Prediction State: Optional- extra prediction state bits

Complications?

- ✓ Complication arise in using 2-bit predictor because it uses information for both the branches taken and not-taken
- ✓ This complication is resolved in PowerPC processors by using both the Target-buffer and Prediction-buffer

Steps involved in using Branch Target Buffer at IF, ID and EXE pipeline stages

- i. IF
- ii. ID
- iii. EXE
- iv. (insert flow chart of Fig. 3.20)

Branch-Target Buffer Flow Chart Explanation

IF Stage: The PC of an instruction is compared with the contents of the buffer

if it is found

then the instruction must be a branch instruction predicted taken

Else It may be a branch predicted not-taken or normal instruction

ID Stage

i. Decode the instruction and

If in the IF Stage, entry was found in the Target-buffer as predicted-branch

then begin fetching immediately from the predicted PC

ii. Check the decoded instruction

If it is Taken-branch

EX Stage performs one of the four possible functions

- i. Where in the IF stage entry was not found in the target buffer and in the ID stage
 - If it is found to be Taken-branch
 - (i-a) then Enter branch-instruction address and next PC into branch-target buffer
 - (i-b) Else Proceed as normal instruction execution
- ii. Where in the IF stage the entry was found in the target-buffer and in the ID stage
 - If it is found to be Taken-branch
 - (ii-a) then correctly predicted , so execute normally without stall
 - (ii-b) Else it is mispredicted, so kill the fetched instruction, restart fetching at an other address and delete entry from the target-buffer

If the correctly predicted branch entry is found in the buffer

Then there will be no branch penalty

Else It suffers at least 2 clock cycle delay as misprediction penalty

- ✓ One clock delay for fetching the wrong instruction and
- ✓ One clock cycle to restart the fetch

Branch-Target Buffer Examples

Inst. in Buffer	Prediction	Actual Branch	Penalty Cycles
Yes	Taken	Taken	0
Yes	Taken	Not-Taken	2
No	-	Taken	2
No	-	Not Taken	0

Branch-Target Buffer Solution

We can compute the penalty by looking at the probability of two events:

- i) Branch predicted taken but end up not take

$$= \% \text{ buffer hit rate} \times \% \text{ incorrect prediction}$$

$$= 0.95 \times 0.1 = 0.095$$
- ii) Branch is taken but is not found in the buffer

$$= \% \text{ incorrect prediction}$$

$$= 0.1$$

The penalty in both the cases is 2 cycles, therefore

$$\text{Branch Penalty} = (0.095 + 0.1) \times 2 = 0.195 \times 2 = 0.39$$

Example: Branch-Target Buffer

Problem:

Consider a branch-target buffer implemented for conditional branches only for pipelined processor

Assuming that:

- ✓ Misprediction penalty = 4 cycles
- ✓ Buffer miss-penalty = 3cycles
- ✓ Hit rate and accuracy each = 90%
- ✓ Branch Frequency = 15%

Solution

The speedup with Branch Target Buffer verses no BTB is expressed as:

$$\text{Speedup} = \text{CPI}_{\text{no BTB}} / \text{CPI}_{\text{BTB}}$$

$$= (\text{CPI}_{\text{base}} + \text{Stalls}_{\text{no BTB}}) / (\text{CPI}_{\text{base}} + \text{Stalls}_{\text{BTB}})$$

The stalls are determined as:

$$\text{Stalls}_{\text{stall}} = \text{Frequency}_{\text{s}} \times \text{Penalty}_{\text{s}}$$

The sum over all the stall cases as the product of frequency of the stall cases and the stall-penalty

- i. $\text{Stalls}_{\text{no BTB}} = 0.15 \times 2 = 0.30$
- ii. To find $\text{Stalls}_{\text{BTB}}$ we have to consider each output from BTB

There exist three possibilities:

- a) Branch misses the BTB:
 $\text{frequency} = 15\% \times 0.1 = 1.5\% = 0.015$
 $\text{Penalty} = 3$
 $\text{Stalls} = 0.045$
- b) Branch can hit and correctly predicted:
 $\text{frequency} = 15\% \times 0.9 (\text{hit}) \times 0.9 (\text{prediction}) = 12.1\% = 0.121$
 $\text{Penalty} = 0$
 $\text{Stalls} = 0$
- c) Branch can hit but incorrectly predicted:
 $\text{frequency} = 15\% \times 0.9 (\text{hit}) \times 0.1 (\text{misprediction}) = 1.3\% = 0.013$
 $\text{Penalty} = 4$
 $\text{Stalls} = 0.052$

$$\text{Stalls}_{\text{BTB}} = 0.045 + 0 + 0.052 = 0.097$$

$$\text{Speedup} = (\text{CPI}_{\text{base}} + \text{Stalls}_{\text{no BTB}}) / (\text{CPI}_{\text{base}} + \text{Stalls}_{\text{BTB}})$$

$$= (1.0 + 0.3) / (1.0 + 0.097)$$

$$= 1.2$$

Improvement in BTB

In order to achieve more instruction delivery, one possible variation in the Branch Target Buffer is:

To store one or more target instructions, instead of or in addition to, the predicted Target Address

Advantages:

- ✓ It possibly allows larger BTB as it permits access to take longer than the time between successive instruction fetches
- ✓ Buffering the actual Target-Instructions allow Branch Folding, i.e.,
 - ZERO cycle Unconditional Branching or some times ZERO Cycle conditional Branching

Lecture 17

Instruction Level Parallelism

(High-performance Instructions delivery - Multiple Issue)

Reducing branch penalties for High-Performance Processors

Branch Target Buffer
Integrated Instruction Fetch Units
Return Address Predictors

Integrated Instruction Fetch Units

Integrated Branch Prediction

- ✓ The Branch-predictor is included in the Instruction Fetch Unit
- ✓ So, it predicts and drive the fetch-pipe

Instruction Prefetch

- ✓ An instruction pre-fetch queue is part of IIFU
- ✓ The queue holds multiple instructions and deliver more than one instructions in one cycle

Instruction memory access and buffering

- ✓ Fetching multiple instructions per clock cycle may require accessing multiple cache lines, which is a complex operation
- ✓ IIFU facilitates to overcome these complexities and hides the cost of crossing cache-blocks
- ✓ IIFU also provides instruction buffering and on-demand issue

Return Address Predictors

The Return-Address predictor predicts the indirect jumps, i.e., the jumps whose address varies at run time

High-level language programs generate such jumps for indirect procedure calls and select or case statements

Summary: Minimizing Control Hazard Penalties

Multiple Instruction-Issue Processors

All of the schemes described so far can at best achieve 1 instruction/cycle

There exist two variations to these schemes:

- ✓ Superscalar processors
- ✓ Very Long Instruction Word (VLIW) processors

Superscalar

The statically scheduled processors use in-order execution

The dynamically scheduled use out-of-order execution

Superscalar concept has been used in:

- ✓ IBM Power2
- ✓ Sun Ultra SPARC
- ✓ Pentium III/4
- ✓ DEC Alpha
- ✓ HP 8000

Very Long Instruction Words VLIW processor

r1 = L r4	r2 = Add r1,M	f1 = Mul f1,f2	r5 = Add r5,4
-----------	---------------	----------------	---------------

VLIW Processors

VLIW includes new features for:

- ✓ predication,
- ✓ rotating registers and
- ✓ speculations, etc.

Typical implementations are:

- ✓ i860, Trimedia, Itanium

We will talk about statically scheduled superscalar today and about compiling for VLIW/EPIC later

Statically Scheduled Superscalar Processor

Instruction Issue Process:

- ✓ The multiple instruction issue is a complex process
- ✓ During instruction fetch, the pipeline receives all the instruction that could potentially issue, called Issue-packet (it may have say from 1 to 4 instructions)

Example: Statically Scheduled Superscalar MIPS Processor

As an example let us consider a MIPS superscalar that has:

- ✓ Number of Instructions issue/clock:
 - 2 instructions - 1 FP operation, 1 Integer operations
 - (The integer operations include Load/store to integer or FP register, branch and Integer ALU operation)

Issuing two instructions per cycle would require Fetch and Decode 64-bits/clock cycle

Fetching two instructions need careful handling of the cache, as either the first instruction may be at end of the cache block or the second instruction may be at the beginning of the cache block

Hazard detection: The restriction of one FP and one Integer makes the hazard checking simple.

We simply have to determine the likelihood of hazards between two instructions in an issue-packet

If this situation exist then the Simple solution is to treat this as a structural hazard (issue only 1 of them)

However, the only difficulties arise when Integer Instruction is a FP load/store/move instruction

- ✓ it may create contention of the FP port and create RAW hazard when second instruction of the pair depends on the first

Example

www.vumultan.com

Fetch Two instructions from the cache

Determine whether 0, 1 or 2 instructions can issue

Issue them to the correct functional unit

Example superscalar pipeline in operation

Let us see how the instructions look like when they go in pair in a pipe

Dynamic Scheduling in Superscalar Processors

1. **2-issue superscalar pipeline**
 - ✓ Here, we do not want to issue instruction to reservation station out of order, as this may lead to the violation of program semantics.
 - ✓ Further, to gain full advantage of Dynamic scheduling remove the constraints of issuing one FP and integer instruction in a clock.
2. **Alternatively, Separate the data structure of FP and integer registers and simultaneously issue both instructions to their respective reservation stations, as long as two issued instructions do not access same registers.**
 - ✓ One approach is: to run this step (assigning a reservation station and update control) in half a clock cycle, so that two instructions can be processed in one clock cycle.
 - ✓ Second approach is: to build logic necessary to handle two instructions at once, including any dependence between the instructions.
 - ✓ Modern superscalar processors issue four or more instructions per clock cycle
 - ✓ Often included both approaches. In addition it is speculated that the Branch prediction is integrated into a dynamically scheduled pipeline. This referred to as Hardware-based speculation

Example

Let us consider a most general 2-issue dynamically scheduled processor and see how a simple loop, which we considered for single-issue Tomasulo, executes on this processor

Recall that our example loop adds a scalar in F2 to each element of a vector in memory

```

Loop: L.D      F0,0(R1)      ; F0=array element
      ADD.D    F4,F0,F2      ; add scalar in F2
      S.D      F4,0(R1)      ; store result
      DADDUI   R1,R1,#-8      ; decrement pointer
                                ; 8 bytes (per DW)
      BNE      R1,R2,LOOP     ; branch R1!= R2
  
```

Let us create a table showing when each instruction issues, begins execution, and write its result to CDB for first three iterations using 2-issue processor using single issue processor

Assume that Both FP and integer operation can be issued on every clock cycle, even if they are dependent

One integer functional unit is used for both ALU operations and effective address calculations and a separate pipeline FP functional unit for each operation type
Issue and write result take one cycle each.

There is dynamic branch prediction hardware and a separate functional unit to evaluate branch conditions

There is one clock for integer ALU, two cycles for load, and three cycles for FP add.

Let us have a look on the clock cycle of issue, execution, and writing result for a dual issue processor

Thus, sustaining one iteration every three cycles would lead to an IPC of $5/3=1.67$ (5 instructions in 3 clocks)

completion rate is: $15/16=0.94$

✓ 15 instructions execute in 16 cycles

Resource usage table

Another example: Overcoming the single integer pipe bottleneck

Now let us consider another example with 2-issue version of the Tomasulo's pipeline to overcome single-integer unit pipe bottleneck

Example 2

In this example, we consider the execution of the same loop, as used in the previous example, but using 2-issue version of the Tomasulo's pipeline to overcome single-integer unit pipe bottleneck. The processor has wider CDBs (2 CDBs).

Similar to the previous example, the activities table, similar to the previous table, shows the clock cycles of issue, execution and writing result for the dual-issue version of the Tomasulo's pipeline.

Notice that dual-issue Tomasulo pipe has:

- ✓ separate functional units for Integer ALU and effective address calculation; and
- ✓ wider CDB

Activity Table for Example 2

Lecture 18

Instruction Level Parallelism

(Hardware-based speculations and exceptions)

Today's Topics

Hardware-based Speculations
 Speculating on the outcome of branches
 Handling Exceptions
 Summary

Recap: Lecture 17

Last time we discussed three basic concepts to accomplish multiple instructions issue:

Branch Target Buffer

- ✓ Branch Target-buffer provides the target branch address at the IF stage
- ✓ Its variation, branch folding, buffers the actual target-instruction instead of or along with target address
- ✓ Both facilitate to minimize branch-hazard stalls allowing multiple instruction issue in one clock cycle

Integrated Instruction Fetch Units

- ✓ Integrated Instruction Fetch Unit (IIFU) integrates the following three functions into a single step
 - Branch Prediction
 - Instruction Prefetch
 - Instruction memory access and buffering

Return Address Predictors

- ✓ Is one that predicts the indirect jumps, i.e., the jumps for indirect procedure calls and select or case statements

Then we discussed the features of:

Superscalar processors

VLIW processors

In the superscalar pipeline processors the multiple instructions issued in one clock cycle can be scheduled using both the static as well as dynamic scheduling techniques

Whereas, the VLIW-based processors schedule multiple instruction issues in one clock cycle using only the static scheduling approaches

Then we discussed the performance enhancement and factors limiting the performance in superscalar pipes

- ✓ statically scheduled and dynamically scheduled

Loop-based example

Last time, in the loop-based example, we observed that the control hazards, which prevent us from starting the next iteration before we know whether the branch was correctly predicted or not, causes one-cycle penalty, on every loop iteration

Today we will focus on the hardware-based speculation to address this limitation

Hardware-based Speculation: Introduction

Hardware-based speculation offers many advantages

- ✓ Can incorporate hardware-based branch prediction
- ✓ Does not require additional bookkeeping code
- ✓ Does not depend on a compiler

This approach has been implemented in the :

- ✓ PowerPC 620
- ✓ MIPS R10000
- ✓ Intel P6, and
- ✓ AMD K5

Hardware Based Speculation: Basics

We have observed that exploiting more instruction level parallelism, increases the burden of maintaining control dependence

Where, the branch prediction reduces the direct stall attributable to branches, a multiple-issue processor may need to execute a branch every clock cycle to maintain maximum performance

Hence, exploiting more parallelism requires that we must overcome the limitations of control dependence

These limitations are overcome by the speculation on the outcome of branches and executing the program for speculations

Here, we:

- ✓ Fetch, Issue and
- ✓ Execute instructions

as if our branch predictions were always correct.

We know that dynamic scheduling without speculation fetches and issues but does not execute such instructions until prediction is checked and found correct

Hardware Support: Speculative Execution

Main idea: allow execution of an instruction dependent on a predicted-taken branch such that there are no consequences (including exceptions such as memory violation) if branch is not actually taken

Hyw i v."{ i "hsr X{ er X'e"vt i gypexzi "mwygxsr "s"geyiwi | gi t xsr w'x ex'wst "t vskveq w' (i.e. memory violation)

This can be achieved: If hardware support for speculation buffers the results and exceptions from instructions, until it is known that the instruction would execute

Hardware Based Speculation: Basics

This shows that: Hardware based speculation combines three key ideas:

- ✓ Dynamic Branch Prediction
 - Dynamic branch prediction facilitates to choose which instruction to execute; i.e., next in sequence or branch
- ✓ Speculation
 - Speculate to allow the execution of the instructions before the control dependence is resolved. Here, the hardware has the ability to undo the instructions hard to do if there are exceptions

- ✓ Dynamic scheduling
 - Dynamic scheduling to deal with the scheduling of different combinations of basic blocks

Thus, the hardware based speculation follows the predicted flow of data values to choose when to execute

To do so, we must separate the bypassing of results among instructions, which (i.e., bypassing) is needed to execute an instruction speculatively, from the actual completion of an instruction

By making this separation we can allow an instruction:

- ✓ to execute and
- ✓ to bypass its result to other instructions without allowing the instruction to perform any update that cannot be undone, until we know that the instruction is no longer speculative

When the instruction is no longer speculative, we allow it to update the register file or memory

This additional step in the instruction execution sequence is called instruction commit

This shows that the basic idea behind implementing the speculation is to allow instructions to execute out-of- order but force them to commit in-order

Hardware Based Speculation: Implementation

In a single issue five stage pipeline: we can ensure that instructions are committed in-order, simply by moving writes to the end of the pipeline. Because when we add speculation, we need to separate the process of completing execution and instruction-commit, as the instructions may finish execution considerably before they are ready to commit

Adding this commit phase to the instruction execution sequence requires some changes to the sequence as well as an additional set of hardware buffers that holds the result of instructions that have finished execution but have not committed

Modified hardware including ROB

Here, the reorder buffer can be operand source, if value not yet committed
Once operand commits, result is found in register file

Mechanism

At issue time, allocate an entry in the ROB to hold result

As each value has a location in the ROB, therefore, use ROB entry number instead of reservation station to rename

However, we can use additional registers for renaming, and ROB only for tracking commits

Instruction results commit to register set in- order

If ROB is implemented as a queue then it is simple to Undo speculated instructions on mispredicted branches or on

exceptions just requires throwing away uncommitted entries

L 'qzpqp'] √y m'πx/ "'Wu.q

Exceptions are not recognized until an instruction becomes ready to commit

VI i "jkyvi "w s{ w'x i "Vsq ewyjs w'l evh{ evi "wygyvi "mghmk"x i "TQD

Here, the basic structure of a MIPS FP unit, using Vsq ewyjs w'epxvni q "w'i | x i r hi h"x" handle speculation.

The mechanism may be further extended to multiple issue by making CDB wider to allow for multiple completions per clock.

Here, the reorder buffer(ROB) provides additional buffer, same way as in reservation wexsr "m"Vsq ewyjs w'"x ex'i | x i r h"x i "vi kmi v'wi x0

The ROB holds the result of an instruction between the time operation associated with the instruction completes and the time instruction commit.

Hence, ROB is a source of operands for instructions just an in reservation station

t vszrh i h"st i ver hw"m"Vs q ewy s w'eksvn q 0

K "Vs q ewy s w'et t vsegl ."sr gi "er "mwygsr "{ vni w'nvi wpx"er}"wyf wi uyi r xj "mwyi h" instructions will find the result in the register file.

Whereas, in speculation the register file is not updated until the instruction commits

Thus the ROB supplies operands in the interval between completion of instruction execution and instruction commit.

VI i "TQD"m'wq ne v's"l i "wsvi "f yjji v'm"l i "Vs q ewy s w'eksvn q 0

ROB consists of four fields,

- ✓ instruction type field
- ✓ destination field
- ✓ the value field
- ✓ the ready field

Reorder Buffer Fields

1. Instruction Type field: It indicates whether:
 - ✓ The instruction is a branch and has no destination,
 - ✓ The instruction is a store, which has a memory address destination) , or
 - ✓ The instruction is a register operation, ALU operation or load, which has register destinations.
2. Destination field: It supplies
 - ✓ the register number (for load and ALU operation) or
 - ✓ the memory address (for stores) where the instruction result should be written.
3. Value field
 - ✓ It is used to hold the value of the instruction result until the instruction commits.
4. Ready field
 - ✓ It indicates that the instruction has completed execution and the value is ready.

[..qomxiu q'] y n'tnx/ "'Hxs'fthm

VI i vi "evi "Hsyv'Ui t w'sj"Ut i gypexzi "Vs q ewy s w'eksvn q 0

1. Issue
 - ✓ Get instruction from the head of the instruction queue
 - ✓ If reservation station and ROB slot free, Then allocate and issue instruction
 - ✓ If not free then stall issue
 - ✓ If operands are available then send them to the reservation station
 - ✓ Else keep track of ROB entry that will produce the operands
2. Execute
 - ✓ Operate on operands (EX)
 - ✓ If both operands ready then execute
 - ✓ If not ready, the watch CDB for result
 - ✓ This checks for RAW hazards
 - ✓ Instructions may take multiple clock cycles here

3. Write result

- ✓ Finish execution (WB)
- ✓ Write on CDB, mark reservation station available
- ✓ Result picked up by ROB entry
- ✓ If the value to be stored is available, then it is written to the value field of the ROB entry for the store.
- ✓ If the value to be stored is not available yet, then the CDB must be monitored until that value is broadcast,
- ✓ At which time the value field of the ROB entry of the store is updated.

4. Commit

- ✓ Commit can occur when an instruction reaches the head of the ROB and its result is present in the buffer.
- ✓ Commit update register or store to memory with ROB result and free up ROB slot
- ✓ If ROB head is an incorrectly predicted branch, then flush ROB
- ✓ If the branch was correctly predicted, then the branch is finished

Example 1

Using the same code segment, as we considered explaining in the Tomasulo's approach, earlier show that what the status table look like when the MUL.D is ready to go to commit.

Assume the same latencies as earlier

- ✓ add is 2 clock cycles,
- ✓ multiply is 10 clock cycles, and
- ✓ divide is 40 clock cycles.

Code

- ✓ L.D F6,34(R2)
- ✓ L.D F2,45(R3)
- ✓ MUL.D F0,F2,F4
- ✓ SUB.D F8,F6,F2
- ✓ DIV.D F10,F0,F6
- ✓ ADD.D F6,F8,F2

Vef p

The table shows that

- ✓ Although the SUB.D instruction has completed execution, it does not commit until the MUL.D commits
- ✓ The reservation station and the register status field contains the same basic

Also note that at the time

- ✓ MUL.D is ready to execute and only two L.D instructions have committed,

although several other have completed execution.

- ✓ The SUB.D and ADD.D will not commit until the MUL.D instruction commits, although the results of the instructions are available and can be used as source for other instructions

Further, here

- ✓ The DIV.D is in execution, but has not completed solely due to its longer latency than MUL.D.

The value column indicates the value being held.

The format #X is used to refer to a value field of ROB entry X.

Reorder buffers 1 and 2 are actually completed but are shown for informational purposes

The table below shows the same example for Tomasulo's approach without speculation, discussed earlier.

Let us discuss the key important difference between a processor with speculation and a processor with dynamic scheduling.

Comparing the two tables we can see that

In the non-speculation case, the ADD.D and SUB.D instructions completed out-of-order, i.e., before the MUL.D completed

The in case of speculative hardware:

- ✓ The reservation stations numbers are replaced with the ROB entry numbers in Qj, Qk and in register status fields
- ✓ And, the DEST. Destination Field is added to reservation station
- ✓ The destination field designates the ROB number that is destination for result

Multiple issue with speculation

A speculative processor can be extended to multiple issue using the same techniques we employed when extending Tomasulo-based processor

VI i vi "evi "X s"gl ep r ki wjsv'q yptp "wyyi "{ n "Vsq ewys w'epksvt q

1. Instruction issue and monitoring the CDBs for instruction completion
2. Maintaining throughput of greater than one instruction per cycle

To show how speculation can improve performance in a multiple issue processor. Let us consider an example.

Example

Consider the execution of the following loop, which searches an array, on two- issue processor, once without speculation and once with speculation.

Loop:

LD	R2,0(R1)	; R2= array element
DADDUI	R2,R2,#1	; increment R2
SD	R2,0(R1)	; store result
DADDUI	R1,R1,#4	; increment pointer
BNE	R2,R3,LOOP	; branch if not last element

Assume that

- ✓ There are separate integer functional units for the effective address calculations, for ALU operations, and for branch condition evaluation.
- ✓ Up to two instructions of any type can commit per clock

Let us consider two tables, for the first three iterations of this loop, for machines with and without speculations

The first table shows time of issue, execution, and writing result for two - issue dynamically scheduled processor, without speculation.

Note that the L.D following the BNE cannot start execution earlier, because it must wait until the branch outcome is determined.

This type of program with data dependent branches that cannot be resolved earlier, shows evaluation allow multiple instructions to execute in the same clock cycle.

WWW.VUMULTAN.COM

The second table shows the time of issue, execution and writing result for a dual-issue version of our pipeline with speculation.

Note that the L.D following the BNE can start execution early because it is speculative.

Comparing the two tables, note that the third branch in the speculative processor executes in 13 clock cycle, while in non-speculative processor it executes in 19 clock cycle. That is, the non-speculative pipelines are falling behind the issue rate rapidly

Exceptions to Hardware-based speculation: Extended discussion

So far, we have been discussing the performance-enhancement using the structure of Vsq ewyps w' Cksvnl q " i | x r hi h" x" l er hp " vt i gypexsr w' js' KR" m" wrnkp -issue and multiple-issue processors

Here, we observed that the store-buffer of thi "Vsq ewyps w'wygyvi "w'i p q mex h"er h"e" Re-Order Buffer is included that incorporates the function of store-buffer

The structure is then further extended to handle multiple-issue by making the CDB wider Now, we will talk about the exceptional situations which may arise when executing a program using dynamic scheduling and how the structure with hardware-based speculation considers these exceptions

We know that the dynamic scheduling without speculation, allows to complete execution out-of-order, where as the structure with speculating-hardware commits in-order

Therefore, if an exceptional situation occurs while exacting an instruction, the ROB in wygyvi "{ n "vt i gypexsr "hsi wr 'xgsq q rler h"l er hp "i gi t xsr w

Let us reconsider the execution of our first e| eq t p "t vskveq "ywnk"Vsq ewyps w'wygyvi " with speculation and without speculation

Here, the instructions SUB.D and ADD.D, occurring after the incomplete instruction OWNF."f y'x'i | i gyx h"i evpn."hsr 'xgsq q rlyr xpx i "wygxsr "OWNF"gsq t p x w'er h" commit

- ✓ in an exceptional case, if MUL.D causes an interrupt, then it is handled as follows we can wait until this interrupt reaches the head of ROB and any pending instruction is flushed out, the speculation is un-done
- ✓ Whereas, in case of dynamic scheduling without speculation, the results in registers F8 (for SUB.D) and in register F6 (for ADD.D) could be overwritten out-of-order, thus the interrupt could not be handled

Furthermore, the exceptions are handled not recognizing then until it is ready to commit. This may be explained by considering our earlier example of the execution of a loop

```

Loop:
L.D      F0,0(R1)
MUL.D    F4,F0,F2
S.D      F4,0(R1)
DADDUI   R1,R1,# -8
BNE      R1,R2, LOOP ;branch if R1=R2

```

Here, if the an exception arises, say due to interrupt from MUL.D, the exception is recorded in the ROB. At the same time, if misprediction arises from the speculated instruction (i.e., BNE) then the exception is flushed out along with the speculated instruction that should not have been executed when the ROB is cleared

Summary

handle execution using speculation, i.e.,

Speculating on the outcome of branches to avoid control hazards, which prevent us from starting the next operation before we know whether the branch was correctly predicted or not

The Main idea is to allow execution of a branch instruction, predicted taken, such that there are no consequences if branch is not actually taken

cause exceptions which stop programs

Software generated interrupt or memory violation are typical examples of exceptions

We found that this can be achieved:

- ✓ by including a buffer that holds the results and exceptions from instructions, until it is known that the instruction would execute
- ✓ Such a buffer is called Re-Order Buffer ROB
- ✓ ROB is used only to track commits
- ✓ The ROB is flushed out if the speculation does not hold or exception is found

Lecture 19

Instruction Level Parallelism (Limitations of ILP and Conclusion)

Today's Topics

Recap

Limitations of ILP

- ✓ Hardware model
- ✓ Effects of branch/jumps
- ✓ finite registers

Performance of Intel P6 Micro-Architecture-based processors

- ✓ Pentium Pro, Pentium II, III and IV

Thread-level Parallelism

Summary

Recap: ILP- Dynamic Scheduling

In the last few lectures we have been discussing the concepts and methodologies, which have been introduced during the last decade, to design high-performance processors

Our focus has been the hardware methods for instruction level parallelism to execute multiple instructions in pipelined datapath

These hardware techniques are referred to as Dynamic Scheduling techniques

These techniques are used to avoid structural, data and control hazards and minimize the number of stalls to achieve better performance

We have discussed dynamic scheduling in integer pipeline datapath and in floating-point pipelined datapath

We discussed the score-file dynamic scheduling in integer and floating-point datapath

The structures implementing these concepts facilitate out-of-order execution to minimize data dependencies thus avoid data hazards without stalls

We also discussed branch-prediction techniques and different types of branch-predictors, used to reduce the number of stalls due to control hazards

The concept of multiple instructions issue was discussed in details

This concept is used to reduce the CPI to less than one, thus, the performance of the processor is enhanced

Last, we discussed hardware-based speculation

It allows to speculate that branch is correctly predicted, thus may execute out-of-order but commit in-order having confirmed that the speculation is correct and no exceptions exist

] √pm " " √..ω " √ Dynamic Scheduling

Today we will conclude our discussion on the dynamic scheduling techniques for Instruction level parallelism by introducing an ideal processor model to study the:

- ✓ Limitations of ILP; and
- ✓ Implementation of these concepts in Intel P6 Micro-architecture

Limitations of the ILP Ideal Processor

To understand the limitations of ILP, let us first define an ideal processor.

- Cr "ini ept vsi wsv/w'sri "{ l rgl "hsi wr x'l ezi "ewjng dsr wernw'sr ILP; and
- The only limits in such a processor are those imposed by the actual data flows through either registers or memory

Assumptions for an Ideal processor

An ideal processor is, therefore, one wherein:

- all control dependencies and
- all but true data dependencies

are eliminated

The control dependencies are eliminated by assuming that the: branch and Jump predictions are perfect, i.e., all conditional branches and jumps (including indirect jumps used for return etc.) are predicted exactly; and the processor has perfect speculation and an unbounded buffer of instructions for execution

All but true data dependencies are eliminated by assuming that:

- There are infinite number of virtual registers available facilitating: register renaming thus avoiding WAW and WAR hazards); and Simultaneous execution of an unlimited number of instructions
- All memory addresses are known exactly facilitating: to move a load before a store, provided that the addresses are not identical

Ideal hardware model

Hence, by combining these assumptions, we can say that in an ideal processor:

- ✓ Can issue unlimited number of instructions, including the load and store instructions, in one cycle
- ✓ All functional units have latencies of one cycle, so the sequence of dependent instruction can issue on successive cycles
- ✓ any instruction, in the execution of a program, can be scheduled on the cycle immediately following the execution of the predecessors on which it depend; and
- ✓ the last dynamically executed instruction in the program, can be scheduled on the very first cycle

Performance of a Nearly Ideal Processor

Now let us examine the ILP in one of the most advanced superscalar processor Alpha 21264

Alpha 21264 has the following features:

- ✓ Issues up to 4 instructions/cycle
- ✓ Initiates execution on up to 6 instructions
- ✓ Supports large set of renaming registers (41-integer and 41 floating-point)
- ✓ Uses large tournament type predictor

In order to examine the ILP in this processor a set of six (6) SPEC92 benchmarks (programs), compiled on MIPS optimizing compiler are run.

The features of these benchmarks are:

The three of these benchmarks are floating point benchmarks:

- ✓ Fpppp, Doduc, Tomcatv

The integer programs are given as follows,

- ✓ gcc, espresso, li

Now let us have a look on the performance Alpha 21264 for average amount of parallelism defined as number of instruction-issues per cycle for these benchmarks

Fig 3.35

Here, you can see that fpppp and tomcatv have extensive parallelism so have high instruction-issues

Where as the doduc parallelism doesn't occur in simple loop as it does in fpppp and tomcatv

The integer program li is a LISP interpreter that has many short dependences so offers lowest parallelism

Now let us discuss how the parameters which define the performance of a realizable processor are limited in ILP

The important parameters to be studied are:

- ✓ Window Size and Issue Count
- ✓ Branch and Jump predictors
- ✓ Finite number of registers

Window size and Issue count

In dynamic scheduling, every pending instruction must look at every completing instruction for either of its operand.

A window in ILP processor is defined as:

Start of the window is the earliest uncompleted instruction and the last instruction in the window determines its size

As each instruction in the window must be kept in the processor till the completion of execution, therefore the total window size is limited by the storage, number of comparisons and issue rate

The number of comparisons required every clock cycle is equal to the product:

maximum completion rate x
window size x
number of operands per instruction

For example, if

maximum completion rate = 6 IPC
window size = 80 instructions
number of operands per instruction = 2 operands

then

maximum comparisons required = $6 \times 80 \times 2 = 960$

In real processors, maximum number of instructions that may issue, execute and commit in the same clock cycle, is smaller than the window size

Now let us see the effect of restricting window size on the instruction-issues per cycle

Here

Here, we assume that the instructions in the window are obtained by perfectly predicting branches and selecting instructions until the window is full.

Here, you can see that the amount of the parallelism uncovered falls sharply with decreasing window size e.g.;

for the benchmark gcc, when window size decreases from 2K to 512 parallelism falls from 35 to 10 IPC and parallelism reduces to almost zero when window size is 4

Also note that the parallelism in the integer and FP program is almost similar for a specific window size

Branch and jump prediction

Now let us discuss effect of realistic branch and jump prediction

Our ideal processor assumes that the branches can be perfectly predicted but no real processor can achieve this

Let us have a look on to this graph which shows the effect of realistic prediction schemes, which we have already discussed

Realistic branch and jump prediction

This graph shows the parallelism for five different levels of predictions

- 1) Perfect
 - At the start of execution all the branches and jumps are perfectly predicted and its gives highest parallelism
- 2) Tournament based branch predictor
 - The second step tournament based branch predictor is considered

This scheme uses 2-bit correlating and 2-bit non-correlating predictor with a selector Predictor buffer consists of 8K entries, each consists of three 2-bit fields, that is, 48K bits, both the correlating and non correlating

It achieves the average accuracy of 97% of the six SPEC92 benchmark

The graph shows the extensive difference among the programs:

- ✓ with loop-level parallelism (tomcatv and fpppp) and
- ✓ those without, i.e., integer programs and doduc

The 3rd level using Standard 2 bit predictor with 512 2-bit entries and 4th level using profile history of the program give almost identical results

The 5th level None - where no branch prediction is used, though jumps are still predicted, the parallelism is largely limited to within a basic look

The effect of finite registers

Another important limiting factor on the Instruction-issues per cycle is the finite registers. Ideal processor eliminates all name dependences among the register references assuming an infinite set of physical registers

To date, the Alpha 21264 has provided the largest number of extended registers i.e. 41 integer and 41 FP registers

In addition 32 integer and 32 FP registers are provided

The graph shows the effect of finite number of registers available for renaming using the same six SPEC 92 benchmarks. ~~Hit~~"5063

Both the number of FP registers and the number of GP registers are increased

Although, having only 32 extra FP and 32 extra GP registers has a significant impact on all the programs.

But, the effect is most dramatic on the FP programs (fpppp and tomcatv) the instruction-issues increases from 10 to 45 IPC when registers increases from 32 to 128

Note that the reduction in available parallelism is significant when fewer than an unbounded number of renaming registers (here, less than 32) are available

For the integer programs the impact of having more than 64 registers is not seen here

This is because of limitation in the window size.

Performance of realizable processors with realistic hardware

Let us consider a processor with the following attributes

- 1) up to 64 instructions per clock with no issue restriction,
- 2) A tournament predictor with 1K entries return predictors
- 3) Perfect memory references done dynamically or through a memory dependence predictors
- 4) register renaming with 64 additional integer and 64 additional FP registers

Limitations on ILP for realizable processors

Intel P6

This configuration is more complex and expensive than any existing implementations, especially in terms of number of instructions issue.

The fig shows the results of the configuration as we vary the window size up to 64 issue per clock

Here, we assume that although there are fewer rename registers than the window size, the number of rename registers equals the issue width.

All the operations have zero latency

The fig shows the effect of the window size for the integer programs is not as severe as for the FP programs.

Putting it all together

The Intel p6 micro-architecture forms the basis for the Pentium pro, Pentium II and Pentium III

These three processors differ in clock rate, cache architecture, and memory interface

VI i "Ri r xyq "t vs."x i "t vs gi wsv'er h"wt i gapi h"gegl i "UT CO w{ i vi "mxi kvexi h"mxs" multichip module

K"x i "Ri r xyq "Kwer hev"UT CO w'evi "ywi h"ew'gegl i w

In the Pentium III, there is either an on chip 256 KB L2 cache or an off chip 512 KB cache.

The P6 micro architecture is dynamically scheduled processor that:

translates each IA-32 instruction to a series of micro-operations (uops) that are executed by the pipeline;

The maximum number of uops that may be generated per clock cycle is six, with four allocated to the first IA-32 instructions

The uops are executed by an out of order speculative pipeline using register renaming and a ROB

Performance of the Pentium pro implementation

The Pentium pro has the smallest set of primary caches among the p6 based microprocessors.

It has high bandwidth interface to the secondary caches.

Branch performance and speculation costs

Branch target addresses are predicted with a 512 entry branch target buffer (BTB).

If the BTB does not hit, a static prediction is used.

Backward branches are predicted taken (and have a one cycle penalty if correctly predicted).

Forward branches are predicted not taken and have no penalty if correctly predicted).

Branch mispredicts have both a direct performance penalty, which is between 10 and 15 cycles.

Also indirect penalty due to the overhead of incorrectly speculated instructions.

Which is essentially impossible to measure.

Here

It shows the fraction of the branches mispredicted either because of BTB misses or because of incorrect predictions.

On average about 20% of the branches either miss or are mispredicted and use the simple static predictors rule.

Overall Performance of P6 Pipeline

Overall performance depends on the rate at which instructions actually complete and commit.

Here

The fig shows the fraction of the time in which zero, one, two or three uops commit.

On the average, one uop commits per cycle.

Here, 23% of the time, three uops commit in a cycle.

This distribution demonstrates the ability of a dynamically scheduled pipeline to fall behind on 55% of the cycles, no uops commit) and later catch up (31% of the cycles have two or three uops committing)

The Pentium III versus Pentium 4

The micro architecture of the Pentium 4, which is called Net Burst, is similar to that of the Pentium III, called the P6 micro architecture.

Both fetch up to three IA-32 instructions per cycle, decode them into micro-ops.

Then sends the uops to an out-of-order execution engine that executes up to three uops per cycle.

There are, however, many differences which allow Net Burst micro architecture to operate at a significantly higher clock rate than the P6 micro architecture

These differences also help to maintain, or close to maintain, the peak to sustained execution throughput.

Differences in Pentium III versus Pentium 4

- 1) NetBurst has a much deeper pipeline than P6, P6 requires about 10 clock cycles time for a simple add instruction, from fetch to the availability of its results
 - ✓ In comparison, Net Burst takes about 20 clock cycles, including 2 cycles reserved simply to drive results across the chip,
- 2) Net Burst uses register renaming (as in the MIPS R10K and the Alpha 21264) rather than the reorder buffer, which is used in P6.
 - ✓ Use of register renaming allows many more outstanding results i.e., potentially up to 128 results versus the 40 permitted in P6.
- 3) There are seven integer execution units in the Net Burst versus five in P6.
 - ✓ In addition an additional integer ALU and an additional address computation unit.
 - ✓ An aggressive ALU (operating at twice the clock rate) and an aggressive data cache lead to lower latencies.
 - The latency for the basic ALU operations is effectively one half of a clock cycle in Net Burst versus one in P6)

- The latency for data loads is effectively two cycles in Net Burst versus three cycles in P6)
- ✓ These high-speed functional units are critical to lowering the potential increase in stalls from the very deep pipeline.
- 4) Net Burst uses a sophisticated trace cache to improve instruction fetch performance, while P6 uses a conventional Prefetch buffer and instruction cache.
- 5) Net Burst has a branch target buffer that is eight times larger and has an improved prediction algorithm
- 6) Net Burst has 8 KB Level-I data cache as compared to P6 that has 16KB Level-I data cache.
 - ✓ However, the Net Burst has larger Level-2 cache (256KB) with higher bandwidth
- 7) Net Burst implements the new SSE2 FP instructions that allow two FP operations per instruction
 - ✓ These operations are structured as 12-bit SIMD or short-vector structure.
 - ✓ This gives Pentium 4 a considerable advantage over Pentium-III on FP code.

Lecture 20

Instruction Level Parallelism (Static Scheduling)

] √pm ""] √..ω"

Recap: Dynamic Scheduling in ILP

Software Approaches to exploit ILP

- ✓ Basic Compiler Techniques
- ✓ Loop unrolling and scheduling
- ✓ Static Branch Prediction

Summary

Recap: Dynamic Scheduling

Our discussions in the last eight (8) lectures have been focused on to the hardware-based approaches to exploit parallelism among instructions

The instructions in a basic block, i.e., straight-line code sequence without branches, are executed in parallel by using a pipelined datapath

Here, we noticed that:

- ✓ The performance of pipelined datapath is limited by its structure and data and control dependences, as they lead to structural, data and control hazards

These hazards are removed by introducing stalls

The stalls degrade the performance of a pipelined datapath by increasing the CPI to more than 1

The number of stalls to overcome hazards in pipelined datapath are reduced or eliminated by introducing additional hardware and using dynamic scheduling techniques

The major hardware-based techniques studied so far are summarized here:

Technique	Hazards type stalls Reduced
Forwarding and bypass	Potential Data Hazard Stalls
Delayed Branching and Branch Scheduling	Control Hazard Stalls
Basic Dynamic Scheduling	Data Hazard Stalls from (score boarding)true dependences
Dynamic Scheduling with renaming	Stalls from: data hazards from anti-hi t i r hi r gi w'er h" *Vsq ewyps w' Ct t vsegl + fromoutput dependences
Dynamic Branch Prediction	Control Hazard stalls
Speculation	Data and Control Hazard stalls
Multiple Instructions issues per cycle	Ideal CPI > 1

Introduction to Static Scheduling in ILP

The multiple-instruction-issues per cycle processors are rated as the high- performance processors

These processors exist in a variety of flavors, such as:

- ✓ Superscalar Processors
- ✓ VLIW processors
- ✓ Vector Processors

The superscalar processors exploit ILP using static as well as dynamic scheduling approaches

The VLIW processors, on the other hand, exploits ILP using static scheduling only

The dynamic scheduling in superscalar processors has already been discussed in detail;

And, the basics of static scheduling for superscalar have been introduced

In a few following lectures our focus will be the detailed study of ILP exploitation through static scheduling

The major software scheduling techniques, under discussion, to reduce the data and control stalls, will be as follows:

Technique	Hazards type stalls Reduced
Basic Compiler scheduling	Data hazard stalls
Loop Unrolling	Control hazard stalls
Compiler dependence	Ideal CPI, Data hazard stalls
Trace Scheduling	Ideal CPI, Data hazard stalls
Compiler Speculation	Ideal CPI, Data and control hazard stalls

Basic Pipeline scheduling

In order to exploit the ILP, we have to keep a pipeline full by a sequence of unrelated instructions which can be overlapped in the pipeline

Thus, a dependent instruction in a sequence, must be separated from the source instruction by a distance equal to the latency of that instruction, $\text{fsv} \mid \text{eq t } \dot{\text{p}} ."$

A FP ALU operation that is using the result of earlier FP ALU operation

- ✓ must be kept 3 cycles away from the earlier instruction; and

A FP ALU operation that is using the result of earlier Load double word operation

- ✓ must be kept 1 cycles away from it

For our further discussions we will assume the following average latencies of the functional units

- ✓ Integer ALU operation latency = 0
- ✓ FP Load latency to FP store = 0;
(here, the result of load can be bypassed without stalling to store)
- ✓ Integer Load latency = 1; whereas
- ✓ FP ALU operation latency to FP store = 2

A compiler performing scheduling to exploit ILP in a program, must take into consideration the latencies of functional units in the pipeline

Let us see, with the help of our earlier example to add a scalar to a vector, how a compiler can increase parallelism by scheduling

Execution a Simple Loop with basic scheduling

Let us consider a simple loop:

```
for (i=1000; i>0; i=i-1)
    x[i] = x[i] + scalar
```

Where, a scalar is added to a vector in 1000 iterations; and the body of each iteration is independent at the compile time

MIPS code without scheduling

The MIPS code without any scheduling look like,

```

L.D      F0, 0(R1)      ;F0 array element
ADD.D    F4, F0, F2      ;add scalar in F2
S.D      F4, 0(R1)      ;store result
DADDUI   R1, R1, #-8     ;decrement pointer 8 bytes
BNE      R1, R2, LOOP    ;branch R1!=R2

```

Notice the data dependencies in ADD and STORE operation which lead to data-hazards; and control hazard due to BNE instruction

Loop execution without Basic Scheduling

Let us assume that the loop is implemented using standard five stage pipeline with branch delay of one clock cycle

Functional units are fully pipelined

The functional units have latencies as shown in the table

Stalls of FP ALU and Load Instruction

Here, the First column shows originating instruction type

Second column is the type of consuming instruction

Last column is the number of intervening clock cycles needed to avoid a stall

Instruction producing result	Instruction using result	Latency in clock cycle
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Single Loop execution (without scheduling for the latencies and stalls for originating viz-a-viz consuming instructions)

Instructions	clock cycles	
L.D F0, 0(R1)	1	
Stall	2	; L.D followed by FP ALU op has latency=1
ADD.D F4, F0, F2	3	
Stall	4	; FP ALU op followed by STORE double
Stall	5	; has latency =2
S.D F4, 0(R1)	6	
DADDUI R1, R1, #-8	7	
Stall	8	; Double ALU has latency = 1
BNE R1, R2, LOOP	9	
Stall	10	; Branch has latency = 1

This code requires 10 clock cycles per iteration.
We can schedule the loop to reduce the stall to 1

Single loop execution With Compiler scheduling

<u>Loop</u>		<u>clock cycles</u>
L.D	F0, 0(R1)	1
DADDUI	R1, R1, #-8	2
ADD.D	F4, F0, F2	3
Stall		4
BNE	R1, R2, LOOP	5 (delayed branch)
S.D	F4, 8(R1)	6 (altered & interchanged with DADDUI)

Explanation

To schedule the delay branch, compiler had to determine that it could swap the DADDUI and S.D by changing the destination address of S.D instruction

You can see that the address 0(R1) and is replaced by 8(R1); as R1 has been decremented by DADDUI

Note that the chain of dependent instructions from L.D to the ADD.D and then ADD.D to S.D determines the clock cycles count; which is for this loop = 6, and for unscheduled execution = 10

In this example, one loop iteration and store back is completed in one array element every 6 clock cycles

but the actual work of operating on the array element takes 3 clock cycles (load, add, and store)

The remaining 3 clock cycles per iteration are the loop-overhead (to evaluate the condition, stall and branch); i.e., the loop over-head is 100% in this example

Loop Unrolling

To eliminate or reduce the impact the loop-overhead, here 3 clock cycles per loop, we have to get more operations within the loop, relative to the number of overhead instructions

A simple way to increase the number of instructions per loop can be to replicate the loop body for number of iterations and adjusting the loop termination code

This approach is known as loop unrolling

Loop Unrolling without scheduling

Let us reconsider our earlier example of loop; and unroll the loop so that there are four (4) copies of the loop body

Assume that R1 is initially a multiple of 32

i.e., the number of loop iterations is a multiple of 4 and R1 contains 8-byte double word

Also, assume that redundant computations are eliminate and registers are not reused

Example: Loop Unrolling without scheduling

LOOP

```

L.D      F0, 0(R1)
ADD.D    F4, F0, F2
S.D      F4, 0(R1)      ; drop ADDUI &BNE
L.D      F6, -8(R1)
ADD.D    F8, F6, F2 L
S.D      F8, -8(R1)      ; drop ADDUI &BNE
L.D      F10, -16(R1)
ADD.D    F12, F10, F2
S.D      F12, -16(R1)    ; drop ADDUI &BNE
L.D      F14, -24(R1)
ADD.D    F16, F14,
S.D      F16, -24(R1)
DADDUI   R1, R1, #-32
BNE      R1, R2, LOOP

```

Loop Unrolling and scheduling

Note that simply replicating the instructions, when the loop is unrolled, results in the use of the same register that could prevent us from effectively scheduling the loop

Here, the DADDUI instruction is merged into the LOAD instruction of the 2nd and 3rd iteration codes; and

BNE is dropped in the first three (3) iterations, i.e., 3 branches and 3 decrement R1 operations are dropped here

The R2 in the instruction BNE R1, R2, Loop must now be set so that 32(R2) is the starting address

Let us have a look on to execution without scheduling

Loop		clock cycles
L.D	F0, 0(R1)	1
stall		2
ADD.D	F4, F0, F2	3
Stall		4
stall		5
S.D	F4, 0(R1) ; drop ADDUI &BNE	6
L.D	F6, -8(R1)	7
Stall		8
ADD.D	F8, F6, F2	9
Stall		10
stall		11
S.D	F8, -8(R1) ; drop ADDUI &BNE	12
L.D	F10, -16(R1)	13
Stall		14
ADD.D	F12, F10, F2	15

Stall		16	
stall		17	
S.D	F12, -16(R1) ; drop ADDUI &BNE		18
L.D	F14,-24(R1)	19	
Stall		20	
ADD.D	F16, F14,		21
Stall		22	
stall		23	
S.D	F16 -24(R1)		24
DADDUI	R1, R1, #-32	25	
Stall		26	
BNE	R1, R2, LOOP		27
Stall			28

Note that, here without scheduling, every operation in the unrolled loop is followed by dependent operations e.g., L.D followed by ADD.D has data dependence; therefore L.D has 1 stall similarly, 2 stalls are for ADD.D, 1 stall for DADDUI and 1 stall for branch. This loop is, therefore, executed in 28 clock cycles (14 for instruction issue and 7 stalls). This unrolled version is currently slower than the scheduled version, discussed earlier, where one iteration completes in 6 cycles so 24 cycles for the 4 iterations.

