Procedural and Loop Instructions

Procedural Calls

Procedures are considered the most important innovation in the development of programming languages. A procedure is a self-contained computer program that is incorporated into a large program. At any point in the program, the procedure may be *invoked* or *called*. The processor is instructed to execute the entire procedure and then return to the point from which the call took place. Procedures are also known as **subroutines**. Below are some characteristics of a procedure (Stallings, 2019):

- It allows the same piece of code to be used multiple times.
- It improves efficient use of storage space in a system.
- It allows large programming tasks to be subdivided into smaller units.
- It greatly eases the programming tasks.
- It can be called from more than one (1) location.
- A procedure call can appear in another procedure, which allows the nesting procedures to an arbitrary depth.
- All procedure call is matched by a return instruction in the called program.

In order to call a procedure from a variety of points, the processor must save the return address so that the return process can take place correctly. The return address can be stored *in a register*, *start of the called procedure*, or at the *top of the stack*.

A **reentrant procedure** is a procedure that can be implemented in such a way that more than one process can execute it at the same time without any conflict. Storing the return address in a register or at the start of the called procedure complicates the utilization of the reentrant procedure. If the parameters are passed via register or memory for a reentrant procedure, a specific set of codes must be responsible for saving the parameters so that the register or memory space is available for other procedural calls (Stallings, 2019).

The procedure mechanism involves two (2) basic instructions, which are:

1. **Call instruction (CALL)**: This branches from the present location to the procedure. The following activities take place inside the microprocessor when the CALL instruction is used (Sharma, 2019):
   - The address of the next instruction that exists in the program is stored in the stack.
   - The instruction queue is emptied to accommodate the instructions of the procedure.
   - The content of the instruction pointer (IP) is then changed with the address of the first instruction of the procedure.
   - The subsequent instructions of the procedure are stored in the instruction queue for execution

   A **Near CALL** is an instruction to call a procedure that is in the same code segment as the CALL instruction that calls it, while the **Far CALL** is used to call a procedure from a different segment.

Common x86 call instructions:

Instruction	Description
CALL Address	This instruction pushes the current IP value onto the stack and causes a jump to the entry point of the procedure by placing the address of the entry point in the IP. This basically calls a procedure of a function.
CALL Label (Near)	This instruction pushes the content of the IP onto the stack and jumps to the offset address in the current code segment.
CALL Label (Far)	This instruction pushes the content of the IP onto the stack and jumps to the offset address in any memory location.
CALL Memory (Near)	This instruction pushes the content of the IP onto the stack and jumps to the offset address in the memory register of the current code segment.
CALL Memory (Far)	This instruction pushes the content of the IP onto the stack and jumps to the offset address of the register at any memory location.

2. **Return instruction (RET)**: This returns the control to the place from which a procedure was called. Thus, it is the last instruction in any procedure. The following activities take place inside the microprocessor when the RET instruction is used (Sharma, 2019):
   - The address of the next instruction in the mainline program, which was previously stored inside the stack, is fetched again and is placed inside the instruction pointer (IP).
   - The instruction queue will again be filled with the subsequent instructions of the mainline program.

Emu8086 return instructions:

Instruction	Description
RET	This instruction returns the address from a near procedure. The return will be done by replacing the instruction pointer (IP) with a word from the top of the stack. Possible algorithms: ▪ Pop from stack: IP

	▪ If immediate operand is present: SP = SP + operand
RETF	This instruction returns the address from a far procedure. The return will be done by replacing the stack pointer (SP) with a word at the top of the stack. Possible algorithms: ▪ Pop from stack: IP or CS ▪ If immediate operand is present: SP = SP + operand

Branch Instructions

A **branch instruction** contains the address of the next instruction to be executed as one of its operands (Stallings, 2019). These instructions are used to implement control flow in program loops and conditionals. A branch instruction is generally classified as follows (JavaTpoint, n.d.):

- **Direct**: The instruction contains the target address.
- **Indirect**: The instruction specifies where the target address is to be found.
- **Relative**: The instruction specifies the differences between the current and the target address.

