Short Notes Chapter 5 [Lecture 13-15]

CALL and RET:

In every processor, instructions are available to divert temporarily and to divert permanently. The instructions for permanent diversion in 8088 are the jump instructions, while the instruction for temporary diversion is the CALL instruction. The word call must be familiar to the readers from subroutine call in higher-level languages. The CALL instruction allows temporary diversion and therefore reusability of code.

When the subroutine completes, we need to return. The RET instruction is used for this purpose. The word return holds in its meaning that we are to return from where we came and need no explicit destination. Therefore, RET takes no arguments and transfers control back to the instruction following the CALL that took us in this subroutine. CALL takes a label as argument and execution starts from that label, until the RET instruction is encountered and it takes execution back to the instruction following the CALL.

Both the instructions CALL and RET are commonly used as a pair, however technically they are independent in their operation. The RET works regardless of the CALL and the CALL works regardless of the RET. If you CALL a subroutine it will not complain if there is no RET present and similarly if you RET without being called it won't complain. It is a logical pair and is used as a pair in every decent code.

Stack:

The operation of placing an element on top of the stack is called pushing the element and the operation of removing an element from the top of the stack is called popping the element. The last thing pushed is popped out first; the last in first out behavior.

The stack operations of pushing and popping only work at the top of the stack. This top of stack is contained in the SP register. **The physical address of the stack is obtained by the SS:SP combination.** The stack segment registers tell where the stack is located and the stack pointer marks the top of stack inside this segment

A decrementing stack moves from higher addresses to lower addresses as elements are added in it while an incrementing stack moves from lower addresses to higher addresses as elements are added.

PUSH and POP:

The operations of placing items on the stack and removing them from there are called push and pop. The push operation copies its operand on the stack, while the pop operation makes a copy from the top of the stack into its operand.

The operation of PUSH is:

SP←SP-2
[SP]←AX

The operation of POP is:

AX←[SP]
SP←SP + 2

PUSH:

PUSH decrements SP (the stack pointer) by two and then transfers a word from the source operand to the top of stack now pointed to by SP. PUSH often is used to place parameters on the stack before calling a procedure; more generally, it is the basic means of storing temporary data on the stack.

POP:

POP transfers the word at the current top of stack (pointed to by SP) to the destination operand and then increments SP by two to point to the new top of stack. POP can be used to move temporary variables from the stack to registers or memory.

The operand of PUSH is called **a source operand** since the data is moving to the stack from the operand, while the operand of POP is called **destination** since data is moving from the stack to the operand.

CALL:

CALL activates an out-of-line procedure, saving information on the stack to permit a RET (return) instruction in the procedure to transfer control back to the instruction following the CALL.

RET:

RET (Return) transfers control from a procedure back to the instruction following the CALL that activated the procedure. RET pops the word at the top of the stack (pointed to by register SP) into the instruction pointer and increments SP by two.

Local Variables:

Another important role of the stack is in the creation of local variables that are only needed while the subroutine is in execution and not afterwards. They should not take permanent space like global variables.

Local variables should be created when the subroutine is called and discarded afterwards. So that the spaced used by them can be reused for the local variables of another subroutine. They only have meaning inside the subroutine and no meaning outside it.
The standard way to start a subroutine which needs to access parameters and has local variables is as under:

```
push bp
mov bp, sp
sub sp, 2
```