Unrolling with scheduling

Now let us see the performance of unrolled loop with scheduling.

As the instructions within a loop from different iterations, can be re-ordered, therefore, Loop Unrolling can also be used to improve scheduling.

Example 3: Unrolling with scheduling

Loop		clock cycles
L.D	F0, 0(R1)	1
L.D	F6,-8(R1)	2
L.D	F10,-16(R1)	3
L.D	F14,-24(R1)	4
ADD.D	F4, F0, F2	5
ADD.D	F8, F6, F2 L	6
ADD.D	F12, F10, F2	7
ADD.D	F16, F14,	8
S.D	F4, 0(R1)	9
S.D	F8, -8(R1)	10
DADDUI	R1, R1, #-32	11
S.D	F12, -16(R1)	12
BNE	R1, R2, LOOP	13
S.D	F16, -24(R1)	14

Here, the unrolled loop of the previous example is scheduled, assuming the same latencies as earlier

are scheduled in sequence

Here, the 4 LOAD instructions of four iterations are executed in sequence within 4 cc.

The STORE instruction of first iteration is issued in the 5th clock cycle, without any stall, as it does not have dependence on the prior 3 LOAD instructions

The branch condition evaluation instruction DADDUI is issued in the 11th clock cycle to avoid stall prior to the BNE, which is issued in the 13th clock cycle

The STORE instruction of the 4th iteration is issued in 14th clock cycle eliminating stall after a branch

Note that the execution time of the unrolled loop has dropped to a total of 14 cycles, i.e., $14/4=3.5$ clocks per element, compared with 7 cycles per iteration before scheduling the loop in this way

Loop unrolling and scheduling: Conclusion

Our discussion so far reveals that the key to the key to perform loop unrolling and scheduling is to know when and how the ordering among instructions be changed

Furthermore, to make the final decision we must:

- ✓ Determine that it was legal to move S.D after the DADDUI and BNE and find the amount to adjust the S.D offset.
- ✓ Determine that unrolling the loop would be useful by finding that the loop iterations were independent except for the loop maintenance code
- ✓ Use different registers to avoid unnecessary constraints that would be focused by using the same registers for different computations
- ✓ Eliminate the extra test and branch instruction and adjust the loop termination and iteration code
- ✓ Determine that the loads and stores in the unrolled loop can be interchanged by observing that the loads and stores from different iterations are independent
- ✓ Schedule the code, preserving any dependencies needed to yield the same result as the original code.

Limits to the gains of Loop unrolling and scheduling

From our discussion on loop unrolling so far, we have observed the following limits to the gains which could be achieved

1. **Loop overhead** When we unrolled the loop, it generated sufficient parallelism among instructions that motivated the loop to be scheduled with no stalls. The scheduling resulted in 14 clock cycles to issue all instruction with the overhead was only 2 cycles

One for DADDUI or DSUBI which maintains the index value; and

The other one for BNE which terminates the loop i.e., the overhead per iteration was 2/4 or 1/2 cycles. This overhead could be reduced to 2/8 or 1/4 cycles per iteration if the loop is unrolled 8 times and to 2/16 or 1/8 if unrolling is for 16 times and so on

2. **Growth in code size** For larger loops the code size growth may be of concern if larger size causes decrease in the cache miss rate
3. **Register Pressure** (Shortfall in Registers) The aggressive unrolling and scheduling of large code size may result in shortfall of registers, which is referred to as the Register Pressure. This is due to the facts that after aggressive scheduling of instructions it may not be possible to allocate registers to live values. Therefore, while unrolling and scheduling the compiler takes care of these limitations

Loop unrolling and scheduling with MULTIPLE ISSUES in Superscalar

Let us see how the loop unrolling and scheduling enhances the performance in case of multiple instructions issue per cycle case already discussed with dynamic scheduling

Here, let us consider the same loop example with two-issues per cycle

One of the instruction may be load/store/branch or integer ALU and the other is FP

The unrolled and scheduled code is as given here

Here, to schedule the loop without delay it is unrolled 5 times (for 5 stage Superscalar pipeline)

After unrolling the loop contains 5 L.D, ADD.D and S.D and one each DADDUI and BNE

Two issues take place in 3rd through 7th cycles as the first L.D with next FP ALU has latency of 2

Loop Unrolling in Superscalar

Integer instruction	FP instruction	Clock cycle
Loop: LD F0,0(R1)		1
LD F6,-8(R1)		2
LD F10,-16(R1)	ADDD F4,F0,F2	3
LD F14,-24(R1)	ADDD F8,F6,F2	4
LD F18,-32(R1)	ADDD F12,F10,F2	5
SD 0(R1),F4	ADDD F16,F14,F2	6
SD -8(R1),F8	ADDD F20,F18,F2	7
SD -16(R1),F12		8
SD -24(R1),F16		9
SUBI R1,R1,#40		10
BNEZ R1,LOOP		11
SD -32(R1),F20		12

The unrolled superscalar (SS) loop runs in 12 clocks, or

$12/5 = 2.4$ clocks per iteration

against $14/4 = 3.5$ for scheduled and unrolled on simple 5-stage MIPS pipeline. Thus, it exhibits an improvement factor of 1.5

In the earlier lectures, while introducing the concept of Dynamic scheduling we discussed Dynamic Branch Predictors

The dynamic branch predictors predict the branch to be taken or not taken by considering the run time behavior of the code

We also introduced the concept of Delayed Branch as the static branch prediction technique

Delayed branches expose a pipeline hazard so that compiler can reduce the penalty associated with the hazard

However, static branch predictors are used in processors where the expectation is that branch behavior is highly predictable at compile time

Being able to accurately predict a branch at compile time, it is also helpful for scheduling data hazards e.g. loop unrolling

Example: Static branch prediction

Let us consider an example that arises from conditional selection branches

LD	R1, 0(R2)
DSUBU	R1, R1, R3
BEQZ	R1, L
OR	R4, R5, R6
DADDU	R10, R4, R3
L: DADDU	R7, R8, R9

Explanation:

- ✓ Here, note the dependence of the DSUBU and BEQZ on the L.D instruction
- ✓ This shows that a stall will be needed after the L.D.
- ✓ If it is predicted that the branch (BEQZ) was almost always taken; and that the value of R7 was not needed on the fall-through path
- ✓ Then the speed of the program could be improved by moving the instruction
L: DADDU R7, R8, R9
to the position after the L.D.
- ✓ On the other hand, if it is predicted that branch (BEQZ) was rarely taken and that the value of R4 was not needed on the taken path, then we could also consider moving the OR instruction after the L.D
- ✓ Furthermore, we can also use the information to better schedule any branch delay as
- ✓ Scheduling depends on knowing the branch behavior

To perform optimization we need to predict the branch statically when the program is compiled.

There are several methods to statically predict the branch behavior

A. The simplest scheme is to predict a branch as taken

This scheme has an average misprediction rate that is equal to the untaken branch frequency, which for the SPEC is 34%

However, the misprediction rate ranges from not very accurate (59%) to highly accurate (9%)

B. To predict on the basis of the branch direction

Choosing backward going branches to be taken and forward going branches to be not taken branches

For SPEC programs, more than half of the forward going branches are taken.

Hence, predicting all the branches as taken is a better approach.

C. To predict branches on the basis of the profile information

This is more accurate technique where the branches are predicted on the basis of the profile information collected from earlier runs

The behavior of these branches is often bi-modal distributed i.e. an individual branch is often highly biased toward taken or untaken.

We can derive the prediction accuracy of a predicted taken strategy and measures the accuracy of the profile scheme

The fig below shows the misprediction rate on SPEC 92 for a profile based predictor

Here, you can see that the misprediction rate varies widely

Fig..4.3

It is generally better for the FP programs than the integer programs

For FP benchmarks the misprediction rate varies from 4% to 9% and

For the integer programs it varies from 5% to 15%

Summary

Today we started with the discussion on the static scheduling techniques to exploit the ILP in pipeline datapath

Here we discussed the basic compiler approach to used to avoid hazards, specially the data and control hazards by inserting stalls

The number of stall are reducing by scheduling the instructions by the compiler

In case of loops, the loops are unrolled to enhance the performance and reduce stalls

The number of stalls are further reduced when stalls unrolled loop is scheduled by repeating each instruction for the number of iteration, but using additional registers

Finally we discussed the impact of static branch prediction on the performance on the scheduled and unrolled loops

We observed that static branch prediction result in decrease in the misprediction rate ranging between 4% to 15%, thus greatly enhances the performance of superscalar processor

Lecture 21

Instruction Level Parallelism (Static Scheduling Multiple Issue Processor)

]√pm ""]√..ω"

Recap: Static Scheduling and Branch Prediction
 Static Multiple Issue: VLIW Approach
 Detecting and enhancing loop level parallelism
 Software pipelining
 Summary

Recap: Static Scheduling

Last time we started discussion on to the static scheduling techniques to exploit the ILP in pipeline datapath
 We noticed that inserting stalls is the basic compiler approach used to avoid the data and control hazards
 However, as the number of stalls degrade the performance so compiler schedule the instructions to avoid hazards and to reduce or eliminate stalls
 Furthermore, we observed that in case of loops, the loops are unrolled to enhance the performance and reduce stalls
 The number of stalls are further reduced when unrolled loop is scheduled by repeating each instruction for the number of iteration, but using additional registers
 Finally, we discussed the impact of static branch prediction on the performance on the scheduled and unrolled loops
 We also observed that in superscalar processor, with multiple issues, the static branch prediction results in decrease in the misprediction rate better than the dynamic branch prediction
 Here, the misprediction rate ranges between 4% to 15%

]√pm "Kū'om"u/z" Scheduling in VLIW processor

We know that the Very Long Instruction Word or VLIW-based processors schedule multiple instruction issues using only the static scheduling
 Today we will extend our discussion on the Static Scheduling as used in VLIW processors

Review of VLIW format

A VLIW contains a fixed set of instructions, say 4-16 instructions
 A VLIW is formatted: Either as one large instruction Or a fixed instruction packet with explicit parallelism among instructions in a set

r1 = L r4	r2 = Add r1, M	f1 = Mul f1, f2	r5 = Add r5, 4
-----------	----------------	-----------------	----------------

VLIW / EPIC Processor

Since there exist explicit parallelism among instructions; VLIW is also referred to as: Explicitly Parallel Instruction Computing EPIC

It can initiate multiple instructions in a cycle by putting operations into wide template or packet by the compiler

A packet may contain 64 - 128 bytes

Multiple-Issue overheads - VLIW vs. Superscalar

- ✓ In superscalar processor Overhead grows with issue-width
 - For two-issue processor the overhead for is minimal
 - For four-issue processor the overhead for is manageable
- ✓ For VLIW the over-head does not grow with the issue-width

The early VLIW machines were rigid in their instruction formats and required recompilation of programs for different versions of the hardware

Certain innovations are made in recent architectures to eliminate the need for recompilation; hence results in performance enhancement

Here, the wider processors are used which employ multiple number of independent functional units; and

The compiler does most of the work in finding and scheduling instructions for parallel execution

Compiler schedules and packs multiple operations into one very long instruction word; and

Hardware simply issues the complete packet given to it by the compiler

Thus, maximum issue-rate is increased

Compiler schedules and packs multiple operations into one very long instruction word;

Hardware simply issues the complete packet given to it by the compiler

Thus, maximum issue-rate is increased

Example: VLIW Processor

Let us consider an example of VLIW processor which can perform maximum five operations in one cycle

These operations include:

- ✓ One integer operation
- ✓ Two floating point operations; and
- ✓ Two memory reference operations

Here, we assume that: the instructions have set of 16-bit to 24-bit fields for each unit with an instruction length ranging from 112 and 168 bits; and

To keep functional unit busy, there must be enough parallelism in the code sequence to fill the available operation slots.

Example: VLIW Loop unrolling

Now let us see how a loop is to be unrolled to execute using multiple-issue with VLIW processor

Here, If unrolling the loop generates straight line code then local scheduling techniques, which operates on single basic block, can be used

Where as If parallelism is required across the branches then complex global scheduling is used to uncover the parallelism

In order to explain these concepts let us reconsider our earlier example MIPS code to add scalar to a vector; i.e., $x[i] = x[i] + s$

Loop	L.D	F0, 0(R1)	;F0 array element
	ADD.D	F4, F0, F2	;add scalar in F2
	S.D	F4, 0(R1)	;store result
	DADDU	R1, R1, #-8	;decrement pointer 8 bytes
	BNE	R1, R2, LOOP	;branch R1! =R2

In order to execute this code using multi-issue VLIW, we may unroll the loop as many times as necessary to eliminate any stalls, ignoring the branch delay, if any

Assumptions

Let us assume that:

The compiler can generate long straight line code using local scheduling to build up VLIW instructions

VLIW processor has sufficient registers and function units to issue up to 5 instructions in one cycle; i.e., 15 registers verses 6 in Superscalar

The loop is unrolled in order to make seven copies of the body, which eliminates all stalls and avoid delays

Loop Unrolling for VLIW Processor

				Clock
LD F0,0(R1)	LD F6,-8(R1)			1
LD F10,-16(R1)	LD F14,-24(R1)			2
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4,F0,F2	ADDD F8,F6,F2	3
LD F26,-48(R1)	ADDD F12,F10,F2	ADDD F16,F14,F2		4
	ADDD F20,F18,F2	ADDD F24,F22,F2		5
SD 0(R1),F4	SD -8(R1),F8	ADDD F28,F26,F2		6
SD -16(R1),F12	SD -24(R1),F16			7
SD -32(R1),F20	SD -40(R1),F24	SUBI R1,R1,#48		8
SD -0(R1),F28	BNEZ R1,LOOP			9

Assumptions

The table shows the VLIW instructions that occupy the copies of the loop instructions in unrolled sequence

Here, we assume that R1 has been initialized to #48 for 7 iterations

[each memory location is 8 byte apart, starting with first value at #48, the 7th value is at #0]

Explanation

Here, the loop has been unrolled for seven (7) iterations to completely empty issue cycles, thus eliminate the stalls

Each instruction comprises two (2) memory reference operations [L.D or S.D], two (2) FP operations [ADD.D] and one (1) integer operation [SUBI]

The multiple-issues per cycle are depicted here showing type of operation in each instruction

Operation types in VLIW

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Int. op/ branch	Clock
LD F0,0(R1)	LD F6,-8(R1)				1
LD F10,-16(R1)	LD F14,-24(R1)				2
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4,F0,F2	ADDD F8,F6,F2		3
LD F26,-48(R1)		ADDD F12,F10,F2	ADDD F16,F14,F2		4
		ADDD F20,F18,F2	ADDD F24,F22,F2		5
		ADDD F28,F26,F2			6
SD 0(R1),F4	SD -8(R1),F8				7
SD -16(R1),F12	SD -24(R1),F16				8
SD -32(R1),F20	SD -40(R1),F24			SUBI R1,R1,#48	9
SD -0(R1),F28				BNEZ R1,LOOP	

Explanation

As our example VLIW processor can handle two memory operation, therefore two (2) L.D operations for 1st and 2nd iteration; are issued in the 1st clock cycle; and the two (2) L.D operations for 3rd and 4th iterations, having no dependence, are issued in 2nd clock cycles

Furthermore, our VLIW processor can handle two (2) FP operations; and two (2) ADD.D operation of the 1st and 2nd iterations have dependence on the L.D instructions of the respective two iterations; and

the L.D instruction has latency of 2 cycles, therefore, the two ADD.D instructions of 1st and 2nd iteration are scheduled in 3rd cycle to eliminate stalls (identified by yellow arrows)

Similarly as the FP ADD and following STORE of the same iteration have latency of 3, therefore, the STORE operation of 1st and 2nd iterations are scheduled in 6th cycle

Moreover, the branch evaluation operation SUBI R1,R1,#48 and the branch instruction BNEZ R1,LOOP have latency of one cycle when branch delay slot is ignored, therefore these two operations are performed in 8th and 9th cycles, respective

Performance Analysis based on Issue Rate

Note that all the 23 instructions of 7 iterations

i.e., $7 \times 3 = 21$ loop instructions and 1 condition evaluation and 1 branch instruction) are issued in 9 cycles

Thus, average issue rate is $23/9 = 2.5$ operations per cycle as compared to 3.5 in case of superscalar discussed earlier

VLIW Processor Efficiency Based on available slots

However, note that here each Instruction Word (VLIW) may have up to 5 operations, thus in 9 instructions 45 slots are available

Therefore, the efficiency of the VLIW processor, which is defined as, the percentage of available slots containing an operation, is $23/45 = 0.51$ or 51%

Performance Analysis Speed of operation

Moreover, the VLIW code runs in 9 cycles resulting in $9/7 = 1.29$ cycles per iteration

This result shows that VLIW is twice as fast as superscalar schedules code

Advanced Compiler Support

Our discussion so far has been focused on the study of basic compiler techniques to exploit ILP

Here, we studied how dependence prevent a loop from being executed in parallel i.e., how Loop Level Parallelism is limited?

LLP vs IVP Loop Level Parallelism (LLP) Verses Instruction Level Parallelism (ILP)

LLP emphasizes more on determining the type of dependence among the operands in a loop across the iterations of loop

ILP emphasis more on dependence among the instructions in a loop

Detecting dependence and enhancing LLP

In general, LLP is analyzed at the source level (i.e., at the higher level)

Whereas the ILP analysis is performed after the compiler has generated the instructions

At the loop level there may exist data and name dependences

Data dependency basically occurs when operand is written at some point and read at a later point

Name dependency can also occur and it can be eliminated by simply renaming technique

Therefore, we will discuss only data dependency

Loop Carried Dependence - Data Dependence in LLP

While dealing LLP we have to consider the data dependency that occurs when:

- ✓ The data accessed in later iteration depends upon the data value produced in the previous iteration

This is referred to as loop carried dependence - LCD

Distinction between LLP and LCD

The examples we have discussed so far had LLP but not LCD

Now let us consider two examples to distinguish between LLP and LCD

Example 1 LLP: Adding a constant to a vector

```
For (i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

Here, you can see that here two dependences occurs

First, there exist dependence between the two uses of $x[i]$ within the same iteration, so this is loop-level dependency but not loop carried

Second, dependency occurs between the successive uses of i in different iterations, which is loop-carried

LLP and LCD

Here, there exist induction variable $x[i]$

Therefore, dependence can be identified through compiler analysis near source level; and

Can be eliminated easily by loop unrolling, as discussed last time

Example 2: LCD Preventing ILP

Now consider another example of loop

```
For ( i=1; i<=100; i=i+1) {
    A[i+1] = A[i] + C[i];    //s1
    B[i+1] = B[i] + A[i+1] //s2
}
```

Here, let us assume that A, B, and C are distinct, non-overlapping arrays

There exist two dependences between the statements of the loop

```
For ( i=1; i<=100; i=i+1)
{
    A[i+1] = A[i] + C[i];    //s1
    B[i+1] = B[i] + A[i+1] //s2
}
```

The statements S1 and S2, use the value computed by each statement in earlier iteration i.e., $A[i+1]$ computed in i^{th} iteration and read in iteration $(i+1)$ but is to be used in the same i^{th} iteration

- ✓ The second statement S2 uses the value $A[i+1]$ computed by first statement S1 in the same i^{th} iteration

Here, the first dependence is loop-carried because:

dependence of each statement S1 or S2 is on the earlier iteration of the same statement.

Thus, if this were the only dependence, the successive iterations are in a sequence

Second dependence is not loop carried as S2 depends on S1 which is within the same iteration.

Thus, if this were the only dependence, the multiple iterations of the loop could be executed in parallel by unrolling, as long as each pair of the statements in an iteration is kept in order

LCD not preventing ILP

Now let us consider another example of loop in which loop carried dependence does not prevent parallelism.

```
For (i=1; 1<=100; i=i+1)
{
  A[i]  = A[i] + B[i]; / S1
  B[i+1] = C[i] + D[i]; /S2
}
```

In order to analyze this example we have to find:

- ✓ The dependence between S1 and S2?
- ✓ Is this loop parallel?

If not, show how to make it parallel

Step 1: Identifying dependence

```
For (i=1; 1<=100; i=i+1)
{
  A[i]  = A[i] + B[i]; / S1
  B[i+1] = C[i] + D[i]; /S2
}
```

Note that firstly there exist loop carried dependence between S1 and S2.

As S1 uses the value B[i] assigned in the previous iteration by S2.

VI yw"U3"hi t i r hw"sr "U4"fyx"U4"hsi wr x"hi t i r h"sr "U30

Step 2: Determining parallelism

Note that a loop is said to be in parallel, if it can be written without cycle in dependence

- ✓ i.e., one statement depends of the previous iteration of a an earlier statement of the loop

Detecting loop level parallelism: Step 2

Our example loop does not have circular dependence

```
For (i=1; 1<=100; i=i+1)
{
  A[i]  = A[i] + B[i]; / S1
  B[i+1] = C[i] + D[i]; /S2
}
```

VI w"wyi "s"x i "v ewsr "x ex"U4"*ri | x"weix q i r x"hsi wr x"l ezi "hi t i r h i r gi "sr"x i " previous iteration of S1

Thus, this loop can be re-organized by partial ordering to expose the Loop Level parallelism

However, in order to obtain this reorganization, the following two observations must be taken in to consideration

Observations to expose parallelism

- 1) Here, in the first iteration the statement S1 depends on the value B[1], which is to be computed prior to initiating the loop
- 2) There is no dependence from S1 to S2, therefore, there would be no cycle in the dependences and loop would be parallel

In order to remove this dependence, identified in the first observation, interchange the two statements such that execution of S2 is not effected

Considering these two observations we can replace the loop with the code given below.

Code enhancing parallelism

```
A[1] = A[1] + B[1];
for (i=1; i<99; i=i+1)
{
    B[i+1] = C[i] + D[i];
    A[i+1] = A[i+1] + B[i+1];
}
B[101] = C[100] + D[100];
```

Note here that S1 does not have dependence on S2, so the two statements are not loop-carried, hence the iterations of the loop can be overlapped

Finding Dependences: Greatest Common Divisor Test

So far we have been taking about the impact of data dependence on the loop level parallelism

We also noticed that Loop Carried dependence is crucial in this case; and is removed to enhance parallelism

Now we will discuss how compiler determines the dependence to enhance parallelism

We know that finding of dependence in a program is important to:

1. Have good scheduling of code.
2. Determine which loop might contain parallelism
3. Eliminate name dependence

How compiler finds dependences

The compiler detects the dependence using dependence analysis algorithm

The dependence analysis algorithm works on assumptions that:

- ✓ Array indices are affine; and
- ✓ There exist Greatest Common Divisor (GCD) of the two affine indices

Finding Dependences

An affine index is defined as follows

An array index is affine if it can be written in the form of an expression:

$$a \times i + b$$

Here, a and b are constant; and i is the loop index variable

E.g. in the loop

```
for (i=1; i<=100; i=i+1)
{
    X[2*i+3] = X[2*i] * 5.0;
}
```

Here, the index value $[2*i+3]$ is affine with $a=2$ and $b=3$

Multi dimensional array

1. A multi-dimensional array index is affine if index of each dimension is affine

For example $x[y[i]]$ the index $[i]$ of y is non-affine

Algorithm: to find dependences

There exists dependence between two references to the same array in the loop. If two affine functions can have the same value for different indices between the bounds of the loop

For example, consider an array element stored with index value $a \times i + b$ and loaded from the same array with index value $c \times i + d$, where i is the for loop index variable that runs m to n , then

A dependence occurs if the following two conditions hold where:

1. Two iterations indices, j and k , are both within the limits of the for loop
i.e., $m \leq j \leq n$, $m \leq k \leq n$
2. the loop stores into an array element indexed by $a \times j + b$ and later fetches from that same array element when it is indexed by $c \times k + d$, then Dependence exist
if: $a \times j + b = c \times k + d$

As the dependence, in reality is very complex, therefore, Greatest Common Divisor (GCD) is a simple test for the absence of a dependence

That is, if a loop-carried dependence exists, then $\text{GCD}(c, a)$ must divide $(d-b)$

Example:

Find whether dependences exist in the following loop

```
for (i=1; i<=100; i=i+1)
{
    X[2*i+3] = X[2*i] * 5.0;
}
```

Solution

Here in $X[2*i+3]$

$a=2$ and $b=3$

and in $X[2*i]$

$c=2$ and $d=0$

Thus $\text{GCD}(a, c) = 2$ and $d-b = -3$

Here, as 2 does not divide -3 so, No dependence is possible

Conclusion GCD Test

The GCD test is sufficient to guarantee that no dependence exist
 However there are cases when GCD test succeeds but no dependence exists
 This arises, in case the GCD test does not take the loop bounds into account
 Please see the class notes for examples

Classifying Dependences

In addition to detecting the presence of a dependence, a compiler wants to classify the type of dependence
 That is, the compiler recognize name dependence and eliminate them at compile time by renaming and copying

Enhancing ILP

To achieve more ILP, compilers can reduce the impact of dependent computations
 The key technique for this is to eliminate or reduce dependent computations by back substitution

Summary

Today we have introduced the concept of loop-carried dependence
 We notice that in most of the cases Loop carried dependence prevent parallelism
 Then we discussed techniques to find dependence
 Here, we define affine indices and introduced greatest common divisor test to determine the loop-carried dependence
 Following this we discussed techniques to eliminate dependent computation
 These techniques are employed in compiler based scheduling

Assignment

Detecting and enhancing loop level parallelism

In this example, dependence information is inexact, as it tells that such a dependence occur.

Now consider one more example,

```
for ( i=1; i<=100; i=i+1)
{
    A[i] = B[i] + C[i];
    D[i] = A[i] * E[i];
}
```

Here the second reference to A need not to be translated to a load instruction.
 As the value is computed and stored by the previous statement.

The second reference to A is a reference to register in which A was computed.
 In general, data dependence analysis tells that one reference may depend on another.

But a complex analysis requires to determine that two reference must be to the exact

same address.

In this example a simple version of this analysis suffices.

As the two references are on the same basic block.

Loop carried dependence occurs most of the time in the form of recurrence.

for (i=2; i<=100; i=i+1)

```
{
    Y[i] = Y[i-1] + Y[i];
}
```

Recurrence exists, when the variable is defined based on the value of that variable in the earlier iteration.

Recurrence detection is important for two reasons.

1. Some architectures have special support for executing recurrences.
2. Some recurrence can be the source of a reasonable amount of parallelism.

Now consider one more loop,

for(i=6; i<=100; i=i+1)

```
{
    Y[i] = Y[i-5] + Y[i];
}
```

Here in ith iteration, the loop reference element is i-5.

This loop have a dependence distance of 5.

Loop carried dependences have a data dependence of 1.

With large dependence distance, more potential of parallelism is obtained by loop unrolling.

Longer distances may provide the enough parallelism to keep the processor busy.

Back Substitution

Back Substitution increases the amount of parallelism, but sometimes it also increases the amount of computation required

These techniques can be applied both:

- ✓ Within a basic block; and
- ✓ Within a loop

Eliminating dependent computations

Within a basic block: Here, algebraic simplifications of expressions and an optimization is used.

This called copy propagation, it eliminates operations that copy values

For example; copy propagation of

```
DADDUI    R1,R2,#4
```

```
DADDUI    R1,R1,#4
```

Results into

```
DADDUI    R1,R2,#8
```

Here, computations are eliminated to remove dependence

Eliminating dependent computations:

Tree-Height Reduction Technique

1. Optimization:

It is also possible to increase the parallelism of the code by possibly increasing the number of operations.

Such optimization is called tree height reduction

For example, the code sequence

```
ADD      R1,R2,R3
```

```
ADD      R4,R1,R6
```

```
ADD      R8,R1,R7
```

Requires three cycles for execution

Because, here all the instructions depend on immediate predecessor and cannot be issued in parallel

2. Associativity:

Now taking the advantage of the associativity, the code can be transformed and written in the form shown as below,

```
ADD      R1,R2,R3
```

```
ADD      R4,R6,R7
```

```
ADD      R8,R1,R4
```

This sequence can be computed in two execution cycles by issuing first 2 instructions in parallel

3. Recurrences:

Recurrences are expressions whose value in one iteration is given by a function that depends on the previous iteration.

Common type of recurrence occurs in: $\text{sum} = \text{sum} + x;$

Assuming an unroll loop with the recurrence of five times.

If the value of x of these five iterations be given by x_1, x_2, x_3, x_4 and x_5 .

Then we can write the value of sum at the end of each unroll as,

$\text{Sum} = \text{sum} + x_1 + x_2 + x_3 + x_4 + x_5;$

Unoptimizing the expressions requires five dependent operations.

And it can be rewritten as,

$\text{Sum} = ((\text{sum} + x_1) + (x_2 + x_3)) + (x_4 + x_5);$

This can be evaluated in only three dependent operations.

Recurrence also occurs from implicit calculations.

With unrolling the dependent computations can be minimised.

Lecture 22

Instruction Level Parallelism (Software pipelining and Trace Scheduling)

Recap:] √pm ""] √..ω"

Recap:

Eliminating Dependent Computations

Software pipelining

Trace Scheduling

Superblocks

Summary

Recap: Lecture 21

Last time we extended our discussion on the Static Scheduling to VLIW processors

A VLIW is formatted as one large instruction or

A fixed instruction packet with explicit parallelism among instructions in a set

The multiple operations are initiated in a cycle by the compiler which place them in a packet

Wider processors having multiple independent functional units are used to eliminate recompilation

Recap: Scheduling in VLIW processor

Compiler finds dependence and schedule instructions for parallel execution

It resulted in the improvement, compared to the superscalar processor in respect of:

- ✓ Average issue rate, i.e., operations issued per cycle; and
- ✓ Execution speed, i.e., the time to complete execution of code

However, the efficiency of VLIW, measured as the percentage of available slots containing an operation, ranges from 40% to 60%

Following this we also distinguished between Instruction Level Parallelism ILP and Loop Level Parallelism LLP

While talking about LLP we found that loop carried dependence prevents LLP but not ILP

At the end we studied the affine-based Greatest Common Divisor (GCD) algorithm to detect dependence in a loop

Continuing our discussion we will exploit how a compiler reduces the dependent computations

Reducing Dependent Computations

In order to achieve more ILP, compiler reduces the dependent computation by using Back Substitution technique

This results in algebraic simplification and optimization which eliminates operations that copy values to simplify the sequence

Copy Propagation

The approach of simplification and optimization is also referred to as the Copy Propagation

For example, in the sequence:

```
DADDUI    R1,R2,#4
```

```
DADDUI    R1,R1,#4
```

Here, the net use of R1 is to hold the result of second DADDUI operation, therefore, Substituting the result of first DADDUI operation in to the second results into

```
DADDUI    R1,R2,#8
```

Here, we have eliminated the multiple use of the register R1 during loop un-rolling

Conclusion:

Particularly, in case of memory access, this technique of reducing computations eliminates:

The multiple increments of array indices

- ✓ During the loop un-rolling; and
- ✓ To move increments across the memory addresses

Tree-Height Reduction Optimization

The copy-propagation technique reduces the number of operations or code length

The Optimization to increase parallelism of the code, however, is possible by

Tree-Height Reduction (Restructuring)

Increase the number of operations while the execution cycles are reduced

Such optimization is called tree- height reduction since

It reduces the height of tree structure representing a computation, making it wider but shorter

Optimization - Tree height reduction

For example, the code sequence

```
ADD      R1,R2,R3
```

```
ADD      R4,R1,R6
```

```
ADD      R8,R1,R7
```

Requires three cycles for execution

Because, here all the instructions depend on immediate predecessor and cannot be issued in parallel

Now taking the advantage of the associativity, the code can be transformed and written in the form shown as below,

```
ADD      R1,R2,R3
```

```
ADD      R4,R6,R7
```

```
ADD      R8,R1,R4
```

This sequence can be computed in two(2) execution cycles by issuing first two instruction in parallel

Conclusion: Detecting and Enhancing LLP

The analysis of LLP focuses on determining data dependence of:

- ✓ Some later iteration on to an earlier iteration

Such dependence is referred to as the Loop-Carried Dependence

Greatest Common Divisor test is defined to find the existence of dependence

Compiler techniques, such as:

- ✓ Copy propagation and
- ✓ Tree-Height Reduction

are discussed to eliminate dependent computations

Uncovering Instruction Level Parallelism

We have already discussed Loop Unrolling as the basic compiler technique to uncover ILP

We have observed that Loop unrolling with compiler scheduling enhances the overall performance of Superscalar and VLIW processors

Loop Unrolling: Review

Loop unrolling generates a sequence of straight line code uncovering parallelism among instructions

Here, to avoid a pipeline stall, the dependent instructions are separated from source instructions, by a distance in clock cycles equal to the pipeline latency of that source instruction

To perform this scheduling, compiler determines both:

- ✓ The amount of ILP available in the program; and
- ✓ The latencies of the functional units in the pipeline.

Advanced Compiler Techniques

Today will discuss two new compiler techniques to uncover the ILP

Software pipelining

Global Code Scheduling

- ✓ Trace scheduling
- ✓ Superblock

Software pipelining

Software pipelining is a technique where the loop is reorganized such that the code for each iteration is made by choosing instructions from different iterations of the original loop

A software pipelined loop interleaves instructions from different iterations without unrolling the loop

Software-pipelined Loop Formation

The instructions are chosen from different loop iterations, separating the dependent instructions within one iteration of the original loop

Example Code:

Let us reconsider our earlier loop unrolling and scheduling example of adding a constant F2 to an array R1

```

Loop  L.D      F0,0(R1)
      ADD.D    F4,F0,F2
      S.D      0(R1),F4
      DADDUI   R1,R1,#-8
      BNE     R1,R2, LOOP
  
```

For software pipelining, the compiler symbolically unroll the loop and schedule them
It selects the instructions from each iteration that do not have dependence among each other

The overhead instructions (DADDUI and BNE) are not replicated

Symbolic Loop Unrolling

The body of the symbolically unrolled loop for three iterations is as follows:

The instructions selected from each iteration are shown in yellow color

```

Iteration i:  L.D      F0,0(R1)
              ADD.D    F4,F0,F2
              S.D      F4,0(R1)
Iteration i+1: L.D      F0,0(R1)
              ADD.D    F4,F0,F2
              S.D      F4,0(R1)
Iteration i+2: L.D      F0,0(R1)
              ADD.D    F4,F0,F2
              S.D      F4,0(R1)
  
```

Software pipeline

The selected instructions from different iterations are put together in a loop with the loop control instructions as shown below

Loop:

S.D	F4,16(R1)	stores in M[i]
ADD.D	F4,F0,F2	adds to M[i-1]
L.D	F0,0(R1)	loads M[i-2]
DADDUI	R1, R1, # -8	
BNE	R1, R2, Loop	

Here, note that for start-up and finish-up we need some code that will not be executed

For start-up we need to run L.D and ADD.D instructions for iteration 1 and L.D for iteration 2

Similarly, for finish-up we need to execute the S.D instruction for iteration 2 and ADD.D and S.D for iteration 3

This loop takes 5 clock cycles to execute per result.

L.D and ADD.D are separated by offset of 16 to run the loop for three iterations (i.e., two iterations less than simple loop unrolling and scheduling case)

Here registers F4,F0 and R1 are reused as there is no data dependence in this case, thus WAR hazard is avoided.

Software pipelining vs. Loop unrolling

Software pipelining can be thought of as symbolic loop unrolling because here, some algorithms use loop unrolling techniques to software-pipeline loops

✓ Software pipelining consumes less space code as compared to loop unrolled

Software unrolling reduces the time, when the loop is not running at peak speed, to once per loop at the beginning and end. E.g.; if a loop is to do 100 iterations with 4 iteration symbolically unrolled; then we pay overhead for $100/4 = 25$ times instead of 100 times

This is shown in the fig. below

Symbolic Loop Unrolling Fill & drain pipe only once per loop vs. once per each unrolled iteration in loop unrolling

Global Code Scheduling

We have observed that:

- ✓ Loop unrolling and code scheduling works well when the loop-body is straight line code
- ✓ Software pipelining works well when the body is a single basic block, as it is easier to find repeatable schedules

Global Code is one where loop body has with internal control flow; i.e., a loop comprises conditional blocks such as if-then-else

The effective scheduling of such a code is complex, since it involves moving instructions across branches into the shortest possible sequence to compact a code fragment

A global code involves both the data dependence and control dependence

In order to enhance the ILP in global code, the loop unrolling and scheduling or software pipeline approaches do not work efficiently

These approaches are suitable to enhance ILP in straight-line or single basic block codes

where internal code of the loop have data dependence and the control dependence of the loop is overcome by loop unrolling

The most commonly used compiler based approaches, to schedule global code, are:

- ✓ Trace scheduling or critical path approach
- ✓ Superblock approach

Complexities of Global code scheduling

Before discussing the Global Code Scheduling techniques, let us first familiarize: the complexities of scheduling assignment instructions internal to a loop in condition branches of a global code

Let us consider a typical global - code fragment that represents an iteration of an unrolled inner loop

Here, moving (scheduling prior to condition evaluation) B or C requires more complex analysis

Effective scheduling of this code may require moving assignments to B and C prior to condition evaluation

Particularly, before movement of B, we must ensure that both the data flow and exception behavior is changed

To see, how compiler ensures data flow, let us have a look on the code sequence for this flow-chart fragment

```

LD      R4, 0(R1)    ;Load A
LD      R4, 0(R1)    ;Load B
DADDU   R4, R4, R5    ; A+B
SD      R4, 0(R1)    ;A=A+B
BNEA    R4, Else_part ;Is A=0?
:      :      ;      T part
SD      0" 0(R2)     ;store to B
J       Join         ; jump to F
Go to ex' 0
x      00            ; x code
:      :
Join:   0            ;after_if
SD      0."2*T5+     ; store c[i]

```

Complexities of Global code scheduling

```

LD      R4, 0(R1)    ;Load A
LD      R4, 0(R1)    ;Load B
DADDU   R4, R4, R5    ; A+B
SD      R4, 0(R1)    ;A=A+B
BNEA    R4, Else_part ;IF A=0?
:      :            ; T part
SD      0"2*T4+      ;store to B
J      Join          ; jump to F
Control flow
x      0             ; x code
:      :
Join:   0             ;after_if
SD      0"2*T5+      ; store c[i]

```

Let us see the effect of moving assignment to B before BNEA

Here, if B is referenced in X or after the IF statement, then moving B before IF will change the data flow

This can be overcome, however, by making a shadow copy of B before IF statement and use the shadow copy in x

However, such a copy is avoided as it slows down the program

```

LD      R4, 0(R1)    ;Load A
LD      R4, 0(R1)    ;Load B
DADDU   R4, R4, R5    ; A+B
SD      R4, 0(R1)    ;A=A+B
BNEA    R4, Else_part ;IF A=0?
:      :            ; THEN part
SD      0"2*T4+      ;store to B
J      Join          ; jump to F
Else_part 0          ;
x      0             ; x code
:      :
Join:   0             ;after_if
SD      0"2*T5+      ; store c[i]

```

Similarly, moving assignment to C before the first branch is more complex

It required two steps:

1. Assignment is moved over the join point of the ELSE part; i.e, in the portion of the THEN part. This movement makes instructions for C control dependent and is not executed if the ELSE path is chosen. Hence, to ensure correct execution, a copy of instruction is made in ELSE part
2. If C assignment is moved to before the IF Test provided it does not affect any data flow. In this case the copy of instruction in ELSE part is avoided

Our above discussion reveals that global code scheduling is an extremely complex problem

Global code scheduling

To simplify the complexities of Global-Code scheduling, we will discuss two methods. These methods rely on the simple principle: Focus the attention of the compiler on a straight line code segment that is estimated to be representing the most frequently executed code path.

Trace scheduling

Trace is a sequence of basic blocks whose operation could be put together into smaller number of instructions.

Trace scheduling is a way to organize global code motion process such that the cost of code motion is incurred by the less frequent paths.

It is useful for processors with a large number of issues per clock, where:

- ✓ Conditional or predicted execution is inappropriate or unsupported; and
- ✓ Simple loop unrolling is not sufficient to uncover ILP to keep processor busy.

Trace scheduling is carried in two steps:

1. Trace selection - to find likely sequence of basic blocks whose operation could be put together into smaller number of instructions.
2. Trace compaction: to squeeze the trace into a small number of wide instructions.

Trace Generation:

- ✓ Since the probability of loop branches-taken is usually high, so trace is generated by loop unrolling.
- ✓ Additionally static branch prediction is employed as taken or not-taken to obtain straight-line code by concatenating many basic blocks.

In our earlier global code fragment, if we take the true path as most frequent, then primary trace could be generated by unrolling the true path n- (say 4-) times. Here, trace exit is jump-off the frequent path and Trace Enter is return to trace.

Code Compaction is the code scheduling, hence, compiler attempts to:

- ✓ Move operations as early as it can in a sequence (trace); and
- ✓ Pack the operations into as few wide instructions (or issue packets) as possible.

Advantages of Trace scheduling

- ✓ Trace scheduling simplifies the decision concerning global code motion
- ✓ Branches are viewed as jumps into [Trace Entrance] or jump out of [Trace Exit] selected trace which the most probable path

Overhead of Trace scheduling

- ✓ Note that when code is moved across trace additional book keeping code is needed on entry or exit point
- ✓ Furthermore, when an entry or exit point is in the middle of a trace, significant overheads of compensation code may make trace scheduling an unattractive approach.

Superblocks

The draw back of Trace is that the entries into and exit out of a trace, in the middle of the trace cause significant complications; and

Compiler requires to generate and track the compensation code

This draw back is overcome by using Superblocks

Superblocks are a form of extended basic blocks, which have a single entry point but allow multiple exits

Therefore it is easier to compact superblocks as compared to a trace

In our earlier example Global Code, superblocks with one entrance can be easily constructed by moving C as shown

Here, Tail Duplication is used to create a separate block that corresponds to portion of the trace after the entry

Here, each unrolling of the loop creates an exit from the superblock to the residual loop that handles the remaining iterations

The residual loop handles the iterations that occur when the unpredicted path is selected

Advantages of Superblocks

This approach reduces the cost of book keeping and scheduling verse the more general trace generation

However, its code size may enlarge more than a trace based approach.

Like trace scheduling, superblocks scheduling may be most appropriate when other techniques fail.

Conclusion Enhancing ILP

All the four approaches

Loop unrolling,

Software pipelining,

Trace scheduling and

Superblocks

aim at trying to increase the amount of ILP, which can be exploited by a processor issuing more than one instruction on every clock cycle.

Numerical problems

In the following code:

1. List all dependences (output,anti,and true) in the
2. Indicate whether the true dependences are loop carried or not?
3. Why the loop is not parallel?

Example 1:

```
For (i=2; i<100; i= i+1)
{
    a[i]   =   b[i] + a[i];    /*s1*/
    c[i-1] =   a[i] + d[i];    /*s2*/
    a[i-1] =   2* b[i];        /*s3*/
    b[i+1] =   2*b[i];         /*s4*/
}
```

Solution:

There are six dependences in the loop.

1. There is antidependence from s1 to s1 on a.
 $a[i] = b[i] + a[i];$ /*s1*/
2. There is true dependence from s2 to s1.
 $a[i] = b[i] + a[i];$ /*s1*/
 $c[i-1] = a[i] + d[i];$ /*s2*/

Here, the value of a in s2 is dependent on the result of a in s1.

- 3 There is loop carried true dependence from s4 to s1 on b.
- 4 There is loop carried true dependence from s4 to s3 on b.
- 5 There is loop carried true dependence from s3 to s3 on b.
- 6 There is loop carried output dependence from s3 to s3 on a

Part b) Indicate whether the true dependences are loop carried or not?

We know that for loops to be parallel, each iteration must be independent of all others.

Here in this case, as dependences 3, 4, 5 are true dependences

They cannot be removed by renaming or any such technique

These dependences are loop carried as the iterations of the loop are not independent.

These factors imply the loop is not parallel as the loop is written.

Loop can be made parallel by rewriting the loop to find a loop that is functionally equivalent to the original loop that can be made parallel

Example 2:

The loop given below is a dot product (assuming the running sum in F2 initially 0) and contains a recurrence.

Assume the pipeline latencies from the table shown below, and a 1 cycle delayed branch.

Considering single issue pipeline.

Instruction producing result	Instruction using result	Latency in clock cycle
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Part a)

Unroll the loop sufficient number of times to schedule it without delay.

Show the schedule after eliminating any redundant overhead instructions.

Foo:

```

L.D      F0,0(R1);    /load X[i]
L.D      F4,0(R2);    /load Y[i]
MUL.D    F0,F0,F4;    /multiply X[i]*Y[i]
ADD.D    F2,F0,F2;    /add sum=sum + x[i] * y[i]
DADDUI   R1,R1,#-8;   /decrement X index i
DADDUI   R2,R2,#-8;   /decrement Y index i
BNEZ     R1,foo ;     /loop if not done

```

Solution:

This code has loop carried dependence from iteration i to i+1.

It also has high latency dependence within and between loop bodies.

Now if we unroll the loop twice in order to avoid any delay, we will get the following result.

```
foo
L.D      F0,0(R1)
L.D      F4,0(R2)
L.D      F6,#-8(R1)
MUL.D    F0,F0,F4      ;1 from L.D F4,0(R2)
L.D      F8,#-8(R2)
DADDUI   R1,R1,#-16
MUL.D    F6,F6,F8      ;1 from L.D F8,-8(R2)
ADD.D    F2,F0,F2      ;3 from MUL.D F0,F0,F4
DADDUI   R2,R2,#-16
Stall
BNEZ     R1,foo
ADD.D    F2,F6,F2      ; in slot, and
                        3 from ADD.D F2,F0,F2
```

Here the dependences chain from one ADD.D to the next ADD.D forces the stall.

Next part: In order to unroll further to schedule eliminating the stall (overhead) we take advantage of commutativity and associativity of dot product of two running sums in the loop

One for even elements and one for odd elements, and combine the two partial sums outside the loop body:

```
foo
L.D      F0,0(R1)
L.D      F6,-8(R1)
L.D      F4,0(R2)
L.D      F8,-8(R2)
MUL.D    F0,F0,F4      ;1 from L.D F4,0(R2)
MUL.D    F6,F6,F8      ;1 from L.D F8,-8(R2)
DADDUI   R1,R1,#-16
DADDUI   R2,R2,#-16
ADD.D    F2,F0,F2      ;3 from MUL.D F0,F0,F4
BNEZ     R1,foo
ADD.D    F2,F0,F2      ;3 from MUL.D F6,F6,F8
                        ;and fill the branch delay slot
ADD.D    F2,F0,F2      ;combine even and odd elements
```

Result discussion:

Here, the code assumes that the loop executes a non zero, even number of times.

The loop itself is stall free, but there are three stalls when the loop exists.

The loop body takes 11 clock cycles.

Part b)

The unrolled and scheduled code for the transferred code - Loop body takes 10 cycles

	integer	Inst	FP Inst	Clock Cycles
Foo	L.D	F0,0(R1)		1
	L.D	F6,-8(R1)		2
	L.D	F4,0(R2)		3
	L.D	F8,-8(R2)		4
	DADDUI	R1,R1,#-16	MUL.D F0,F0,F4	5
	DADDUI	R2,R2,#-16	MUL.D F6,F6,F8	6
	stall			7
	stall			8
	BNEZ	R1,foo	ADD.D F2,F0,F2	9
			ADD.D F2,F0,F2	10
Bar:	..	..		
			ADD.D F2,F0,F2	14

Example 3:

Consider a code

For (i=2; i<=100; i+=2)

a[i] = a[50*i+1]

Using GCD test, normalize the loop.

Start index at 1 and increment it by 1 on every iteration.

Write the normalized version of the loop then use GCD test to see if there is dependence.

Solution:

By normalizing the loop it leads to a modified C code as shown below,

For (i=1; i<=50; i++) { ;divide i by 2

a[2*i] = a[(100*i)+1] ; multiple constant by 2

}

The GCD test shows the potential for dependences within an array indexed by the function, $ai + b$ and $cj + d$

If the condition $(d-b) \bmod \gcd(c,a) = 0$ is satisfied

Applying GCD test, in that case we will get,

$a = 2, b = 0; c = 100, d = 1$

allows us to determine dependence in loop.

Thus gcd will be, $\gcd(2,100) = 2$

And: $d - b = 1$

Here, as 1 is factor of 2.

Thus, GCD test indicates that there is a dependence in the code.

In reality, there is no dependence in the code.

Since the loop load its value from

e]323a"e]423a e]7223a and

again these values to

e]4a"e]6a" 0e]322a

Lecture 23

Instruction Level Parallelism (Hardware Support at Compile Time)

Recap

H/W Support at Compile Time

Conditional/Predicated Instructions

H/W based Compiler Speculation

Summary

Recap: H/W and S/W Exploitation

We have studied both the Dynamic and Static scheduling techniques to exploit ILP for single or multiple instructions issue per clock cycle and to enhance the processor performance

The dynamic approaches use hardware modification which results in superscalar and VLIW processors

Furthermore, the pipeline structure enhancement such as

1. Branch predictors minimize the stalls due to the control hazards
2. Branch predictors minimize the stalls due to the control hazards

The static scheduling approaches include

1. Loop unrolling
2. Software Pipelining
3. Trace Scheduling
4. Superblock Scheduling

These techniques are focused to increase ILP by exploiting processor issuing more than one instruction every cycle

These techniques give better performance when the behavior of the branches is correctly predictable at the compile time

Otherwise, the parallelism could not be completely exposed at the compile time

This is due to the following two reasons

1. Control dependences limits the amount of the parallelism that can be exploited; and
2. Dependence between memory reference instructions could prevent code movement necessary to increase parallelism

Hardware Support for VLIW

These limitations, particularly for VLIW processor, could be overcome by providing hardware support at the compile time

Today, we will introduce some hardware support-based techniques to help:
overcoming these limitations; and
to expose more parallelism at the compile time

The most commonly used such techniques are:

Extension of the Instruction Set by including Conditional or Predicated (base something on something) Instructions

Hardware speculation to enhance the ability of compiler to move code over branches, while preserving exceptional behavior

To allow the compiler to reorder load/store instruction when no conflict is suspected but not certain

1: Instruction Set Extension

The extended instruction set including Conditional or Predicated Instructions

- ✓ allow the compiler to group instructions across branches
- ✓ eliminate branches
- ✓ convert control dependence into data dependence

These approaches are equally useful for hardware-intensive as well as software-intensive scheme, i.e., the dynamic as well as static scheduling

As discussed earlier, Predicate registers are included, in the structure of IA64 processor, to implement predicated instructions to improve performance

Conditional Instructions

Now let us discuss the concept behind introducing the conditional instructions in the instruction set

- ✓ The conditional instructions have an extra operand a one-bit predicate register
- ✓ A condition is evaluated as part of instruction execution to set the value of predicate-register

Conditional or predicted Instructions

In HPL-PD from HP Lab, the value of the predicate register is set by the instruction to-predicate operation;

$p1 = \text{CMPP} \leq r1, r2$

Here the predicate register p1 is set if r2 is $\leq$ r1

If condition is true ($p1=1$), the instruction is executed normally

If the condition is false ($p1=0$), the instruction execution continues as if the instructions were a no-operation

Typical conditional instructions for pipeline processors are:

Conditional Move CMOVZ R1, R2, R3;

- ✓ it moves the value from one register to another if the condition is true; i.e., third operand the predicate register R3 is Zero
- ✓ Such instructions are used to eliminate branch code sequence

Conditional ADD (R8) ADD R1, R2, R3

- ✓ assumes that the $R1 = R2 + R3$ occurs if the predicate register R8 is 1

Conditional Load LWC R1, 0(R2),

- ✓ R3 assumes that the load occurs unless the third operand R3 is Zero

The LW instruction, or a short block of code, following the branch can be converted to LWC and moved up to second issue slot to improve the execution time for several cycles

Example 1: Conditional or predicated Instructions

Let us consider the conditional statement: If $(A==0)$ { $S=T$; }
i.e., the value S is to be replaced by T if the value A is zero

Assuming the register R1, R2, R3 holds the value of A, S and T respectively.

The code to implement this conditional statements can be written as:

```
BNEZ R1, L      ; No-op if A (R1) != 0
ADDU R2, R3, R0 ; Else replace S (R2) by T (R3)
```

The IF statement can be implemented by the conditional move as:

```
CMOVZ R2, R3, R1
Move   R3 to R2
if the third operand R1=0
```

Here, notice that using the Conditional instruction CMOVZ, the next operation is determined by the contents of the third register instead of condition evaluation i.e., the control dependence has been converted to data dependence.

This transformation has moved the place to resolve dependence in a pipelined processor

We know that, in a pipelined processor the dependence for branches is resolved near the front of the pipe

Whereas, the conditional instruction resolve the dependence where the register-write occurs

This transformation is also used for vector computers, where it is called if-conversion

The if-conversion replaces conditional branches with predicated operations

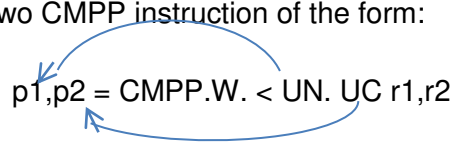
For example: Let see the code generated for the following two if-then-else statements

```
If (a<b) then c=a; else c=b;
If (d<e) then f=d; else f=e;
```

might be written as two VLIW instructions as

Alternatively two CMPP instruction of the form:

p1,p2 = CMPP.W. < UN. UC r1,r2



Example 2: Absolute Value Function

Conditional move is also used to implement the absolute value function: $A = \text{abs}(B)$

Which is implemented as:

if $B < 0$ { $A = -B$; } else { $A = B$; }

Where this statement can be implemented as

pair of conditional moves

CMOVZ R2, -R3, R1

CMOVZ R2, R3, R4

or one unconditional move $A = B$

and one conditional move $A = -B$

Conditional MOVE Instruction

The Conditional moves enables us to eliminate the branch and improves the pipeline behavior

It is useful for short sequences.

It has certain limitations

When predication is used to eliminate branches that guard the execution of large blocks of code, many conditional moves are needed

To remedy this inefficiency, some architectures supports full predication, i.e.; the execution of all the instructions is controlled by a predicated move

Full predictions allows to convert the large blocks of code that are branch dependent.

For example, if-then-else statement within a loop converted to predicated execution.

Where code executes only if the value of the condition is true.

The code in the else case executes only if the value of the condition is false.

Predicated LOAD Instructions

Now let us consider another example code sequence for a two-issue superscalar

The superscalar can issue a combination of one memory reference and one ALU operation or a branch by itself, every cycle.

2-Issue Instruction Code

First instruction slot	Second instruction slot
LW R1, 40(R2)	ADD R3,R4,R5
	ADD R6,R3,R7
BEQZ R10,L	
LW R8, 0(R10)	
LW R9, 0(R8)	

- ✓ How the code can be improved using a predicated form LW? And
- ✓ We can say that the transformation is speculative

Here, the second LW after the branch LW R9, 0(R8), depends on the prior load LW R8, 0(R10)

Therefore, this sequence wastes a memory operation slot in second cycle; and will incur a data dependence stall if the branch (BEQZ R10, L) is not taken.

Let us consider the predicated version load word LWC:

LWC R8, 0(R10), R10

and assume the load occurs unless the third operand R10 is 0

The LW instruction, LW R8, 0(R10), immediately following the branch, can be converted to an predicated load:

LWC R8, 0(R10), R10 and

moved up to the second issue slot; i.e.; prior to the BEQZ R10, L

First instruction slot	Second instruction slot
LW R1,40 R2)	ADD R3,R4,R5
LWC R8,0(R10),R10	ADD R6,R3,R7
BEQZ R10,L	
LW R9,0 (R8)	

Note that it improves the execution time by:

- ✓ eliminating the one instruction issue slot; and
- ✓ reducing the pipeline stall for last instruction in the sequence

Furthermore, note that if compiler mispredicated the branch, the mispredicated instruction will have no effect and will not improve the running time

Therefore we can say that the transformation of conditional load is speculative.

Advantages of predicated Code

Predicated or conditional instructions are extremely useful:

- ✓ for implementing short alternative control flows
- ✓ for eliminating some unpredictable branches; and
- ✓ for reducing the overhead of global code scheduling

Limitations on the conditional Instructions

To move an instruction across a branch and making it conditional will: slow the program whenever the move instruction would not have been normally executed

Predicating a control dependent and eliminating a branch may slow down the processor if that code would not have been executed

Predicated instructions are useful when predicate could be evaluated early

Conditional instruction results in a stall for data hazard, if the condition evaluation and predicated instructions are not separated

Conditional instruction may have some speed penalty compared to unconditional instructions

The use of Conditional instruction is limited when the control flow involves more than a simple alternative sequence

For example, moving an instruction across multiple branches requires making it conditional on both the branches

Thus, it requires to specify additional instructions to compute the controlling predicate

Architectures with Conditional Instructions

For these limitations most of the architectures include only a few, mostly CMOV, conditional instructions

The MIPS, Alpha, PowerPC, SPARC and Intel (Pentium) all support Conditional Move
IA-64 Micro Architecture supports full predication of all the instructions

Introduction to Compiler Speculation

In our earlier discussion, we have noticed that: where the programs have branches, which may be predicted at the compile time, either from the program structure or from program profile

Here, the compiler speculates to: either improve the scheduling and/or to increase the issue rate. However, the predicated instructions may help to speculate

These instructions are more useful when they can eliminate control dependence by if-conversion

Hardware support Speculation

Furthermore, in most of the cases we would have to move the speculated instructions before the condition evaluation

But, this cannot be done by predication

Rather, it motivates to have the following capabilities to speculate ambitiously

Compiler speculation with hardware support

- 1) The ability to find instruction can be speculatively moved and not affect the program data flow
- 2) The ability to ignore exceptions in speculated instructions, until we know such exception would really occur
- 3) The ability to speculatively interchange loads and stores, or stores and stores, which may have address conflicts.

Note the first one is the compiler's capability where as the other two can be achieved by Hardware support

Hardware speculation approach supports reordering loads and stores.

Hardware based speculative movement of instructions is done by checking for potential address conflicts at runtime; and

It allows the compiler to reorder loads and stores when it suspects they do not conflict.

Methods to preserve exceptions

The following are four hardware methods to support more ambitious speculation without introducing erroneous exception behavior

The key to these methods is to observe that the results of speculated sequence that is mispredicted will not be used in the final computation, for this purpose the exceptions are preserved

Methods to provide Hardware support

- 1) The hardware and operating system cooperatively ignores the exception for speculative instructions for the incorrect program

Here, the exception behavior for the correct program is preserved and for incorrect one is ignored

VI nWet t vsegl "nWyi h"ewE" jewXq s hi "yr hi vX i "t vskvegsr xsp

Methods to Preserve exceptions

The examples of exceptions that indicate a program error and normally cause termination are memory protection violation

The examples of exceptions that handle the program error and normally resumed are page fault

- 2) Speculative instructions that never raise exception are used; and checks are introduced to determine when exception should occur
- 3) C'wi x'sj'f nWgepi h" t smsr "f nW "evi "exegl i h"sXvi Wyp'vi knwi v'VI i wi "f nWevi "{ vxi r" by speculated instructions when the instruction causes exceptions.

The poison bits cause a fault when a normal instruction attempts to use the register

This approach suggests to track the exceptions as they occur but postpone any terminating exception until a value is actually used

The scheme adds a poison bit to every register and another bit to every instruction to indicate if whether the instruction is speculative

The poison bit of the destination register is set whenever a speculative instruction results in terminating exception

And all other exceptions are handled immediately

If the speculative instruction uses a register with poison bit on; the destination register has its poison bit on

Now if the normal instruction attempts a register source with poison bit on, the instruction causes a fault

- 4) A mechanism is provided to indicate: that an instruction is speculative; and the hardware buffers the instruction result until it is certain that the instruction is no longer speculative.

Summary

Both the hardware and software mechanisms provides approaches to exploit ILP.

There are certain limitation on both mechanisms

We will discuss these limitations next time

Lecture 24

Instruction Level Parallelism (Concluding Instruction Level Parallelism)

Recap

Recap

Compile Time H/W Support:

- ✓ To Preserve Exceptions - Typical Examples
- ✓ For memory Reference Speculation

Speculation Mechanism: H/W Vs. S/W

Summary

Recap: Compile Time H/W Support

Last time we discussed the methods to provide H/W support for exposing more parallelism at the compile time

We introduced the concept of extension in the Instruction set by including Conditional or predicated instructions

We found that such instructions can be used to eliminate branches and to convert control dependence to data dependence which is relatively simple to handle

Thus, it improves the processing performance

We also introduced the hardware and software-based abilities required to: move the speculated instructions before the branch condition evaluation while preserving the exception behavior

We further introduced four methods to support speculation; and, move of instruction such that the miss-predicted speculated sequence is not used in the final execution;

But, the exception behavior is preserved to take care of the exceptions that may be used later

However, in order to study these methods we distinguish between the exceptions that indicate program error and:

- ✓ normally cause termination; or
- ✓ handle the error to resume normally

Typical example of the behavior-exception that indicate the program error and terminates is memory protection violation

The result of a program that gets such an exception is not well defined, therefore if such an exception arise in speculated instructions, we cannot take the exception, hence need not preserve such an instruction

The example of the behavior-exception that indicates the program error, which could be handled and program normally resumes is Page Fault in virtual memory

Such exceptions are preserved and are processed for speculative instruction as if they were normal instructions when the instruction is no longer speculative

Methods to preserve exception behavior

Let us study in details, the methods to preserve exceptions.

The four methods which we introduced last time are:

1. Fast mode approach

2. Speculative-instructions method
3. Poison-bit Register method
4. Hardware (Re-order) buffering

Methods 1: Fast Mode

The simplest approach to preserve exceptions is one where hardware works cooperatively with the operating system and: handle presumable exceptions under the program control

Here, the H/w and S/w preserves the exception behavior for the correct program, but ignores the exception behavior for the incorrect programs

Before we proceed further, let us define a correct and an incorrect program

Correct Program

- ✓ A program may be correct if the instruction generating the terminating exception is speculative; and the speculative result is simply unused
- ✓ i.e., the speculative instruction returns a value that is not harmful to the program

Incorrect Program

- ✓ A incorrect program is one which previously have received a terminating exception, and
- ✓ will get an incorrect result in the present instruction
- ✓ Thus, the exception behavior is mispredicted

Fast Mode to preserve exception

- ✓ Hence, the exception behavior as a result of speculated instruction will not be used in the final computation if mispredicted;
- ✓ Now let us consider a typical correct code fragment for the if-then-else statement of the form:

Example Code fragment for Fast Mode

```
If (A==0) A = B;
Else A = A + 4;
```

And see how this code can be modified to preserve behavior exceptions

Assuming that A is at 0(R3) and B is at 0(R2); the code based on the speculation that the THEN clause is almost always executed, can be written as follows:

```
LD      R1,0(R3) ;load A
BNEZ    R1,L1    ;test A
LD      R1,0(R2) ;then clause
J       L2       ;skip else
L1:     DADDUI   R1,R1,#4 ;else clause
L2:     SD       R1,0(R3) ;store A
LD      R1,0(R3) ;load A
```

Speculate that THEN clause is almost always executed i.e., program will rarely branch to else clause

```

        BNEZ    R1,L1    ;test A
        LD      R1,0(R2) ;then clause
        J       L2       ;skip else
L1:     DADDUI  R1,R1,#4  ;else clause
L2:     SD      R1,0(R3)  ;store A

```

therefore the value B at 0(R2) must be preserved

Revised Code fragment with Exceptions preserved

Now in order to preserve the speculation we use a temporary register say R14, to avoid destroying R1 when B is being loaded as in initial code

```

        LD      R1,0(R3)    ;load A
        LD      R14,0(R2)   ;speculative load B
        BEQZ    R1,L3       ;other branch of if
        DADDUI  R14,R1,#4    ;else clause
L3:     SD      R14,0(R3)    ;non-speculative store
L2:     SD      R1,0(R3)     ;store A

```

The speculative value B at 0(R2) is preserved in temporary register R14 to be used in THEN clause

Example Code fragment

- ✓ Here, is a new code that preserve the speculation in a temporary register R14, to avoid destroying R1 when B is loaded

```

        LD      R1,0(R3)    ;load A
        LD      R14,0(R2)   ;speculative load B
        BEQZ    R1,L3       ;other branch of IF
        DADDUI  R14, R1,#4   ;else clause
L3:     SD      R14,0(R3)    ;THEN clause: non-speculative store A

```

Method 2: Speculative Instructions

This approach introduces speculative version of instructions, such as

- ✓ Speculative Load sLD
- ✓ Speculative Check SPECCK

These instructions are used to preserve exception behavior exactly rather than speculatively

These instructions

- ✓ For sLD, the processor checks for such exceptions

The combined use of two or more speculative instructions, such as sLD and SPECCK, preserve the exception behavior exactly

Example 2

Now let us reconsider our earlier example of IF-THEN-ELSE statement to see how it can be coded, using speculative load and check instructions to preserve exception behavior exactly

Here, we assume that the temporary register R14 is unused and available

Revised code using speculative instructions

```

LD      R1,0(R3)    ;load A
sLD     R14,0(R2)   ;speculative,
                    no termination

BNEZ    R1, L1      ;test A
SPECCK  0(R2)      ;speculative check
J       L2          ;skip else
L1:     DADDUI      R1,R1,#4 ;else clause
L2:     SD          R1,0(R3) ;store A

```

Speculate Branch taken/not-taken (in this example Not-Taken) and save value B at 0(R2) in temporary register R14; i.e., the basic block for THEN (in this case the value B) is maintained

Check if the speculation is correct or not, i.e., Branch has taken or otherwise

Here, the load instruction speculate in respect of the branch instruction whether it will be taken or not-taken

The speculation check requires to maintain a basic block for THEN clause, thus preserve the exception behavior

Notice that extra code is required to check for possible exception thus, result in overhead

Method 3: Poison Bit

The basic concept behind this approach is that the speculative movement of instructions, before branch or before reordering of load/store, must not cause exceptions

If an exceptional condition occurs during a speculative operation, the exception is not raised

Here, the register to indicate whether the instruction is speculative

These bits are used to track exceptions as they occur; but postpones any terminating exception until a value is actually used

The speculative bits are simply propagated by speculative instructions

When a bit is set in the result register it indicates that an exceptional condition has occurred

When non-speculative operation encounters a register with speculative bit being set, an exception is raised

The poison-bits cause a fault when a normal instruction attempts to use the register

Sequence of steps to preserve exception behavior using poison bits

1. The poison bit of the destination register is set whenever a speculative instruction results in terminating exception
2. All other exceptions are handled immediately
3. IF the speculative instruction uses a register with poison-bit turned ON; the destination register of the instruction simply has its poison-bit turned ON
4. IF the normal instruction attempts to use a register source with poison-bit turned ON, the instruction causes a fault

Resulting Behavior: Poison-bits Approach

Thus, any program that would have generated an exception still generates one;
 Although, at the first instant where a result is used by an instruction that is not speculative
 Furthermore, STORES are never speculative as the poison-bits exist on registers only

Example 3: Poison Bits Approach

Let us once again consider our earlier example and:

- ✓ See how it would be compiled with speculative instructions SLD and register R14 with poison-bit; and
- ✓ Show where an exception for the speculative memory reference would be recognized

IF-THEN-ELSE Statement Reconsidered

```
IF (A==0) THEN A = B;
ELSE A = A + 4;
```

Assume A is at 0(R3) and B is at 0(R2); and that R14 is unused and available

```
LD      R1,0(R3) ;load A
sLD     R14, 0(R2) ;
BEQZ    R1,L3
DADDUI  R14,R1,#4
L3:     SD      R14,0(R3);
```

Value B at 0(R2) is speculatively loaded in R14
 The poison bit of R14 is turned on if terminating exception is generated by speculative LOAD

When non-speculative SW occurs, exception for speculative LW is raised if poison bit for R14 is ON

Note here, If the speculative sLD generates a terminating exception, the poison bit of the R14 will be turned on

When the non-speculative SW instruction occurs, it will raise an exception if the poison bit for R14 is turned on

Furthermore, to facilitate the operating system to save the user register on a context switch if the poison bit is set,

a special instruction is included in the instruction-set to set/save the state of the poison-bit

Method 4: Hardware Buffering

1. A hardware buffer, such as the reorder buffer, is provided and the compiler:
2. Marks instructions as speculative
3. Includes an indicator of how many branches the instruction was moved speculatively
 - What branch action (Taken / not-taken) is assumed by the compiler?
 - The compiler accomplishes this by using either of the following two approaches:
 - For single branch: only 1 bit is used to tell whether the speculated instruction has come from the branch taken or not-taken path

- Alternatively: The original location of the speculative instruction is marked by a sentinel; i.e., looked out or guarded. It tells the hardware that the earlier speculative instruction is no longer speculative, so the values may be committed
- 4. The instructions marked as speculative are placed in the re-order buffer
- 5. The re-order buffer is flushed when the branch is resolved
- 6. The speculative instructions are not allowed to commit until:
 - Either the branches have been speculatively moved and are ready to commit;
 - Or corresponding sentinel is reached
- 7. At this stage, we should know whether the speculated instruction should have been executed or not
- 8. If it has been executed and has generated terminating exception then the program should be terminated
 - If not yet executed then exception is ignored

Memory Reference Speculation

With so much discussion in respect of preserve exception behavior of speculated instructions using hardware support, let us extend our discussion on the compile time hardware support for memory reference speculation

H/W support for Memory Disambiguation (Address Certainty)

While reducing the critical path length of the code, i.e., to optimize codes, it may be desirable to move the LOAD(s) across STORE(s)

For example, in the following code segment if the latency of LOAD is high, it may be desirable to move LOAD before STORE

In order to optimize codes with memory reference instructions, the critical path length of the code is reduced

This is accomplished mostly by moving memory reference instructions with longer latency, such as the LOAD across the instruction which has relatively lesser latency, i.e., STORE instruction

For example, in the following code segment;

if the latency of LOAD is high, it may be desirable to move LOAD before STORE for desirable optimization

However, this optimization is not valid if LOAD and STORE reference the same location; i.e., if R2 and R4 contain the same address

And, the compiler is not certain about this at the compile time

This issue can be resolved by providing run-time memory address (certainty) disambiguation

In order to resolve this issue, two special instructions are included in the ISA

View "Load Speculatively"

LDS R1, 0(R2) ;Load Speculatively

- This instruction initiates a load a normal load instruction
- A log entry is made in a table to store the memory location

LDV R2, 0(R2) ;Load Verify

- This instruction checks to see if store to memory location has occurred since the LDS
- View "Load Verify"

Here, LOAD is speculatively moved across the STORE, and

The LDV instruction is left at the original place of LOAD instruction (i.e., after the STORE instruction, in this example) which acts like a guardian and checks for the address conflict

When the speculated load LDS is executed, the hardware saves the address of the accessed memory location in log table and

1. If the subsequent STORE changes the location before the check instruction LDV, speculation is unsuccessful
2. If the subsequent STORE does not change the location before the check instruction LDV, then speculation is successful

Now, let us see how the speculation failure is handled?

It is handled in one of the two ways!

1. If only the load instruction was speculated then it is sufficient to redo the load
 - This is accomplished by using LDV instruction, which supplies target register in addition to memory address
2. If the additional instruction (LDV) is also speculated then
 - re-execute all the speculated instruction, starting with the LOAD, using a fix-up sequence
 - Here, the check instruction (LDV) specifies the address of the fix-up code

We have discussed the use hardware-intensive support with software intensive approaches to achieve ILP

Here, we observed that the improvement of code by using predicted version of load-word (LWC) instructions, which facilitate to move load instruction before branch

Example: Conditional Moving up the Branch

This approach improves the execution time

However, such transformations are speculative and no improvement in the run-time is possible if the compiler mispredict the branch

This motivates to use conditional move instruction to:

- eliminate branch; and to
- guard against the memory access violation that may terminate the program

In order to explain the use of conditional move instruction, let us reconsider the example code sequence for a 2-issue superscalar, which we discussed last time

Here, the second LW after the branch depends of the prior load, therefore, this sequence wastes a memory operation slot and incur a data dependence stall if branch is not taken

First instruction slot	second instruction slot
LW R1,40 R2)	ADD R3,R4,R5
	ADD R6,R3,R7
BEQZ R10,L	
LW R8,0(R10)	
LW R9,0 R8)	

Here, we discussed that if the LW immediately following the Branch is converted to predicted version of load-word (LWC) instruction, move it up by two slot, and assume load occurs unless third operand (R10) is ZERO, then it improve the execution time

First instruction slot	second instruction slot
LW R1,40 R2)	ADD R3,R4,R5
LW C R8,0(R10), R10	ADD R6,R3,R7
BEQZ R10,L	
BEQZ R10,L	
LW R9,0 R8)	

Here, the branch is written to skip the LW instruction if $R10 = 0$; and this instruction $LW\ R8, 0(R10)$ is executed unconditionally

This is likely to cause a protection exception which should not occur

Now let us see how we can re-write the code using conditional move instruction

The code should be written assuming that the loads, which are no longer control dependent, cannot raise an exception if they should not have been executed, i.e.,

The branch must guard against a memory access violation

As in this example code, a violation would terminate the program, therefore, while re-writing the code, we must ensure that the branch is written to skip the LW instruction if $R10 = 0$; and this instruction $LW\ R8, 0(R10)$ is executed unconditionally

Here, in order to guard the load by conditional move instruction, we need two unassigned registers;

- One of the register must contain a safe address for the load (Let us use register R29) instruction $[LW\ R8, 0(R10)]$
- The other register must save the original contents of R8 (Let us use the register R30 for this purpose)

L my .xq'J√z' pAYq ū'qp'b√pq

```

DADDI    R29,R0,#1000    ;initialize R29 to a safe address
LW        R1,40(R2)       ;first load instruction of original code
MOV       R30,R8          ;save R8 in unused R30
CMOVNZ   R29,R10,R10
LW        R8,0(R29)       ;speculative load
CMOVZ    R8,R30,R10
BEQZ     R10,L
LW        R9,0(R8)

```

R29 is unused and contain a safe address

$R29 \leftarrow R10$ if R10 contains a safe address 2

If $R10=0$ load is incorrectly speculated so restore R8

Here, the Load after the branch can be speculated using one more conditional move, and for this one more unused register will be needed

However, there is a significant conditional instruction overhead

Using R31 as the unused register, the branch free code is as follows

```

ADDI     R29,R0,#1000
LW       R1,40(R2)
MOV      R30,R8
MOV      R31,R9          ;save R9 in unused R31
CMOVNZ   R29,R10,R10
LW       R8,0(R29)
LW       R9,0(R8)        ;load speculated
CMOVZ    R8,R30,R10
CMOVZ    R9,R31,R10      ;restore R9, if needed

```

Summary: Hardware versus software speculation

Now while concluding our discussion on exploiting the ILP using hardware and software techniques we can say that both the approaches have certain limitations

These limitations can be summarized as follows:

1. To speculate extensively, we must be able to ascertain the memory references, which is difficult to do at the compile time
 - In a hardware based schemes, dynamic run time certainty of memory address is required. This is not possible in hardware. Hence, we have to move loads past/before stores at run time
2. Hardware based speculation works better when:
 - when control flow is unpredictable
 - And when hardware-based branch prediction is superior to software-based branch prediction, which is done at compile time
3. Hardware based speculation maintains a completely precise exception model for speculated instructions
4. Hardware based speculation does not require compensation which is needed by ambitious software speculation mechanism
5. Compiler based approaches have the ability to see further in the code sequence, Thus, may provide better code scheduling than a purely hardware driven approach; for example, use of conditional move instruction
6. Hardware based speculation with dynamic scheduling does not require different code sequences to achieve good performance for different implementation of architecture. However, the major disadvantage of hardware support for speculation is the extra hardware resources and Complexity of the structure

Lecture 25

Memory Hierarchy Design (Storage Technologies Trends and Caching)

] √pm ""] √..ω"

Recap: Processor Performance
What is outside the processor
Storage Technologies
RAM and Enhanced DRAM
Disk Storage
Summary

Recap: Processor Performance

Data path and control design of processor
Hardware-based and software-based techniques to expose the ILP

What is outside processor?

Performance of a computer

- ✓ what is outside the processor?
- ✓ how the outside systems influence the performance of processors?

Whatever is outside the processor is referred to as the I/O system

The I/O systems include:

- ✓ Memory System
- ✓ Buses and controllers

Memory System: An introduction

Memory system design for high performance computers

Design goal of memory system for high performance computers is to present the user with as much memory as is available

Fast memory is expensive and cheap memory is slow, therefore a memory hierarchy is organized into several levels

- ✓ cost as low as of the cheapest memory and
- ✓ speed as fast as of the fastest memory

Principle of Memory hierarchy

What is the principle of memory hierarchy? Where each type of the module is located in the memory system?

Principle of Locality

Advantage of the principle of locality

An average access speed that is very close to the speed that is offered by the fastest technology and

- ✓ Cost that is very close to the cheapest technology

Storage Type in Memory System

The semiconductor memories such as registers, Static RAM and Dynamic RAM fast in access speed but are expensive

Used in small size and placed either inside or closest to the processor

Secondary storage devices which offer huge storage at lowest cost per bit are placed farthest away from the processor

Levels of the Memory Hierarchy

Memory Hierarchy Pyramid

Storage Systems

Classification of memory systems based on different attributes and their design

Attributes:

- ✓ Material Semiconductor, magnetic, optical
- ✓ Accessing Random, Sequential, Hybrid
- ✓ Store/Retrieve ROM, RMM and RWM

Random-Access Memory (RAM)

Key features

- ✓ RAM is packaged as a chip.
- ✓ Basic storage unit is a cell (one bit per cell).
- ✓ Multiple RAM chips form a memory

Types of RAM

- ✓ Static RAM (SRAM)
- ✓ Dynamic RAM (DRAM)

Static RAM: Basic Cell

Write:

1. Drive bit lines (bit=1, bit=0)
2. Select row

Read:

1. Pre-charge bit and bit to Vdd
2. Select row
3. Cell pulls one line low
4. Sense amp on column detects difference between bit and bit

SRAM Organization: 16-word x 4-bit

Dynamic Random Access Memory (DRAM)

Each cell stores bit with a capacitor and transistor.

Value must be refreshed every 10-100 ms.

Sensitive to disturbances.

Slower and cheaper than SRAM.

Basic DRAM Cell

Write:

- ✓ Drive bit line and Select row

Read:

- ✓ Pre-charge bit line to Vdd and Select row
- ✓ Cell and bit line share charges
 - Here, very small voltage changes occurs on the bit line therefore Sense amplifier is used to detect changes
- ✓ Apply Write to restore the value

Refresh

- ✓ Just do a dummy read to every cell.

The state of the art DRAM cell only has one transistor. The bit is stored in a tiny transistor.

The write operation is very simple. Just drive the bit line and select the row by turning on this pass transistor.

For read, we will need to precharge this bit line to high and then turn on the pass transistor.

This will cause a small voltage change on the bit line and a very sensitive amplifier will be used to measure this small voltage change with respect to a reference bit line.

Once again, the value we stored will be destroyed by the read operation so an automatic write back has to be performed at the end of every read.

DRAM Organization 16 words x 8bit

Reading DRAM Supercell (2,1)

Step 1(a): Row access strobe (RAS) selects row 2.

Step 1(b): Row 2 copied from DRAM array to row buffer.

Step 2(a): Column access strobe (CAS) selects column 1.

Step 2(b): Supercell (2,1) copied from buffer to data lines, and eventually back to the CPU.

64 MB DRAM Memory Module [8x8MB DRAM Chips]

Enhanced DRAMs

1. Fast Page Mode DRAM (FPM DRAM)

- ✓ In normal DRAM, we can only read and write M-bit at a time because only one row and one column is selected at any time by the row and column address
- ✓ In other words, for each M-bit memory access, we have to provide a row address followed by a column address.

Page Mode DRAM: Motivation

Regular DRAM Organization:

- ✓ N rows x N column x M-bit
- ✓ Read & Write M-bit at a time
- ✓ Each M-bit access requires a RAS / CAS cycle

Fast Page Mode DRAM

- ✓ P" | " O" i k m i v " x s" wezi " e" row

2. Extended data out DRAM (EDO DRAM)

- ✓ It is the Enhanced FPM DRAM with more closely spaced CAS signals

3. Synchronous DRAM (SDRAM)

- ✓ It is driven with rising clock edge instead of asynchronous control signals.

4. Double Data Rate synchronous DRAM (DDR SDRAM)

- ✓ It is Enhancement of SDRAM that uses both clock edges as control signals.

5. Video RAM (VRAM)

- ✓ It is like FPM DRAM, but output is produced by shifting row buffer; and is
- ✓ Dual ported to allow concurrent reads and writes

Nonvolatile Memories

DRAM and SRAM are volatile memories

- ✓ Lose information if powered off.

Nonvolatile memories retain value even if powered off.

- ✓ Generic name is read-only memory (ROM).
- ✓ Misleading because some ROMs can be read and modified.

Types of ROMs

- ✓ Programmable ROM (PROM)
- ✓ Erasable programmable ROM (EPROM)
- ✓ Electrically erasable PROM (EEPROM)
- ✓ Flash memory

Firmware

- ✓ Program stored in a ROM
 - Boot time code, BIOS (basic input/output system)
 - graphics cards, disk controllers

Disk Geometry

Disk Geometry (Multiple-Platter View)

Aligned tracks form a cylinder.

Disk Capacity

Capacity: maximum number of bits that can be stored.

- ✓ Recording density (bits/in): number of bits that can be squeezed into a 1 inch segment of a track.
- ✓ Track density (tracks/in): number of tracks that can be squeezed into a 1 inch radial segment.
- ✓ Aerial density (bits/in²): product of recording and track density.

Disk Operation (Multi-Platter View)

Disk Access Time

Average time to access some target sector approximated by :

- ✓ $T_{\text{access}} = T_{\text{avg seek}} + T_{\text{avg rotation}} + T_{\text{avg transfer}}$

Seek time ($T_{\text{avg seek}}$)

- ✓ Time to position heads over cylinder containing target sector
- ✓ Typical $T_{\text{avg seek}} = 9 \text{ ms}$

Rotational latency ($T_{\text{avg rotation}}$)

- ✓ Time waiting for first bit of target sector to pass under r/w head.
- ✓ $T_{\text{avg rotation}} = \frac{1}{2} \times \frac{1}{\text{RPMs}} \times 60 \text{ sec/1 min}$

Transfer time ($T_{\text{avg transfer}}$)

- ✓ Time to read the bits in the target sector.

$T_{\text{avg transfer}} = \frac{1}{\text{RPM}} \times \frac{1}{(\text{avg \# sectors/track})} \times 60 \text{ secs/1 min.}$

Disk Access Time Example

Given:

- ✓ Rotational rate = 7,200 RPM
- ✓ Average seek time = 9 ms.
- ✓ Avg # sectors/track = 400.

Derived:

- ✓ $T_{\text{avg rotation}} = \frac{1}{2} \times \left(\frac{60 \text{ secs}}{7200 \text{ RPM}} \right) \times 1000 \text{ ms/sec} = 4 \text{ ms.}$
- ✓ $T_{\text{avg transfer}} = \frac{60}{7200 \text{ RPM}} \times \frac{1}{400 \text{ secs/track}} \times 1000 \text{ ms/sec} = 0.02 \text{ ms}$
- ✓ $T_{\text{access}} = 9 \text{ ms} + 4 \text{ ms} + 0.02 \text{ ms}$

Important points:

- ✓ Access time dominated by seek time and rotational latency.
- ✓ First bit in a sector is the most expensive, the rest are free.
- ✓ SRAM access time is about 4 ns/doubleword, DRAM about 60 ns
 - Disk is about 40,000 times slower than SRAM,
 - 2,500 times slower than DRAM.

Logical Disk Blocks

The set of available sectors is modeled as a sequence of b-sized logical blocks (0, 1, 2, ...)

Mapping between logical blocks and actual (physical) sectors

- ✓ Maintained by hardware/firmware device called disk controller.
- ✓ Converts requests for logical blocks into (surface, track, sector) triples.

CPU-Memory Gap

Summary

Memory hierarchy organization

Design of basic memory modules of DRAM and SRAM

Design and working of disk storages

Gap between the speed of processor and the storage devices - DRAM, SRAM and Disk is increasing with time

Lecture 26
Memory Hierarchy Design
(Concept of Caching and Principle of Locality)

]√pm "]√..ω"

Recap: Storage trends and memory hierarchy
Concept of Cache Memory
Principle of Locality
Cache Addressing Techniques
RAM vs. Cache Transaction
Summary

Recap: Storage Devices

Design features of semiconductor memories
SRAM
DRAM
Magnetic disk storage

Speed and Cost per byte

DRAM is slow but cheap relative to SRAM
Main memory of the processor to hold moderately large amount of data and instructions
Disk storage is slowest and cheapest
Secondary storage to hold bulk of data and instructions

CPU-Memory Access-Time

The gap between the speed of DRAM and Disk with respect to the speed of processor, as compared to that of the SRAM, is increasing very fast with time

CPU-Memory Gap

Memory Hierarchy Principles

The speed of DRAM and CPU complement each other

Organize memory in hierarchy, based on the Concept of Caching; and

- ✓ Principle of Locality

1: Concept of Caching

staging area or temporary-place to:

- ✓ store frequently-used subset of the data or instructions from the relatively cheaper, larger and slower memory; and
- ✓ To avoid having to go to the main memory every time this information is needed

Caching and Memory Hierarchy

Memory devices of different type are used for each value k the device level

the faster, smaller device at level k , serves as a cache for the larger, slower device at level $k+1$

The programs tend to access the data or instructions at level k more often than they access the data at level $k+1$

Storage at level $k+1$ can be slower, but larger and cheaper per bit

A large pool of memory that costs as much as the cheap storage at the highest level (near the bottom in hierarchy)

serves data or instructions at the rate of the fast storage at the lowest level (near the top in hierarchy)

Examples of Caching in the Hierarchy

2: Principle of Locality

Programs access a relatively small portion of the address space at any instant of time

E.g.; we all have a lot of friends, but at any given time most of us can only keep in touch with a small group of them

Types of Locality

Temporal locality is the locality in time which says if an item is referenced, it will tend to be referenced again soon.

Spatial locality is the locality in space. It says if an item is referenced, items whose addresses are close by tend to be referenced soon

A well-written program tends to reuse data and instructions which are:

- ✓ either near those they have used recently
- ✓ or that were recently referenced themselves

Spatial locality: Items with nearby addresses (i.e., nearby in space) be located at the same level, as they tend to be referenced close together in time

Temporal locality: Recently referenced items (i.e., referenced close in time) be placed at the same memory level, as they are likely to be referenced in the near future

Locality Example: Program

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

Spatial Locality: All the array-elements $a[i]$ or data, reference in succession at each loop iteration, so all the array elements be located at the same level. All the instructions of the loop are referenced repeatedly in sequence therefore be located at the same level

Temporal Locality: The data, sum is referred each iteration; i.e., recently referred data is referred in each iteration. The Instructions of a loop, $\text{sum} += a[i]$ Cycle through loop repeatedly

Based on Locality Principle How Memory Hierarchy works?

The memory hierarchy will keep the more recently accessed data items closer to the processor because chances are the processor will access them again soon
NOT ONLY do we move the item that has just been accessed closer to the processor, but we ALSO move the data items that are adjacent to it

Hierarchy List

Register File	Level 0	Datapath
L1	Level 1	Cache on Chip
L2	Level 2	External Cache
Main memory	Level 3	System Board DRAM
Disk cache	Level 4	Disk drive
Disk	Level 5	Magnetic disk
Optical	Level 6	CDs etc- bulk storage
Tape	Level 7	Huge cheapest Storage

Intel Processor Cache

80386 no on chip cache

80486 8k byte lines

Pentium (all versions)

- ✓ two on chip L1 caches
- ✓ Data & instructions

Pentium 4

L1 caches Two 8k bytes

L2 cache 256k

- ✓ Feeding both L1 caches

Cache Devices

Cache device is a small SRAM which is made directly accessible to the processor; and DRAM, which is accessible by the cache as well as by the user or programmer, is placed at the next higher level as the Main-Memory

Larger storage such as disk, is placed away from the main memory

Caching in a Memory Hierarchy

Cache Organization

Cache Addressing Direct Addressing

Memory Hierarchy Terminology

Hit: the data the processor wants to access appears in some block in the upper level (example: Block X)

- ✓ Hit Rate: the fraction of memory access that are found in the upper level (i.e., HIT)
- ✓ Hit Time: Time to access the upper level which consists of
 - i. RAM access time
 - ii. Time to determine if this is hit or miss

Miss: data needed by the processor is not found in the upper level and has to be retrieved from a block in the lower level (Block Y)

- ✓ Miss Rate = $1 - (\text{Hit Rate})$
- ✓ Miss Penalty is the sum of the time:
 - i. to replace a block in the upper level
 - ii. to deliver the block the processor

Recommendation: Hit Time must be much much smaller than Miss Penalty, otherwise no need for memory hierarchy

Cache Hit

CPU needs object d, which is stored in some block b, say 14 of the (k+1) memory and corresponding block 2 of the cache at level k

- ✓ Program finds block b (14) in the cache at level k.
- ✓ Object d is transferred to CPU

Cache Miss

Program needs object A, which is stored in some block C say block 12 at level K+1
Cache miss

- ✓ Block C (12 from K+1) is not at level k- It is cache Miss
- ✓ Hence, level k cache must fetch it from level k+1; and
- ✓ transfer object A to the CPU

Placement and Replacement Policies

If level k cache is full, then some current block must be replaced (evicted), which one is to be replaced? This is the problem of replacement policy.

- ✓ Cache design that defines the relationship of cache addresses with the higher level memory addresses
- ✓ Placement policy that determines where can the new block go? and
- ✓ Replacement policy that defines which block should be evicted?

Types of misses

Cold (compulsory) miss: Occurs when the cache is empty; at the beginning of the cache access.

Capacity miss: Occurs when the set of active cache blocks (working set) is larger than the cache.

Conflict miss: Occurs when the level k cache is large enough, but multiple data objects all map to the same level k block.

Example: Conflict Miss

If the placement policy is based on the direct addressing, then:

- ✓ Block n at level $k+1$ must be placed in block $(n \bmod 4)$ at level k

In this case, referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time as $8 \bmod 4 = 0$, as both the blocks 0 and 8 of level $k+1$ are placed at the location 00 at level k

Cache Design

We have observed that more than one blocks from the level $k+1$ memory (say of the main memory), having N blocks, may be placed at the same location (given by $N \bmod M$) in the level- k memory (say cache) having M blocks

Hence, a tag must be associated with each block in the level- k (cache) memory to identify its position in the level $k+1$ memory (Main memory)

Direct Mapping Example

The 16 MB main memory has 24 address bus

It is organized in 32-bit blocks

16 K word (64 KB) cache requires 16-bit address and 8-bit tag

Direct Mapping Address Structure

24 bit address

2 bit word identifier (4 byte block)

22 bit block identifier for the main memory

- ✓ 8 bit tag ($=22-14$)
- ✓ 14 bit slot or line or index value for cache

No two blocks in the same line have the same Tag field

Check contents of cache by finding line and checking Tag

Direct Mapping Cache Organization

Cache Design: Another Example

Let us consider another example with realistic numbers:

Assume we have a 1 KB direct mapped cache with block size equals to 32 bytes

In other words, each block associated with the cache tag will have 32 bytes in it (Row 1).

Address Translation Direct Mapped Cache

Assume the k+1 level main memory of 4GB, with Block Size equals to 32 bytes, and a k level cache of 1Kbyte

Cache Design

With Block Size equals to 32 bytes, the 5 least significant bits of the address will be used as byte select within the cache block.

Since the cache size is 1K byte, the upper 32 minus 10 bits, or 22 bits of the address will be stored as cache tag

The rest of the address bits in the middle, that is bit 5 through 9, will be used as Cache Index to select the proper cache block entry

Lecture 27

Memory Hierarchy Design (Cache Design Techniques)

] √pm ""] √..ω"

Recap: Caching and Locality
Cache Performance Metrics
Cache Designs
Addressing Techniques
Summary

Recap: Memory Hierarchy Principles

High speed storage at the cheapest cost per byte
Different types of memory modules are organized in hierarchy, based on the:

- ✓ Concept of Caching
- ✓ Principle of Locality

Recap: Concept of Caching

A small, fastest and most expensive storage is used as the staging area or temporary-place

- ✓ Store frequently-used subset of the data or instructions from the relatively cheaper, larger and slower memory; and
- ✓ To avoid having to go to the main memory every time this information is needed

Recap: Principle of Locality

To obtain data or instructions of a program, the processor accesses a relatively small portion of the address space at any instant of time
There are two different types of locality

- ✓ Temporal locality
- ✓ Spatial locality

Recap: Working of Memory Hierarchy

The memory hierarchy will keep the more recently accessed data items closer to the processor because chances are the processor will access them again soon.
NOT ONLY do we move the item that has just been accessed closer to the processor, but we ALSO move the data items that are adjacent to it

Recap: Cache Devices

Cache device is a small SRAM which is made directly accessible to the processor
Cache sits between normal main memory and CPU as data and instruction caches and may be located on CPU chip or as a module
Data transfer between cache - CPU, and cache- Main memory is performed by the cache controller
Cache and main memory is organized in equal sized blocks

Recap: Cache/Main Memory Data Transfer

An address-tag is associated with each cache block that defines the relationship of the cache block with the higher-level memory (say main memory)

Data Transfer between CPU and Caches takes place as the word transfer

Data transfer between Cache and the Main memory takes place as the block transfer

Recap: Cache operation

CPU requests contents of main memory location

Controller checks cache blocks for this data

If present, i.e., HIT, it gets data or instruction from cache - fast

If not present, i.e., MISS, it reads required block from main memory to cache, then deliver from cache to CPU

Cache Memory Performance

Miss rate, Miss Penalty, and Average access time are the major trade-off of Cache Memory performance

Miss Rate: is the fraction of memory accesses that are not found in the level-k memory or say the cache

✓ Miss Rate = number of misses / total memory accesses

As, Hit rate is defined as the fraction of memory access that are found in the level-k memory or say the cache, therefore Miss Rate = $1 - \text{Hit Rate}$

Miss Penalty is the memory stall cycles i.e., the number of cycles CPU is stalled for a memory access; and is determined by the sum of:

1. The Cycles (time) to replace a block in the cache, upper level and
2. The Cycles (time) to deliver the block to the processor

Average Access Time = Hit Time x (Hit Rate) + Miss Penalty x Miss Rate

The performance of a CPU is the product of clock cycle time and sum of CPU clock cycles and memory stall cycles

CPU Execution Time = (CPU Clock Cycles + Memory Stall Cycles) x clock cycle time

Where,

Memory stall cycles =

= Number of Misses x Miss Penalty

= IC x (Misses / Instructions) x Miss Penalty

= IC x [(Memory Access / Instructions)] x Miss Rate x Miss Penalty

Number of cycles for memory read and for memory write may be different,

Miss penalty for read may be different from the write

Memory Stall Clock Cycles = Memory read stall cycles + Memory Write stall cycles

Cache Performance Example

Assume a computer has CPI=1.0 when all memory accesses are hit; the only data accesses are load/store access; and these are 50% of the total instructions

If the miss rate is 2% and miss penalty is 25 clock cycles, how much faster the computer will be if all instructions are HIT

Execution Time for all Hit = IC x 1.0 x cycle time

CPU Execution time with real cache = CPU Execution time + Memory Stall time

Memory Stall Cycles =

= IC x (Instruction access + data access) per instruction x miss rate x miss penalty

= IC (1 + 0.5) x 0.02 x 25

= IC x 0.75

CPU Execution time (with cache) =

= (IC x 1.0 + IC x 0.75) x clock time

= 1.75 x IC x Cycle time

Computer with no cache misses is 1.75 times faster

Block Size Tradeoff: Miss Rate

Miss rate probably will go to infinity.
It is true that if an item is accessed,
it is likely that it will be accessed
again soon.

This is called the ping pong effect

The data is acting like a ping pong
ball bouncing in and out of the
cache.

MISS RATE is not the only cache
performance metrics, we have to
worry about the miss penalty

Block Size Tradeoff: Miss Penalty

Block Size Tradeoff: Average Access Time

Block size passes a certain point,
the miss rate actually goes up.

Block size passes a certain point,
the miss rate actually goes up.

Average Access Time

Performance metric than the miss
rate or miss penalty

Block Size Tradeoff: Average Access Time

Not only is the miss penalty is increasing,
Miss rate is increasing as well

How do you Design a Cache?

read: $\text{data} \leftarrow \text{Mem}[\text{Physical Address}]$
write: $\text{Mem}[\text{Physical Address}] \leftarrow \text{Data}$

Categories of Cache Organization

Cache can be organized in three different way based on the block placement policy

The block placement policy defines where a block from main (Physical) memory be placed in the cache

There exist three block placement policies namely:

- ✓ Direct Mapped
- ✓ Fully Associative
- ✓ Set Associative

Direct Mapped Cache Organization

Direct mapped cache organization is one where each block has only one place it can appear in the cache.

Direct Mapping Example

For example, a computer uses 16 MB main memory and 1MB cache.

These memories are organized in 4-byte blocks

That is, the main memory has 4M Blocks and the cache is organized in 256K blocks each of 4-byte

Each block in the main as well as in cache is address by a line-number

Now in order to understand the placement policy, the main memory is logically divided into 16 sections each of 256K blocks (lines)

Each section is represented by a Tag number

Direct Mapping Characteristics

Each block of main memory maps to only one cache line

e.g., the block number at line 339CH from any of the 16 sections must be place at line number 0CE7 with corresponding tag in the cache

This shows that the direct mapping is given by: (Block address) MOD (Number of Blocks in the cache)

$$(00339C) \text{ MOD } (3FFF) = 0CE7$$

$$(16339C) \text{ MOD } (3FFF) = 0CE7$$

Direct Mapping Address Structure

24 bit address

2 bit word identifier (4 byte block)

22 bit block identifier for the main memory

- ✓ 8 bit tag ($=22-14$)

- ✓ 14 bit slot or line or index value for cache

No two blocks in the same line have the same Tag field

Check contents of cache by finding line and checking Tag

Direct Mapping Cache Organization

Least Significant w bits identify unique word

Next Significant s bits specify one memory block

The MSBs are split into a cache line field r and a tag of $s-r$ (most significant)

Cache Design: Another Example

Let us consider another example with realistic numbers:

Assume we have a 1 KB direct mapped cache with block size equals to 32 bytes

In other words, each block associated with the cache tag will have 32 bytes in it (Row 1).

Address Translation Direct Mapped Cache

Assume the $k+1$ level main memory of 4GB, with Block Size equals to 32 bytes, and a k level cache of 1Kbyte

Direct Mapping pros & cons

Simple, Inexpensive

Fixed location for given block

- ✓ If a program accesses 2 blocks that map to the same line repeatedly, cache misses are very high

Xeph" f" d" w" m" g" y" h" s" w" i" " j" b" i" " g" e" l" i" " g" s" r" x" i" r" w" e" v" i" " z" e" p" h" " i" | t" p" e" r" e" x" s" r" "

Associative Mapping

A main memory block can load into any line of cache

Memory address is interpreted as tag and word

Tag uniquely identifies block of memory

Gzi vj " p" n" i" w" x" e" k" w" i" | e" q" m" i" h" j" s" v" e" q" e" x" l

Fully Associative Cache Organization

Forget about the Cache Index

Place any block of the main memory any where in the cache

Store all the upper bits of the address (except Byte select) that is associated with the cache block as the cache tag and have one comparator for every entry.

Associative Mapping Example

Associative Mapping: Address Structure

22 bit tag stored with each 32 bit block of data

Compare tag field with tag entry in cache to check for hit

Least significant 2 bits of address identify which 16 bit word is required from 32 bit data block

e.g.

Address	Tag	Data	Cache line
FFFFC	3FFFFC	24682468	3FFF

Fully Associative

While the direct mapped cache is on the simple end of the cache design spectrum, the fully associative cache is on the most complex end.

It is the N-way set associative cache carried to the extreme where N in this case is set to the number of cache entries in the cache.

We just store all the upper bits of the address (except Byte select) that is associated with the cache block as the cache tag and have one comparator for every entry.

Fully Associative Cache Organization

The address is sent to all entries at once and compared in parallel and only the one that matches are sent to the output.

This is called an associative lookup.

Hardware intensive.

Fully associative cache is limited to 64 or less entries.

Conflict miss is zero for a fully associative cache. Assume we have 64 entries here. The first 64 items we accessed can fit in.

But when we try to bring in the 65th item, we will need to throw one of them out to make room for the new item.

This brings us to the cache misses of type Capacity Miss

Set Associative Mapping Summary

Address length = $(s + w)$ bits

Number of addressable units = 2^{s+w} words or bytes

Block size = line size = 2^w words or bytes

Number of blocks in main memory = 2^d

Number of lines in set = k

Number of sets = $v = 2^d$

Number of lines in cache = $k v = k * 2^d$

Size of tag = $(s - d)$ bits

Set Associative Mapping

Cache is divided into a number of sets

Each set contains a number of lines

A given block maps to any line in a given set

e.g. Block B can be in any line of set i

e.g. 2 lines per set

2 way associative mapping

A given block can be in one of 2 lines in only one set

Set Associative Cache

This organization allows to place a block in a restricted set of places in the cache, where a set is a group of blocks in the cache at each index value

Here a block is first mapped onto a set (i.e., mapped at an index value and then it can be placed anywhere within that set

The set is usually chosen by bit-selection; i.e.,

$(\text{block address}) \text{ MOD } (\text{Number of sets in cache})$

If there are n -blocks in a set, the cache placement is referred to as the n -way set associative

This is called a 2-way set associative cache because there are two cache entries for each cache index. Essentially, you have two direct mapped cache works in parallel.

This is how it works: the cache index selects a set from the cache. The two tags in the set are compared in parallel with the upper bits of the memory address.

If neither tag matches the incoming address tag, we have a cache miss.

Otherwise, we have a cache hit and we will select the data on the side where the tag matches occur.

This is simple enough. What are its disadvantages?

Set Associative Mapping Address Structure

Use set field to determine cache set to look in

Compare tag field to see if we have a hit, e.g

Address	Tag	Data	Set number
1FF 7FFC	1FF	12345678	1FFF
001 7FFC	001	11223344	1FFF

Two Way Set Associative Mapping Example

Two-way Set Associative Cache

Let us consider this example 2-way set Associative Cache

Here, two cache entries are possible for each index

i.e., two direct mapped caches are working in parallel

Working of Two-way Set Associative Cache

Let us see how it works?

- ✓ The cache index selects a set from the cache. The two tags in the set are compared in parallel with the upper bits of the memory address.
- ✓ If neither tag matches the incoming address tag, we have a cache miss
- ✓ Otherwise, we have a cache hit and we will select the data on the side where the tag matches occur.

This is simple enough. What are its disadvantages?

Disadvantage of Set Associative Cache

First of all, a N-way set associative cache will need N comparators instead of just one comparator (use the right side of the diagram for direct mapped cache).

A N-way set associative cache will also be slower than a direct mapped cache because of this extra multiplexer delay.

Finally, for a N-way set associative cache, the data will be available AFTER the hit/miss signal becomes valid because the hit/miss is needed to control the data MUX.

For a direct mapped cache, that is everything before the MUX on the right or left side, the cache block will be available BEFORE the hit/miss signal (AND gate output) because the data does not have to go through the comparator.

This can be an important consideration because the processor can now go ahead and use the data without knowing if it is a Hit or Miss.

If it assumes that it is a hit; it will be ahead by 90% of the time as cache hit rate is in the upper 90% range, and for other 10% of the time that it is wrong, just make sure that it can recover

We cannot play this speculation game with a N-way set - associative cache because as we said earlier, the data will not be available to until the hit/miss signal is valid.

Lecture 28

Memory Hierarchy Design (Cache Design and policies)

Recap: Cache Addressing Techniques

Placement and Replacement Policies

Cache Write Strategy

Cache Performance Enhancement

Summary

Recap: Block Size Trade off

Impact of block size on the cache performance and categories of cache design

The trade-off of the block size verses the Miss rate, Miss Penalty, and Average access time, the basic CPU performance matrices

- ✓ The larger block size reduces the miss rate, but If block size is too big relative to cache size, miss rate will go up; and
- ✓ Miss penalty will go up as the block size increases; and
- ✓ Combining these two parameters, the third parameter, Average Access Time

Recap: Cache Organizations

Cache organizations

Block placement policy, we studied three cache organizations.

Recap: Cache Organizations

Direct Mapped where each block has only one place it can appear in the cache

Conflict Miss

Fully Associative Mapped where any block of the main memory can be placed anywhere in the cache; and

Set Associative Mapped which allows to place a block in a set of places in the cache

Block placement: Where can a block be placed in the upper level?

1. Block placement: Where can a block be placed in the upper level?
2. Block identification: How is a block found if it is in the upper level?
3. Block replacement: Which block should be replaced on a miss?
4. Write strategy: What happens on a write?

Block Placement Policy

Fully Associative: Block can be placed anywhere in the upper level (Cache)

- ✓ E.g. Block 12 from the main memory can be placed at block 2, 6 or any of the 8 block locations in cache

Set Associative: Block can be placed anywhere in a set in upper level (cache)

The set number in the upper level given as:

(Block No) MOD (number of sets)

E.g., an 8-block, 2-way set associative mapped cache, has 4 sets [0-3] each of two blocks; therefore, block 12 or 16 of main memory can go anywhere in

set # 0 as $(12 \text{ MOD } 4 = 0)$ and

$(16 \text{ MOD } 4 = 0)$

Similarly, block 14 can be placed at any of the 2 locations in set#2 ($14 \text{ MOD } 4 = 2$)

Direct Mapped: (1 way associative) Block can be placed at only one specific location in upper level (Cache)

The location in the cache is given by:

Block number MOD No. of cache blocks

E.g., the block 12 or block 20 can be placed at location 4 in cache having 8 blocks as:

$12 \text{ MOD } 8 = 4$

Block Identification

How a block is found if it is in the upper level? Tag/Block

A TAG is associated with each block frame

The TAG gives the block address

All possible TAGS, where a block may be placed are checked in parallel

Valid bit is used to identify whether the block contains correct data

No need to check index or block offset

Direct Mapped:

✓ Lower Level (Main) memory: 4GB 32-bit address

Block Replacement Policy

In case of cache miss, a new block needs to be brought in

If the existing block locations, as defined by Block placement policy, they are filled, then an existing block has to be fired based on

- ✓ Cache mapping; and
- ✓ some block replacement policy

Random: replace any block

- ✓ it is simple and easiest to implement
- ✓ The candidate for replacement are randomly selected
- ✓ Some designers use pseudo random block numbers

Least Recently Used (LRU): replace the block either never used or used long ago

- ✓ It reduces the chances of throwing out information that may be needed soon
- ✓ Here, the access time and number of times a block is accessed is recorded
- ✓ The block replaced is one that has not been used for longest time
- ✓ E.g., if the blocks are accessed in the sequence 0,2,3,0, 4,3,0,1,8,0 the victim to replace is block 2

First-in, First-out (FIFO): the block first place in the cache is thrown out first; e.g., if the blocks are accessed in the sequence 2,3,4,5,3,4

- ✓ then to bring in a new block in the cache, the block 2 will be thrown out as it is the oldest accessed block in the sequence
- ✓ FIFO is used as approximation to LRU as LRU can be complicated to calculate

Conclusion:

Associativity	2-way		4-way		8-way	
Size	LRU	Random	LRU	Random	LRU	Random
16 KB	5.0%	5.7%	4.7%	5.3%	4.4%	5.0%
64 KB	1.9%	2.0%	1.5%	1.7%	1.4%	1.5%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

Write Strategy

Must not overwrite a cache block unless main memory is up to date

Multiple CPUs may have individual caches

I/O may address main memory directly

Memory is accessed for read and write purposes

The instruction cache accesses are read

Instruction issue dominates the cache traffic as the writes are typically 10% of the cache access

Furthermore, the data cache are 10%-20% of the overall memory access are write

In order to store data in cache, the common case fast

Fortunately, the common case, i.e., the cache read, is easy to make fast as:

Read can be optimized by making the tag-checking and data-transfer in parallel

Thus, the cache performance is good

However, in case of cache-write, the cache contents modification cannot begin until the tag is checked for address-hit

Therefore the cache-write cannot begin in parallel with the tag checking

Another complication is that the processor specifies the size of write which is usually a portion of the block

Therefore, the write needs consideration

- ✓ Write back the information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced
 - Pros and Cons
 - No write to the lower level for repeated writes to cache
 - a dirty bit is commonly used to indicate the status as the cache block is modified (dirty) or not modified (clean)
 - Reduce memory-bandwidth requirements, hence the reduces the memory power requirements
- ✓ Write through_ The information is written to both the block in the cache and to the block in the lower-level memory
 - Pros and Cons
 - Simplifies the replacement procedure
 - the block is always clean, so unlike write-back strategy the read misses cannot result in writes to the lower level
 - Simplifies the data-coherency as the next lower level has the most recent copy (we will discuss this later)

Write Buffer for Write Through

Write buffer is just a FIFO: Typical number of entries: 4

Once the data is written into the write buffer and assuming a cache hit, the CPU is done with the write.

The real memory behind the scene

DRAM cycle time sets the upper limit on how frequent you can write to the main memory.

The write buffer works as long as the frequency of store, with respect to the time, is not too high, i.e.,

Store frequency $\ll 1 / \text{DRAM write cycle}$

If the stores are too close together or the CPU time is so much faster than the DRAM cycle time, you can end up overflowing the write buffer and the CPU must stop and wait.

time approaches 1 over the DRAM Write Cycle Time, i.e.,

The CPU Cycle Time $\leq$ DRAM Write Cycle Time

We call this Write Buffer Saturation

Write Buffer Saturation

In that case, it does NOT matter how big you make the write buffer, the write buffer will still overflow because you are simply feeding things in it faster than you can empty it

There are two solutions to this problem:

The first solution is to get rid of this write buffer and replace this write through cache with a write back cache

Write-Miss Policy

In case of write-miss, two options are used, these options are :

Write Allocate: A block is allocated on a write-miss, followed by the write hit action

No-write Allocate: Usually the write-misses do not affect the cache, rather the block is modified only in the lower level memory, i.e.,

Now we do a 16-bit write to memory location 0x000000 and causes a cache miss in our 1KB direct mapped cache that has 32-byte block select.

Assume we do a 16-bit write to memory location 0x000000 and causes a cache miss in our 1KB direct mapped cache that has 32-byte block select.

After we write the cache tag into the cache and write the 16-bit data into Byte 0 and Byte 1, do we have to read the rest of the block (Byte 2, 3, ... Byte 31) from memory?

If we do read the rest of the block in, it is called write allocate. But stop and think for a second. Is it really necessary to bring in the rest of the block on a write miss?

True, the principle of spatial locality implies that we are likely to access them soon.

But the type of access we are going to do is likely to be another write.

So if even if we do read in the data, we may end up overwriting them anyway so it is a common practice to NOT read in the rest of the block on a write miss.

Allocate, you better have some way to tell the processor the rest of the block is no longer valid.

This brings us to the topic of sub-blocking.

The blocks stay out of the cache in no-write allocate until the program tries to read the blocks, but

The blocks that are only written will still be in the cache with write allocate

Let us discuss it with the help of example

Now we do a 16-bit write to memory location 0x000000 and causes a cache miss in our 1KB direct mapped cache that has 32-byte block select.

Assume we do a 16-bit write to memory location 0x000000 and causes a cache miss in our 1KB direct mapped cache that has 32-byte block select.

Assume we do a 16-bit write to memory location 0x000000 and causes a cache miss in our 1KB direct mapped cache that has 32-byte block select

After we write the cache tag into the cache and write the 16-bit data into Byte 0 and Byte 1, do we have to read the rest of the block (Byte 2, 3, ... Byte 31) from memory?

If we do read the rest of the block in, it is called write allocate.

True, the principle of spatial locality implies that we are likely to access them soon.

But the type of access we are going to do is likely to be another write.

So if even if we do read in the data, we may end up overwriting them anyway so it is a common practice to NOT read in the rest of the block on a write miss.

If you allocate, you better have some way to tell the processor the rest of the block is no longer valid.

This brings us to the topic of sub-blocking.

As the principle of spatial locality implies that we are likely to access them soon.

But the type of access we are going to do is likely to be another write.

So even if we do read in the data, we may end up overwriting them anyway so it is a common practice to NOT read in the rest of the block on a write miss.

Allocate, you better have some way to tell the processor the rest of the block is no longer valid.

Example: No write-allocate verses write allocate

Let us consider a fully associative write-back cache with cache entries that start empty

Consider the following sequence of five memory operations and find

The number of hits and misses when using no-write allocate verses write allocate

Write Mem [100]

Write Mem [100]

Read Mem [200]

Write Mem [200]

Write Mem [100]

For no-write allocate, the address [100] is not in the cache (i.e., its tag is not in the cache)

So the first two writes will result in MISSES

Address [200] is also not in the cache, the read is also miss

The subsequent write [200] is a hit

The last write [100] is still a miss

The result is 4 MISSES and 1 HIT

For the write-allocate policy

The first access to 100 and 200 are MISSES

The rest are HITS as [100] and [200] are both found in the cache

The result is 2 MISSES and 3 HITS

Conclusion

- ✓ Either write miss policy could be used with the write-through or write-back
- ✓ Normally
- ✓ Write-back caches use write-allocate, hoping that the subsequent write to the block will be captured by the cache
- ✓ Write-through caches often use No Write Allocate, the reason is that even if there is a subsequent write to the block, the write must go to the lower level memory

Lecture 29

Memory Hierarchy Design

Cache Performance Enhancement by: Reducing Cache Miss Penalty

] √pm ""] √..ω"

Recap: Cache Design
Cache Performance
Reducing Miss Penalty
Summary

Yqom.AT qy √“ 'Oq“m'ot 'Kq"usz q“ ""J √zoq“z"

Block placement: Where can a block be placed in the upper level?
Block identification: How is a block found if it is in the upper level?
Block replacement: Which block should be replaced on a miss?
Write strategy: What happens on a write?

Recap: Write Buffer for Write Through

cache write strategies
write back
write through
use of write-buffer

Recap: Write Buffer for Write Through

level-2 cache is introduced in between the Level-1 cache and the DRAM main memory

- ✓ Write Allocate and
- ✓ No-Write Allocate

Recap: Write Miss Policies

Write Allocate:

- ✓ A block is allocated in the cache on a write miss, i.e., the block to be written is available in the cache

No-Write Allocate:

- ✓ The blocks stay out of the cache until the program tries to read the blocks; i.e., the block is modified only in the lower level memory

Impact of Caches on CPU Performance

CPU Execution Time equation

$$\text{CPU (ex-Time)} = (\text{CPU}_{\text{Exe. clock cycle}} + \text{Memory}_{\text{Stall cycles}}) \times \text{Clock}_{\text{Cycle Time}}$$

Example

Assumptions

- ✓ the cache miss penalty of 100 clock cycles
- ✓ all instructions normally take 1 clock cycle
- ✓ Average miss rate is 2%

- ✓ Average memory references per instruction = 1.5
- ✓ Average number of cache misses per 1000 inst. = 30

Find the impact of cache on performance of CPU considering both the misses per instruction and miss rate

CPU Time =

$$(\text{CPU Exe. clock cycle} + \text{Memory Stall cycles}) \times \text{Clock Cycle Time}$$

CPU Time_{with cache} (including cache miss)

$$= (\text{IC} \times (1.0 + (30/1000 \times 100)) \times \text{clock cycle time})$$

$$= \text{IC} \times 4.00 \times \text{clock cycle time}$$

CPU Time_{with cache} (including miss rate)

$$= (\text{IC} \times (1.0 + (1.5 \times 2\% \times 100)) \times \text{clock cycle time})$$

$$= \text{IC} \times 4.00 \times \text{clock cycle time}$$

Cache Performance (Review)

Number of Misses or miss rate

Cost per Miss or miss penalty

Memory stall clock cycles equal to the sum of

- ✓ $\text{IC} \times \text{Reads per inst.} \times \text{Read miss rate} \times \text{Read Miss Penalty}$; and
- ✓ $\text{IC} \times \text{writes per inst.} \times \text{Write Miss Rate} \times \text{Write Miss Penalty}$
- ✓ $(\text{Number of reads} \times \text{read miss rate} \times \text{read miss penalty})$

+

$$(\text{Number of write} \times \text{write miss rate} \times \text{write miss penalty})$$

Averaging the read and write miss rate

Memory stall clock cycles =

$$\text{Number of memory access} \times \text{Misses rate} \times \text{miss penalty}$$

Average Memory Access Time =

$$\text{Hit Time} \times \text{Misses rate} \times \text{miss penalty}$$

Note that the average memory access time is an indirect measure of the CPU performance and is not substitute for the Execution Time

However, this formula can decide about the split caches (i.e., instruction cache and data cache) or unified cache

E.g., if we have to find out which of these two types of caches has lower miss rate we can use this formula as follows:

Example:

Statement: Let us consider 32KB unified cache with misses per 1000 instruction equals 43.3 and instruction/data split caches each of 16KB with instruction cache misses per 1000 as 3.82 and data cache as 40.9; Assume that

- ✓ 36% of the instructions are data transfer instructions;
- ✓ 74% of memory references are instruction references; and
- ✓ hit takes 1 clock cycle where the miss penalty is 100 cycles and
- ✓ a load or store takes one extra cycle on unified cache

Assuming write-through caches with write-buffer and ignore stalls due to write buffer
Find the average memory access time in each case

Note to solve this problem we first find the miss rate and then average memory access time

Solution:

1. Miss Rate = (Misses/1000) / (Accesses/ inst.)
 - ✓ Miss Rate_{16KB Inst} = (3.82/1000) / 1.0 = 0.0038
 - ✓ Miss Rate_{16KB data} = (40.9/1000) / 0.36 = 0.114
 - ✓ As about 74% of the memory access are instructions therefore overall miss rate for split caches = (74% x 0.0038) + (26% x 0.114) = 0.0324
 - ✓ Miss Rate_{32KB unified} = (43.3/1000) / (1+0.36) = 0.0318
 - ✓ i.e., the unified cache has slightly lower miss rate
2. Average Memory Access Time

$$= \%inst \times (\text{Hit time} + \text{Inst. Miss rate} \times \text{miss penalty})$$

$$+ \%data \times (\text{Hit time} + \text{data Miss rate} \times \text{miss penalty})$$
 - ✓ Average Memory Access Time_{split}

$$= 74\% \times (1 + 0.0038 \times 100) + 26\% \times (1 + 0.114 \times 100) = 4.24$$
 - ✓ Average Memory Access Time_{unified}

$$= 74\% \times (1 + 0.0318 \times 100) + 26\% \times (1 + 0.0318 \times 100) = 4.44$$
 - ✓ i.e., the split caches have slightly better average access time and also avoids Structural Hazards

Improving Cache Performance

Average memory access time gives framework to optimize the cache performance

The Average memory access time formula:

Average Memory Access time = Hit Time + Miss Rate x Miss Penalty

Four General Options

1. Reduce the miss penalty,
2. Reduce the miss rate,
3. Reduce miss Penalty or miss rate via Parallelism
4. Reduce the time to hit in the cache

Reducing Miss Penalty

1. Multilevel Caches
2. Critical Word first and Early Restart
3. Priority to Read Misses Over write Misses
4. Merging Write Buffers
5. Victim Caches

1: Multilevel Caches (to reduce Miss Penalty)

This technique ignores the CPU but concentrates on the interface between cache and main memory

Multiple levels of caches

Tradeoff between cache size (cache effectiveness and cost (access time) a small fastest memory is used as level-1 cache

Performance Analysis

Average access Time is:

Access Time_{average}

$$= \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Penalty}_{L1}$$

Where, Miss Penalty_{L1}

$$= \text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2}$$

The Average memory access time

$$= \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times (\text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2})$$

Stall/instruction_{average}

$$= \text{Misses per instruction}_{L1} \times \text{Hit Time}_{L2} + \text{Misses per instruction}_{L2} \times \text{Miss Penalty}_{L2}$$

Local Miss Rate: Measure of misses in a cache divided by the total number of misses in this cache.

Global Miss Rate: Measure of the number of misses in the cache divided by the total number of memory access generated by the CPU

$$\checkmark \text{ 1}^{\text{st}} \text{ level cache} = \text{Miss Rate}_{L1}$$

$$\checkmark \text{ 2}^{\text{nd}} \text{ level cache} = \text{Miss Rate}_{L1} \times \text{Miss Rate}_{L2}$$

Example: Find the local and global miss rates, the Average Memory Access Time and Average memory stall cycles per instruction, given that for 1000 reference with 40 misses in L1 cache and 20 in L2 Cache;

Assume:

- ✓ miss penalty for L2 cache-memory = 100 clock cycles
- ✓ hit time for L2 cache is 10 clock cycle
- ✓ Hit time for L1 cache is 1 Clock cycle
- ✓ Memory Reference per instruction = 1.5

Solution:

Miss rate for either local or global

$$L1 \text{ cache is same} = 4\% [(40/1000) \times 100]$$

$$\text{Local Miss Rate for L2} = 50\% [20/40]$$

$$\text{Global Miss Rate for L2 cache} = 2\% [(20/1000) \times 100]$$

Average Memory Access time

$$= \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Penalty}_{L1} \dots (1)$$

where, Miss Penalty_{L1}

$$= \text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2} \dots (2)$$

Substituting 2 in 1 we get,

Average Memory Access time

$$= \text{Hit Time L1} + \text{Miss Rate L1} \times (\text{Hit Time L2} + \text{Miss Rate L2} \times \text{Miss Penalty L2})$$

$$= 1 + 4\% \times (10 + 50\% \times 100) = 3.4 \text{ cycles}$$

Average Memory Stalls per instruction (i.e., miss penalty)

$$= \text{Misses per instruction L1} \times \text{Hit Time L2}$$

$$+ \text{Misses per instruction L2} \times \text{Miss Penalty L2}$$

For Memory references per instruction = 1.5

- ✓ Misses per instruction for L1 = $40 \times 1.5 = 60$ per 1000 instructions
- ✓ Misses per instruction for L2 = $20 \times 1.5 = 30$ per 1000 instructions

Average Memory Stalls per instruction

$$= (60/1000) \times 10 + (30/1000) \times 100$$

$$= 0.6 + 3.0 = 3.6 \text{ clock cycles}$$

i.e., the average miss penalty using multi level caches reduces by a factor of $100/3.6 = 28$ relative to the single level cache

Miss rate verses cache size for multilevel caches

The miss-rate of single level cache verses size is plotted against the local and global miss rates of 2nd level cache using 32 KB 1st level cache L2 unified cache is 2-way set associative with LRU replacement

2: Critical word first and early restart

The CPU normally needs one word of a block at a time. If the first word of a block is the critical word, the CPU can restart the block load as soon as the critical word arrives.

The CPU normally needs one word of a block at a time. If the first word of a block is the critical word, the CPU can restart the block load as soon as the critical word arrives.

Early Restart: Request the words in a block in normal order. As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution.

Critical Word First: The CPU requests the critical word first. As soon as the critical word arrives, the CPU restarts the block load. The CPU continues filling the rest of the words in the block afterwards.

Example:

Consider a computer using 64-byte [8 word] cache blocks

An L2 cache takes 11 clock cycles to get first 8-byte (critical word) and then 2 clock cycles per 8-byte word to get the rest of the block (and 2 issues per cycle)

1. with critical word first (assuming no other access to the rest of the block)
2. without critical word first (assuming following instructions read data sequentially 8-byte words at a time from the rest of the block; i.e., block load is required in this case)

Solution:

1. With Critical word first:

Average miss penalty

$$= \text{Miss Penalty of critical word} + \text{Miss penalty of the remaining words of the block}$$

$$= 11 \times 1 + (8-1) \times 2 = 11 + 14 = 25 \text{ clock cycles}$$

2. Without critical word first (it requires block load)

$$= [\text{Miss Penalty of first word} + \text{miss penalty of the remaining words of the block}]$$

$$+ \text{clock cycles to issue the load}$$

$$= [11 \times 1 + (8-1) \times 2] + 8/4 = 25 + 4 = 29 \text{ clock cycles}$$

2 issues/cycle so 4 cycles for 8 issues

Drawback: This technique is generally useful only in large blocks, therefore the programs exhibiting spatial locality may face a problem in accessing the data or instruction from the memory, as the next miss is to the remainder of the block

3: Priority to Read Miss over the Write misses

This technique reduces the average miss penalty by considering the overlap between the CPU and cache miss-penalty

We have already discussed how the write-buffer reduces the write stalls in write-through caches

The write-buffers ensure that write to memory does not stall the processor;

Furthermore, write-buffer may hold the updated value of a location needed on a read miss and the processor is blocked till read returns

Thus, the write buffers do complicate memory access and needs consideration

Let us consider an example program segment and discuss how the complexities of write buffer can be simplified by giving priority to read misses over the write misses

Consider the code:

```
SW R3, 512 (R0)          ; M[512] ← R3 Cache index 0
LW R1, 1024 (R0)         ; R1 ← M[1024] Cache index 0
LW R2, 512(R0)           ; R2 ← M[512] Cache index 0
```

Assuming that the code is implemented using a direct mapped, write-through cache that maps 512 and 1024 to same block and a 4-word write buffer

Find if value in R2 always be equal to value in R3

Note that this is case of RAW data hazard

Let us see how the cache access is performed

The Data in R3 is placed in write-buffer after first instruction; i.e., after the store instruction using the index 512

The following load instruction (i.e., the 2nd instruction) uses the same index 512; and is therefore a miss

- ✓ The second load instruction (i.e., the 3rd instruction is sequence) tries to put the value in location 512 into register R2; this also results into a miss
- ✓ When completed writing to location 512 in memory, the read of location 512 will put the old value into the cache and then into R2.
- ✓ Thus R3 would not equal to R2, this is because the reads are served before write have been completed, i.e.,

Write through with write buffers offer RAW conflicts with main memory reads on cache misses; and

If processor simply waits for write buffer to empty, it might increase read miss penalty (say for old MIPS 1000) by 50%)

The simplest way to come of this dilemma is to give priority to read miss; i.e.,

- ✓ either Wait for write buffer to empty
- ✓ or Check write buffer contents on a read miss; and if there are no conflicts and memory system is available then let the memory access continue for read

Note that by giving priority to Read Miss the cost (penalty) of write by the processor in the write-back register can also be reduce

In Write-back caches, the read miss may require replacing a dirty block, which is

- ✓ Normally done by Writing dirty block to memory, and then do the read
- ✓ Better alternative is to Copy the dirty block to a write-buffer, then do the read, and then do the write

In this case, the CPU stall less since it restarts as soon as read is done, hence reduces the miss-penalty

4: Merging Write Buffer

Write-through cache rely on write-buffer as all stores must be sent to the lower level of hierarchy

Even the write back caches use a simple buffer when a block is replaced

In Normal mode of operation, the write buffer absorbs write from CPU; and commit it to memory in the background

However, here the problem, particularly in write-through caches, is that small write-buffer may end up stalling processor if they fill up; and the Processor needs to wait till write committed to memory

This problem is resolved by Merging cache-block entries in the write buffer, because:

- ✓ Multiword writes are usually faster than writes performed one at a time
- ✓ Y vni wywyeq "q shij) "sr i "{ svh"m"e" f psgo="Vl yw' 0

If a write buffer already contains some words from the given data block we will merge current modified word with the block parts already in the buffer

That is, If the buffer contains other modified blocks the address can be checked to see if the address of this new data matches the address of valid write buffer entry

Then the new data are combined with the existing entry - it is called Write Merge

Note that here, the CPU continues to work while the write-buffer prepares to write the word to memory

This technique, therefore, reduces the number of stalls due to write-buffer being full; hence

reduces the miss penalty through improvement in efficiency of write-buffer

5: Victim Caches: Reducing Miss Penalty

Another way to reduce the miss penalty is to remember what was discarded as it may need again.

This method reduces the miss penalty since the discarded data has already been fetched so it can be used again at small cost

The victim cache contains only discarded blocks because of some earlier miss; and are checked on another miss to see if they have the desired data before going to the next lower-level memory

If the desired data (or instruction) is found then the victim block and cache block are swapped

This recycling requires small fully associative cache between a cache and its refill path - called the victim cache as shown in the following figure

Summary

Extra people are { i }gsq i "s"gsq i "e"sr k

VI i "wi gsr h"xi gl rnyi ." "EvngpY svh"Hmw'er h"GevpTi wevx ."m'xl i "mxp ver gi

VI i "X nh"q i x sh." t vsnx) "s"vi eh"q mw'szi v'xl i "{ xnfq mw ."m'xl i "jezsvnmq "VI i "jsywl " xi gl rnyi ." q i vkmk"{ vnf-yjji v. m'eguyemxer gi

Combining sequential writes into a single block for fast memory transfer

VI i "jnx "xi gl rnyi ."zmxq "gegl i "m'wezeki

All these methods help reducing

Miss penalty; however, the first one multi level caches, are the most important and efficient

However, reducing miss rate and hit rate to improve the memory hierarchy performance are also important metrics

We will take up these metrics next time till then

Lecture 30
Memory Hierarchy Design
Cache Performance Enhancement
(Reducing Miss Rate)

] √pm ""] √..ω"

Recap: Reducing Miss Penalty
Classification of Cache Misses
Reducing Cache Miss Rate
Summary

Recap: Improving Cache Performance

The miss penalty
The miss rate
The miss Penalty or miss rate via Parallelism
The time to hit in the cache

Recap: Reducing Miss Penalty

1. Multilevel Caches
 - ✓ VI i "q svi "X i "q i winv
2. Critical Word first and Early Restart
 - ✓ Reduces miss-penalty
3. Priority to Read Misses Over writes
 - ✓ Favoritism
4. Merging Write Buffers
 - ✓ Acquaintance
5. Victim Caches
 - ✓ Salvage

Cache Misses

Compulsory Misses (cold start or first reference misses)

- ✓ Block must be brought into the cache

Capacity Misses

- ✓ Capacity misses occur in Fully Associative cache

Conflict Misses (collision or interference misses)

- ✓ Many blocks map to the same address or set

Reducing Miss Rate

1. Larger Block Size
2. Larger Caches
3. Higher Associativity
4. Way Prediction and Pseudo-associativity
5. Compiler Optimization

1: Larger Block Size

Reduce the miss rate

Spatial locality

Larger block have maximum number of data or instructions

In small cache , larger blocks may increase

Assumption

80 clock cycles of over head

Delivers 16 bytes every 2 clock cycle

Hit time of 1 clock cycle

Solution:

Average memory access time = Hit time + Miss Rate x Miss Penalty

4KB cache, the miss rate = 7.24%

Miss penalty = 80 + 4 = 84 clocks

Average Memory Access Time = $1 + (7.25\% \times 84) = 7.082$ Clock cycles

Copy table 5.18 pp 428

Latency and bandwidth of the lower level memory

High latency and high bandwidth

2: Large Cache Size

Reduce Capacity misses

In 2001

2nd level and 3rd-level caches

Drawback

Longer hit time

Higher cost (access time)

3: Higher Associativity

The CCT for 2 way to 8-way associative

$\text{CCT 2way} = 1.36 \times \text{CCT 1way}$

$\text{CCT 4way} = 1.44 \times \text{CCT 1way}$

$\text{CCT 8way} = 1.56 \times \text{CCT 1way}$

4: Way Prediction and Pseudo-associativity

Fast hit time of direct-mapped caches

Lower conflict misses of set-associative caches

Reduces the conflict misses

Way Prediction

- ✓ Block in a set
- ✓ Steps
 1. Extra bits
 - 2-way prediction
 - 4-way prediction
 2. Multiplexer
 - Single tag

Other blocks for matches in subsequent clock cycles

Alpha 21264

Latency of 1 clock cycle

3 clock cycles

Pseudo-associative

- ✓ Column associative caches
- ✓ Miss
- ✓ $R_{w_i} \rightarrow y_{h-w_i} \rightarrow x$

Pseudo Associative caches

- ✓ Performance
- ✓ $U_{ps} \{ i \rightarrow v \} \rightarrow m$

5: Compiler Optimization

Reduce both the data caches misses and instruction cache misses

Instruction

Code optimization

- ✓ Reordering the Procedures
- ✓ Using Cache-line Alignment

The code reordering

- ✓ Determines conflicts

The code-line alignment method:

- ✓ Decreases cache miss
- ✓ Entry point is at the beginning of a cache block

Data misses are reduced

Spatial locality

Temporal locality

Array calculation

- ✓ loop interchange
- ✓ blocking

program having nested loops that access data in non-sequential order for j ($0 \rightarrow 100$) and in sequential order for i ($0 \rightarrow 5000$)

Using Loop Interchange

First Version:

```

for (k = 0; k < 100; k = k+1)
    for (j = 0; j < 100; j = j+1)
        for (i = 0; i < 5000; i = i+1)
            x[i][j] = 2 * x[i][j];

```

Reordered version:

```

for (k = 0; k < 100; k = k+1)
    for (i = 0; i < 5000; i = i+1)
        for (j = 0; j < 100; j = j+1)
            x[i][j] = 2 * x[i][j];

```

Using Blocking

- ✓ Example of improving Temporal Locality
- ✓ program to perform matrix multiplication
- ✓ Ts{ "q ersv'svhi v " "s{by-row) Espq r "q ersv'svhi v "
- ✓ Iteration for matrix multiplication

/* Initial version of matrix multiplication code */

```

for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        {r = 0;
        for (k = 0; k < N; k = k+1) {
            r = r + y[i][k]*z[k][j];  };
        x[i][j] = r;
    };

```

Summary

Large block size to reduce compulsory misses

Large cache size to reduce capacity misses

Higher associativity to reduce conflict misses

The way-prediction techniques checks a section of cache for hit and then on miss it checks the rest of the cache

The final technique loop interchange and blocking, is a software approach to optimize the cache performance

Next time we will talk about the way to enhance performance by having processor and memory operate in parallel till then

Example: Avg. Memory Access Time vs. Miss Rate

Example: assume CCT = 1.10 for 2-way, 1.12 for 4-way, 1.14 for 8-way vs. CCT direct mapped

(Red means A.M.A.T. not improved by more associativity)

WWW.VUMULTAN.COM

Lecture 31

Memory Hierarchy Design

Cache Performance Enhancement by (Miss Penalty/Rate Parallelism and Hit time)

] √pm ""] √..ω"

Recap: Reducing Miss Rate
Reducing Miss Penalty or Miss Rate using Parallelism
Reducing Hit Time
Summary Cache Optimizatio

Recap: 3 C Model and Reducing Miss Rate

Large block size to reduce compulsory misses
Large cache size to reduce capacity misses
Higher associativity to reduce conflict misses
In addition we discussed the way-prediction technique that checks a section of cache for hit first; and then on miss, it checks the rest of the cache
At the end we discussed the compiler based techniques, namely the loop interchange and blocking, to optimize the cache performance
Today, we will talk about the other ways to enhance cache performance
These methods include:

1. Reducing Miss Penalty or Miss Rate via parallelism Overlapping the execution of instructions with activities in the memory hierarchy
2. Reducing the hit time

Reducing Miss Penalty or Rate via Parallelism

The basic idea is to reduce miss penalty or miss rate by performing multiple outstanding memory operations through overlapping the memory activities and the instruction execution activities which take place in the processor
This can be accomplished by using three different techniques

1. Non-blocking Caches: reduces stall on misses
2. Hardware Prefetch: reduces number of misses
3. Software (compiler controlled) Prefetch: reduces number of misses

Non-blocking Caches

Memory hierarchy
Instruction and data caches are decoupled
Out-of-order execution CPU
Non-blocking or lockup-free
VI i " I nlyr hi v'q rww
J nlyr hi v'q yprp "q rww "
O nlyr hi v'q rww "
Complexity
1000 and later a miss to address 1032
Complexity of the cache controller increases

Memory system can service multiple misses
Pentium Pro

Hardware Prefetch: Reduces Misses

2 blocks

Requested block

Unavailable

Available in the stream buffer

1. Original cache request is cancelled
2. Block is read from the stream buffer and
3. Next pre-fetch request is issued

Fetches next block

Jouppi in 1990

1. 15% - 25% of misses
2. 4 blocks stream buffer
3. 43% for stream fetching
4. 50% at the same address
5. 16-block stream buffer
6. 72% misses

Efficient for data caches

Hardware identifies stream of accesses

Palacharla & Kessler in 1994

8 stream buffers

50% to 70%

64KB, 4-way set associative caches

Example: For UltraSPARC III, 64 KB data cache and 256 KB cache has average miss per 1000 instruction of 36.9 and 326 respectively, assuming

1. prefetching hit rate equals 20%
2. hit time is 1 clock cycle,
3. miss penalty is 15 cycles
4. 1 extra clock cycle if the data misses the cache but are found in prefetch buffer
5. data references equal 22%

solution:

1. Miss rate_{prefetch}

$$= [\text{Misses}/1000] / \text{data references}$$

$$= [36.9 / 1000] / [22 / 100]$$

$$= 36.9/220 = 16.7\%$$
2. Average memory access time_{prefetch}

$$= \text{Hit Time} + \text{Miss rate} \times \text{Prefetch hit time} \times 1$$

$$+ \text{Miss rate} \times (1 - \text{prefetch hit time}) \times \text{miss penalty}$$

$$\begin{aligned} \text{Average memory access time}_{\text{prefetch}} &= 1 + (16.7\% \times 20\% \times 1) + (16.7\% (1 - 20\%) \times 15) \\ &= 3.046 \end{aligned}$$

- a) Effective Miss rate_{pre fetched 64K}

$$= [\text{Average memory access time} - \text{Hit time}] / \text{Miss Penalty}$$

$$= [3.064 - 1] / [15] = 2.064 / 15 = 13.6\%$$

- b) From the given data, 256KB data cache yields miss rate:

$$\text{Miss rate}_{256\text{KB}} = 33.6 / (22\% \times 1000) = 14.8 \%$$

Software (Compiler Controlled) Prefetch:

Two variants of prefetch

- ✓ Register Prefetch: Load data into register (HP RISC)
- ✓ Cache Prefetch: Load data only into cache and not the register (MIPS IV, PowerPC, SPARC v. 9)

Issues and limitations of compiler prefetching

1. Register or Cache prefetching
 - Faulting prefetch
 - Non-Faulting prefetch
2. Compiler prefetching
 - Semantically Invisible,
 - Cannot cause virtual memory faults
3. Caches do not stall
4. Issuing the prefetch instructions

Summary reducing Cache Miss Penalty or miss rate via parallelism

The non-blocking caches

Bandwidth behind the cache Instruction-level parallelism

Hardware and Software prefetching

Final Component to reduce Average memory Access Time- Hit Time

five techniques to reduce the miss penalty

five methods to reduce the miss rate

three approaches to reduce the miss penalty

miss rate in parallel to reduce the average memory access time

Hit time

Reducing Hit Time

1. Clock rate of the processor
2. Cache Access time

The four commonly used techniques

1. Small and Simple Caches
2. Avoiding Address translation during Indexing
3. Pipelined Cache Access
4. Trace Cache

Small and Simple Caches

While discussing the cache design, we have observed the most time consuming operation in cache hit is to compare long address tag

This operation definitely require bigger circuit which is always slow as compared to smaller circuits

Thus an obvious approach to keep the hit time small is to keep the cache Smaller and simpler

The cache be kept small to fit it on the same chip as the processor to avoid the time penalty going off chip; and

Simpler, such as the direct-mapped the tag comparison can be overlapped with the data transmission

The impact of cache size and complexity, i.e., associativity and number of read/write ports is shown in this graph

Size of L1 caches

The 16KB L1 cache on Pentium III

8KB on Pentium IV

Avoiding Address Translation during Indexing

Small and simple cache

Translation Look-aside Buffer (TLB)

Virtual address from CPU

Virtual memory system

Two levels of address mapping

Address of the virtual memory to main memory and

- ✓ Main memory to cache

Virtual address for the cache

Directly mapped

Virtual caches

Physical cache

Eliminated from cache hit

Amdahl's rule

Virtual caches we have to consider two issues

Limitation

- ✓ Virtually addressed caches

Reasons are

- ✓ Protection
- ✓ Page address
- ✓ Physical Address difference
- ✓ virtual address refers to different physical address
- ✓ Aliasing/synonyms
- ✓ Two different virtual addresses
- ✓ Two processes
- ✓ The hardware solution to synonym
- ✓ 2-way set associative cache

The software solution is to share some address bit

Pipelined Cache Access

Cache hit time

Pipelining the cache access

Latency of the first level cache-hit

Fast cycle time which increases the bandwidth of instructions

- ✓ for Pentium I takes 1 clock-cycle to access the instruction cache;
- ✓ for Pentium Pro through Pentium II its takes 2 clock cycles, and
- ✓ for Pentium IV it takes 4 clock cycles

Trace Caches

Multiple-issue processors
 Trace caches
 Dynamic sequence
 Load into the cache block
 Branch prediction
 Instruction prefetching
 No wasted words and no conflicts
 Conflicts are avoided
 Complicates the address mapping
 No longer aligned to power 2 multiples of words
 Requirement for address mapping
 Demerit
 Same instruction may be stored multiple times

Summary Cache Optimization

5 methods to reduce the miss penalty
 7 ways to reduce 3Cs
 3 methods for reducing miss rate and miss penalty via parallelism; and
 4 techniques to reduce hit time

Technique	Miss Penalty	MR Rate	HT Time	Complexity	Comments
Miss Penalty					
Multilevel caches	+			2	Costly H/W
Early Restart & Critical Word 1st	+			2	Widely Used
Priority to Read Misses	+			1	Trivial for Uni-processor
Merging write buffer	+			1	used with w/t through
Victim Caches	+	+		2	

Technique	Miss Penalty	MR Rate	HT Time	Complexity	Comments
Miss Rate					
Larger Block Size	-	+		0	Trivial
Larger Cache Size	-	+		1	Widely Used
Higher Associativity	+			1	Widely Used
Pseudo-Associative	+			2	Used in L2
Way Predicted		+		2	Used in I-Cache
Compiler Reduce Misses	+			0	S/W approach

Technique	Miss Penalty	MR Rate	HT Time	Complexity	Comments
Parallelism					
Non-Blocking	+			3	out-of-order CPU
HW Prefetching	+	+		2inst	
of Instr/Data				3 data	
Compiler Controlled Prefetching	+	+		3	need non- blocking cache
Hit Time					
Avoiding Address Translation			+	2	Widely Used
Trace Cache			+	3	Used in P

Lecture 32

Memory Hierarchy Design (Main and Virtual Memories)

] √pm ""] √..ω"

Recap: Memory Hierarchy and Cache performance

Main Memory Performance

Virtual Memory Performance

Summary

Recap: Memory Hierarchy

design goal of memory system

Low cost as of cheapest memory fast speed as of fastest memory

The fastest, smallest and most costly memories

The slowest, biggest and cheapest memories

- ✓ Average access speed
- ✓ Cost
- ✓ Cheapest technology

Semiconductor memories

Static and Dynamic RAMs

Upper levels in the memory hierarchy

Recap: Caches Design

The Caches use Static Random Access Memory

Main Memory is Dynamic Random Access Memory (DRAM)

(~8 ms, <5% time)

The magnetic, optical or other medias

virtual memory

Cache and main memory are organized in equal sized blocks

Word transfer

Block transfer

The CPU requests contents of main memory

Word transfer is fast

Recap: Cache Performance

If misses

Miss penalty

Cache design and the performance

Techniques

- ✓ Miss rate
- ✓ Miss penalty
- ✓ Hit time

Main Memory Organization

Organizations of main memory

Source for Caches

Destination virtual memory

DRAM logical organization (4 M Bit)

Main Memory Performance

Performance of DRAM

1. Fast page mode DRAM
2. Synchronous DRAM
3. Double Data Rate DRAM

Fast page mode: Optimizes sequential access

Synchronous DRAM (SDRAM): Avoid handshaking

Double Data Rate (DDR) DRAM: Transmit data

Latency: Average memory access time

Bandwidth: Number of bytes read/write per unit time

- ✓ Access Time
- ✓ Cycle Time

Inputs/outputs and multiprocessors

Low-latency memory

Multiprocessor demand higher bandwidth

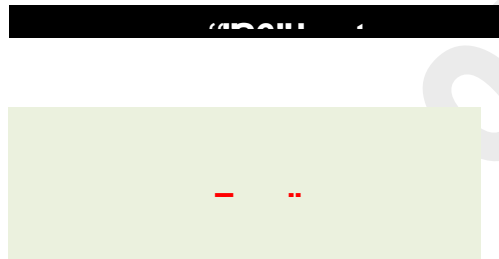
2nd level caches with larger block size

Improving Main Memory Performance

The most commonly used techniques are

- ✓ Wider Main Memory
- ✓ Simple Interleaved Memory
- ✓ Independent Memory Banks

S5"

**1: Wider Main Memory: Example**

4 words (i.e. 32 byte) block

- ✓ Time to send address = 4 clock cycles
- ✓ Time to send the data word = 4 clock cycles
- ✓ Access time per word = 56 clock cycles
- ✓ Miss Penalty =

$$\text{No. of words} \times [\text{time to: send address} + \text{send data word} + \text{access word}]$$

For 1 word organization

Miss Penalty = $4 \times (4 + 4 + 56) = 4 \times (64)$
 = 256 Clock Cycles;

The memory bandwidth = bytes/clock cycle
 = $32/256 = 1/8$ byte /cycle

For 4-word organization

Miss Penalty = $1 \times (4 + 4 + 56) = 64$ Clock
 Cycles; and
 Memory bandwidth = $32/64 = 1/2$
 bytes/cycle;

2: Interleaved Memory

bank 0 has all word whose: Address MOD 4 = 0
 bank 1 has all word whose: Address MOD 4 = 1
 bank 2 has all word whose: Address MOD 4 = 2
 bank 3 has all word whose: Address MOD 4 = 3

Example:

Bandwidth Calculation:

bandwidth of 4 words interleaved memory using the time model as used in case of wider memory

The miss penalty for 4-word interleave memory is:

$$\begin{aligned}
 &= \text{time to send address} + \text{time to access} + \\
 &\quad \text{number of banks} \times \text{time to send data} \\
 &= 4 + 56 + 4 \times 4 = 76 \text{ clock cycles}
 \end{aligned}$$

$$\text{Bandwidth} = 32/76 = 0.4 \text{ byte per clock}$$

$$\text{Bandwidth} = 32/256 = 1/8 = 0.125 \text{ byte per clock}$$

3: Independent Memory Banks

Memory banks offer independent accesses

Multiprocessors

I/O

CPU with Hit under n Misses

Non-blocking Caches

- ✓ An input device may use one controller and one bank
- ✓ The cache read may use another and
- ✓ The cache write still another

Summary: Main Memory Bandwidth

Using memory banks

Making memory and its bus wider

Doing both

How many the banks should be there?

Summary: Main Memory Bandwidth Enhancement

This decision is essential to ensure that

if memory is being accessed sequentially
(e.g. when processing an array)

then by the time you try to read a second word from a bank,
the first access has finished

Otherwise it will return to original bank before it has the next word ready

8 banks, each of 64-bit

Access time of 10 clock cycle

- ✓ Clock cycle 1
- ✓ Bank 0 after 10 clock cycles
- ✓ After 10 clock cycles,
- ✓ The bank 0 would fetch the next desired word
- ✓ 7 banks sequentially till the 18th clock cycle
- ✓ 18th clock
- ✓ Bank 0
- ✓ CPU cannot start fetching
- ✓ Clock cycle 20

- ✓ 10 clock cycles again
- ✓ Pyq f i V'sj" f er o" "Pyq f i V'sj" g s go" g} g p w x" eggi ww{ sh" m" f er o

Virtual Memory

Multiple processes
Single process
Exceed physical memory available
Increasing gap
High cost of main memory
Physical DRAM as a cache for the disk
Single level store
Single level storage
Virtual Memory System
Manages two levels of memory hierarchy
Main memory and secondary storage
Segments, named as a page
Page
Block
Contiguous pages

Attributes

- ✓ Protection
- ✓ Relocation

Protection

- Operate in different address space
- Different permissions
- Cannot access privileged information

Relocation

- ✓ Simplifies loading of program
- ✓ Allows to place a program anywhere

Hardware

Software

Cache verses Virtual memory

- Page or segment is used for block
- Page fault or address fault is used for miss
- CPU produces virtual address
- The virtual addresses are translated to the main memory or physical addresses
- Address translation
- Mapping of virtual address to the physical address
- Page table
- Physical address of the segment or the page

Replacement on cache miss
Page fault
The size of processor address
Cache size is independent of the processor address
Secondary storage
Lower-level backing store for main memory
File system occupies the space on secondary storage

Issues of Virtual Memory Design

Line size
Large, since disk better at transferring large blocks
Associativity
High, (fully associative) to minimize miss rate
Write Strategy
Write through or write back
miss rate: Extremely low. $\ll 1\%$
hit time: Must match cache/main memory performance
miss latency: Very high. $\sim 20\text{ms}$
tag storage overhead: Low, relative to block size

Typical System with Virtual Memory

- The CPU generates the Virtual Address
- Operating system manages a lookup table
- Location of the page or segment
- Virtual addresses to physical addresses

What happens when a virtual address is not in memory?

- Indicates virtual address not in memory
- OS exception handler invoked
- Current process suspends
- OS has full control over placement

WWW.VUMULTAN.COM

WWW.VUMULTAN.COM

Lecture 33

Memory Hierarchy Design (Virtual Memory System)

] √pm ""] √..ω"

Recap: Main memory and Virtual memory Design
Virtual Memory Address Translation
Virtual Memory Performance
Protection of multiple processes sharing memory
Summary

Recap: Memory Hierarchy

Main memory organization
Organized using banks of memory arrays
Dual Inline Memory Modules - DIMMs
Fast page mode
Synchronous
Double Data Rate DRAMs

Recap: Main Memory Performance

Fast page mode
Synchronous DRAM (SDRAM)
Double Data Rate (DDR) DRAM
latency and bandwidth

Recap: Main Memory Performance

concern of caches
bandwidth
Inputs/outputs and multiprocessors
Wider Main Memory
Simple Interleaved Memory
Independent Memory Banks

Recap: Virtual Memory

Multiple processes
Dedicate a full address space
Virtual Memory
Fix-sized fragment
Variable-sized fragment
Contiguous pages in virtual memory
Physically available on the main memory
Protection and Relocation
Protection
Relocation

Recap: Cache versus Virtual memory

Page fault or address fault
 CPU produces virtual address
 Mapping of a virtual address
 Replacement
 The size of processor address
 Secondary storage
 The page replacement strategies
 FIFO First in-First Out
 LRU Least recently Used
 Approximation to LRU

- ✓ Bit
- ✓ Resets the reference bit
- ✓ Page with a reference bit

VM Write strategies may be:

Write Back

Write Through

Write through is impossible because:

- ✓ Too long access to disk
- ✓ The write buffer
- ✓ The I/O system

Recap: Virtual Memory operation

The CPU generates the Virtual Address
 Lookup table
 Location of the page or segment
 Virtual addresses to physical addresses
 page fault

- ✓ The OS has full control over placement
- ✓ OS exception handler is invoked
- ✓ current process suspends the data is to the main memory by the OS

The contents of the page table are updated

VM Address Translation Concept

where $M < N$
 Assuming n -bit virtual address, m -bit physical address and p -bit page offset, the virtual and physical address limits and the page size can be expressed as

- ✓ Virtual address limit = $N = 2^n$
- ✓ Physical address limit = $M = 2^m$
- ✓ page size (bytes) = $PS = 2^p$

page offset

page number

VM Address Translation Concept

Page Table

Page Table Operation: 3 steps

1. Translation
2. Computing Physical Address
3. Checking Protection

Address Translation via Page Table

Simple Memory System Example

Addressing

- ✓ 14-bit virtual addresses
- ✓ 12-bit physical address
- ✓ Page size = 64 bytes

Simple Memory System Page Table

Only show first 16 entries

Fast Address Translation

large and in the main memory

miss penalty

one memory access to obtain the physical address

second to get the data

Miss penalty can be reduced

Fast Translation with a TLB

Address Translation with a TLB

WWW.VUMULTAN.COM

Fully associative placement policy

Violation against protection information in the TLB

The physical address

The page offset

A full physical address

Merits of TLB:

- ✓ Fully associative, set associative, or direct mapped
- ✓ 128 - 256 entries
- ✓ Mid-range machines use small n-way set associative organizations.

Address Translation Example

virtual address of 64 bits

- ✓ Physical Address: 41 bits
- ✓ TLB direct mapped with 256 entries
- ✓ First Level Caches: direct mapped with 8KB entries, block size 64 byte
- ✓ Second Level Cache: direct mapped 4MB direct mapped; block size 64 bytes

Address Translation Example VA L2

VM Protection Process

VM Protection Mechanisms

The address translation mechanism

Protection attribute bits

(PTE) and TLB

- ✓ Protection
- ✓ Does not have permission
- ✓ An exception is raised

Protection mechanism

The address is said to be valid if

$\text{Base} \leq \text{address} \leq \text{Bound}$

Base and Bound register

Page tables each pointing to the distinct pages of memory

Prevented from modifying these tables

Summary

Cache memories:

HW-management

Separate instruction and data caches permits simultaneous instruction fetch and data access

Four questions:

- ✓ Block placement
- ✓ Block identification
- ✓ Block replacement
- ✓ Write strategy

Virtual memory:

- ✓ Software-management
- ✓ Very high miss penalty

- ✓ => miss rate must be very low

Also supports:

- ✓ program loading
- ✓ memory protection
- ✓ Multiprogramming

Memory hierarchy organization

Modules of DRAM and SRAM

design and working of disk storages

DRAM, SRAM and Disk

Recap: Memory Hierarchy Principles

Concept of Caching

Principle of Locality

Recap: Principle of Locality

Data or instructions

Processor access a relatively small portion of the address space

Fastest memory closet to the processor

Recap: Types of Locality

Temporal locality

Spatial locality

Recap: Improving Cache Performance

The miss penalty

The miss rate

The miss Penalty or miss rate via Parallelism

The time to hit in the cache

Recap: Reducing Miss Penalty

Multilevel Caches

Critical Word first and Early Restart

Priority to Read Misses Over writes

Merging Write Buffers

Victim Caches

Recap: Reducing Miss Penalty

Oypri zi pgegl i w "

- ✓ the more the merrier

"EvngepY svh" Hw' er h" Gevj "T i wev .

t svw) "s" i eh" q nw'szi v\ i "{ vni "q nw ."

- ✓ Favoritism

q i vknk"{ vni-f yjji v.

- ✓ acquaintance

✓ salvage

✓ salvage

Reducing miss penalty

Reducing miss rate

Cache-misses and methods to reduce the miss rate

Summary Cache Optimization

5 methods to reduce the miss penalty

7 ways to reduce 3Cs

3 methods for reducing miss rate and miss penalty via parallelism; and

4 techniques to reduce hit time

The performance of these methods is summarized here

Lecture 34

Multiprocessors

(Shared Memory Architectures)

Recap:

Parallel Processing
Parallel Processing Architectures
Symmetric Shared Memory
Distributed Shared Memory
Performance of Parallel Architectures
Summary

Recap

So far our focus have been to study the performance of a single instruction stream computers; and methodologies to enhance the performance of such machines
We studied how
the Instruction Level Parallelism is exploited among the instructions of a stream; and
the control, data and memory dependencies are resolved

Recap: ILP

These characteristics are realized through:
Pipelining the datapath
Superscalar Architecture
Very Long Instruction Word (VLIW) Architecture
Out-of-Order execution

Parallel Processing and Parallel Architecture

However, further improvements in the performance may be achieved by exploiting parallelism among multiple instruction streams, which uses:
Multithreading, i.e., number of instruction streams running on one CPU
Multiprocessing, i.e., streams running on multiple CPUs where each CPU can itself be multithreaded

Furthermore, while evaluating the performance enhancement due to parallel processing

two important challenges are to be taken into consideration

- ✓ Limited parallelism available in program
- ✓ High cost of communication

These limitations make it difficult to achieve good speedup in any parallel processor
For example, if a portion of the program is sequential, it limits the speedup; this can be understood by the following example:

Example: What fraction of original computation can be sequential to achieve speedup of 80 with 100 processors?

Cr w{ i v"VI i "Cq hel p w{e{ "wex w\l ex

$$\text{Speedup} = 1 / (\text{Fraction}_{\text{Enhanced}} / \text{Speedup}_{\text{Enhanced}}) + (1 - \text{Fraction}_{\text{Enhanced}})$$

Here, the fraction enhanced is the fraction in parallel, therefore speedup can be expressed as

$$80 = 1 / [(\text{Fraction}_{\text{parallel}}/100 + (1 - \text{Fraction}_{\text{parallel}}))]$$

Simplifying the expression, we get

$$0.8 * \text{Fraction}_{\text{parallel}} + 80 * (1 - \text{Fraction}_{\text{parallel}}) = 1$$

$$80 - 79.2 * \text{Fraction}_{\text{parallel}} = 1$$

$$\text{Fraction}_{\text{parallel}} = (80 - 1) / 79.2 = 0.9975$$

i.e., to achieve speedup of 80 with 100 processors only 0.25% sequential allowed!

The second major challenge in parallel processing is the communication cost that involves the latency of remote access

Now let us consider another example to explain the impact of communication cost on the performance of parallel computers

Example: Consider an application running on 32-processors multiprocessor, with 40 nsec. time to handle remote memory reference

Assume instruction per cycle for all memory reference hit is 2 and processor clock rate is 1GHz, and find:

How fast is the multiprocessor when there is no communication versus 0.2% of the instructions involve remote access?

Solution: The effective CPI for multiprocessor with remote reference is:

$$\text{CPI} = \text{Base CPI} + \text{Remote request rate} \times \text{remote access cost}$$

Introduction to Parallel Processing

Substituting the values we get:

$$\begin{aligned} \text{CPI} &= [1/\text{Base IPC}] + 0.2\% \times \text{remote request cost} \\ &= [1/2] + 0.2\% \times (400 \text{ cycle}) \\ &= 0.5 + 0.8 = 1.3 \end{aligned}$$

And, CPI without remote reference

$$= 1/\text{Base IPC} = 0.5$$

Hence, the multiprocessor with all local reference is $1.3/0.5 = 2.6$ times faster as compare to that with no remote reference

Considering these limitations let us explore how improvement in computer performance can be accomplished using Parallel Processing Architecture

Parallel Architecture is a collection of processing elements that cooperate and communicate to solve larger problems fast

Parallel Computers extend the traditional computer architecture with a communication architecture to achieve synchronization between threads and consistency of data in cache

Parallel Computer Categories

In 1966, Flynn proposed simple categorization of computers that is still valid today

This categorization forms the basis to implement the programming and communication models for parallel computing

Flynn looked at the parallelism in the instruction and in the data streams called for by the instructions and proposed the following four categories:

1. SISD (Single Instruction Single Data)
 - ✓ This category is Uniprocessor
2. SIMD (Single Instruction Multiple Data)
 - ✓ Same instruction is executed by multiple processors using different data streams
 - ✓ Each processor has its own data memory (i.e., multiple data memories) but there is a single instruction memory and single control processor
 - ✓ Illiac-IV and CM-2 are the typical examples of SIMD architecture, which offer:
 - i. Simple programming model
 - ii. Low overhead
 - iii. Flexibility
 - iv. All custom integrated circuits
3. MISD (Multiple Instruction Single Data)
 - ✓ Multiple processors or consecutive functional units are working on a single data stream
 - ✓ (However, no commercial multiprocessor of this type is available till date)
4. MIMD (Multiple Instruction Multiple Data)
 - ✓ Each processor fetches its own instructions and operates on its own data
 - ✓ Examples: Sun Enterprise 5000, Cray T3D, SGI Origin. The characteristics these machines are:
 - i. Flexibility: it can function as Single-user multiprocessor or as multi-programmed multiprocessor running many programs simultaneously
 - ii. Use of off-the-shelf microprocessors

MIMD and Thread Level Parallelism

MIMD machines have multiple processors and can be used as:

- ✓ Either each processor executing different processes in a multi-program environment
- ✓ Or multiple processors execute a single program sharing the code and most of their address space

In the later case, where multiple processes share code and data, such processes are referred to as the threads

Threads may be

- ✓ either large-scale independent processes, such as independent programs, running in multi-programmed fashion
- ✓ Or parallel iterations of a loops having thousands of instructions, automatically generated by a compiler

This parallelism in the threads is called Thread Level Parallelism

MIMD Classification

Based on the memory organization and interconnect strategy, the MIMD machines are classified as:

- ✓ Centralized Shared Memory Architecture
- ✓ Distributed Memory Architecture

Centralized Shared-Memory

The Centralized Shared Memory design, shown here, illustrates the interconnection of main memory and I/O systems to the processor-cache subsystems

In small-level designs, with less than a dozens processor-cache, subsystems share the same physical centralized memory connected by a bus; while

In larger designs, i.e., the designs with a few dozens processor- cache subsystems, the single bus is replaced with multiple buses or even a switch are used.

However, the key architectural property of the Centralized Shared Memory design is the Uniform Memory Access (UMA);

i.e., the access time to all memory from all the processors is same

Furthermore, the single main memory has a symmetric relationship to all the processors

These multiprocessors, therefore are referred to as the Symmetric (Shared Memory) Multi-Processors (SMP)

This style of architecture is also sometimes called the Uniform Memory Access (UMA) as it offers uniform access time to all the memory from all the processors

Decentralized or Distributed Memory

The decentralized or distributed memory design style of multiprocessor architecture is shown here

It consists of number of individual nodes containing a processors, some memory and I/O and an interface to an interconnection network that connects all the nodes

The individual nodes contain a small number of processors which may be interconnected by a small bus or a different interconnection technology

Furthermore, it is a cost effective way to scale the memory bandwidth if most of the accesses are to the local memory in the node

Thus, the distributed memory provides more memory bandwidth and lower memory latency

This makes the design more attractive for small number of processors

The disadvantage of the distributed memory is that the data communication between the processors is more complex than in a shared memory system. There is no direct connection between the processors

Parallel Architecture Issues

While studying parallel architecture we will be considering the following fundamental issues that characterize parallel machines:

- ✓ How large is a collection of processor?
- ✓ How powerful are processing elements?
- ✓ How do they cooperate and communicate?
- ✓ How are data transmitted?
- ✓ What type of interconnection?
- ✓ What are HW and SW primitives for programmer? And
- ✓ How does it translate into performance?

Issues of Parallel Machines

These issues can be classified as:

1. Naming
2. Synchronization
3. Latency and Bandwidth

Fundamental Issue #1: Naming

Naming deals with:

- ✓ How to solve large problem fast?
- ✓ What data is shared?
- ✓ How it is addressed?

- ✓ What operations can access data?
- ✓ How processes refer to each other?

The segmented shared address space locations are named uniformly for all processes of the parallel program as: <process number, address>

Choice of naming affects:

- ✓ Code produced by a compiler as for message passing via load, the compiler just remembers address or keep track of processor number and local virtual address
- ✓ Replication of data, because in case of cache memory hierarchy the replication and consistency through load or via SW is affected by naming
- ✓ Global Physical and Virtual address space, as naming determines if the address space of each process can be configured to contain all shared data of the parallel program

Issue #2: Synchronization

In parallel machines to achieve synchronization between two processes, the processes must coordinate:

- ✓ Message passing implicitly coordinates with transmission or arrival of data
- ✓ Shared addresses explicitly coordinate through additional operations, e.g., write a flag, awaken a thread, interrupt a processor

Issue #3: Latency and Bandwidth

Bandwidth

- ✓ Need high bandwidth in parallel communication; however, bandwidth cannot be scaled, but stays close to the requirements
- ✓ Match limits in network, memory, and processor
- ✓ Overhead to communicate is a problem in many machines

Latency

- ✓ Affects performance, since processor may have to wait
- ✓ Affects ease of programming, since requires more thought to overlap communication and computation

Latency Hiding

- ✓ As the latency increases the programming system burden, therefore mechanisms are found to help hide latency

Examples:

- ✓ overlap message send with computation
- ✓ prefetch data
- ✓ switch to other tasks

Framework for Parallel processing

The framework for parallel architecture is defined as a two layer representation

These layers define Programming and Communication Models

These models present sharing of address space and message passing in parallel architecture

The shared address space model at:

- ✓ The communication layer, defines the communication via memory to handle load, store, and etc.; and at
- ✓ The programming layer, it defines handling several processors operating on several data sets simultaneously; and to exchange information globally and simultaneously

Message passing model at the:

- ✓ communication layer defines sending and receiving messages and library calls; and at the
- ✓ programming layer provides a multiprogramming model to conduct lots of jobs without I/O communication simultaneously

Shared Address Space Architecture (for Decentralized Memory Architecture)

Shared Address space is referred to as the Distributed Shared Memory DSM; where, each processor can name every physical location in the machine; and each process can name all data it shares with other processes

Data transfer takes place via load and store

Data size is defined as: byte, word, ... or cache blocks

Uses virtual memory to map virtual to local or remote physical

Processes on multiple processors are time shared

Multiple private address spaces offer message passing multicomputer via separate address space

Shared Address Space Architecture Programming Model

The programming model defines how to share code, private stack, some shared heap, some private heap

Here, Process is defined as virtual address space plus one or more threads of control. Multiple processes can overlap, but ALL threads share a process address space, i.e., portions of address spaces of processes are shared; and Writes to shared address space by one thread are visible to reads of all threads in other processes as well.

There exist number of shared address space architectures.

The most popular architectures are:

1. Main Frame Computers
2. Minicomputers Symmetric Multi Processors (SMP)
3. Dance Hall
4. Distributed Memory Non-Uniform Multiprocessor Architecture (NUMA)

1: Main Frame Architecture Shared Address Space Architecture

The main frame architecture was motivated by multiprogramming.

As shown here, it extends crossbar for processor interface to memory modules and I/O. Initially this architecture was limited by processor cost; but, later by the cost of crossbar. IBM S/390 (now z-Server) is typical example of cross-bar architecture.

2: Minicomputers (SMP) Architecture

The minicomputer architecture was also motivated by multiprogramming and multi-transaction processing.

As shown here, as all the components are on shared bus and all memory locations have equal access time so this architecture is referred to as the Symmetric Multi Processors (SMPs).

However, the bus is bandwidth bottleneck as sharing is limited by Bandwidth when we add processors, I/O

Furthermore, caching is key to the coherence problem we will talk about this later

The typical example of SMP architecture is Intl Pentium Pro Quad

Intel Pentium Pro Quad

Here, all the coherence and multi-processing is glued in processor module

It is highly integrated and have low latency and bandwidth

3: Dance Hall Architecture

All processors are on one side of the network and all memories on the other side

As we have notice that in the cross-bar architecture the major cost is of interconnect and in SMPs bus bandwidth is the bottleneck

This architecture offers solution to both the problems through its scalable interconnect network where the bandwidth is scalable

However, the interconnect network has larger access latency; and caching is key to the coherence problem

4: Distribute Memory Architecture

It is a large scale multiprocessor architecture where Memory is distributed with Non-Uniform Access Time

This architecture is therefore referred to as Non Uniform Memory Access (NUMA)

Cray T3E is a typical example of NUMA architecture

Cray T3E is a typical example of NUMA architecture which scales up to 1024 processors with 480 MB/sec. links

Here, the non-local references are accessed using communication requests generated automatically by the memory controller in the external I/Os

Here no hardware coherence mechanism is employed rather directory based cache-coherence protocols are used We will discuss this in detail later

Message Passing Architecture

So far we have been talking about the programming and communication models of shared-memory address space architecture and their evolution

Now let us discuss the programming and communication models of Message passing Architecture

The programming model depicted here illustrates that the whole computers (CPU, memory, I/O devices) communicate as explicit I/O operations

Message Passing Architecture: Programming Model

Note that the message passing is essentially NUMA but it is integrated at I/O devices vs. memory system

Here, the local memory is directly accessed, i.e., it directly accesses the private address space (e.g., the processor P directly access the local address X); and

Communication takes place via explicit message passing, i.e., via send/receive

Send specifies local buffer and the receiving process on remote computer

Receive specifies sending process on remote computer and local buffer to place data (i.e., address Y on Processor Q)

- ✓ Usually send includes process tag and receive has rule on tag: match 1, match any

Send and receive is memory-memory copy, where each supplies local address, AND does pair-wise synchronization

The synchronization is achieved as follows:

receive wait for send when

- ✓ send completes
- ✓ buffer free and
- ✓ request accepted

Message Passing Architecture: Communication Model

The high-level block diagram for complete computer as a building block, similar to the distributed memory spared address space is shown here to describe the communication abstraction

Here, the communication is integrated at IO level, and not into memory system

It has networks of workstations (clusters), but tighter integration

It is easier to build than scalable shared address space machines

Typical example of Message Passing Machines are IBM SP shown here

IBM SP: Message Passing Machine

Made out of essentially complete RS6000 workstations

Network interface integrated in I/O bus

Bandwidth is limited by I/O bus

Summary

Today we have explored how further improvement in computer performance can be accomplished using Parallel Processing Architectures

Parallel Architecture is a collection of processing elements that cooperate and communicate to solve larger problems fast

Then we described the four categories of Parallel Architecture as: SISD, SIMD, MISD and MIMD architecture

We noticed that based on the memory organization and interconnect strategy, the MIMD machines are classified as:

- ✓ Centralized Shared Memory Architecture and Distributed Memory Architecture

We also introduced the framework to describe parallel architecture as a two layer representation: Programming and Communication models

These models present sharing of address space and message passing in parallel architecture

The advantages of Shared-Memory Communication model are as follows:

- ✓ Ease of programming when communication patterns are complex or vary dynamically during execution
- ✓ Lower communication overhead, better use of BW for small items
- ✓ HW-controlled caching to reduce remote comm. by caching of all data, both shared and private

The advantages of message passing communication model are as follows:

- ✓ Communication is explicit and simpler to understand where as in shared memory it can be hard to know when communicating and when not, and how costly it is
- ✓ Easier to use sender-initiated communication, which may have some advantages in performance
- ✓ Synchronization is associated with sending messages

Lecture 35

Multiprocessors (Cache Coherence Problem)

Recap:] √pm ""] √..ω"

Recap:

Multiprocessor Cache Coherence

Enforcing Coherence in:

Symmetric Shared Memory Architecture

Distributed Memory Architecture

Performance of Cache Coherence Schemes

Summary

Recap: Parallel Processing Architecture

Last time we introduced the concept of Parallel Processing to improve the computer performance

Parallel Architecture is a collection of processing elements that cooperate and communicate to solve larger problems fast

There are four categories of computers which form the basis to implement the programming and communication models for parallel computing

These categories are:

- ✓ SISD (Single Instruction Single Data)
- ✓ SIMD (Single Instruction Multiple Data)
- ✓ MISD (Multiple Instruction Single Data)
- ✓ MIMD (Multiple Instruction Multiple Data)

The MIMD machines implement Parallel processing architecture

Recap: MIMD Classification

We noticed that based on the memory organization and interconnect strategy, the MIMD machines are classified as:

- ✓ Centralized Shared Memory Architecture
- ✓ Here, the subsystems share the same physical centralized memory connected by a bus. The key architectural property of this design is the Uniform Memory Access (UMA); i.e., the access time to all memory from all the processors is same

Recap: MIMD Classification

Distributed Memory Architecture

- ✓ It consists of number of individual nodes containing a processors, some memory and I/O and an interface to an interconnection network that connects all the nodes
- ✓ The distributed memory provides more memory bandwidth and lower memory latency

Recap: Framework for Parallel processing

Last time we also studied a framework for parallel architecture

The framework defines the programming and communication Models for centralized shared-memory and distributed memory parallel processing architectures

These models present address space sharing and message passing in parallel architecture

Here, we noticed that the shared-memory communication model has compatibility with the SMP hardware; and

offers ease of programming when communication patterns are complex or vary dynamically during execution

While the message-passing communication model has explicit Communication which is simple to understand; and is easier to use sender-initiated communication

Multiprocessor Cache Sharing

Today, we will look into the sharing of caches for multi-processing in the symmetric shared-memory architecture

The symmetric shared memory architecture is one where each processor has the same relationship to the single memory

Small-scale shared-memory machines usually support caching of both the private data as well as the shared data

The private data is used by a single processor, while the shared data is replicated in the caches of the multiple processors for their simultaneous use

It is obvious that the program behavior for caching of private data is identical to the that of a Uniprocessor, as no other processor uses the same data, i.e., no other processor cache has copy of the same data

Multiprocessor Cache Coherence

Whereas when shared data are cached the shared value may be replicated in multiple caches

This results in reduction in access latency and fulfill the bandwidth requirements, but, due to difference in the communication for load/store and strategy to write in the caches, values in different caches may not be consistent, i.e.,

There may be conflict (or inconsistency) for the shared data being read by the multiple processors simultaneously

This conflict or contention in caching of sheared data is referred to as the cache coherence problem

Informally, we can say that memory system is coherent if any read of a data item returns the most recently written value of that data item

This definition contains two aspects of memory behavior:

- ✓ Coherence that defines what value can be returned by a read?
- ✓ Consistency that determines when a written value will be returned by a read?

Let us explain the cache coherence problem with the help of a typical shared memory architecture shown here!

Cache Coherency Problem?

Note that here the processors P1, P2, P3 see old values in their caches as there exist several alternative to write to caches!

For example, in write-back caches, value written back to memory depends on which cache flushes or writes back value (and when);

i.e., value returned depends on the program order, program issue order or order of completion etc.

The cache coherency problem exists even on uniprocessors where due interaction between caches and I/O devices the infrequent software solutions work well

However, the problem is performance-critical in multiprocessors where the order among multiple processes is crucial and needs to be treated as a basic hardware design issue

Order among multiple processes?

Now let us discuss what does order among multiple processes means!

Firstly, let us consider a single shared memory, with no caches

- ✓ Here, every read/write to a location accesses the same physical location and the operation completes at the time when it does so

This means that a single shared memory, with no caches, imposes a serial or total order on operations to the location, i.e.,

- ✓ the operations to the location from a given processor are in program order; and
- ✓ the order of operations to the location from different processors is some interleaving that preserves the individual program orders

Now, let us discuss the case of a single shared memory, with caches

Here, the latest means the most recent in a serial order with operations to a location from a given processor in program order

Note that for the serial order to be consistent, all processors must see writes to the location in the same order

Formal Definition of Coherence!

With this much discussion on the cache coherence problem, we can say that

A memory system is coherent if the results of any execution of a program are such that for each location, it is possible to construct a hypothetical serial order of all operations to the location that is consistent with the results of the execution

In a coherent system

- ✓ the operations issued by any particular process occur in the order issued by that process, and
- ✓ the value returned by a read is the value written by the last write to that location in the serial order

Features of Coherent System

Two features of a coherent system are:

- ✓ write propagation: value written must become visible to others, i.e., any write must eventually be seen by a read
- ✓ write serialization: writes to a location seen in the same order by all

Cache Coherence on buses

Bus transactions and Cache state transitions are the fundamentals of Uniprocessor systems

Bus transaction passes through three phases: arbitration, command/address, data transfer

Cache State transition deals with every block as a finite state machine

The write-through, write no-allocate caches have two states: valid, invalid

write-back caches have one more state: q shijinh** hnx +

Multiprocessor cache Coherence

Multiprocessors extend both the bus transaction and state transition to implement cache coherence

Coherence with write-through caches!

Here, the controller snoops on bus events (write transactions) and invalidate / update cache

As in case of write-through, the memory is always up-to-date therefore invalidation causes next read to miss and fetch new value from memory, so the bus transaction is indeed write propagation

The Bus transactions impose write serialization as the writes are seen in the same order

Cache Coherence Protocols

In a coherent multiprocessor, the caches provide both the relocation (migration) and replication (duplication) of shared data items

There exist protocols which use different techniques to track the sharing status to maintain coherence for multiprocessor

The protocols are referred to as the Cache Coherence Protocols

Potential HW Coherency Solutions

The two fundamental classes of Coherence protocols are:

- ✓ Snooping Protocols: All cache controllers monitor or snoop (spy) on the bus to determine whether or not they have a copy of the block that is requested on the bus

- ✓ Directory-Based Protocols: The sharing status of a block of physical memory is kept in one location, called directory

The Snoopy solutions:

- ✓ Send all requests for data to all processors
- ✓ Processors snoop to see if they have a copy and respond accordingly
- ✓ Requires broadcast, since caching information is at processors
- ✓ Works well with bus (natural broadcast medium)
- ✓ Dominates for small scale machines (most of the market)

Directory-Based Schemes

- ✓ Keep track of what is being shared in one centralized place
- ✓ Distributed memory employs distributed directory for scalability and to avoid bottlenecks
- ✓ Send point-to-point requests to processors via network
- ✓ Scales better than Snooping
- ✓ Actually existed BEFORE Snooping-based schemes

Basic Snooping Protocols

There are two ways to maintain coherence requirements using snooping protocols. These techniques are: write invalidate and write broadcast

1. Write Invalidate Method

- This method ensures that processor has exclusive access to the data item before it write that item and all other cached copies are invalidated or canceled on write
- Exclusive excess ensures that no other readable or writeable copies of an item exist when the write occurs

1. Write Invalidate Protocol

Uses Multiple readers and single writer

For Write to shared data:

- ✓ an invalidate information is sent to all caches
- ✓ Considering this information, the controller snoop and invalidate any copies

For Read Miss, in case of:

- ✓ Write-through: memory is always up-to-date, so no problem; and
- ✓ Write-back: it snoop in caches to find most recent copy

Example: The following table shows the working of invalidation protocol for snooping bus with write-back cache

Here, we assume that both the caches of CPU A and B do not initially hold X, and that the value of X in the memory is 0 (First row)

Here, to see how this protocol ensures coherence, we consider a write followed by a read by another processor

As the write requires exclusive access, any copy held by the reading processor must be invalidated; thus

When the read occurs it misses in the cache and is forced to a new copy of data

Furthermore, the exclusive write access prevents any other processor from being writing simultaneously

In the table, the CPU and memory contents show the value after the processor activity

A blank indicates no activity or no copy cached and bus activity have completed

When 2nd miss by B occurs, the CPU A responds with the value cancelling the response from memory

The values given in the 4th row show the invalidation for the memory location x when A attempts to write 1

This update of the memory, which occurs when block becomes shared, simplifies the protocol

2: Write Broadcast Protocol

The alternative to Write Invalidate protocol is the write update or write broadcast protocol. Instead of invalidating this protocol updates all the cached copies of a data item when that item is written.

This protocol is particularly used for write through caches, here for

Write to shared data the processors snoop, and update any copies by broadcasting on bus

Example: The following table shows the working of write update protocol for snooping bus with write-back cache

Here, we assume that both the caches of CPU A and B do not initially hold X, and that the value of X in the memory is 0 (First row)

The CPU and memory contents show the value after the processor and bus activity have both completed

As shown in the 4th row, when CPA writes a 1 to memory X it update the value in caches of A and B and the memory

Write Invalidate versus Broadcast

Invalidate requires one transaction for multiple writes to the same word

Invalidate uses spatial locality: one transaction for write to different words in the same block

Broadcast has lower latency between write and read

An Example Snooping Protocol

A bus based protocol is usually implemented by incorporating a finite state machine controller in each node

This controller responds to the request from the processor and from the bus based on:

- ✓ the type of the request
- ✓ Whether it is hit or miss in the cache
- ✓ State of the cache block specified in the request

Each block of memory is in one of the three states:

- ✓ (Shared) Clean in all caches and up-to-date in memory
- ✓ OR (Exclusive) Dirty in exactly one cache
- ✓ OR Not in any caches

Each cache block is in one of the three state (track these):

- ✓ Shared : block can be read
- ✓ OR Exclusive : cache has only copy, its writeable, and dirty
- ✓ OR Invalid : block contains no data

Read misses: cause all caches to snoop bus

Writes to clean line are treated as misses

Finite State Machine for Write Invalidation Protocol and write Back Caches

Now let discuss the finite-state Transition for a single cache block using a write invalidation protocol and write back caches

The state machine has three states:

- ✓ Invalid
- ✓ Shared (read only) and
- ✓ Exclusive (read/write)

Here, the cache states are shown in circles where access permitted by the CPU without a state transition shown in parenthesis

The stimulus causing the state transition is shown on the transition arc in yellow and the bus action generated as part of the state transition is shown in orange

The state in each cache node represents the state of the selected cache block specified by the processor or bus request

In reality there is only one state-transition diagram but for simplicity the states of the protocol are duplicated here to represent:

- ✓ Transition based on the CPU request
- ✓ Transition based on the bus request

Now let us discuss the state-transition based on the actions of CPU associated with the cache, shown state machine -I

Snoopy-Cache State Machine-I: for CPU requests for each cache block

Note that a read miss in the exclusive or shared state and a write miss in the exclusive state occurs when the address requested by the CPU does not match the address in the cache block

Further an attempt to write a block in the shared state always generates miss even if the block is present in the cache, since the block must be made exclusive

Here, note that in case of read hit, the shared and exclusive states read data in cache and address the conflict miss

Snoopy-Cache State Machine-I

The invalid state places the read miss on the bus;

For write hit, the exclusive state writes the data in cache and shared state place write miss on bus

In case of write miss, the invalid state places the miss on the bus shared and exclusive states address the conflict miss;

the shared state places write miss on the bus, while

the exclusive state write-back block and then places write miss on the bus

Snoopy-Cache State Machine-II

Finite State Machine for Write Invalidation Protocol and write Back Caches

Now let us discuss the state-transition based on the actions of bus request associated with the cache, shown as state machine-II

Here, when ever a bus transaction occurs, all caches that contain the cache block specified in the bus transaction take the action as shown in this state machine

Here, the protocol assumes that

Memory provides data on a read miss for a block that is clean in all caches

Note that read miss, the shared state take no action, and allows the memory to service read miss;

where as the exclusive state, attempts to share the data, places the cache block on the bus and change the state to shared.

For the write miss, the shared state attempts to write shared block and invalidates the block

Whereas, the exclusive state attempts to write block that is exclusive elsewhere; write back the cache block and make the state invalid

Summary

Today, we talked about sharing of caches for multi-processing in the symmetric shared-memory architecture

We studied the cache coherence problem and studied two methods to resolve the problem

Here, we discussed the write invalidation and write broadcasting schemes

At the end we discussed the finite state machine for the implementation of snooping algorithm

We will further explain the snooping protocol with the help of example next time.

Lecture 36

Multiprocessors

(Cnot q"J√t q“qz oq'W√nxy " "J√z' p)

]√pm ""]√..ω”

Recap:

Example of Invalidation Scheme

Coherence in Distributed Memory Architecture

Performance of Cache Coherence Schemes

Summary

Recap: Cache Coherence Problem

Last time we discussed the sharing of caches for multi-processing in the symmetric shared-memory architecture, wherein each processor has the same relationship to the single memory

Here, we distinguished between the private data and shared data, i.e.,

- ✓ the data used by a single processor and
- ✓ the data replicated in the caches of the multiple processors for their simultaneous use

Then we discussed cache coherence problem in symmetric shared memory which results due to inconsistency or conflict in caching of shared data, being read by the multiple processors simultaneously

We studied the cache coherence problem with the help of a typical shared memory architecture where each of the processor contained write-back cache

In write-back caches, values written back to memory depend on which cache flushes or writes back the value and when?

We noticed that the cache coherency problem exists even on uniprocessors due to interaction between caches and I/O devices

However, in multiprocessors the problem is performance-critical where the order among multiple processes is crucial, i.e.,

Recap: Order among multiple processes

For single shared memory, with no caches, a serial or total order is imposed on operations to the location; and for

single shared memory, with caches, the serial order be consistent, i.e., all processors must see writes to the location in the same order

Considering this we can say that in a coherent system:

- ✓ the operations issued by any particular process occur in the order issued by that process, and
- ✓ the value returned by a read is the value written by the last write to that location in the serial order

Then we talked about write propagation and write serialization as the two features of the coherent system

Recap: Multiprocessor cache Coherence

We also noticed that to implement cache coherence the multiprocessors extend both the bus transaction and state transition

The cache controller snoops on bus events (write transactions) and invalidate / update cache

Then we discussed the cache coherence protocols, which use different techniques to track the sharing status and maintain coherence for multiprocessor

Recap: Coherency Solutions

The two fundamental classes of Coherence protocols are:

- ✓ Snooping Protocols: All cache controllers monitor or snoop (spy) on the bus to determine whether or not they have a copy of the block that is requested on the bus
- ✓ Directory-Based Protocols: The sharing status of a block of physical memory is kept in one location, called directory

Recap: Basic Snooping Protocols

The snooping protocols are implemented using two techniques: write invalidate and write broadcast

The Write Invalidate method ensures that processor has exclusive access to the data item before it write that item and all other cached copies are invalidated or canceled on write

The write broadcast approach, on the other hand, updates all the cached copies of a data item when that item is written

Recap: Write Invalidate versus Broadcast

We noticed that

- ✓ Invalidate requires one transaction for multiple writes to the same word; and it uses spatial locality, i.e., one transaction for write to different words in the same block; and
- ✓ Broadcast has lower latency between write and read

Then we discussed the finite state machine controller implementing the snooping protocols

Recap: An Example Snooping Protocol

This controller responds to the request from the processor and from the bus based on:

- ✓ the type of the request
- ✓ Its hit or miss status in the cache; and
- ✓ State of the cache block specified in the request

Furthermore, each block of memory is in one of the three states: Shared, Exclusive or Invalid (Not in any caches) and each cache block tracks these three states

Example: Working of Finite State Machine Controller

Today we will continue our discussion on the finite state machine controller for the implementation of snooping protocol;

and will try to understand its working with the help of example

Here, we assume that two processors P1 and P2 each having its own cache, share the main memory connected on bus

The status of the processors, bus transaction and the memory is depicted in a table for each step of the state machine

Here, the state of the machine for each processor and cache address and value cached, the bus action and shared-memory status is shown for each step of operation

Initially the cache state is invalid (i.e., the block of memory is not in the cache); and memory blocks A1 and A2 map to the same cache block where the address A1 is not equal to A2

At Step 1 P1 writes 10 to A1

- I. write miss on bus occurs and the state transition from invalid to exclusive takes place

At Step 2 P1 reads A1

- I. CPU read HITs occurs, hence the FSM Stays in exclusive state

At Step 3: P2 reads A1

- I. As P2 is initially in invalid state, therefore, read miss on the bus occurs; the controller state changes from invalid to Shared

- II. P1 being in Exclusive state, remote read write-back is asserted and the state changes from exclusive to Shared; and
- III. the value (10) is read 1 from the shared-memory at address A1, into P1 and P2 caches at A1; and both P1 and P2 controllers are in shared state

At Step 4: P2 write 20 to A2

- I. P1 find a remote write, so the state of the controller changes from shared to Invalid
- II. P2 find a CPU write, so places write miss on the bus and changes the state from shared to exclusive and writes value 20 to A1
- III. The memory address to A1 with value A1

At Step 5: P2 write 40 to A2

- ✓ P2 being in Exclusive state, CPU write Miss occurs, and initiates write-back to P2 at A2
- ✓ P2 remains in Exclusive state, with address A2 and value 40

Implementation Complications

With this example, we have observed that the finite state machine implementation of the snooping protocols works well

However, the following implementation complications have been observed

- ✓ Write Races
- ✓ Interventions and invalidations

Write Races occur when one processor wants to update the cache but another processor may get bus first and then write the same cache block!

We know that bus transaction is a two step process:

- ✓ Arbitrate for bus
- ✓ Place miss on bus and complete operation

If miss occurs to block while waiting for bus, handle miss, i.e. invalidate, and then restart.

Furthermore, to overcome the write races, split transaction bus, so that

- ✓ it can have multiple outstanding transactions for a block
- ✓ Multiple misses can interleave, allowing two caches to grab block in the Exclusive state
- ✓ Must track and prevent multiple misses for one block

Snooping Cache Conflict

In snooping cache method, the CPU assess the cache and the bus transaction checks the cache tags

Processors continuously snoop on address bus and if the address matches tag, it either invalidate or update

Since every bus transaction checks cache tags; therefore there could be interference with CPU

There are two ways to reduce the interference; the methods are:

1. Duplicate set of tags for L1 caches
 - CPU uses a different set of tags
 - The CPU gets stalled during cache access when snoop has detected a copy in the cache and tags need to be updated
2. Multi-level caches with inclusion: i.e., L2 cache already duplicate, provided L2 obeys inclusion with L1 cache; here
 - Content of primary cache (L1) is in secondary cache (L2)
 - Most CPU activity directed to L1
 - Snoop activity directed to L2
 - If snoop gets a hit then it arbitrates L1 to update and possibly get data; this will stall CPU
 - Exclusion contention

Snooping Cache Variations

MESI Protocol:

This protocol contains four (4) states

- ✓ Modified
- ✓ Exclusive
- ✓ Shared
- ✓ Invalid

Exclusive now means exclusively cached but clean upon loading

Four State Machine

Bus serializes writes, getting bus ensures no one else can perform memory operation

On a miss in a write back cache, may have the desired copy and its dirty, so must reply

Add extra state bit to cache to determine shared or not

Add 4th state Modify that Modifies for exclusive writes

Snooping Cache Variations: Berkeley Protocol

The main idea is to allow cache to cache transfers on the shared bus

the cache that has the block in a Dirty state is the owner of that block:

The last one who writes, is the owner

The owner responsible to transfer data if read occurs and to update main memory; If a

block is not owned by any cache, memory is the owner

Summary Snooping Cache Variations: Summary

Illinois Protocol

- ✓ Private Dirty
- ✓ Private Clean
- ✓ Shared
- ✓ Invalid

If read sourced from memory, then Private Clean
if read sourced from other cache, then Shared
Can write in cache if held private clean or dirty

MESI Protocol

- ✓ Modified (private, ≠Memory)
- ✓ eXclusive (private, =Memory)
- ✓ Shared (shared, =Memory)
- ✓ Invalid

Snoop Cache Extensions

Extensions:

- A. Berkeley Protocol:
 - ✓ Fourth State: Ownership
 - ✓ Shared-Exclusive
- B. MESI Protocol:
 - ✓ Clean exclusive state (no miss for private data on write)
- C. Illinois Protocol:
 - ✓ Cache supplies data when shared state (no memory access)

Larger Microprocessors

Use separate Memory per Processor

Local or Remote access via memory controller

1 Cache Coherency solution is using non-cached pages

Alternative is to use: directory containing information for every block in memory that tracks state of every block in every cache, which caches have a copies of block, dirty vs. clean, etc

The use of information per memory block vs. per cache block has some plus and minus points

- ✓ PLUS: In memory => simpler protocol as compared to centralized/one location
- ✓ MINUS: In memory => directory is function of memory size) as compared to simple protocol where director is function of cache size

Directory Based Protocol: Distributed Shared Memory**Directory Based Protocol**

The director base protocol is similar to Snoopy Protocol:

The Three states of the protocol are:

- ✓ Shared: → 1 processors have data, memory up-to-date
- ✓ Uncached (no processor has it; not valid in any cache)
- ✓ Exclusive: → 1 processor (owner) has data; memory out-of-date

In addition to cache state, must track which processors have data when in the shared state (usually bit vector, 1 if processor has copy)

Keep it simple(r):

- ✓ Writes to non-exclusive data => write miss
- ✓ Processor blocks until access completes
- ✓ Assume messages received and acted upon in order sent

Ps" f yw"er h"hsr x" { er x"s" f vsehgewx<

- ✓ interconnect no longer single arbitration point
- ✓ all messages have explicit responses

Typically 3 processors involved

- ✓ Local node where a request originates
- ✓ Home node where the memory location of an address resides
- ✓ Remote node has a copy of a cache block, whether exclusive or shared

Example messages are as follows: Here P is used for processor number, A for address

<u>Message type</u>	<u>Source</u>	<u>Destination</u>	<u>Msg Content</u>
Read miss Processor P reads data at address A; make P a read sharer and arrange to send data back	Local cache	Home directory	P, A
Write miss Processor P writes data at address A; make P the exclusive owner and arrange to send data back	Local cache	Home directory	P, A
Invalidate Invalidate a shared copy at address A.	Home directory	Remote caches	A
Fetch Fetch the block at address A and send it to its home directory	Home directory	Remote cache	A
Fetch/Invalidate Fetch the block at address A and send it to its home directory; invalidate the block in the cache	Home directory	Remote cache	A
Data value reply Return a data value from the home memory (read miss response)	Home directory	Local cache	Data
Data write-back Write-back a data value for address A (invalidate response)	Remote cache	Home directory	A, Data

State Transition Diagram for an Individual Cache Block in a Directory Based System

States identical to snoopy case;

Transactions very similar.

Transactions are caused by read misses, write misses, invalidates, data fetch requests

Generates read miss & write miss messages to home directory.

Write misses that were broadcast on the bus for snooping results in explicit invalidate & data fetch requests.

Note: on a write, a cache block is bigger, so need to read the full cache block

CPU - Cache State Machine

State machine for CPU requests for each memory block

Invalid state if in memory

State Transition Diagram for the Directory

Here, the same states & structure is shown as the transition diagram for an individual cache

Two actions performed are:

1. update of directory state and
2. send messages to satisfy requests

The controller tracks all copies of memory block; and also indicates an action that updates the sharing set, called Sharers, as well as sending a message

Directory State Machine

State machine for Directory requests for each memory block

Un-cached state if in memory

Example Directory Protocol

Message sent to directory causes two actions:

- ✓ Update the directory
- ✓ More messages to satisfy request

Block is in Uncached state: the copy in memory is the current value; only possible requests for that block are:

- ✓ Read miss: requesting processor sent data from memory & requestor made only sharing node; state of block made Shared
- ✓ Write miss: requesting processor is sent the value & becomes the Sharing node. The block is made Exclusive to indicate that the only valid copy is cached. Sharers indicates the identity of the owner.

Block is Shared state => the memory value is up-to-date; the read miss and write miss activities are:

- ✓ Read miss: requesting processor is sent back the data from memory & requesting processor is added to the sharing set.
- ✓ Write miss: requesting processor is sent the value. All processors in the set Sharers are sent invalidate messages, & Sharers is set to identity of requesting processor. The state of the block is made Exclusive.

Block is Exclusive: Current value of the block is held in the cache of the processor identified by the set Sharers (the owner), three possible directory requests:

- ✓ Read Miss
- ✓ Data Write back
- ✓ Write Miss

Read miss:

- ✓ Cache to transition to Shared; and
- ✓ Causes owner to send data to directory, where it is written to memory & sent back to requesting processor
- ✓ Identity of requesting processor is added to set Sharers, which still contains the identity of the processor that was the owner (since it still has a readable copy). State is shared.

Data write-back:

- ✓ Owner processor is replacing the block and hence must write it back, making memory copy up-to-date
- ✓ The block is now Uncached, and the Sharer set is empty.

Write miss:

- ✓ Block has a new owner.
- ✓ A message is sent to old owner causing the cache to send the value of the block to the directory from which it is sent to the requesting processor, which becomes the new owner.
- ✓ A sharer is set to identity of new owner, and state of block is made Exclusive.

Summary

Caches contain all information on state of cached memory blocks
 Snooping and Directory Protocols are similar;
 However, bus makes snooping easier because of broadcast
 Directory has extra data structure to keep track of state of all cache blocks

Lecture 37

Multiprocessors

(Performance and Synchronization)

] ✓pm ""] ✓..ω"

Performance of Multiprocessors with

- ✓ Symmetric Shared-Memory
- ✓ Distributed Shared Memory

Synchronization in Parallel Architecture

Conclusion

Recap: Cache Coherence Problem

So far we have discussed the sharing of caches for multi-processing in the:

- ✓ Symmetric shared-memory architecture
- ✓ Distributed shared memory architecture

We have studied cache coherence problem in symmetric and distributed shared-memory multiprocessors; and have noticed that this problem is indeed performance-critical

Recap: Multiprocessor cache Coherence

Last time we also studied the cache coherence protocols, which use different techniques to track the sharing status and maintain coherence without performance degrading

These protocols are classified as:

- ✓ Snooping Protocols
- ✓ Directory-Based Protocols

These protocols are implemented using a FSM controller

Recap: Snooping Protocols

Snooping protocols employ write invalidate and write broadcast techniques

Here, the block of memory is in one of the three states, and each cached-block tracks these three states; and the controller responds to the read/write request for a block of memory or cached block, both from the processor and from the bus

Recap: Implementation Complications of snoopy protocols

The three states of the basic FSM are: Shared, Exclusive or Invalid

However, the complications such as: write races, interventions and invalidation have been observed in the implementation of snoopy protocols; and

to overcome these complications number of variations in the FSM controller have been suggested

These variations are: MESI Protocol, Barkley Protocol and Illinois Protocol

Recap: Variations in snoopy protocols

These variations resulted in four (4) states FSM controller

- ✓ The states of MESI Protocol are: Modify, Exclusive, Shared and Invalid
- ✓ The states of Barkley Protocol are: Owned- Exclusive, Owned-Shared, Shared and Invalid; and of
- ✓ Illinois Protocol are: Private Dirty, Private clean, shared and Invalid

Recap: Directory based Protocols

The larger multiprocessor systems employ distributed shared-memory , i.e., a separate memory per processor is provided

Here, the Cache Coherency is achieved using non-cached pages or directory containing information for every block in memory

The directory-based protocol tracks state of every block in every cache and finds the caches having copies of block being dirty or clean

The directory-based protocol tracks state of every block in every cache and finds the caches having copies of block being dirty or clean

Similar to the Snoopy Protocol, the directory-based protocol are implemented by FSM having three states: Shared, Uncached and Exclusive

Recap: Directory Based Protocols

These protocols involve three processors or nodes, namely: local, home and remote nodes

- ✓ Local node originates the request
- ✓ Home node stores the memory location of an address
- ✓ Remote node holds a copy of a cache block, whether exclusive or shared

The transactions are caused by the messages such as: read misses, write misses, invalidates or data fetch requests

These messages are sent to the directory to cause actions such as: update directory state and to satisfy requests

The controller tracks all copies of memory block; and indicates an action that updates the sharing set

Example: Working of Finite State Machine Controller

Now are going to discuss the state transition and messages generated by FSM controller in each state to implement the directory-based protocols.

We consider an example distributed shared-memory multiprocessor having two processors P1 and P2 where each processor has its own cache, memory and directory. Here, if the required data is not in the cache and is available in memory associated with the respective processor, then the state machine is said to be in Uncached state; and transition to other states is caused by messages such as: read miss, write miss, invalidates and data fetch request.

Dealing with read/write misses

A1 and A2 map to the same cache block

Let us assume that initially the cache states are Uncached (i.e., the block of data is in memory); and at the first step P1 writes 10 to address A1, here the following three activities take place:

1. The bus action is write miss and the processor P1 places the address A1 on the bus;
2. the data value reply message is sent to the controller, P1 is inserted in the directory sharer-set {P1}; and
3. the state transition from Uncached to exclusive takes place, these operations are shown here in the red color.

At Step 2: P1 reads A1; CPU read HITs occurs, hence the FSM stays in exclusive state.

At Step 3: P2 reads A1

- i. Read miss occurs on the bus as P2 is initially in Uncached state; the controller states of P1 and P2 change from Uncached to Shared
- ii. P1 being in Exclusive state, remote read write-back is asserted and the state changes from exclusive to Shared; and
- iii. The value (10) is read 1 from the shared-memory at address A1, into P1 and P2 caches at A1; and both P1 and P2 controllers are inserted in sharer-set {P1,P2}

Working of FSM Controller

At Step 4: P2 write 20 to A2

- i. As A1 and A2 maps to the same cache block; P1 find a remote write, so the state of the controller changes from shared to Invalid
- ii. P2 find a CPU write, so places write miss on the bus and changes the state from shared to exclusive and writes value 20 to A1
- iii. The director addresses to A1 with sharer-set containing {P2}

A1 and A2 map to the same cache block

At Step 5: P2 write 40 to A2

- i. P2 being in Exclusive state, P2 write Miss at A2 occurs
- ii. Director of A2 is in exclusive state and places P2 in the sharer-set {P2}
- iii. P2 write-back 20 at A1 completes; the directory at A1 is in Uncached state; the sharer-set is empty and value 20 is placed in the memory
- iv. P2 remains in Exclusive state, with address A2 and value 40

A1 and A2 map to the same cache block

Performance of Multiprocessors: Symmetric Shared-Memory Architecture

In bus-based multiprocessor using an invalidation protocols, several phenomenon combine to determine performance:

Overall cache performance is combination of the behavior of the Uniprocessor cache miss-traffic and the traffic caused by the communication due to invalidation and subsequent cache miss

Changing processor count, cache size and block size effect these two components of miss rate

- ✓ The misses arising from inter-processor communication, called coherence misses, can be from two sources true sharing and false sharing.

True Sharing: The so-called true sharing misses arise from communication of data through cache-coherence mechanism

- ✓ The first write by processor to a shared cache-block caused an invalidation to establish ownership of that block
- ✓ When another processor attempts to read modified word, a miss occurs and the resultant block is transferred
- ✓ Both the misses are classified as true-sharing misses, as they arise from the sharing of data

False Sharing: it arise from the use of invalidation-base coherence algorithm with a single valid bit per cache block

- ✓ False sharing occurs when a block is invalidated and a subsequent reference causes a miss. i.e.,
- ✓ the word being written and the word read are different and the invalidation does not cause a new value to be communicated, but only causes an extra cache miss
- ✓ Here, the block is shared but no word in the block is shared and the miss would not occur if the block size were a single word

Example of True and False Sharing:

Considering the previous example, assume the words A1 and A2 are in the same cache block, which is in the shared state in the caches of P1 and P2

Let us identify the true-sharing miss and false sharing miss for the following sequence of events

Time	P1	P2
1	Write A1	
2		Read A2
3	Write A1	
4		Read A2
5	Write A2	

Event 1: P1 Write A1 is true sharing miss, since A1 was read by P2 and needs to be invalidated from P2

Event 2: P2 Read A2 is false sharing miss, since A2 was invalidated by the write of A1 in P1, but the value of A1 is not used in P2

Event 3: P1 Write A1 is false sharing miss; since the block containing A1 is marked shared due to read in P2; but P2 did not read A1

Event 4: P2 Write A2 is false sharing miss; since the block containing A2 is marked shared due to read in P2 (event 2); but P2 did not Write A2

Event 5: P1 Read A2 is true sharing miss; since the value being read by P2 was written by P2 (in event 4)

Performance of Multiprocessors

Distributed Shared-Memory Architecture

The performance of directory-based multiprocessors depends on many of the same factors (such as processor count, cache size and block size etc.) that influence the performance of bus-based multiprocessor

In addition, the location of requested data item which depends on both the initial allocation and sharing pattern also influence the performance of distributed shared-memory architecture

Here, the distribution of memory requests between local memory and remote memory is key to the performance, because it affects both the consumption of both global bandwidth and latency seen by the requests

This can be visualized from these figures

Here the cache misses are separated into the local and remote requests

The graphs for data miss rate vs. cache size, obtained using same benchmarks, show that miss rate decrease as cache size grow

Note that there is a steady decrease in the local miss rate while the decline in the remote miss rate depend on coherence misses

In all cases shown here, the decrease in the local miss rate is larger than the decrease in the remote miss rate

The graphs for data miss rate vs. block size, obtained using same benchmarks, show that miss rate decrease as block size increases

Synchronization

Why Synchronization?

- ✓ While using multiprocessor architecture, we need to know when it is safe for different processes to use shared data
- ✓ This is accomplished by using the synchronization mechanisms

These mechanisms are built with user-level software routines that rely on the hardware supplied synchronization instructions

For small multiprocessors Uninterruptable instruction are used to fetch and update memory which is referred to as the atomic operation

For large scale multiprocessors, synchronization can be a bottleneck

Several techniques have been proposed to reduce contention and latency of synchronization

Here, we will examine the hardware primitives to implement synchronization and then construct synchronization routines

Hardware Primitives: Uninterruptable Instructions

The basic requirement to implement synchronization in a multiprocessor is the set of hardware primitives with the ability to atomically read and modify a memory location,

- ✓ i.e., read and modify are performed in one step

One typical operation that interchanges a value in a register for a value in memory is referred to as Atomic exchange

There are number of other atomic primitives that can be used to implement synchronization

The key property of these atomic primitives is that they read and update a memory value atomically

The other such operations used in many old multiprocessors is Test-and-Set and fetch-and-increment etc.

Now let us understand how the atomic operation work?

Atomic Exchange: To see how we can use this primitive to build synchronization, let us assume we want to build a simple lock where

- ✓ 0 indicates that lock is free; and
- ✓ 1 indicates that lock is unavailable

To implement synchronization, a processor tries to set the lock by exchange of 1, which is in the register, with the memory address corresponding to the lock

The value returned from the exchange instruction is 1 if some other processor had already claimed access, otherwise the value returned is 0; i.e.,

The synchronization is locked and unavailable if some other processor had already claimed access; otherwise the value returned is 0

In the later case, where the value returned is 0, the value is changed to 1, preventing any competing exchange from also retrieving 0

Example:

Consider two processors trying to exchange simultaneously

This race is broken when one of the processor exchange first and returns 0, and the second processor will return 1 when it does the exchange

Test-and-set: tests a value and sets it if the value passes the test

Fetch-and-increment: it returns the value of a memory location and atomically increments it

Key to the atomic operations is that each operation is indivisible

Implementing a single atomic instruction in hardware is complex and is hard to have read & write in one instruction; therefore

In the recent multiprocessor pair of instructions is used the two instructions are:

- ✓ Load linked (or load locked) and
- ✓ Store conditional

Here, the second instruction returns a value from which it can be deduced as if the instruction were executed as atomic

Note that:

- ✓ Load linked (LL) returns the initial value
- ✓ Store conditional (SC) returns 1 if it succeeds (no other store to same memory location since proceeding load) and 0 otherwise

These instructions are used in sequence:

- ✓ If the contents of memory location, specified by the LL are changed before the SC to the same address occurs, then the SC fails
- ✓ The store conditional returns a value 1 or 0 indicating whether the SC was successful or not

Let us consider an example program segment showing implementation of atomic exchange on memory location specified by the contents of register R1

Example doing atomic swap with LL & SC:

```

try:  MOV  R3,R4      ; mov exchange value
ll    R2,0(R1)        ; load linked
sc    R3,0(R1)        ; store conditional
beqz  R3,try          ; branch store fails (R3 = 0)
mov   R4,R2           ; put load value in R4

```

At the end of this sequence, the contents of R4 and memory location specified by R1 have been atomically exchanged

The LL SC primitive can be used to build other primitives, e.g., the atomic fetch and increment can be constructed as:

Example doing fetch & increment with LL & SC:

```

try:  ll    R2,0(R1)    ; load linked
      addi  R2,R2,#1    ; increment (OK if reg reg)
      sc    R2,0(R1)    ; store conditional
      beqz  R2,try      ; branch store fails (R2 = 0)

```

As the SC instruction simply checks that its address matches that in the link register, therefore, register-register instructions can safely be placed after the LL instruction; however, the number of instructions in between LL and SC must be kept small

Summary:

In this series of four lectures on multiprocessors we have studied how improvement in computer performance can be accomplished using Parallel Processing Architectures. Parallel Architecture is a collection of processing elements that cooperate and communicate to solve larger problems fast.

Then we described the four categories of Parallel Architecture as: SISD, SIMD, MISD and MIMD architecture.

We noticed that based on the memory organization and interconnect strategy, the MIMD machines are classified as:

- ✓ Centralized Shared Memory Architecture
- ✓ Distributed Memory Architecture

We also introduced the framework to describe parallel architecture as a two layer representation: Programming and Communication models

We talked about sharing of caches for multi-processing in the symmetric shared-memory architecture in details

Here, we studied the cache coherence problem and introduced two methods, write invalidation and write broadcasting schemes, to resolve the problem

We also discussed the finite state machine for the implementation of snooping algorithm

Today we have discussed FSM controller to implement Directory Based Protocols which involve three processors or nodes, namely: local, home and remote nodes

We discussed the state transition and messages generated by FSM controller in each state to implement the directory-based protocols

We have also discussed in details the performance of distributed and centralized shared-memory architecture

Concluding our discussion on the multiprocessor, we can say that multiprocessors are highly effective for multi-programmed work loads

More recently, multiprocessors have proved very effective for commercial workloads such as web searching

The centralized memory architecture, also known as Symmetric Multiprocessors (SMPs) maintain a single centralized memory with uniform access time; while

In contrast, the Distributed Shared-Memory Multiprocessor (DSMs) have non uniform memory architecture and can achieve greater scalability

The advantages of these two architecture, i.e., maximizing uniform memory access while allowing greater scalability can be partially combined in the Sun Microsystems's Wildfire architecture, shown here

Here, note that large SMPs (such as E6000) are used as nodes to maximize uniform memory access and greater scalability is achieved by using Wildfire Interface (WFI)

Each E6000 can accept up to 15 processors or I/O Boards on Giga-plane bus interconnect

WFI can connect 2 or 4 E6000 multiprocessors by replacing one I/O board with WFI board

You may look into further details of the Sun Microsystems's Wildfire architecture from literature and study its performance

Lecture 38

Input Output Systems (Storage and I/O Systems)

Recap:

Recap:

Disk Storage Systems

Interfacing Storage Devices

Conclusion

Recap: Multiprocessing

In last four lectures we discussed how the computer performance can be improved by Parallel Architectures

Parallel Architecture is a collection of processing elements that cooperate and communicate to solve larger problems fast

Parallel architectures are implemented as: SIMD, MISD and MIMD machines, where the MIMD machines facilitate complete parallel processing

The MIMD machines are classified as:

- ✓ Centralized Shared Memory Architecture
- ✓ Distributed Memory Architecture

The centralized memory architecture, maintain a single centralized memory with uniform access time

In contrast, the distributed Shared-Memory multiprocessors have non uniform memory architecture but offer greater scalability

The sharing of caches for multi-processing introduces cache coherence problem

In Centralized shared-memory architecture, the cache coherence problem is resolved by using write invalidation and write broadcasting schemes those implement Snooping algorithm

In Distributed shared-memory architecture, the cache coherence problem is resolved by using Directory Based Protocols

Recap: outside processor

Today the :

- ✓ Processing Power doubles every 18 months
- ✓ Memory Size doubles every 18 months; and
- ✓ Disk positioning rate (Seek + Rotate) doubles every 10 Years

Recall the 2nd lecture, where we discussed the quantitative principles to define the computer performance, we noticed that the execution time of CPU is not the only measure of computer performance

Introduction: outside the processor

The overall performance of a computer is measured by its throughput, which is very much influenced by the systems external to the processor

As we have already pointed out in 25th lecture that measuring the overall performance of a powerful Uni-processor or a parallel processing architecture without considering the

I/O devices and their interconnection, is just like trying to determine the road performance of a car, which is fitted with powerful engine but is without wheels

The effect of neglecting the I/Os on the overall performance of a computer system can best be visualized by Amdahl's Law which identifies that: system speed-up limited by the slowest part!

Let us consider computer whose response time is 10% longer than the CPU time

If the CPU time is speeded up by a factor of 10 then neglecting the I/Os, the overall speed up as determined using the Amdahl's Law is 5; i.e.,

Half of what we would have achieved if both the CPU time and I/O time were sped up 10 times

In other words we can say 50% loss in the speed-up

Similarly, if the CPU time is speeded up 100 times and neglecting the I/Os, the overall speed up is 10; i.e.,

10% of what we would have achieved if both the CPU time and I/O time were sped up 100 times

In other words we can say that ignoring the I/Os there is 90% loss in the speed-up

Thus, I/O performance increasingly limits the system performance and efficiency

After having detailed discussion on the performance enhancement of:

- ✓ instruction Set Architecture
- ✓ computer hardware
- ✓ instruction level parallelism
- ✓ memory hierarchy systems and
- ✓ parallel processing architecture

We are, now, going to focus our discussion on the study of the systems outside processor, i.e., I/O systems

I/O System

An I/O system comprises storage I/Os and Communication I/Os

The Storage I/Os consist of Secondary and Tertiary Storage Devices; and
 The communication I/O consists of I/O Bus system which interconnect the microprocessor and memory with the I/O devices
 Today we will talk about the storage I/O
 The secondary and tertiary storages include: magnetic disk, magnetic tape automated tape libraries, CDs, and DVDs
 These devices offer bulk data storage, but on the contrary are too large for embedded applications

Disk Storages: Technology Trends

As you can see from the plot shown here that extensive improvement have been made in the disk capacity;
 before 1990 disk capacity doubled every 36 months; and now every 18 months;

Storage Technology Drivers

This improvement in the technology trend is driven by the prevailing computing paradigm
 In 1950s computing observed migration from batch to on-line processing where as
 In 1990s on-line processing migrated to ubiquitous computing; i.e.,

- ✓ global wireless infrastructure
- ✓ nationwide fiber optical network with wireless tails

This development in processing effected the storage industry and motivated to develop:

- ✓ the smaller, cheaper, more reliable and lower power embedded storages for ubiquitous computing
- ✓ high capacity, hierarchically managed storages as data utilities

Before discussing the storage technologies, let us perceive the historical perspective of magnetic storages

Historical Perspective

1956 - early 1970s

- ✓ IBM Ramac and Winchester were developed for mainframe computers as proprietary interfaces
- ✓ Steady shrink in form factor: 27 in. to 14 in.

1970s developments

- ✓ 5.25 inch floppy disk form-factor (microcode into mainframe)
- ✓ early emergence of industry standard disk interfaces
 - ST506, SASI, SMD, ESDI

Disk History

Capacity of Unit Shown Megabytes; and
Data density: M bit/sq. in.

Historical Perspective

Early 1980s: Era of PCs and first generation workstations; and

Mid 1980s: Era of Client/server computing and Centralized storage on file server

This voyage of computing from first generation to client/server resulted in end of proprietary interfaces and:

- ✓ Accelerated disk downsizing: 8 inch to 5.25 inch
- ✓ Mass market disk drives become a reality
- ✓ industry standards: SCSI, IPI, IDE
- ✓ 5.25 inch drives for standalone PCs,

Late 1980s - Early 1990s: Era of Laptops, note-books, (palmtops)

- ✓ 3.5 inch, 2.5 inch, (1.8 inch form factors)
- ✓ Form factor plus capacity drives market,

Challenged by DRAM, flash RAM in PCMCIA cards

- ✓ unattractive M Bytes per cubic inch
- ✓ unattractive M Bytes per cubic inch

Optical disk failed on performance but found slot (CD ROM)

Disk History

1989:
63 Mbit/sq. in
60,000 MBytes

1997:
1450 Mbit/sq. in
2300 MBytes

1997:
3090 Mbit/sq. in
8100 MBytes

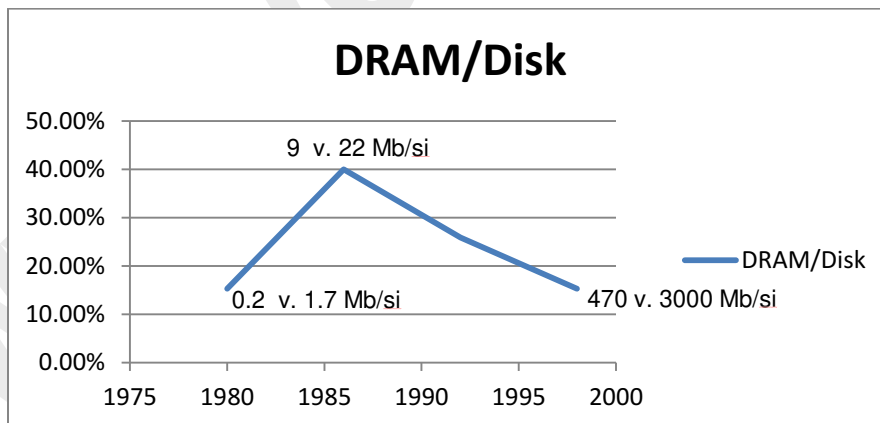
A huge disk drive changed to palm-drive

DRAM as % of Disk over time MBits per square inch:

In 1974, the use of DRAM was only 10% of the disk storage

It reached to the peak in 1986 when DRAM was 40% of the disk storage

This trend once again started reducing and was up to 15% in 1998



Alternative Data Storage Technologies: Early 1990s

Technology	Cap (MB)	BPI	TPI	BPI*TPI (Million)	Data Xfer (KByte/s)	Access Time
Conventional Tape:						
Cartridge (.25")	150	12000	104	1.2	92	min
IBM 3490 (.5")	800	22860	38	0.9	3000	sec
Helical Scan Tape:						
Video (8mm)	4600	43200	1638	71	492	45 secs
DAT (4mm)	1300	61000	1870	114	183	20 secs
Magnetic & Optical Disk:						
Hard Disk (5.25")	1200	33528	1880	63	3000	18 ms
IBM 3390 (10.5")	3800	27940	2235	62	4250	20 ms
Sony MO (5.25")	640	24130	18796	454	88	100 ms

Devices: Magnetic Disks

Purpose:

- ✓ Long-term, nonvolatile storage
- ✓ Large, inexpensive, slow level in the storage hierarchy

Characteristics:

- ✓ Seek Time (~8 ms avg)
 - positional latency
 - rotational latency

Transfer rate

- ✓ About a sector per ms (5-15 MB/s)
- ✓ Blocks

Capacity

- ✓ Gigabytes
- ✓ Quadruples every 3 years (aerodynamics)

Devices: Magnetic Disks

Speed: 7200 RPM = 120 RPS => 8 ms per rev

Ave rot. latency = 4 ms

128 sectors per track => 0.25 ms per sector

1 KB per sector => 16 MB / s

Tape vs. Disk

Longitudinal tape uses same technology as hard disk; tracks its density improvements

Disk head flies above surface, tape head lies on surface

Disk fixed, tape removable

Inherent cost-performance based on geometries: disk Vs. Tape

Disk: fixed rotating platters with gaps

(random access, limited area, 1 media / reader)

Current Drawbacks to Tape

Tape wear out:

- ✓ Helical 100s of passes to 1000s for longitudinal

Head wear out:

- ✓ 2000 hours for helical

Both must be accounted for in economic / reliability model

Long rewind, eject, load, spin-up times; not inherent,
just no need in marketplace (so far)

I/O Performance Parameters

Diversity: Which I/O device can connect to the CPU

Capacity: How many I/O devices can connect to the CPU

Latency: Overall response time to complete a task

Bandwidth: Number of task completed in specified time - throughput

The parameters diversity that refers to which I/O device and capacity means how many I/O devices can connect to the CPU are the I/O performance measures having no counterpart in CPU performance metrics.

In addition, the latency (response time) and bandwidth (throughput) also apply to the I/O system.

An I/O system is said to be in equilibrium state when the rate at which the I/O requests from CPU arriving, at the input of I/O queue (buffer) equals the rate at which the requests departs the queue after being fulfilled by the I/O device.

I/O Vs. CPU Performance

The parameters diversity refers to I/O device and capacity

It identifies how many I/O devices can connect to the CPU

Note that the I/O performance measures have no counterpart in CPU performance metrics

In addition, the latency (response time) and bandwidth (throughput) also apply to the I/O system

Recall from our discussion in the 3rd lecture, where we studied that an I/O system works on the principle of producer-server model

This model comprises an area called queue, wherein the tasks accumulate waiting to be serviced and the device performing the requested service, called server.

Producer creates tasks to be processed and place them in a FIFO buffer queue.

Server takes the task from buffer and perform them

The response time is the time task takes from the moment it arrives in the buffer to the time the server finishes the task

Disk I/O Performance Measure

The metrics of disk I/O performance are:

- ✓ Response Time is the time to Queue + Device Service time
- ✓ Throughput: the percent of the total bandwidth

The graph shows the relationship between the response time and throughput of disk I/O system

Here, the minimum response time achieves only 10% of the throughput

The response time of 100% throughput takes 7-8 times the minimum response time

I/O Transaction Time: Performance parameters

The interaction time or transaction time of a computer is sum of three times:

- ✓ Entry Time: the time for user to enter a command average 0.25 sec; from keyboard 4.0 sec.

- ✓ System Response Time: time between when user enters the command and system responds
- ✓ Think Time: the time from reception of the command until the user enters the next command

Response Time vs. Productivity

Example: Let us see what happens to transaction time as system response time shrinks from 1.0 sec to 0.3 sec?

Assume:

- ✓ with Keyboard the entry time is 4.0 sec and think time is 9.4 sec; and
- ✓ With Graphics: 0.25 sec entry, 1.6 sec think time

The upper part of graph showing the response time for conventional use (keyboard) depicts that:

$1.0 - 0.3 = 0.7\text{sec}$ off response saves 4.9 sec (34%)

And, lower graphs for graphics saves 2.0 sec (70%) of total time per transaction; i.e., shrinkage in the response time results in greater productivity

Processor Interface Issues

Processor interface

- ✓ Isolated I/O
- ✓ Memory mapped I/O
- ✓ Interrupts

I/O Control Structures

- ✓ Polling
- ✓ Interrupts
- ✓ DMA
- ✓ I/O Controllers
- ✓ I/O Processors

Capacity, Access Time, Bandwidth

Interconnections

- ✓ Busses

I/O - Processor Interface

Isolated I/O Bus is implemented as:

- ✓ Independent I/O bus
- ✓ common memory & I/O bus

It requires separate I/O instructions (in, out)

Independent I/O Bus

Common Memory & I/O Bus

WWW.VUMULTAN.COM

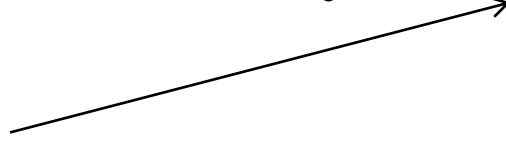
Memory Mapped I/O

Programmed I/O (Polling)

Interrupt Driven Data Transfer

Direct Memory Access

CPU sends a starting address, direction, and length count to DMAC. Then issues "start".



DMAC provides handshake signals for Peripheral Controller, and Memory Addresses and handshake signals for Memory.

Input / Output Processors

I/O Processor looks in memory for commands

Input / Output Processors

1. CPU issues instruction to IOP
- 2-3 IOP steals memory cycles.
Device to/from memory transfers are controlled by the IOP directly.
- 4 IOP interrupts CPU when done

Summary

Disk industry growing rapidly, improves:

- ✓ bandwidth 40%/yr ,
- ✓ areal density 60%/year, \$/MB faster?

queue + controller + seek + rotate + transfer

Advertised average seek time benchmark much greater than average seek time in practice

Response time vs. Bandwidth tradeoffs

Value of faster response time:

- ✓ 0.7sec off response saves 4.9 sec and 2.0 sec (70%) total time per transaction
=> greater productivity
- ✓ everyone gets more done with faster response, but novice with fast response = expert with slow

Processor Interface: today peripheral processors, DMA, I/O bus, interrupts

Lecture 39

Input Output Systems (Bus Structures Connecting I/O Devices)

Recap: "I/O"

Recap:

I/O interconnect Trends

Bus-based Interconnect

Bus Standards

Conclusion

Recap: I/O System

Last time we noticed that the overall performance of a computer is measured by its throughput, which is very much influenced by the systems external to the processor

The effect of neglecting the I/Os on the overall performance of a computer system can best be visualized by Amdahl's Law which identifies that: system speed-up limited by the slowest part!

We noticed that an I/O system comprises storage I/Os and Communication I/Os

The Storage I/Os consist of Secondary and Tertiary Storage Devices; and

The communication I/O consists of I/O Bus system which interconnect the microprocessor and memory with the I/O devices

The development in processing effected the storage industry and motivated to develop:

- ✓ The smaller, cheaper, more reliable and lower power embedded storages for ubiquitous computing; and high capacity, hierarchically managed storages as data utilities

We noticed that diversity, capacity, latency and bandwidth are the most important parameters of I/O performance measurement

I/O system works on the principle of producer-server model, which comprises an area called queue, wherein the tasks accumulate while waiting to be serviced

The metrics of disk I/O performance are:

- ✓ Response Time, which is the time to Queue + Device Service time; and
- ✓ Throughput, which is the percent of the total bandwidth

Example:

Comparing the performance of different I/Os. Assume the following parameters, and compare the time to read and write a 64Kbyte block to flash memory and disk

Flash memory takes:

- ✓ 65 ns to read 1 byte
- ✓ 1.5 μ sec. to write 1 byte and
- ✓ 5 msec. to erase 4KB

Disk Storage has:

- ✓ Average seek time = 4.0 msec.
- ✓ Average rotational delay = 8.3 msec.
- ✓ Transfer time = 4.2 MB/sec;
- ✓ Controller overhead = 0.1 msec.

Average read or write time for disk is same and is calculated as:

$$\begin{aligned}
 &= \text{Average seek time} + \text{Average rotational delay} + \text{Transfer time} + \text{Controller overhead} \\
 &= 4.0 \text{ ms} + 8.3 \text{ ms} + 64\text{KB}/4.2 \text{ MB/sec} + 0.1 \text{ ms} \\
 &= 27.3 \text{ msec.}
 \end{aligned}$$

Read time for flash is the ratio of the flash size to the read bandwidth:

$$= 64\text{KB}/1\text{B}/65\text{ns} = 4.3 \text{ ms}$$

Flash is about 6 times faster than the disk for reading 64KB

Write time for flash is sum of the erase time and the ratio of the flash size to the write bandwidth:

$$\begin{aligned}
 &= (64\text{KB}/4\text{KB}/5\text{ms}) + (64\text{KB}/1\text{B}/1.5\mu\text{s}) \\
 &= 178.3 \text{ ms}
 \end{aligned}$$

The disk is about 6 times faster than the flash for writing 64KB

Interconnect Trends

The I/O interconnect is the glue that interfaces computer system components

I/O interconnects are facilitated using High speed hardware interfaces and logical protocols

Based on the desired communication distance, bandwidth, latency and reliability, interconnects are classified as used:

Backplanes, channels, Networks

Bus-Based Interconnect

Communication on different interconnects is done via buses

Bus is a shared communication link between subsystems

The advantages of using buses are:

- ✓ Low cost: a single set of wires is shared multiple ways
- ✓ Versatility: Easy to add new devices & peripherals may even be ported between computers using common bus

Disadvantage

- ✓ The major disadvantage of a bus is that it creates a communication bottleneck, possibly limiting the maximum I/O throughput
- ✓ In server systems, where I/O is frequent, design a bus-system capable of meeting the demand of the processor is a real challenge

Bus speed is limited by physical factors, such as:

- ✓ the bus length
- ✓ the bus loading, i.e., number of devices connected to a bus

These physical limits prevent arbitrary bus speedup, which make the bus design difficult

Bus systems are classified into Two generic types as:

- ✓ I/O busses: are lengthy, facilitate to connect many types of devices, offer wide range in the data bandwidth, and follow a bus standard (I/O bus is sometimes called a channel)
- ✓ CPU memory buses: high speed, matched to the memory system to maximize memory CPU bandwidth, single device (sometimes called a backplane)

Bus Transactions

Bus transactions are usually defined with reference to the memory, i.e., what they do with memory memory read or memory write

Bus transaction includes two parts: Sending the address and Receiving the data

Read Transaction:

- ✓ Address is first sent down the bus to the memory together with asserting the read signal

The memory responds by sending the data and de-asserting the wait signal

Write Transaction:

- ✓ Address and data are sent down the bus to the memory together with asserting the write signal
- ✓ The memory stores the data and de-asserting the wait signal

Bus Transition Protocols

Bus transition or bus Communication Protocols specify the sequence of events and timing requirements in transferring information

Synchronous Bus Transfers: follows a sequence of operations relative to a common clock

Asynchronous Bus Transfers is not clocked and uses control lines (req., ack.) which provide handshaking among the devices having bus transition

Synchronous Bus Protocols

The address transmitted in the 1st clock, using control lines to indicate the type of request; the read begins when NOT READ is asserted

The data are not ready until the wait signal is reasserted

Asynchronous Handshake

The asynchronous bus is not clocked, rather it is self timed

Hand shaking protocols are used between the bus sender and receiver

t0 : Master has obtained control and asserts address, direction, data; Waits a specified amount of time for slaves to decode target

t1: Master asserts request line

t2: Slave asserts ack, indicating data received

t3: Master releases req

t4: Slave releases ack

Read Transaction

Read Transaction

t0 : Master has obtained control and asserts address, direction, data; Waits a specified amount of time for slaves to decode target\

t1: Master asserts request line

t2: Slave asserts ack, indicating ready to transmit data

t3: Master releases req, data received

t4: Slave releases ack

Bus Arbitration Protocols

Having understood the bus transactions, the most important is to understand how is a bus reserved by a device that wishes to communicate when multiple devices need the bus access?

This is accomplished by introducing one or more bus masters into the system

A Bus Master has ability to control the bus requests and initiate a bus transaction

Bus Slave is module activated by the master for transaction

In a simple system processor is a bus master as it initiates a bus request; and memory is usually a slave

Alternately, a system may have multiple bus masters, each of which may initiate a bus transfer to the same slave

This will create chaos; as it is similar to when number of students (masters) in a class room start asking questions to the instructor (slave)

How the instructor will overcome this situation?

The instructor must have a protocol to decide who is the next (master) to talk

Similarly, the protocol to manage the bus transaction by more than one masters is referred to as Bus Arbitration Protocol

Bus Arbitration Protocol provide the mechanism for arbitrating (deciding) access to the bus so that it is used in a cooperative manner

Here, a device or processor (master) wanting to use the bus signals a bus-request and is later granted the bus

Once the bus is granted the master uses the bus and when finished the transaction signals the arbiter that bus is no more required

The arbiter then may grant the bus to another master

The multiple-master bus have a set of three control lines for performing the request, grant and release operation

The bus arbitration schemes usually try to balance two factors:

- ✓ Bus-priority: every device has certain priority; the device with highest priority should be serviced first
- ✓ Fairness: every device that want to use the bus is guaranteed to get the bus eventually

The bus arbitration schemes can be classified as:

- ✓ Daisy Chain Arbitration
- ✓ Centralized Parallel Arbitration
- ✓ Distributed Arbitration

Bus Arbitration Schemes

The bus-grant line is run through the devices from highest-to-lowest priority

If the device has requested bus access, it uses the grant line to determine access has been given to it

Bus Arbitration Schemes

Sequence of Daisy Chain Arbitration

1. Signal the request line
2. Wait for a transition on the grant line from low-to-high (it indicates that bus is being reassigned)
3. Intercept the grant signal, and do not allow the lower priority devices to see it. (stop asserting the request line)
4. Use the bus
5. Signal that the bus is no longer required by asserting the release line

Centralized Parallel Arbitration

- ✓ This scheme uses multiple request lines
- ✓ The devices independently request the bus
- ✓ A centralized arbiter chooses from among the devices, request the bus access and notify the selected device that is now the bus-master

Distributed Arbitration schemes are classified as:

- ✓ Distributed arbitration by self-selection
 - This scheme also uses multiple request line
 - The devices requesting the bus access determine who will be granted the access
 - Here, each device wanting the access places a code indicating its identity on the bus
 - By examining this code, the devices can determine the highest priority device that has made request
- ✓ Distributed arbitration by Collision Detection
 - In this scheme each device independently request the bus
 - Multiple simultaneous requests result in collision
 - A device is selected among the collided devices based on the priority

Bus Options: Design Decisions

Option	High performance	Low cost
Bus width	Separate address & data lines	Multiplex address & data lines
Data width	Wider is faster (e.g., 32 bits)	Narrower is cheaper (e.g., 8 bits)
Transfer size	Multiple words has less bus overhead	Single-word transfer is simpler
Bus masters	Multiple (requires arbitration)	Single master (no arbitration)
Split transaction?	Yes separate Request and Reply packets gets higher bandwidth	No continuous connection is cheaper and has lower latency (needs multiple masters)
Clocking	Synchronous	Asynchronous

Bus Design Decisions

The decisions regarding design of a bus system depend on:

1. Bus Bandwidth
2. Data width
3. Transfer size

Based on the bus bandwidth; separate address and data buses are used for high performance while the multiplexed address and data line are used for low cost design

Based on the data width; wider (64-bit) data bus is recommended for high performance systems and narrow (8-bit) offers cheap solution

Based on the transfer size, multiple word are transferred for high performance computing as it offers less overhead while single word transfer is used for low cost design as it is simple

Split transaction, Bus masters, and clocking are other important parameters in bus design decisions

Based on the bus masters, multiple master are used in high performance computing; and single master that involve no arbitration is used for low cost systems

Split transaction is used for high performance design where separate requests and reply packets get higher bandwidth; it involves multiple masters

The synchronous multiple masters protocols are described hereafter

Synchronous Bus Protocols- Multiple Masters

Where as bus has multiple masters, the multiple processors or I/O devices can initiate bus transaction

Here, the bus can offer higher bandwidth using packets as opposed to holding the bus for full transaction

This technique is called a split transaction or pipelined bus

Here, the bus events are divided into number of requests and replies; so the bus can be used in time between request and reply

The split transaction makes the bus available for other masters while the memory reads the word from requested address

Bus Standards

Clock rate: 5 MHz / 10 MHz (fast) / 20 MHz (ultra)

Width: $n = 8$ bits / 16 bits (wide);

up to $n = 32$ bits

For $n = 8$ bits, the bus is divided into two 8-bit channels

SCSI controller and the SCSI disks

SCSI: Small Computer System Interface

Bus Free: No device is currently accessing the bus

Arbitration: When the SCSI bus goes free, multiple devices may request (arbitrate for) the bus; fixed priority by address

Selection: informs the target that it will participate (Reselection if disconnected)

Command: the initiator reads the SCSI command bytes from host memory and sends them to the target

Data Transfer: data in or out, initiator: target

Message Phase: message in or out, initiator: target (identify, save/restore data pointer, disconnect, command complete)

Status Phase: target, just before command complete

1993 I/O Bus Survey (P&H, 2nd Ed)

1993 MP Server Memory Bus Survey

Lecture 40

Input Output Systems (RAID and I/O System Design)

Recap: "I/O"

Recap:

Redundant Array of Inexpensive Disks

I/O Benchmarks

I/O System Design

Conclusion

Yqom. **RAID** "pq wq "...q"r/y nzoq

Last time we compared the performance of disk storage and flash memory

We noticed that flash is six times faster than the disk for read and the disk is six times faster than the flash for data write

Then we discussed the trends in I/O inter-connects as: the networks, channels and backplanes

The networks offer message-based narrow-pathway for distributed processors over long distance

Recap: I/O Interconnects

The backplanes offer memory-mapped wide pathway for centralized processing over short distance

The interconnects are implemented via buses

The buses are classified in two major categories as the I/O bus and CPU-Memory bus

The channels are implemented using I/O buses and backplanes using CPU-Memory buses

Recap: I/O buses

Then we discussed the bus transition protocols which specify the sequence of events and timing requirements in transferring information as synchronous or asynchronous communication

Y i "eps" hmgw h"fyw"ef nexs "t vsxgsp" "x i "t vsxgsp" s"i w i zi "x i "fyw" f}"e" device that wishes to communicate when multiple devices need the bus access

Here, we noticed that the bus arbitration schemes usually try to balance two factors:

Recap: I/O System

Bus-priority: the device with highest priority should be serviced first

Fairness: every device that want to use the bus is guaranteed to get the bus eventually

The three bus arbitration schemes are:

- ✓ Daisy Chain Arbitration
- ✓ Centralized Parallel Arbitration
- ✓ Distributed Arbitration

Storage I/O Performance

Now having discussed the basic types of storage devices and the ways to interconnect them to the CPU, we are going to look into the ways to evaluate the performance of storage I/O systems

We know that if a storage device crashes then prime objective of a storage device should be to remember the original information to make storage device reliable

The reliability of a system can be improved by using the following four methods

Reliability Improvement

Fault Avoidance prevent fault occurrence by construction

Fault Tolerance providing service complying with the service specification by redundancy

Error Removal minimizing the presence of errors by verification

Error Forecasting to estimate the presence, creation and consequence of errors by evaluation

Reliability, availability and dependability

The performance of storage I/Os is measured in terms of its reliability, availability and dependability

These terminologies have been defined by Laprie; in the paper entitled

Fi t i r h e f p "E s q t y n k " e r h " H e y p V s p v e r g i < E s r g i t x w e r h " V q m s p k } =

Published in the Digest of papers of 15th Annual Symposium on Fault Tolerant Computing (1985)

Dependability

Laprie defined dependability as the quality of delivered service such that reliance can justifiably be placed on this service;

Where the service delivered by a system is its observed actual behavior and the system failure occurs when actual behavior deviates from the specified behavior

Note that a user perceives a system alternating between two states of delivered service; these states are:

- ✓ Service Accomplishment service is delivered as specified and
- ✓ Service Interruption delivered service is different from the specified service

Quantifying the transitions between service accomplishment and service interruption is the measure of the dependability

The dependability is measured in terms of the measure of:

- ✓ module reliability, which is the measure of the continuous service accomplishment;
- ✓ and, module availability, which is the measure of the swinging between the accomplishment and interruption states of delivered service

Measuring Reliability

Now before we discuss the reliable and dependable designs of the storage I/O let us understand the terminologies used to measure reliability, availability and dependability

The reliability of a module is the measure of the time to failure from a reference initial instant

In other words we can say the Mean Time To Failure (MTTF) of a storage module, a disk, is the measure of reliability; and

The reciprocal of the MTTF is the rate of failure; and

The service interruption is measured as the Mean Time To Repair (MTTR)

Now let us understand, with the help of an example, how can we use these terminologies to measure the availability of a disk subsystem

Example

Consider a disk subsystem comprising the following component

- ✓ 10 disks
- ✓ 1 SCSI controller
- ✓ 1 SCSI cable
- ✓ 1 power supply
- ✓ 1 fan

For the given values of MTTF of each component; find the system failure rate and hence the system MTTF

10 disks, each with MTTF = 1,000,000 Hrs

1 SCSI controller with MTTF = 500,000 Hrs

1 SCSI cable with MTTF = 1,000,000 Hrs

1 power supply with MTTF = 200,000 Hrs

1 fan with MTTF = 200,000 Hrs

Solution:

System Failure Rate

$$= 10 (1/1,000,000) + 1/500,000 + 1/1,000,000 + 1/200,000 + 1/200,000$$

$$= 23 / 1,000,000 \text{ Hrs}$$

System MTTF = 1/Failure Rate = 1,000,000/23

$$= 43,500 \text{ Hrs} = 5 \text{ years}$$

Availability

The availability of a module is the measure of the service accomplishment with respect to the swinging between the two states of accomplishment and interruption

The module availability, therefore can be quantified as the ratio of the MTTF and Mean Time Between Failure MTBF (which is equal to the sum of MTTF and MTTR); i.e.,

$$\text{Availability} = \text{MTTF} / (\text{MTTF} + \text{MTTR})$$

$$= \text{MTTF} / \text{MTBF}$$

Network Attached Storages and Reliability

Last time we discussed the disk storages and their interface with the processor using channel and backplane interconnects; and talked about the impact of disk storages and interconnects on the overall performance of the complete computing system

Today we will discuss the network interconnects to interface multiple processors located at a long distance and need high performance storage service

A network provides well defined physical and logical interfaces; i.e., interconnect separate CPU and storage system

The networks are capable of sustaining high bandwidth transfer and their file-server Operating system supports remote file access

Hence, the network attached storages are more vulnerable to the reliability and their dependability is very high

Network Attached Storage

Network Attached Storages and Reliability

So to improve both the availability and performance of storage system, disk arrays are introduced, which contain many low cost disks

The throughput of disk arrays is improved by having high bandwidth disk system which employ many small disk drives; and

VI i "X vsykl t yX"sj"e"hmoe"ewe} "w"mgvi ewi h"f } "l ezmk"q er } "wq ep"evq w'sr "wq ep"5022 " 30 -"hmohvzi w\ex i v\X er "sri "s rk"evq "sr "e"ek i v"m"36 " 46 -"er h

Manufacturing Advantages of Disk Arrays

Replace Small # of Large Disks with Large # of Small Disks! (1988 Disks)

Network Attached Storages and Reliability

Simply spreading the data over many disk forces access to may several disks and hence improve the throughput

The drawback to an array with more devices is that dependability and hence the reliability decreases generally N devices have $1/N$ reliability

Array Reliability: Example

Reliability of N disks = Reliability of 1 Disk $\div$ N

Disk system MTTF = 50,000 Hours $\div$ 70 disks
= 700 hours

Drops from 6 years to 1 month!

However, the dependability can be improved by adding redundant disks to the array to tolerate faults

Arrays (without redundancy) too unreliable to be useful!

Subsystem Organization

Redundant Arrays of Disks

In a disk array, files are "striped" across multiple spindles

Adding redundant disk to achieve high fault tolerance yields high data availability

Here, if Disks fails, the contents are reconstructed from data redundantly stored in the array

However the drawbacks of redundant disk are:

- ✓ Capacity penalty to store it
- ✓ Bandwidth penalty to update

System-Level Availability

Redundant Arrays of Disks

These systems are known as RAID:

- ✓ Redundant Array of Inexpensive Disks or
- ✓ Redundant Array of Independent Disks

There exist several different approaches to include redundant disks in the disk array

These approaches are usually classified by numerical value which identifies the RAID level

Each of these techniques have different overheads and performance

The fault tolerance and overhead in redundant disk for RAID having 8 disks of user data is as given below:

RAID Level	No. of disk faults survived	Corresponding check disks
0. No Redundancy	0	0
1. Mirrored	1	8
2. Memory Style ECC	1	4
3. Bit Interleaved Parity	1	1
4. Block Interleaved Parity	1	1
5. Block interleaved distributed parity	1	1
7. P+Q Redundancy	2	2

RAID 0 Non Redundant Striped

RAID 0 is the disk array without any redundant disk

However, here the data is striped across a set of disks which makes the collection appears to the software as a single large disk

Note that the taxonomy RAID 0 is a misnomer as there is no redundant disk; but as the data stripping is used here, so it is normally referred to as the RAID

RAID 1: Disk Mirroring/Shadowing

Gepl "hmo"m]yp "hyt pgei h"sr s"nw\$w ehs{ " "

Targeted for high I/O rate

Whenever data are written to one disk those data are also written to redundant disk

If a disk fails, the system just goes to the mirror, so there are 8 survivals in this example provided one disk of mirrored pair fails

It is the most expensive solution: 100% capacity overhead

One Logical write = two physical writes

If the data worth 4 disk is to be striped and stored on 8 disks, there are two way to strip the data

1. RAID 1+0

Create 4 pairs of disks, each organized as RAID 1
and then strip data across the 4 RAID pairs

2. RAID 0+1

Create two sets of 4-disks, each organized as RAID 0
and Mirror write to both RAID 0

Note that since 2001, as there is no commercial implementation of RAID 2, we will not discuss this technique

RAID 3: Bit-Interleaved Parity Disk

Rather than having a complete copy of the original disk, we can achieve desired dependability by adding enough redundant information to restore the lost information on failure

RAID 3 uses one extra disk, called Parity disk, that holds the check information in case of failure

RAID 3 act logically as single high capacity, high transfer rate disk

The arms are synchronized logically and spindles rotationally

Here, every read or write access goes to all the disk
 For every read access, the parity is computed across recovery group to protect against hard disk failures
 Note that for the RAID 3 shown here, there is 33% capacity cost for parity
 However, the wider arrays reduce capacity costs, but decreases expected availability and increases reconstruction time

RAID 4: Block-Interleaved Parity and

RAID 5: Distributed Block-Interleaved Parity

Both the RAID 4 and RAID 5 levels use the same ratio of data disk to parity disk as RAID 3, but they access data differently

The distribution of data in RAID 4 versus RAID 5 is shown here.

In the Block-Interleaved Parity RAID 4, the parity disk is associated to each data block, identical to RAID 3

So it supports a mixture of small read and small writes and large read and large writes

However, one drawback of this system is that the parity disk must be updated on every write, which is bottleneck for back-to-back write

This bottleneck is resolved in Block interleaved parity RAID 5, where the parity disk is distributed among the blocks

Note from the RAID 5 organization shown here that the parity associated each row of the data block is no longer restricted to a single disk

Hence, this organization allows multiple writes to occur simultaneously as long as the stripe-units are not located in the same disk

For example:

1st write to block 8 must also access its parity block P2 (i.e., two reads from two disks the 1st and 3rd disks) and

RAID 4: Block-Interleaved Parity

RAID 5: Distributed Block-Interleaved Parity

RAID 4 and RAID

2nd write to block 5 imply an update in P1 (i.e., two reads from two disks the 2nd and 4th disks)

Thus, the two write could occur at the same time in parallel

Where as we you look into the organization of RAID 4, both the P1 and P2 are on the same disk (5th disk) so it would be bottleneck and could not be written simultaneously

RAID 4 and RAID 5

In RAID 4 and RAID 5, the parity is stored as blocks and is associated with a set of data blocks

In RAID 3 every access goes to all the disks while the levels 4 and 5 use smaller accesses which allow independent access to occur in parallel

In RAID 4 and RAID 5 error detection information in each sector is checked
 of information needed to recalculate the parity

Let us compare the recalculation of parity on small write for RAID level 3, 4 and 5

Let us assume that we have 4 blocks of data and block of parity

The parity calculation of RAID 3, shown here, is straightforward

Small Writes update on RAID 3**RAID 3 verses RAID 4 and RAID 5**

Note that here the new data D0 comes directly from CPU, so disk are not involved in reading it

The small writes in case of RAID 4/5 are as shown here

Which bit will change

RAID 4/RAID 5: Small Writes**RAID 3 versus RAID 4 and RAID 5**

Once it has been checked, then the old parity P is read and corresponding bits are

This is accomplished by the logical EX-ORs

involving just 2 disks

Hence we can say that one (1) Logical Write in RAID 4 and RAID 5 is equivalent to 2 Physical Reads and 2 Physical Writes

System Availability: Orthogonal RAIDs

Lecture 41

Networks and Clusters

(Networks: Interconnection and Topology)

14pm "Topics

Recap:
 A Simple Network
 Network Topology
 Internetworking
 Summary

Recap: I/O Systems and Storages

Last time we concluded our discussion on the storage I/Os and communication I/Os. Here, we noticed that the dependability, reliability, availability of the storage I/Os mostly influence the overall performance of computer systems. Dependability is the quality of delivered service such that confidence can be placed on this service; and measured by quantifying the transitions between service accomplishment and service interruption.

Recap: Dependability, reliability and Availability

The dependability is measured in terms of the reliability and availability of a module. The reliability of a module is the measure of the continuous service accomplishment or the measure of the time to failure, from a reference initial instant. The availability of a module is the measure of the service accomplishment with respect to the swinging between the accomplishment and interruption states.

Recap: I/O and Storage Systems & Network Attached Storages

The storages are interfaced with the processor using channel and backplane and network interconnects. The networks are capable of sustaining high bandwidth transfer and their file-server operating system supports remote file access. Hence, the network attached storages have very high dependability, but are more vulnerable to the reliability, so to improve the availability and performance of network attached storage system, disk arrays are introduced. Here, the data is striped across a set of disks which makes the collection appear to the software as a single large disk. The throughput of disk arrays is improved due to many small disk drives having high bandwidth. The drawback to an array with more devices is that dependability of the device increases, hence, the reliability decreases; as generally N devices have $1/N$ reliability.

Recap: Redundant Arrays of Disks

The dependability of disk array is improved by adding redundant disks to the array to tolerate faults. Such a disk array is called Redundant Array of Inexpensive Disk RAID.

There exist several different approaches to include redundant disks in the disk array. These approaches are usually classified by numerical value which identifies the RAID level.

RAID 0 is the disk array without any redundant disk, but employs the stripping of data across a set of disks.

RAID 3 or Bit-Interleaved Parity Disk employs a parity disk for each group of data; the parity computed across recovery group to protect against hard disk failures.

RAID 4 or Block Interleaved Parity and the RAID 5 or Block Interleaved Distributed Parity, both use the same ratio of data disk to parity disk as RAID 3, but they access data differently.

In RAID 4 level, the parity disk is associated to each data block, identical to it is associated to each data group in RAID 3, so it supports a mixture of both the small and large reads and writes.

In RAID 5 level, the parity disk is associated to each data block.

The data blocks are distributed among different disks in each row;

i.e., the striped data units are not located in the same disk.

This allows simultaneous read and write of more than one block.

Interconnection Networks

Till now the focus of our studies has been the architecture of the components of a single computer and their performance.

Now today and in the following a few lectures we will talk about how to connect computers together forming network of computers.

The formation of a generic interconnection network is depicted here.

The standard components of a computer network are:

- ✓ Computer nodes (also called host or end system)
- ✓ H/W and S/W interface
- ✓ Links to the interconnection network
- ✓ Interconnection networks (also called network or communication subnet)

The coordinated use of interconnected computers in a machine room is referred to as the cluster

The connection of two or more interconnection networks is called Internetworking

The typical example of Internetworking is the Internet

Internetworking relies on the communication standards to convert information from one kind of network to another

Depending on the number of nodes and their proximity or nearness the interconnections are designated as:

Local Area Network-LAN: Hundreds of computer distributed in a building within a distance of up to a few kilometers

Wide Area Network-WAN: Interconnection of thousands of computers distributed throughout the world at a maximum distance of thousands of kilometer Automatic Teller Machine (ATM) is a typical example

System Area Network-SAN: Interconnection network of hundreds of nodes within the machine room; so the distance of the link is less than 100 meters

SAN is basically the cluster

It defines the interconnection of components within a single computer

In order to discuss the complexities and performance of networks, let us consider a simple interconnection model of two computers and understand the implications of network parameters

Networks Communication Model

The communication model depicted here shows that two machines are connected via two unidirectional wires with a FIFO (queue) at the end to hold the data

Here, each machine wants receive a word or message from the other

The machine A to get data from B, it sends a request to B, which responds by sending a reply along with the data

In order to send a request and reply a message contains extra information beyond data, as shown in the example message format

Networks Message Format

Here, a 1 bit header specifies the message as a request (header=0) or reply (header=1)
The request carries the address of the data word and the reply the data word

Networks Interconnection Software

Interconnection networks involve software to establish communication

For the simple network considered here, the software is invoked to translate the request and reply messages

The network software:

- ✓ cooperate with the operating system to distinguish between the processes on the other networks
- ✓ protect the processes running on networks
- ✓ Ensures reliable delivery of message, i.e., to ensure that the message is neither distorted nor lost in transit

It is worth mentioning here that reliability the message format is modified by adding an error detection code (checksum or CRC) and using 2-bit header as shown here

This information is calculated at the sending-end and is added to the message;
then at the receiving-end this message is checked; and the receiver sends an acknowledgment if the message passes the test

Furthermore, to ensure reliable deliver of message, the sender activates a timer each time a message is sent;

The sender copies the data into an operating system buffer to resend the message if the timer expires, as it is presumed to be lost

At the receiving end the message is copied into the operating system buffer

The checksum is checked, if it passes the test acknowledgment is sent otherwise the

message is deleted from the buffer

Networks Interconnection Protocol

So far we have been talking about the acknowledgment - protocol for reliable communication on a simple network

However, there are many more issues of reliable communication; e.g.

- ✓ Two machines from two different manufacturers might be using different byte-order within a word (Big Endian or Little Endian) so the software must have to reverse the order accordingly
- ✓ The duplicate delivery of message should be guarded against the late delivery of the original message, if it was stuck in the network
- ✓ The order or the sequence of the message should not change; so sequence number should be included in the message
- ✓ Checksums should be incorporated

Networks Performance Model

Having discussed the issues of protection, reliability and network protocols, let us understand the performance model of networks

The performance of a network can be modeled at any level, i.e., inside a chip, between chips on PCB and between computers in a cluster, through the interconnection performance parameters

These parameters are:

- ✓ Bandwidth: the maximum rate at which the network can propagate information
- ✓ Time of Flight: time of the first bit of the message from time departed to the time it arrives at the receiver
- ✓ Transmission Time: The time of the message to pass through the network not including the time of flight; in other words, it is the time between the first and the last bit of the message arrives at the receiver
- ✓ Transport Latency: The sum of time of flight and transmission time
- ✓ Sender Overhead: time for the processor to inject the message into the network, including both the hardware and software components
- ✓ Receiver Overhead: the time for the receiver processor to pull the message from the interconnection network, including both the hardware and software components

Based on the network performance parameters, the

Total Latency of a message =

Sender overhead + time to flight + (message size / bandwidth) + Receiver overhead

Interconnection Network Media Hierarchy

Just as the memory hierarchy, there is hierarchy of media to interconnect computer

The interconnect media varies in cost, performance and reliability based on the maximum distance between nodes

The three most popular media are:

- ✓ Twisted Pair (of Copper wire)

✓ Coaxial Cable

✓ Fiber Optics

Twisted Pair of Copper Wire

Two insulated copper wires, about 1mm thick twisted together to reduce the electrical interference

The original twisted pair telephone line gives the data transfer rate of a few mega-bits per sec and is referred to as Level-1 or Category-1 UTP (Unshielded Twisted Pair)

Level 3 or Cat-3 UTP is good for 10M bits/sec Ethernet and Cat-5 for 100M bits/sec and up to 1000 M bits/sec when distance is limited to 100 meters

(Insert fig. 8.11 (a) pp 802)

Coaxial Cable

Coaxial Cable is a stiff single copper wire surrounded by insulating material which is covered by cylindrical sheath woven as a braided mesh

A 50 ohm base-band coaxial cable can deliver 10 M bits/sec over a kilometer

Coaxial cable can deliver higher rate over a few kilometers and offers high bandwidth and good noise immunity. (Insert fig. 8.11 (b) pp 802)

Fiber Optics

Unlike the twisted pair or coaxial cable, Fibers are one-way of simplex media

Two fibers are used for 2-way or full-duplex connections

Fiber Optics contains a glass fiber core surrounded by cladding to confine light, which is covered by a protecting buffer. (Insert fig. 8.11 (c) pp 802)

A light source (LED or laser) and a light detector (photo diode) are employed as transmitter and receiver

As we know that light bends or refracts at interfaces and can spread slowly as it travels down the cable

However, if the diameter of the cable is equal to or less than one wavelength, then it is transferred into straight line (here the angle of refraction is more than the critical angle and total reflection takes place)

Fiber Optic Cables are of two forms: Single-mode and Multimode Fiber:

Multimode Fiber:

- ✓ It uses inexpensive light source with wavelength larger than that of light, and
- ✓ offers wider dispersion where some wave frequencies have different propagation velocities
- ✓ Its dispersion is therefore limited to a few hundred meters at 1000 M bit/sec or up to a few kilo-meters at 100 M bit/sec

Single-mode Fiber:

- ✓ It uses more expensive lasers with single wavelength
- ✓ It can transmit G bits/sec for hundreds of kilo-meters
- ✓ The drawbacks of single mode fiber are:
 - It is more difficult to attach connectors
 - It is less reliable and more expensive and has restrictions on the degree it can be bent
 - The cost, distance and bandwidth are affected by the power of light

source

Interconnection Networks

So far we have discussed connecting two computers over private lines

However, interconnecting hundreds of computers is more interesting and challenging

The bus-based LAN or Ethernet is the simplest way to interconnect more than two computers sharing a single media

Bus Based Networks Interconnection

It is Simple and cost-effective for small-scale multiprocessors

However, the bus bandwidth limits the number of processors

The bus-based interconnect is more challenging also as it requires coordination and arbitration as more than one computer may need the same media simultaneously

Interconnection Networks

However, if the network is small, spread over a few hundred meters, centralized arbitration may be used

so we have to go for distributed arbitration

If two nodes get hold of the media and transmit simultaneously, it leads to collision

So to avoid collision, different techniques such as collision detection and token passing are used

Alternative to sharing media is to use switching

Switch have a dedicated line which it provides in turns to all destinations

Switching allow point-to-point communication much faster than the shared media

Switches are also called data switching exchanges, multistage interconnection networks or interface message processor (IMPs)

Network Topology

With this much discussion regarding sharing of media using buses and switches, let us discuss the topologies used to construct computer networks

There exist numerous topologies of SANs, LANs and WANs, however, the most popular switch-based topologies are classified as: at present are:

- ✓ Centralized Switch Topologies
- ✓ Distributed Switch Topologies

Today we will be talking about

- ✓ The basic Centralized Switch Topologies as:

- Crossbar
- Multistage
- ✓ And, the Distributed Switch Topologies as:
 - 2D Grid or mesh
 - 2D Torus
 - Hypercube tree

Crossbar Switch Topology

A crossbar switch is a non-blocking switch that facilitate unidirectional interconnection of all the inputs (any processor) to any output to the other processor

The interconnection of 2x2 crossbar switch are shown here [Fig. 8.13 c]

As you can see that 2 nodes A, B can pass information equally to outputs C and D

i.e., here no connections block any connection between other processor or memory units

The organization of a crossbar topology for 8-nodes (processors) is shown here

Note that a crossbar uses n^2 switches where n is the number of processors

Here, the links are unidirectional, i.e., the data comes in at one (left) link and goes out at other (right) link

The routing depends on the style of addressing

In source-based routing where message specifies the path to the destination, the message includes the sequence of out-bound arcs to reach the destination

Thus, once an outgoing link (arc) is picked, the portion of the routing sequence is dropped from the packet

In destination-based routing the message simply contains the destination address; and a program running in the switch decides from routing table which port to take for a given address

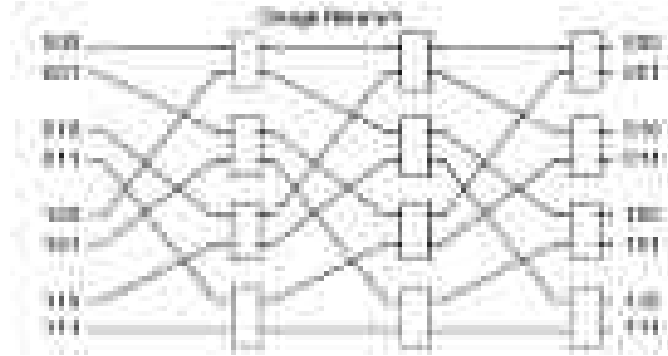
A crossbar switch offers low latency and high throughput

Multistage Interconnection Topology

An intermediate class of networks which lies between crossbar and bus based networks

- ✓ Performance: more scalable than bus
- ✓ Cost: more scalable than crossbar

Built from small (e.g., 2x2 crossbar) switch nodes, with a regular interconnection pattern
The Omega Topology, depicted here is a typical implementation of Multistage Topology

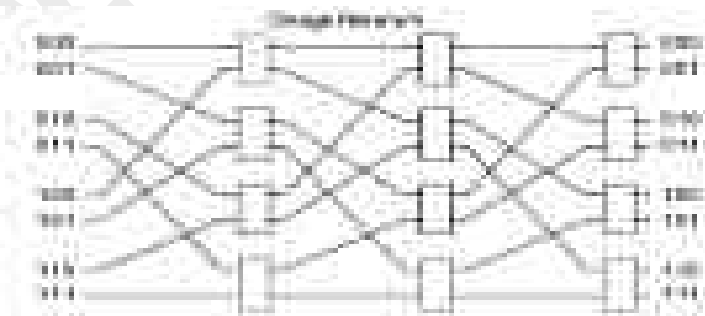


Here each switch is a 2x2 crossbar
It has $\log_2 n$ identical stages
Here, the switches used are $n/2 \log_2 n$ versus n^2 in crossbar

Omega Interconnection Topology

Here, the connection occur between the messages and depend on the pattern of communication and may give rise to blocking

For example, for the Omega network shown, a message from P1 to P7 blocks while waiting for a message from P0 to P6 as has to follow the same path



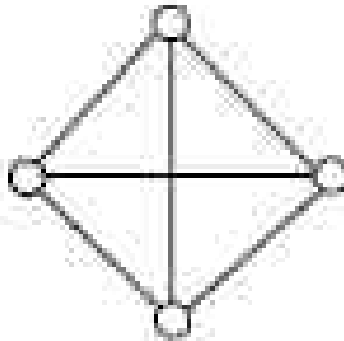
Fully Connected Switching Network

So far we have been discussing centralized switching topologies

The distributed switching network is one where the switching is distributed throughout the network

An ideal situation may be when the distributed switching allows to interconnect all nodes to each other, as shown here

Such a network is called fully connected network



Terminology of Distributed Network

Degree: Number of link to each node

Diameter: Number of nodes between source and destination

For fully connected Network

- ✓ Diameter = 1
- ✓ Deg = $K - 1$ where K is the number of nodes
- ✓ Links = $K * (K - 1) / 2$
- ✓ Bisects = $K * K / 5$

Distributed Switch Topologies

The simplest possible, low cost alternative to the fully interconnected network topology, is a distributed switch ring network, shown here

As shown, a small switch is placed at every computer connected to the ring

Here, as only two nodes are connected to a particular node, therefore the message will have to hop along intermediate nodes until it arrives at the final destination

Ring Network

For example, where the 1st node is to connect to the 4th node, it hops the 2nd and 3rd nodes

A variation to the Ring Network, to simplify the arbitration, is the Token Ring

Here, a single slot (token) goes around the ring to determine which node is allowed to send the message. A node can send a message if it gets a token

The ring network has the following measure

- ✓ Degree: 2
- ✓ Diameter : $N/2$
- ✓ Bisect: 2
- ✓ Bandwidth = N
- ✓ Latency = $N/2$

For example, where the 1st node is to connect to the 4th node, it hops the 2nd and 3rd nodes

A variation to the Ring Network, to simplify the arbitration, is the Token Ring

Here, a single slot (token) goes around the ring to determine which node is allowed to send the message a node can send a message if it gets a token

2D Grid and 2D Torus Mesh

Connecting the switches associated with each node to the switches on the left and right and up and down and also connecting the switches to the top and bottom rows gives a grid structure

Also connecting the switches of left and right columns give 2D Torus Mesh

Lecture 42

Networks and Clusters

,Uq' √“w”] √..√x/s 'hzp'P'q'zq' √“wzs”22J√z' p-

] √pm ”] √..ω”

Recap:

U{ ng] "Vst spskinw0Esr x h

- ✓ Centralized Switch Topology
- ✓ Distributed Switch Topology

Cluster

Summary

Recap: Lecture 41

Last time we discussed:

- ✓ The formation of generic interconnection networks and their categorization;
- ✓ The networks communication model, performance, media, software, protocols, subnet and networks topologies

Here, we noticed that a generic interconnection network comprises:

- ✓ Computer nodes (host or end system)
- ✓ H/W and S/W interface
- ✓ Links to the interconnection network and
- ✓ Communication subnet

The interconnections are classified based on the number of processors or nodes and the distance between them as:

- ✓ Local Area Network-LAN
- ✓ Wide Area Network-WAN
- ✓ System Area Network-SAN

The interconnect communication model shows that two machines are connected via two unidirectional wires with a FIFO (queue) at the end to hold the data

The communication software separates the header and trailer from the message and identifies the request, reply, their acknowledgments and error checking codes

The communication protocols suggest the sequence of steps to reliable communication

The network performance that defines the latency of the message as the sum of the:

Sender overhead, time to flight, receiver overhead and the ratio of the message size to the bandwidth

We also discussed the properties and performance of interconnect network media or link the unshielded twisted pair (UTP), coaxial cable and fiber optics

At the end we discussed the formation of bus-based and switch-based communication subnets and introduced the network topologies

Here, we observed that the bus-based LAN or Ethernet is the simplest way to interconnect more than two computers sharing a single media

However, the interconnect sharing media are challenging as it requires coordination and arbitration when more than one computer needs the same media simultaneously

Alternative to sharing media is to use a switch to provide a dedicated line to all destinations in order; and facilitates point-to-point communication much faster than the shared media

A switch provides unidirectional inter-connection of input to any one of multiple output terminals

The switch that facilitates unidirectional inter-connection of every processor to all the processors in the network is referred to as the non-blocking switch

The Crossbar switch is typical example of non-blocking switch; an is employed in the centralized switching topology

Last time we discussed the crossbar topology in detail and noticed that a crossbar uses n^2 switches to interconnect n processors in a network

Here the routing, to establish interconnection between two node at a time, depends on the addressing style

i.e., source-based routing where message specifies the path to the destination or destination-based routing where the message simply contains the destination address and a program running in the switch selects the port to take for a given destination

Multistage Interconnect Network

Today, continuing our discussion on the centralized switching topologies, we will discuss an intermediate class of network interconnect which lies between crossbar and bus-based networks

This interconnect topology is referred to as the Multistage network topology

A centralized multistage network, shown here, is built from number of large switch boxes, placed at multiple stages to interconnect all of the nodes

Multistage Interconnection Topology

Each stage contains number of small crossbar switches and allows the straight or cross connections through the switch, as shown

The number of stages are related to the number of nodes and the size of the crossbar switch

Consequently, its performance and cost are more scalable than bus-based networks. The number of identical stages (N_s) in the network having n nodes and switches of size $m \times m$, in each stage, is given as:

$$N_s = \log_m n$$

And, the number of switches per stage is n/m

Thus, the total number of switches used in multistage network of n nodes is $n/m \log_m n$ i.e., its cost is

$$O(n \log n) \text{ as compared } O(n^2) \text{ for crossbar}$$

To understand the design and working of multistage networks, let us consider Omega Network, depicted here, as a typical implementation of multistage network

Omega Topology

Here, 8 nodes (processors), are addressed using 3-bit code and 3 stages of 2×2 crossbar switches

number of identical stages $\lceil \log_2 8 \rceil = 3$

And, switches per stage $\lceil n/m \rceil = 8/2 = 4$

Omega Topology: Multistage Interconnect

let us see how the switches at each stage operate to establish connection

Note that for the 8-nodes Omega Network the node address is of 3 bits, which is equal to number of stages of the switch

Omega Network: Example

Here, the 3-bit code $a_2a_1a_0$ represents 3 stages of the network, as stage $S_2S_1S_0$, from left to right

To find the connection pattern XOR the source and destination, e.g.,

Src (010) $\rightarrow$ dest (110) then XOR results

100 $\rightarrow$ Cross (S2) Straight (S1) Straight (S0)

The switch connections are shown Green Circles

Thus, the generalized rule to find the switch connection can be summarized as

- ✓ For the stage i
 - IF the source and destination differ in i^{th} bit
 - THEN Connection Cross the switch in the i^{th} ~~we~~ i
 - ELSE Connection is Straight in the i^{th} ~~we~~ i

Characteristics of Omega

There exist an single path from source to destination, thus contrary to the non-blocking crossbar network, the omega network is blocking network

This is shown here as:

- ✓ the path 010 $\rightarrow$ 110 (red) and
- ✓ the path 110 $\rightarrow$ 100 (blue)

Have blockage as the S2 for 110 has to wait till 010 has passed otherwise it results in collision

However, in order to minimize collisions and to improve fault tolerance to achieve high reliability and dependability extra pathways can be added

Butterfly Network

Alternative to the Omega Topology of multistage switching, is Butterfly Network shown here

Here, irrespective of the source address, for the destination $a_2a_1a_0$, the i^{th} stage switch sends to:

Upper port if $a_i = 0$ and to

Lower port if $a_i = 1$

Distributed Switch Networks

So far we have been discussing the Centralized switching topologies

The distributed switching network is one where the switches are distributed throughout the network and they allow interconnection of one node to:

- ✓ either all the nodes
- ✓ or to a limited number of nodes

A network where each node interconnects all nodes of the network is called, Fully connected network

There exist different interconnects for distributed switch networks

Before discussing these interconnects, let us understand the parameters of interconnect performance measure

Interconnect Performance Measure Criteria

Latency: Number of Links and must be small

Bandwidth: The number of messages or the length of messages; it should be large

Node Degree: Number of links connected to a node

Diameter: Maximum distance between any two processors, i.e., the number of nodes between source and destination; this is indeed the measure of maximum latency

Bisect: The imaginary line that divides the interconnect into roughly two equal parts, each having half the nodes

Bisection Bandwidth: Sum of the bandwidth of lines crossing the imaginary bisection line. It measures the volume of communication allowed between any two halves of network with equal number of nodes

Parameters of Interconnect Performance Measure

Note that the bisect is not clear in non-symmetric networks, therefore, in order to draw the bisect line the bisection band-width is considered

Furthermore, it is to be noted that the bisection bandwidth is the worst-case metric of non-symmetric interconnect

Therefore, the division or the bisect line that makes the bandwidth worst is chosen

Distributed Switch Topologies

Based on the concept of distributed-switch interconnects, there exist numerous topologies

The most popular and commercially available are:

- ✓ Linear Array and Ring
- ✓ Fully Connected
- ✓ 2D Mesh and Torus
- ✓ Hypercube
- ✓ Tree

Linear Array / Ring

The simplest possible, low cost distributed switch network topology, is a linear array and ring network

As shown here, the Linear Array networks is one where a small switch is placed at every node (processor)

The switch at the i^{th} node connects the i^{th} node to the:

- ✓ $(i-1)^{\text{th}}$ node except for $i=1$, and
- ✓ $(i+1)^{\text{th}}$ node except for $i=n$

In the linear array, as the i^{th} node is connected to $(i-1)^{\text{th}}$ and $(i+1)^{\text{th}}$ node, therefore the message will have to hop along intermediate node until it arrives at the final destination at $(i \pm m)$ where $m > 1$

For example, where the message is to pass from 1st node is to the 4th node, it hops the 2nd and 3rd nodes

The ring network is established by establishing an interconnect between the 1st and the n^{th} nodes in the linear array network

Ring /Token Ring

Like linear array, in Ring network some messages hop along the intermediate nodes until they reach destination

However, it allows many transfers simultaneously; the 1st node can send to the 2nd at the same time as the 3rd can send to 4th and so on.

A variation called Token Ring is used in the Ring Network, to simplify the arbitration in the ring topology

Ring Network

Here, a single slot (token) goes around the ring to determine which node is allowed to send the message a node can send a message if it gets a token

The common performance of an n - node linear array and ring network are as follows:

- ✓ Cost: Cheap as the cost is $O(n)$
- ✓ Bandwidth: Overall bandwidth is high
- ✓ Latency: High as it is of $O(N)$

Performance: Array verses Ring

Linear Array

- ✓ Degree: 2
- ✓ Diameter : N
- ✓ Bisection width = 1
- ✓ Bandwidth = $N-1$
- ✓ Mean Latency = $N/2$
- ✓ Asymmetric
- ✓ Heterogeneous

Ring

- ✓ Degree: 2
- ✓ Diameter : $N/2$
- ✓ Bisection Width = 2
- ✓ Bandwidth = N
- ✓ Latency = $N/2$
- ✓ Symmetric
- ✓ Homogeneous

Fully Connected

A straight forward, symmetric but expensive distributed network, equivalent of the crossbar network, is Fully connected network

Here, every node has a direct link with all the other nodes of the network

As shown here the node A is interconnected with the nodes B, C, and D

Its performance metrics are as follows:

- ✓ Diameter = 1
- ✓ Degree = $n-1$
- ✓ Links = $n * (n-1)/2$
- ✓ Bisection bandwidth is
- ✓ proportional to $(n/2)^2$

2D Mesh and 2D Torus

Two dimensional Mesh or Grid is an example of asymmetric network topology and uses bisection bandwidth as the performance metric

Here, the nodes (processors) are arranged in a array structure forming 2D Grid or Mesh

An example 3x4 mesh structure is shown here

A switch is associated to each (processor) node [shown as blue circle]

Each switch has one port for the processor and four ports to interconnect the processor to the four nearest-neighbor nodes, i.e., the nodes to the left - right and up - down position

This structure is sometimes also referred to as NEWS communication pattern, representing North, East, West and South communication

Note that here the switches associated with the top/bottom rows or left/right columns

Connecting the unused ports of switches of the top/bottom rows and the left/right columns forms 2D Torus, using wraparound links, as shown here

The performance metrics of n-node 2D Mesh / Torus are as follows

- ✓ Degree = 4
- ✓ Frequency = $4 \times P$
- ✓ Diameter = $\sqrt{N}$
- ✓ Bandwidth = N
- ✓ Asymmetric

Tree Network Topology

Another example of distributed switch network is the Tree Topology

Here, the switches associated with each node have the number of ports equal to the number of branches of the tree plus one for the processor

A Binary Tree structure shown here has two branches of the root node and branch nodes

The performance metrics of N-nodes Tree Network are as follows

- ✓ Cost: It is cheap as cost is $O(N)$
- ✓ Flooding: $O(N)$
- ✓ Latency: $O(\log_{deg} N)$
- ✓ Diameter: $2 \log_{deg} N$
- ✓ Bisection Width: 1

Tree Network: Bottlenecks

The root node and the branch nodes of the leaf-nodes are the bottleneck. For example, leaf-nodes 1, 2 of the branch nodes 9 and 3, 4 of branch node 10, may be interconnected simultaneously, but the leaf-nodes 1, 3 and 2, 4 cannot, as there may be collision at branch nodes 13, 9 and 10

Fat Tree Network

To avoid root being the bottleneck, multiple paths are provided between any two nodes, as shown here. This structure is called the Fat Trees

Here, the black dots show the processor-memory nodes connected through the multiple stages of 2x2 crossbar, 4x (2+2) crossbar and 8 x (4+4) crossbars switches and so on. This 3D switching increases the bandwidth via extra links at each level over a simple tree. In CM-5 the concept of Fat Tree is used as a Centralized switching Network.

Hypercube Network Topology

Another example of a distributed switch network is the Hypercube topology, which is also called a binary n-cube, as it has 2 nodes of n-cubes.

It is an n-dimensional interconnect for 2^n nodes.

As can be seen from the figure here, that for 16 nodes; the hypercube is a 4D structure as

$$N=16 = 2^4 \text{ therefore } n=4$$

It requires n ports per switch plus one for the processor; this has n nearest neighbors nodes.

Thus, it minimizes hops and has a latency of $O(\log_2 N)$; the other performance metrics are:

- ✓ Nodes = $N = 2^n$
- ✓ Degree = n; Diameter = n
- ✓ Links = $n * 2^{(n-1)}$
- ✓ Bisection width = $2^{(n-1)}$

The other topologies, such as tree, mesh etc., can be embedded in a hypercube.

Note that the bisection bandwidth is good but it is difficult to layout in 3D space.

Hypercube has been popular in early message passing machines, e.g., Intel iPSC, NCUBE etc.

K-ary n-cube Network Topology

Rather than having just 2 nodes of n-cubes in the binary hypercube, the generalization of hypercube is to interconnect k nodes of n-cubes in a string.

The total number of nodes: $N = k^n$

A 64 node, where $64 = 4^3$ [4-ary 3-cube] structure is shown here.

This structure allows for a wider channel but requires more hops.

$$64 = 4^3 \text{ [4-ary 3-cube]}$$

(3-cube is a 16 nodes binary hypercube)

Comparing Network Topologies

The relative cost and performance of topologies discussed, based on the bisection bandwidth and number of links for 64 nodes network is given in the table here

Evaluation Category	Bus	Ring	2D Torus	Fully Connected
Performance:				
Bisection Bandwidth	1	2	16	1024
Cost				
Ports/switch	N/A	3	5	64
Total Links	1	128	192	2080

Here, bus is used as the standard reference at unit cost, all transfers are done by taking the time units equal to the number of messages

Where as the fully connected network has all nodes at equal distance therefore the number of links and ports per switch are maximum and all transfers are done in parallel taking only unit time

The nodes for ring topology are differing distances

Here, bus is used as the standard reference at unit cost, all transfers are done by taking the time units equal to the number of messages

Where as the fully connected network has all nodes at equal distance therefore the number of links and ports per switch are maximum and all transfers are done in parallel taking only unit time

The nodes for ring topology are differing distances

Internetworking

Internetworking deals with the communication of computers on independent and incompatible networks reliably and efficiently

The software standards are the basic enabling technologies of internetworking (Transmission Control Protocol/Internet Protocol) TCP/IP is the most popular internetworking standard

The detailed discussion on Internetworking is beyond the scope of this course

Cluster

Internetworking deals with the communication of computers on independent and incompatible networks reliably and efficiently

The software standards are the basic enabling technologies of internetworking (Transmission Control Protocol/Internet Protocol) TCP/IP is the most popular internetworking standard

The detailed discussion on Internetworking is beyond the scope of this course

Summary

Today, we discussed an intermediate class of network interconnect which lies between crossbar and bus-based networks, referred to as the Multistage Switch network topology

A multistage centralized switch is built from number of large switch boxes, placed at number of stages to interconnect all of the nodes

Here, The number of identical stages (N_s) in the network having n nodes and switches in each stage are of size $m \times m$ is as given as:

$$N_s = \log_m n$$

Lecture 43

Networks and Clusters (Internetworks and Clusters)

] √pm ""] √..ω"

Recap:
Internetworks
Cluster
Case Studies
Summary

Recap:

In our last two lectures on Networks and Cluster we discussed:

- ✓ The formation of generic interconnection networks and their categorization, the networks communication model, performance, media, software, protocols, subnet and networks topologies
- ✓ Here, we noticed that a generic interconnection network comprises: Computer nodes, H/W and S/W interface, Links to the interconnection network and Communication subnet

The interconnect communication model shows that two machines are connected via two unidirectional wires with a FIFO (queue) at the end to hold the data

The communication software separates the header and trailer from the message and identifies the request, reply, their acknowledgments and error checking codes

The communication protocols suggest the sequence of steps to reliable communication

We also discussed:

- ✓ The properties and performance of interconnect network media or link the unshielded twisted pair (UTP), coaxial cable and fiber optics
- ✓ The formation of bus-based and switch-based communication subnets and introduced the network topologies
- ✓ The bus-based communication subnets share the common media where arbitration is the bottleneck
- ✓ Alternative to sharing media is to use a switch to provide a dedicated line to all destinations in order; and facilitates point-to-point communication much faster than the shared media
- ✓ The switch-based networks are classified as the centralized and distributed switch networks

Here the routing, to establish interconnection between two node at a time, depends on the addressing style: source-based routing and destination-based routing

The performance of a distributed network is measured in terms of:

- ✓ Latency - number of Links between source and destination
- ✓ Bandwidth - number or length of messages passing per sec.
- ✓ Degree - number of links connected to a node
- ✓ Diameter - number of nodes between source and destination; this is indeed the measure of maximum latency

- ✓ Bisection - the imaginary line that divides the interconnect into roughly two equal parts, each having half the nodes
- ✓ Bisection Bandwidth: the volume of communication allowed between any two halves of network with equal number of nodes

Last time, we discussed an intermediate class of network interconnect Multistage Switch network

It is built from number of large switch boxes each containing number of small crossbar

The performance of Multi-stage switch lies between performance of non-locking crossbar and bus-based networks

Following the discussion on centralized switch topologies we studied the distributed-switch interconnects; which are categorized as the fully-connected and partially-connected, symmetric or asymmetric interconnects

The distributed-switch interconnect topologies, such as: linear array, ring, 2D mesh/torus and hypercube were studied

We also discussed the relative cost and performance of these topologies, based on the bisection bandwidth and number of links for 64 nodes network; which is shown in the following table

<u>Evaluation Category</u>	<u>Bus</u>	<u>Ring</u>	<u>2D Torus</u>	<u>Fully Connected</u>
<u>Performance:</u>				
Bisection B/W	1	2	16	1024
<u>Cost:</u>				
Ports/switch	N/A	3	5	64
Total Links	1	128	192	2080

Internetworking

So far we have been talking about the design styles, topologies and performance of interconnection networks

Now we are doing to talk about the connection of two or more interconnection networks, called Internetworking; the Internet is typical example of Internetworking

Internetworking deals with the communication of computers on independent and incompatible networks reliably and efficiently

Internetworking relies on the communication standards to convert information from one kind of network to another

These standards are composed of hierarchy of layers, where each layer is responsible for a portion of overall communication

Each computer, network and switch implements its layer of standards, called the Protocol Families or Protocol suites, and facilitates applications to work with any inter-connection

OSI: 7- Layer Model

The Open Systems Interconnect (OSI) developed a 7-layer model, which describes a network as the series of layers; with

Application layer at the top (i.e., layer 7) and Physical layer at the bottom (layer 1) and presentation, session, transport, network and data link layers in between the top to bottom layers, as layer-6 down to layer-2, respectively

The OSI model, layer-7, the Application layer is used for applications specifically written to run over the network, e.g., Network File System (NFS) etc.

The layer-6, Presentation layer translates from application to network format and vice versa

The layer-5, Session Layer, establishes maintains and ends the sessions across the network

The layer-4, Transport Layer, facilitates additional connection below the session layer; the protocol is referred to as the Transmission Control Protocol - TCP

The layer-3, Network Layer, translates the network address and names to their physical address; e.g., computer name to Media Access Control (MAC); the layer-3 protocol is referred to as Internet Protocol or IP

The layer-2, Data Link layer, turns packets into raw bits and at the receiving end turns bits into packets; the example protocol is Ethernet

The layer-1, Physical Layer, transmits raw bit-stream over physical cable/media; IEEE 802 is typical example physical layer protocol

TCP/IP Families

The protocol family divides the responsibilities among the layers, with each layer offering services needed by the layer above

The Transmission Control Protocol/Internet Protocol - TCP/IP is the most popular internetworking standard

TCP/IP is the basis of Internet, which connect the tens of millions of computers around the world

The protocol at each level is implemented by adding headers and trailers at the sending layer and removing at the receiving layer

The original message, from the top layer, includes a header and trailer sent by the lower-level protocol

The next-lower protocol in turn adds its own header (and possibly trailer) to the message and so on

If the message is too large for a particular layer, then it is broken into smaller messages; this division of message and addition of header and trailer continues till the message descends to the physical transmission media

The message is then sent to the destination

Each level of protocol family at the receiving end, from bottom to the top layer, checks the message at its level and remove its header and trailer, and pass it on to the next higher level

The message is rebuilt by putting the pieces together

This nesting of protocols layers is referred to as the Protocol Stack as it reflects the Last-in First out nature of addition and removal of the header and trailer

A typical TCP/IP datagram, containing header and message, is depicted here

Fig. 8.27 pp 835 Text book

The standard IP and TCP headers are 20 byte each, stacked as shown

However, the length can optionally be increased which is specified by the length field (L)

VI i "p r k l "s j" x i " { l s p "hexekveq "w r h i r x j n h " f } " e " w i t e v e x i " j n h " N i r k l " m " R " l i e h i v . " { l p " x i " V E R " l i e h i V n g y h i w l w n j s v q e x s r " m " x i " w i u y i r g i w r y q f i v j n h " }

Internetworking

As the detailed discussion on the TCP / IP is beyond the scope of this course on Computer Architecture, therefore

{ i " e v i " p e z m k " x l w h m g y w s r " l i v i " e r h " e v i " k s m k " s " x e p " e f s y x " g p y w i v . " x i " p e w " x t r o " s j " }

our study of t l i " P i x s w o r h " E p y w i v =

rather the last topic of this course on Advance Computer Architecture
However, the students interested in further study of Internetworks may consult literature and books on Computer Networks and Internetworking

Clusters System Area Networks

The coordinated use of interconnected computers in a machine room is referred to as the cluster or System Area Network

Massively parallel machine providing high bandwidth can be built from off-the-shelf components, instead of depending on the custom machines or networks

A cluster , i.e., a collection or bunch of desk-top computer and disk offers low cost computing infrastructure that could tackle very large problems and applications, such as: databases, file servers, Web servers, simulation and multiprogramming and batch processing

The clusters face some performance confront such as:

- ✓ Non-standard connections
- ✓ Division of memory

Let us talk about these confronts one by one

Non-Standard Confront: As you know that the multiprocessors are usually connected memory bus which offers high bandwidth and low latency; and

Contrary to this, the clusters are connected using I/O bus of the computer, thus have large conflicts at high speed

Division of Memory: A large single program running on a cluster of N machines requires N independent memory units and N copies of operating system; on the other hand,

A shared address multiprocessor allows to use almost all memory in the computer

However, contrary to these challenges, clusters have advantages in respect of dependability and scalability

The weakness of separate memories for program size in case of cluster, as discussed earlier, is indeed a strength in terms of system availability and expandability (or say the scalability)

Furthermore, as the cluster consists of independent computers connected through LAN, and cluster software is a layer that runs on top of the local operating system, therefore, it is easier as compared to the multiprocessor, to replace any computer without bringing down the all computer of the cluster, hence a cluster offers high dependability and scalability

Furthermore, it is easier to expand a cluster, therefore, it is attractive to the world wide web service providers

Cluster Design Examples

In order to study practical aspects of cluster designs, we are going to discuss different cluster design comprising: 32 processors, 32 GB DRAM, and 32 or 64 disks

For different cluster designs let us consider P-III processors operating at clock rate of 700 MHz and 1000 MHz include large L2 cache ranging from 256 KB to 1MB

However, note that due to larger die size, the processor chip price with 1 MB cache is double as compared to that of with 256KB Cache chip

In cluster design, the higher chip price of chip matters little; but the objective is to minimize cost for desired performance target

We are considering following four cases:

1. Cost of cluster hardware with local disk
2. Cost of cluster hardware with disk over SAN (system or storage area network)
3. Cost of cluster options that is more realistic
4. Cost and Performance of a cluster for transaction processing

Example 1: (Cluster Design Examples)

In order to discuss the first case, Cost of cluster hardware with local disk, let us consider three logical organizations of clusters:

- a. Uniprocessor Cluster
- b. 2-way SMP (Symmetric Shared Memory Processor) cluster
- c. 8-way SMP cluster

a. Uniprocessor Cluster Design

The Uniprocessor cluster organization, shown here, consists of 32 xSeries 300 computer (for 32 processors). Fig 8.34 a pp 846

As the maximum memory for this computer is 1.5 GB, so it easily allows desired 32 GB (1 x 32) memory

As each computer has 2 disk drives each of 36.4 GB so it yield $32 \times 2 \times 36.4 = 2330$ GB

The organization uses the built-in slots for storage, so computer can accept its own G-bit hot adapter, hence 32 cables are available for the IGB Ethernet switch

However, as the switch has 30 ports, therefore 2 switches are used

These two switches are connected together with 4 cables, leaving 56 ports for 32 computers

VI i "ver hev" vego" w 3; " | " 52 " | " 94 " JY " | " F " | " J d " er h" ger " egg shexi " 54" uniprocessors computers, 34 rack units (32 for computers and 2 for switches)

This design is cost effective

b. 2-way SMP Cluster Design

In we use the 2-processor computer of xSeries 330, as shown here, every thing is halved

Fig 8.34 (b) pp 846

Here, 32 processor need only 16 computers, a single 30-port switch can work as there are 16 cables to be interfaced

Furthermore, the rack size 18 RU instead of 44 RU which is less than half the standard size

c. 8-way SMP Cluster Design

The 8-processor computer of xSeries 370, as shown here, only 4 computer are used at they contain $4 \times 8 = 32$ processors

Fig 8.34 (c) pp 846

The maximum memory is 32 GB but we need only 8 GB per computer and for 4 computers only 8-port switch is sufficient

However, at 2 disk per computer, the 4 computers can hold 8 disks with maximum capacity per disk 73.4 GB; hence we need expansion storage box which can hold up to 14 disks; and 2 racks are needed in place of 1 in the previous cases

Comparison of 3-Cluster Designs

The price of three clusters with a total of 32 processors, 32 GB memory and 2.3 Tetra-byte disk is shown here

Fig 8.35 848

Note that the network cost decreases as the size of the SMP increases; because the memory buses supply more of the inter-processor communication

Furthermore, the 4 of the 8-way SMP cost more than 32 Uniprocessor computers

The price of three clusters with a total of 32 processors, 32 GB memory and 2.3 Tetra-byte disk is shown here

Fig 8.35 848

Note that the network cost decreases as the size of the SMP increases; because the memory buses supply more of the inter-processor communication

Furthermore, the 4 of the 8-way SMP cost more than 32 Uniprocessor computers

Example 2: (Cluster Design Examples)

Now let us discuss the 2nd case, Cost of cluster hardware using SAN (storage area network) for disks

In the previous we set the disks local to the computer which reduces the cost and space However, it offers the following problems for the operator

1. No protection against single disk failure
2. State in each computer must be manages separately;

This results in system-down state on the disk failure

To overcome this problem, a RAID controller and Fiber Channel Arbitrated Loop (FC-AL), is used as the storage area network (SAN)

In this case all the SCSI disks are replaced FC-AL disk behind the RAID storage server

Note that FC-AL can be connected in a loop with up to 127 devices

The price comparison for the three clusters, using SAN, show here

Fig. 8.37 pp 850

Illustrates that the cost of SAN also shrinks as the servers increase in number of the processors per computer

Example 3: (Cluster Design Examples)

Now let us discuss the 3rd case, cluster design considering other costs

In the first two design we considered the cost of hardware only

However, the software (data base) cost and hardware maintenance cost (cost of operator to keep the machine running) has not been considered

The other costs include the cost of backup tapes, cost of space to house the servers

A complete comparison of the earlier three clusters including the other cost is shown here

Fig. 8.39 pp 852

It shows that 2-way SMP using SAN is lowest in total price

However, over the 3 years, the cost of operator will be more than the cost of the hardware; so we must reduce the purchase cost of the old computers to reduce the overall cost

Example 4: (Cluster Design Examples)

Now let us discuss the 4th case, cluster design for transaction processing, shown here

Fig. 8.40 pp 853

The cluster has 32 P-III processors, using the same IBM computer as the basic building block which was employed in earlier design

The key differences are:

- ✓ Disk Size: Small and fast disks are used as this structure cares more about I/Os per second (IOPS)
- ✓ RAID: No RAID is required as this as the performance benchmark doesn't include the human cost
- ✓ Memory: Maximum DRAM is packed into the servers, so each of the four 8-way SMPs is stuffed with maximum of 32 GB, yielding 128 GB
- ✓ Processor: 900 MHz P-III with 2 MB L2 cache is used

The cost performance analysis of this structure shows that almost half of the cost is in software, installation and maintenance

Summarizing the cluster design examples we conclude that as the cost of purchase is less than half the cost of the ownership, therefore, the cost of hardware solves only a part of the problem

Summary

In this module on the Networks and clusters we studied that:

The formation of a generic interconnection network that comprises

- ✓ Computer nodes (host or end system)
- ✓ H/W and S/W interface

- ✓ Links to the interconnection network and
- ✓ Communication subnet

The interconnections are designated as:

- ✓ Local Area Network-LAN
- ✓ Wide Area Network-WAN
- ✓ System (or Storage) Area Network-SAN

While talking about the interconnect model, software and protocols we studied that:

- ✓ The interconnect communication model shows that two machines are connected via two unidirectional wires with a FIFO (queue) at the end to hold the data
- ✓ The communication software separates the header and trailer from the message and identifies the request, reply, their acknowledgments and error checking codes
- ✓ The communication protocols suggest the sequence of steps to reliable communication

Then we studied the network performance that defines the latency of the message as the sum of Sender overhead, time to flight, receiver overhead and the ratio of the message size to the bandwidth

We also discussed the properties and performance of interconnect network media or link the unshielded twisted pair (UTP), coaxial cable and fiber optics; and the formation of bus-based and switch-based communication subnets and introduced the network topologies

We also studied an intermediate class of network interconnect, which lies between crossbar and bus-based networks, referred to as the Multistage Switch that is built from number of large switch boxes each containing small crossbar switches; here

The number of identical stages (N_s) of large switch boxes each having $m \times m$ crossbar switches, in the network having n nodes, is equal to $\log_m n$; and, the switches per stage is n/m

The cost of multistage switch network is of $O(n \log n)$ which is considerable small as compared to that of crossbar network that is of $O(n^2)$ when n is large

The typical examples of multistage switching topologies discussed are Omega and Butterfly networks

Following the discussion on centralized switch topologies we considered the distributed-switch interconnects where the switches are distributed among the nodes (processor)

We classified the distributed switch interconnects as the fully and partially connected, symmetric and asymmetric interconnects

Then we discussed the linear array, ring, 2D mesh/torus and hypercube topologies and their performance measures

The relative cost and performance of these topologies, based on the bisection bandwidth and number of links for 64 nodes network is as follows

Today we discussed Internetworking, i.e., the connection of two or more interconnection networks to communicate reliably and efficiently

Internetworking relies on the communication standards composed of hierarchy of layers

The internet communication Protocol Families facilitates applications to work with any inter-connection

The Transmission Control Protocol/Internet Protocol - TCP/IP is the most popular internetworking standard

The protocol at each level is implemented by adding headers and trailers at the sending layer and removing at the receiving layer

Have introduced the basic concept of internetworking we discussed the computer cluster which is coordinated use of interconnected computers in a machine room

Here, we studied Non-standard connections and Division of memory as the performance confront of clusters; furthermore

Contrary to these challenges, clusters have advantages in respect of dependability and scalability

At the end we studied the practical aspects of cluster designs through four examples

Conclusion

Today we have completed our discussion on almost all topics related to this course on Advanced Computer Architecture

In the following last two lecture we will review the complete course following some case studies

Lecture 44

Putting It All Together (Case Studies)

] √pm ""] √..ω"

Case Studies

- ✓ Power PC 750 Architecture
- ✓ Power PC 970 Architecture
- ✓ Intel Pentium VI Architecture

Summary

PowerPC 750 - General

PowerPC 750 is an implementation of PowerPC microprocessor family of reduced instruction set computer (RISC) microprocessors

750 implements the 32-bit portion of the PowerPC architecture

It provides 32-bit effective addresses for:

- ✓ Integer data types of 8, 16, and 32 bits
- ✓ Floating-point data types of 32 and 64 bits

It is high-performance, superscalar micro-processor architecture that has Six execution units and two register files. It can:

- ✓ fetch from the instruction cache as many as four instructions per cycle
- ✓ dispatch as many as two instructions per clock
- ✓ execute as many as six instructions per clock

PowerPC Instructions

Instructions are encoded as single-word (32-bit)

Instruction formats are consistent among all instruction types, permitting efficient decoding to occur in parallel with operand accesses

This fixed instruction length and consistent format greatly simplifies instruction pipelining

Integer instructions are:

- ✓ Integer arithmetic, Integer compare, logical, rotate and shift

Floating-point instructions are:

- ✓ Floating-point arithmetic, multiply/add, rounding and conversion, compare, status and control instructions

Load/store instructions are:

- ✓ Integer and Floating-point load and store; and atomic memory operations (lwarx and stwcx) instructions

Flow control instructions are:

- ✓ branching, condition register logical, trap, and other instructions that affect the instruction flow

Processor control instructions are:

- ✓ used for synchronizing memory accesses and management of caches, TLBs, and the segment registers

Memory control instructions are:

- ✓ provide control of caches, TLBs, and SRs

PowerPC 750 Block Diagram

WWW.VUMULTAN.COM

PowerPC 750 Instruction Flow

Now let discuss the instruction flow in PowerPC 750, which includes:

- ✓ Instruction fetch,
- ✓ Instruction decode and
- ✓ Instruction dispatch

The instruction flow in PowerPC 750 is illustrated here with the help of block diagram
PowerPC 750 allows maximum four instruction fetch per clock cycle

PowerPC 750: Instruction Flow (decode/dispatch)

PowerPC 750 Instruction Fetch

However, the number of clock cycles necessary to request instructions from the memory system depends on where exactly is the:

1. branch target instruction cache
2. on-chip instruction L1 cache
3. L2 cache

Having understood the instruction let us discuss how the PowerPC decodes and dispatch the instruction

PowerPC 750 Decode/Dispatch

Refer to the instruction flow diagram again and note that:

- ✓ Instructions can be dispatched only from the two lowest instruction queue entries, IQ0 and IQ1

- ✓ A maximum of two instructions can be dispatched per clock cycle (although an additional branch instruction can be handled by the Branch Processing Unit-BPU)
- ✓ Only one instruction can be dispatched to each execution unit per clock cycle

Note that to facilitate dispatch:

- ✓ There must be a vacancy in the specified execution unit
- ✓ A rename register must be available for each destination operand specified by the instruction
- ✓ There must be an open position in the completion queue; If no entry is available, the instruction remains in the IQ.

PowerPC 750: Superscalar Pipeline

PowerPC 750 Execution Units

Refer to the PowerPC 750 superscalar pipeline shown here and note that it contains two integer units (IUs),

- ✓ IU1 can execute any integer instruction
- ✓ IU2 can execute all integer instructions except multiply and divide

Which share thirty-two GPRs for integer operands and a Single-entry reservation station for each

Furthermore, there exist

- ✓ One three-stage floating point unit (FPU) that allows both single- and double-precision operations
- ✓ Hardware support for demormalized numbers and Single-entry reservation station are provided
- ✓ Thirty-two 64-bit FPRs for single- or double-precision operands

Two-stage LSU (Load/Store Unit) contains

- ✓ Two-entry reservation station
- ✓ Single-cycle, pipelined cache access
- ✓ Three-entry store queue

Supports both big- and little-endian modes

It performs alignment and precision conversion for floating-point data and sign extension for integer data

PowerPC 750: Completion Unit

Completion unit retires an instruction from the six-entry reorder buffer (completion queue) when:

1. All instructions ahead of it have been completed, and
2. The instruction has finished execution, and
3. No exceptions are pending

The completion unit guarantees sequential programming model (precise exception model)

Monitors all dispatched instructions and retires them in order

Tracks unresolved branches and flushes instructions from the mispredicted branch

Retires as many as two instructions per clock

PowerPC 750 Rename Buffers

750 provides rename registers for holding instruction results before the completion commits them to the architected register

Refer to the instruction flow diagram again and note that there are six GPR rename registers, six FPR rename registers, and one each for the CR, LR, and CTR

When an instruction is dispatched to its execution unit, a rename register for the results of that instruction is assigned

Dispatcher also provides a tag to the execution unit identifying the rename register that forwards the required data for an instruction

When the source data reaches the rename register, execution can begin

Results are transferred from the rename registers to the architected registers by the completion unit when an instruction is retired from completion queue

Results of squashed instructions are flushed from the rename registers

PowerPC 750 Branch Prediction Unit

Featuring both static and dynamic branch predictions, only one is used at any given time

Static branch prediction

- ✓ It is defined by the PowerPC architecture and involves encoding the branch instructions
- ✓ The PowerPC architecture provides a field in branch instructions (the BO field) to allow software to hint whether a branch is likely to be taken
- ✓ Rather than delaying instruction processing until the condition is known, the 750 uses the instruction encoding to predict whether the branch is likely to be taken and begins fetching and executing along that path

Dynamic branch prediction:

- ✓ 750 use the 512-entry Branch history table (BHT) with two bits per entry
- ✓ Allows prediction as: Not-taken, strongly not-taken, taken, strongly taken

PowerPC 750 Branch Target Cache - BTC

750 uses the BTC to reduce time required for fetching target instructions when branch is predicted to be taken

Branch Target Instruction Cache (BTIC)

- ✓ 64-entry (16-set, four-way set-associative)
- ✓ Cache of branch instructions that have been encountered in branch/loop code sequences
- ✓ BTIC hit: instructions are fetched into the instruction queue a cycle sooner than it can be made available from the instruction cache

PowerPC 750 Multiple Branch Prediction

The 750 executes through two levels of prediction

Instructions from the first unresolved branch can execute, but they cannot complete until the branch is resolved.

If a second branch instruction is encountered in the predicted instruction stream, it can be predicted

Instructions can be fetched, but not executed, from the second branch

No action can be taken for a third branch instruction until at least one of the two previous branch instructions is resolved

PowerPC 750 Cache

Separate on-chip instruction and data caches

- ✓ 32-Kbyte, eight-way set-associative instruction
- ✓ Pseudo least-recently-used (PLRU) replacement
- ✓ 32-byte (eight-word) cache block
- ✓ Physically indexed/physical tags
- ✓ Cache write-back or write-through operation per-block basis
- ✓ Caches can be disabled in software
- ✓ Caches can be locked in software
- ✓ Data cache coherency (MEI) maintained in hardware
- ✓ The critical double word is made available to the requesting unit
- ✓ The cache is non-blocking

PowerPC 750: Data and Instruction Cache Organization

Data Cache:

- ✓ 32-word blocks (5 bits)
- ✓ 128 sets (7 bits)
- ✓ Tags (20 bits)
- ✓ State: MEI

Instruction Cache

- ✓ 32-word blocks (5 bits)
- ✓ 128 sets (7 bits)
- ✓ Tags (20 bits)
- ✓ Valid/Not valid
- ✓ Not snooped

PowerPC 750: Multiprocessing

750 Multiprocessing support features include:

- ✓ Hardware-enforced, three-state cache coherency protocol (MEI) for data cache.
- ✓ Load/store with reservation instruction pair for atomic memory references, semaphores, and other multiprocessor operations

VI i "972 wX e-state cache-coherency protocol (MEI) supports the Modified, Exclusive, and Invalid states

PowerPC 750

Three states:

- ✓ Invalid
- ✓ Modified
- ✓ Exclusive

No shared state

Caching-Inhibited Reads

PowerPC 970 FX

PowerPC 970 FX: Organization

64-bit implementation of the PowerPC® AS Architecture (version 2.01)

Vector/SIMD Multimedia eXtension

Deeply pipelined design consisting:

- ✓ 16 stages for most fixed-point register-register operations
- ✓ 18 stages for most load and store operations (assuming an L1 D-cache hit)
- ✓ Up to 25 stages for floating point operations
- ✓ The VALU.
- ✓ 19 stages for vector permute operations

Dynamic instruction cracking

- ✓ Some complex instructions are broken into two simpler, more RISC-like instructions!
- ✓ Allows for simpler inner core dataflow

PowerPC 970 FX: General

Aggressive branch prediction

- ✓ Prediction for up to two branches per cycle
- ✓ Support for up to 16 predicted branches in flight
- ✓ Prediction support for branch direction and branch addresses

In-order dispatch of up to five operations into distributed issue queue structure

Out-of-order issue of up to 10 operations into 10 execution pipelines

- ✓ Two load or store operations
- ✓ Two fixed-point register-register operations
- ✓ Two floating-point operations
- ✓ One branch operation
- ✓ One condition register operation
- ✓ One vector permute operation
- ✓ One vector ALU operation

Register renaming

Cache coherency protocol: MERSI (modified/exclusive/recent/shared/invalid)

Large number of instructions in flight (theoretical maximum of 215 instructions)

Up to 16 instructions in the instruction fetch unit (fetch buffer and overflow buffer)

Up to 32 instructions in the instruction fetch buffer in instruction decode unit

Up to 35 instructions in three decode pipe stages and four dispatch buffers

Up to 100 instructions in the inner-core (after dispatch)

Up to 32 stores queued in the store queue (STQ) (available for forwarding)

Fast, selective flush of incorrect speculative instructions and results

Specific focus on storage latency management

Out-of-order and speculative issue of load operations

Support for up to eight outstanding L1 cache line misses

Hardware initiated instruction prefetching from L2 cache

Software initiated data stream prefetching with support for up to eight active streams

Critical word forwarding / critical sector first

New branch processing / prediction hints for branch instructions

PowerPC 970 FX: Instruction Fetch

64KB, direct-mapped instruction cache (I-cache)

- ✓ 128-byte lines (broken into four 32-byte sectors)
- ✓ Dedicated 32-byte read/write interface from L2 cache critical sector first reload policy

Four-entry, 128-byte, instruction prefetch queue above the I-cache; hardware-initiated prefetches

Fetch 32-byte aligned block of eight instructions per cycle

PowerPC 970 FX: Branch Prediction

Scan all eight fetched instructions for branches each cycle

Predict up to two branches per cycle

Three-table prediction structure

- ✓ Local (16K entries, 1-bit each); Taken/Not taken
- ✓ Global (16K entries, 1-bit each) 11-bit history XORed with branch instruction address; Taken/ Not taken
- ✓ Selector (16K entries, 1-bit each) indexed as above; Use local predictor/Use global predictor

This combination of branch prediction tables has been shown to produce very accurate predictions on a wide range of workload types.

16-entry link stack for address prediction (with stack recovery)- predict the target address for a branch to link instruction that it believes corresponds to a subroutine return (pushed into the stack earlier)

32-entry count cache for address prediction (indexed by the address of Branch Conditional to Count Register (bcctr) instructions)

PowerPC 970 FX: Instruction Decode and Preprocessing

Three cycle pipeline to decode and preprocess instructions; Cracking one instruction into two internal operations

Cracked and micro-coded instructions have access to four renamed emulation GPRs (eGPRs), one renamed emulation FPR (eFPR), and one renamed emulation CR (eCR) field (in addition to architected facilities)

8-entry (16 bytes per entry) instruction fetch buffer (up to eight instructions in, five instructions out each cycle)

PowerPC 970 FX: Instruction Dispatch and Completion Control

Four dispatch buffers which can hold up to four dispatch groups when the global completion table (GCT) is full

20-entry global completion table

- ✓ Group-oriented tracking associates a five operation dispatch group with a single GCT entry
- ✓ Tracks internal operations from dispatch to completion for up to 100 operations
- ✓ Capable of restoring the machine state for any of the instructions in flight
- ✓ Very fast restoration for instructions on group boundaries (i.e., branches)
- ✓ Slower for instructions contained within a group

Supports precise exceptions

PowerPC 970 FX: Branch and Condition Register Execution Pipeline

One branch execution pipeline

- ✓ Computes actual branch address and branch direction for comparison with prediction
- ✓ Redirects instruction fetching if either prediction was incorrect
- ✓ Assists in training/maintaining the branch table predictors, the link stack, and the count cache

One condition register logical pipeline

- ✓ Executes CR logical instructions and the CR movement operations
- ✓ Executes some Move to/from Special Purpose Register (mtspr and mfspr) instructions also

Out-of-order issue with bias towards oldest operations first

PowerPC 970 FX: Data Stream Prefetch

Eight (modeable) data prefetch streams supported in hardware.

Eight hardware streams are only available if vector prefetch instructions are disabled

Four vector prefetch streams supported using four of the eight hardware streams.

The vector prefetch mapping algorithm supports the most commonly used forms of vector prefetch instructions

Intel P-VI: General

The P6 family of processors is the generation of processors that succeeds the Pentium® line of Intel processors

VI w't vs gi wsv'jeq n'p'q i r xw'k x i p w'h} r eq on execution micro-architecture

- ✓ Multiple branch prediction
- ✓ Data flow analysis
- ✓ Speculative execution

Intel P-VI: Three Engines and Interface with Memory**Intel P-VI: Major Units**

The FETCH/DECODE unit:

- ✓ An in-order unit that takes as input the user program instruction stream from the instruction cache, and
- ✓ Fi gshi w'x i q "m's"e"wi vinw'sj" -st i vexsr w"* st w'x ex'vi t vi wi r'x' i "hexej{s{ "sj" that instruction stream

The pre-fetch is speculative

The DISPATCH/EXECUTE unit:

- ✓ An out-of-order unit that accepts the dataflow stream, schedules execution of the instructions, and stores the results of these speculative executions

The RETIRE unit:

- ✓ An in-order unit that writes the results of speculative executions to permanent architectural state

The BUS INTERFACE unit:

- ✓ The bus interface unit communicates directly with the L2 (second level) cache supporting up to four concurrent cache accesses.
- ✓ The bus interface unit also controls a transaction bus, with MESI snooping protocol, to system memory

Intel P-VI: Inside Fetch

The L1 Instruction Cache fetches the cache line corresponding to the index from the Next_IP and presents 16 aligned bytes to the decoder.

The decoder converts the Intel CISC instructions into RISC-like instructions, one logical destination per operation

Some of the CISC instructions are converted into RISC-like instructions. Some instructions are decoded into one-to-many instructions. Some instructions require microcode

Some of the CISC instructions are converted into RISC-like instructions. Some instructions are decoded into one-to-many instructions. Some instructions require microcode

Some of the CISC instructions are converted into RISC-like instructions. Some instructions are decoded into one-to-many instructions. Some instructions require microcode

The instruction pool is implemented as an array of Content Addressable Memory called the Re-Order Buffer (ROB).

Intel P-VI: Inside Dispatch /Execute

VI i "F~~mt~~ ex~~l~~ "yr ~~nl~~wi ~~p~~ g~~w~~" st w~~j~~vsq "l i "m~~w~~xyg~~xsr~~"ssphi t ending upon their status
K~~x~~ i "w~~w~~xyw~~m~~nh~~g~~exi w~~x~~ ex"e" st "l ew~~e~~p~~s~~j" ~~nv~~st i ver h~~w~~ i r "l i "h~~mt~~ ex~~l~~ "yr ~~nl~~gl i gow~~x~~s"
wi i "h~~x~~ i "i | i gy~~xsr~~ "vi wsyvgi "ri i hi h~~f~~ } "l ex" st "n~~e~~ps~~w~~ezen~~ef~~ p
K~~f~~ s~~x~~ "ev i "xyi ."l i "Ti wi vz~~xsr~~ "U~~xsr~~ "vi q sz i w~~x~~ ex" st "er h~~w~~ i r h~~w~~ ~~nl~~to the resource
where it is executed

VI i "vi w'p'w'sj"x i " st "ei "xi v'vi xvr i h"x"x i "t ssp

There are five ports on the Reservation Station, and the multiple resources are accessed as shown

The P6 family of processors can schedule (in an out-of-order fashion) at a peak rate of 5 instructions per cycle, but a more typical

Psixologiya "s"i "st" "f" vergli w

The Branch Target Buffer (BTB) will correctly predict most of these branches

Overgl "st w'evi" tagged (in the in-order pipeline) with their fall-through address and the destination that was predicted for them

But if mispredicted, then the Jump Execution Unit (JEU) changes the status of all of the instructions in the instruction pool

In that case the proper branch destination is provided to the BTB which restarts the whole pipeline from the new target address

Intel P-VI: Inside Retire

Once removed, the instruction is placed in the "retire" pool.

Once removed, the instruction is placed in the "retire" pool. The Retire Unit must also re-impose the original program order on them.

The Retire Unit must first read the instruction pool to find the potential candidates for retirement and determine which of these candidates are next in the original program order.

The Retire Unit must first read the instruction pool to find the potential candidates for retirement and determine which of these candidates are next in the original program order.

Once removed, the instruction is placed in the "retire" pool. The Retire Unit must also re-impose the original program order on them.

Once removed, the instruction is placed in the "retire" pool. The Retire Unit must also re-impose the original program order on them.

Intel P-VI: Bus Interface Unit

Stores are never performed speculatively since there is no transparent way to undo them.

Stores are never performed speculatively since there is no transparent way to undo them.

Stores are never performed speculatively since there is no transparent way to undo them.

Stores are also never re-ordered among themselves.

A store is dispatched only when both the address and the data are available and there are no older stores awaiting dispatch.

A study of the importance of memory access reordering concluded:

- ✓ Stores must be constrained from passing other stores, for only a small impact on performance.
- ✓ Stores can be constrained from passing loads, for an inconsequential performance loss.
- ✓ Constraining loads from passing other loads or stores has a significant impact on performance.

The Memory Order Buffer (MOB) allows loads to pass other loads and stores by acting like a reservation station and re-order buffer
It holds suspended loads and stores and re-dispatches them when a blocking condition (dependency or resource) disappears

Summary

Today we have studied four advance computer architecture:

- ✓ PowerPC 750 and 970 FX
- ✓ Intel P-VI

With this we have completed our discussion on all topic of Advanced Computer Architecture

Next time, in the last lecture we will review all concepts we have studied in our earlier lectures

Lecture 45

Putting It All Together (Review: Lecture 1 - 43)

]

Module 1: Introduction
 Module 2: Instruction Set Architecture
 Module 3: Computer hardware design
 Module 4: Instruction Level Parallelism Dynamic
 Module 5: Instruction Level Parallelism Static
 Module 6: Memory Hierarchy system
 Module 7: Multiprocessing
 Module 8: I/O Systems
 Module 9: Networks and Clusters

Module 1: Introduction and Quantitative Principles

We started this course distinguishing the computer organization and computer architecture
 Architecture refers to those attributes of a computer visible to the programmer or compiler writer; e.g., instruction set, memory addressing techniques, I/O mechanisms
 Organization refers to how the features of a computer are implemented; e.g., control signals are generated using the principles of FSM or microprogramming
 The architecture of the members of a processor family are same whereas organization of same architecture may differ between different members of the family

Module 1: Introduction Computer Development

We also introduced the computers developments with academic and commercial perspectives
 Academically, modern computer developments have their infancy in 1944-49, when John von Neumann introduced the concept of stored-program computer, referred to as Electronic Discrete Variable Automatic Computer EDVAC
 Commercially, the first machine was built by Eckert-Mauchly Computer Corporation in 1949
 In 1971, Intel introduced first cheap microprocessor 4004 and then 80 x 86 series
 In 1998, more than 350 million microprocessors with different instruction set architectures were in use; this number has risen to more than a billion in 2006
 Technological developments, from vacuum tubes to VLSI circuits, dynamic memory and network technology gave birth to four different generations of computers
 This course has viewed the Computer Architecture from four perspectives

- ✓ Processor Design
- ✓ Memory Hierarchy
- ✓ Input/output and storages
- ✓ Multiprocessor and Network interconnection

Module 1: Quantitative Principles

The key to the quantitative analysis in determining the effectiveness of the entire computing system is the computer hardware and software performance

In this respect, we discussed:

- ✓ Price-performance design
- ✓ CPU performance metrics
- ✓ CPU benchmark suites

Module 1: Price-Performance Design

The issue of cost-performance is complex one

At one extreme, high-performance computers designer may not give importance to the cost in achieving the performance goal.

At the other end, low-cost designer may sacrifice performance to some extent.

The price-performance design lies between these extremes where the designer balances cost and hence price versus performance.

Module1: CPU Benchmark Suites

In order to compare the performance of two machines, a user can simply compare the execution time of the same workload running on both the machines.

In practice users want to know, without running their own programs, that how well the machine will perform on their workload.

This is accomplished by evaluating the machine using a set of benchmarks the programs specifically chosen to measure the performance.

Five levels of programs are used as benchmarks:

- ✓ Real Applications scientific programs evaluate the performance of a machine
- ✓ Modified Applications the real applications with certain blocks modified to focus desired aspects of application,
- ✓ Kernels the small key pieces extracted from the real program
- ✓ Toy benchmarks small codes normally used as beginning programming assignments.
- ✓ Synthetic benchmarks the small section of Artificially created program

Module1: Quantitative Principles of Performance Measurement

Quantitatively the performance of a system can be enhanced by speedup of a fraction of system based on the concept of the common case first

defines the Speedup due to enhancement E that accelerates a fraction F of the task as:

Module 1: Amdahl's Law

$$\text{Speedup (E)} = \text{Ex Time without Enhancement} / \text{Ex Time with Enhancement}$$

$$= \text{Performance with Enhancement} / \text{Performance without Enhancement}$$

Module 2: Instruction Set Architecture

The three pillars of computer architecture are:

- ✓ hardware,
- ✓ instruction set
- ✓ software

Hardware facilitates to run the software and instruction set is the interface between the hardware and software

While talking about the Instruction set architecture the focus of our discussion has been:

- ✓ ISA Taxonomy
- ✓ Types of operands
- ✓ Types of operations
- ✓ Memory Addressing modes

Module 2: Taxonomy of Instruction Set

The taxonomy of Instruction set was defined as:

- ✓ Stack Architecture:
- ✓ Accumulator Architecture
- ✓ General Purpose Register Architecture
 - Register memory
 - Register Register (load/store)
 - Memory Memory Architecture (Obsolete)

Module 2: Types of Operands and Operations

Operands Types:

- ✓ Integer, FP and Character

Operand Size:

- ✓ Half word, word, double word

Classification of operations:

- ✓ Arithmetic, data transfer, control
- ✓ and support operations

Module 2: Types of Operands addressing modes

Operand Addressing Modes:

- ✓ Immediate, register, direct (absolute) and Indirect

Classification of Indirect Addressing:

- ✓ Register, indexed, relative (i.e. with displacement) and memory

Special Addressing Modes:

- ✓ Auto-increment, auto-decrement and scaled

Control Instruction Addressing modes:

- ✓ Branch, jump and procedure call/return

Module 3: Computer Hardware design

Basic building blocks of a computer
Sub-systems of CPU: Datapath and Control
Processor design steps
Processor design parameters
Hardware design process
Timing signals
Uni-bus, 2-bus and 3-bus structures
3-bus based single cycles data path

Sub-systems of Central Processing Unit

The Central Processing Unit (CPU) consists of two main sub-systems:

- ✓ Data path: The path that facilitates the transfer of information from one part (register/memory/ IO) to the other part of the system
- ✓ Control: the hardware that generates signals to control the sequence of steps and direct the flow of information through the data path

Module 3: Datapath Implementations

The datapath is the arithmetic organ of the Von-Neumann architecture. It is responsible for executing the instructions of a program.

Typically, the datapath may be implemented as:

- ✓ Unibus structure
- ✓ 2-bus structure
- ✓ 3-bus structure

Based on the concepts of single cycle, multiple cycle and pipelined architecture
It consists of registers, internal buses, arithmetic units and shifters

Each register in the register file has:

- a load control line that enables data load to register
- a set of tri-state buffers between its output and the bus
- a read control line that enables its buffer and place the register on the bus

Module 3: Single/Multiple Cycle Approach

In the Single Cycle implementation, the cycle time is set to accommodate the longest instruction, the Load instruction.

In the Multiple Cycles implementation, the cycle time is set to accomplish longest step, the memory read/write

Consequently, the cycle time for the Single Cycle implementation can be five times longer than the multiple cycle implementation.

Module 3: Pipelined Datapath

Pipelining is a fundamental concept

Where an instructions is completed in multiple steps using distinct resources

It utilizes capabilities of the Datapath by

Starting next instruction while working on the current one

The pipelined datapath may encounter three types of hazards

- ✓ Structural, Data and Control

Module 3: Pipeline Hazards

Structural hazards occur when same resource is accessed by more than one instructions; e.g., One memory port or one register write port

It can be removed by using either multiple resources or inserting stall

Stall degrades the pipeline performance

Data Hazards occur when attempt is made to read invalid data

Data hazard can be removed by using stall and forwarding techniques

Control hazards occur when an attempt is made to branch prior to the evaluation of the condition

Four ways to handle control hazards

Module 3: 4 ways to handle control hazard

1. Stall until branch direction is clear
2. Predict Branch Not Taken
3. Execute successor instructions in sequence
 - ✓ Predict Branch Taken
4. Delayed Branch
 - ✓ Define branch to take place AFTER a following instruction

Module 4: Instruction Level parallelism

Simple pipeline facilitates in-order execution

Whereas, in order to enhance the performance of the pipeline, we want to begin execution as soon as the data operands are available, i.e., out-of-order execution

Out-of-order execution may introduce data hazards of type WAR and WAW

Instruction Level Parallelism can be achieved by Hardware or Software

In SW parallelism, the dependencies are defined by program result in hazards if HW cannot resolve

HW exploiting ILP works when dependence cannot be determined at run time

These hardware techniques to exploit ILP are referred to as Dynamic Scheduling techniques

Dynamic scheduling is accomplished by dividing the ID stage into two parts

- ✓ Issue the instruction in-order
- ✓ Read operand out-of-order

Structural and data dependencies are checked at ID stage

It facilitates out-of-order execution which results in out-of-order completion

We discussed the score-board technique for dynamic scheduling in integer and floating-point datapath

The structures implementing these concepts facilitate out-of-order execution to minimize data dependencies thus avoid data hazards without stalls

Tomasulo's Approach for IBM 360/91 to achieve high Performance without special compilers. Here, the control and buffers are distributed with Function Units (FU)

Registers in instructions are replaced by values or pointers to reservation stations (RS) ; i.e., the registers are renamed

Unlike Scoreboard, Tomasulo can have multiple loads outstanding

We also discussed branch-prediction techniques and different types of branch-predictors, used to reduce the number of stalls due to control hazards

The concept of multiple instructions issue was discussed in details

This concept is used to reduce the CPI to less than one, thus, the performance of the processor is enhanced

Yi "wyhnh" i | x r wsr w' x s" x i " Vsq ewyjs w' wwygyvi " f } mgyhmk" l evh{ evi -based speculation. It allows to speculate that branch is correctly predicted, thus may execute out-of-order but commit in-order having confirmed that the speculation is correct and no exceptions exist. The major hardware-based techniques studied are summarized here:

Technique	Hazards type stalls Reduced
Forwarding and bypass	Potential Data Hazard Stalls
Delayed Branching and Branch Scheduling	Control Hazard Stalls
Basic Dynamic Scheduling (score boarding)	Data Hazard Stalls from true dependences
Dynamic Scheduling with renaming *Vsq ewyjs w'Ct t vsegl +	Stalls from: data hazards from anti-dependences and from output dependences
Dynamic Branch Prediction	Control Hazard stalls
Speculation	Data and Control Hazard stalls
Multiple Instructions issues per cycle	Ideal CPI > 1

Module 5: Static Approach for ILP

The multiple-instruction-issues per cycle processors are rated as the high- performance processors

These processors exist in a variety of flavors, such as:

- ✓ Superscalar Processors
- ✓ VLIW processors
- ✓ Vector Processors

The superscalar processors exploit ILP using static as well as dynamic scheduling approaches

The VLIW processors, on the other hand, exploits ILP using static scheduling only

The major software scheduling techniques, under discussion, to reduce the data and control stalls, are as follows:

Technique	Hazards type stalls Reduced
Basic Compiler scheduling	Data hazard stalls
Loop Unrolling	Control hazard stalls
Compiler dependence	Ideal CPI, Data hazard stalls
Trace Scheduling	Ideal CPI, Data hazard stalls
Compiler Speculation	Ideal CPI, Data and control hazard stalls

Module 6: Memory Hierarchy System

Here, we discussed how the gap between the speed of processor and the storage devices - DRAM, SRAM and Disk is increasing with time

We studied that in order to obtain high speed storage at the cheapest cost per byte, different types of memory modules are organize in hierarchy, based on the:

- ✓ Concept of Caching and
- ✓ Principle of Locality

The principle of locality states that to obtain data or instructions of a program, the processor access, at any instant of time, a relatively small portion of the address space of the fastest memory closet to the processor

There are two different types of locality:

- ✓ Temporal locality is the locality in time
- ✓ Spatial locality is the locality in space

Concept of caching states that a small, fastest and most expensive storage be used as the staging area or temporary-place to:

- ✓ Store frequently-used subset of the data or instructions from the relatively cheaper, larger and slower memory; and
- ✓ To avoid having to go to the main memory every time this information is needed

The performance of cache is limited by different types of penalties

Then we that talked four options to improve the cache performance

These options are used to reduce:

- ✓ the miss penalty
- ✓ the miss rate
- ✓ the miss Penalty or miss rate
- ✓ the time to hit in the cache
- via Parallelism

Module 7: Multiprocessing

In this series of four lectures on multiprocessors we have studied, how improvement in computer performance can be accomplished using Parallel Processing Architectures?

Parallel Architecture is a collection of processing elements that cooperate and communicate to solve larger problems fast

Then we described the four categories of Parallel Architecture as: SISD, SIMD, MISD and MIMD architecture

We noticed that based on the memory organization and interconnect strategy, the MIMD machines are classified as:

- ✓ Centralized Shared Memory Architecture
- ✓ Distributed Memory Architecture

We also introduced the framework to describe parallel architecture as a two layer representation: Programming and Communication models

We talked about sharing of caches for multi-processing in the symmetric shared-memory architecture in details

Here, we studied the cache coherence problem and introduced two methods, write invalidation and write broadcasting schemes, to resolve the problem

We also discussed the finite state machine for the implementation of snooping algorithm

Module 8: I/O Systems

The overall performance of a computer is measured by its throughput, which is very much influenced by the systems external to the processor

The effect of neglecting the I/Os on the overall performance of a computer system can best be visualized by Amdahl's Law which identifies that: system speed-up limited by the slowest part!

Then we discussed the trends in I/O inter-connects as: the networks, channels and backplanes

The networks offer message-based narrow-pathway for distributed processors over long distance

The backplanes offer memory-mapped wide pathway for centralized processing over short distance

The interconnects are implemented via buses

The buses are classified in two major categories as the I/O bus and CPU-Memory bus

The channels are implemented using I/O buses and backplanes using CPU-Memory buses

We discussed the bus transition protocols which specify the sequence of events and timing requirements in transferring information as synchronous or asynchronous communication

Y i "eps"wyhinh"fywef nexsr "t vsxsgpw" "x i "t vsxsgpw"s"i wi vzi "x i "f ywf" } "e"hi zngi " that wishes to communicates when multiple devices need the bus access

Here, we noticed that the bus arbitration schemes usually try to balance two factors:

Bus-priority: the device with highest priority should be serviced first

Fairness: every device that want to use the bus is guaranteed to get the bus eventually

The three bus arbitration schemes are:

- ✓ Daisy Chain Arbitration
- ✓ Centralized Parallel Arbitration
- ✓ Distributed Arbitration

Having discussed the basic types of storage devices and the ways to interconnect them to the CPU, we studied the ways to evaluate the performance of storage I/O systems. Here, we noticed that the reliability of a system can be improved by using the following four methods

Fault Avoidance	Prevent fault occurrence by construction
Fault Tolerance	providing service complying with the service specification by redundancy
Error Removal	minimizing the presence of errors by verification
Error Forecasting	to estimate the presence, creation and consequence of errors by evaluation

Module 9: Networks and Clusters

The formation of a generic interconnection network that comprises

- ✓ Computer nodes (host or end system)
- ✓ H/W and S/W interface
- ✓ Links to the interconnection network and
- ✓ Communication subnet

The interconnections are designated as:

- ✓ Local Area Network-LAN
- ✓ Wide Area Network-WAN
- ✓ System (or Storage) Area Network-SAN

While talking about the interconnect model, software and protocols we studied that:

- ✓ The interconnect communication model shows that two machines are connected via two unidirectional wires with a FIFO (queue) at the end to hold the data
- ✓ The communication software separates the header and trailer from the message and identifies the request, reply, their acknowledgments and error checking codes
- ✓ The communication protocols suggest the sequence of steps to reliable communication

We classified the distributed switch interconnects as the fully and partially connected, symmetric and asymmetric interconnects

Then we discussed the linear array, ring, 2D mesh/torus and hypercube topologies and their performance measures

We also discussed Internetworking, i.e., the connection of two or more interconnection networks to communicate reliably and efficiently

Internetworking relies on the communication standards composed of hierarchy of layers

The internet communication Protocol Families facilitates applications to work with any inter-connection