In computer organization, a *branch* may also refer to the act of switching execution to a different instruction sequence as a result of executing a branch instruction. Note that a branch can either be *forward – instructions with higher addresses* or *backward – instructions with lower addresses*. A branch instruction computes the target address in one of the following ways (JavaTpoint, n.d.):

- The target address is the sum of a constant and the address of the branch instructions itself.
- The target address is the absolute address given as an operand to the instruction.
- The target address is the address found in the link register.
- The target address is the address found in the count register.

Below are the two (2) categories of branch instructions:

1. **Unconditional branches** – These instructions always result in branching.
2. **Conditional branches** – These branch instructions may or may not cause branching, depending on some specified conditions. The address of the next instruction following a conditional branch instruction cannot be confirmed until the branch condition has been evaluated (Ledin, 2020).
   - If the branch condition is not satisfied, the processor will execute the instruction that follows the conditional branch.
   - If the branch condition is satisfied, the next instruction is at the address indicated in the conditional branch instruction.

Jump Instructions

Jump instructions facilitate the transfer of the program sequence to the memory address given in the operand, based on the specified flag. These instructions are technically used for changing the execution flow of instructions in the processor by jumping to any instruction in between the code. Jump instructions are divided into two types (Sharma, 2019):

- **Unconditional Jump Instructions** – These are used to jump on a particular location unconditionally. Thus, there is no specific condition that needs to be satisfied for the jump to take place. There are three (3) types of unconditional jump:
  1. **Near JMP** – This procedure targets memory location within the same code segment (intra-segment).
  2. **Short JMP** – This procedure targets memory location within the same code segment (intra-segment), but the offset is 1byte long.
  3. **Far JMP** – This procedure targets memory location that is outside the segment (inter-segment), and the size of the instruction pointer is a double word.
- **Conditional Jump Instructions** – In implementing conditional jumps, the processor must check for a specific condition for the jump to take place. Each status flag, or a combination of the status flags, is tested for a conditional jump. The following are the status bits of each flag and its corresponding name: *C – carry flag*, *P – parity flag*, *A – auxiliary carry flag*, *Z – zero flag*, *S – sign flag*, and *O – overflow flag*.

Some Emu8086 jump instructions:

Instructions	Description	Algorithm
JA Label	Short jump if the 1st operand is above the 2nd operand (as set by the CMP instruction); Unsigned	If C=0 and Z=0, then jump.
JAE Label	Short jump if the 1st operand is above or equal to the 2nd operand (as set by the CMP instruction); Unsigned	If C=0, then jump.
JB Label	Short jump if 1st operand is below the 2nd operand (as set by the CMP instruction); Unsigned	If C=1, then jump.
JBE Label	Short jump if 1st operand is below or equal to the 2nd operand (as set by CMP instruction); Unsigned	If C=1 or Z=1, then jump.
JC Label	Short jump if carry flag is set to 1	If C=1, then jump.

JE Label	Short jump if 1st operand is equal to 2nd operand (as set by the CMP instruction); Either signed or unsigned	If Z=1, then jump.
JG Label	Short jump if 1st operand is greater than the 2nd operand (as set by the CMP instruction); Signed	If Z=0 and S=F, then jump.
JMP Label	Unconditional jump; Transfers control to another part of the program; 4byte address may be entered in the form: *1234h:5678h*, where the 1st value is a segment and the 2nd value is an offset	Always jump
JNP Label	Short jump if no parity (odd); Only the 8 low bits of the result are checked; Set by the CMP, SUB, ADD, TEST, AND, OR, or XOR instructions	If P=0, then jump.
JP Label	Short jump if parity (even); Only the 8 low bits of the result are checked; Set by the CMP, SUB, ADD, TEST, AND, OR, or XOR instructions	If P=1, then jump.
JNS Label	Short jump if not signed (if positive); Set by CMP, SUB, ADD, TEST, AND, OR, or XOR instructions.	If S=0, then jump.
JS Label	Short jump is signed (if negative); Set by CMP, SUB, ADD, TEST, AND, OR, or XOR instructions.	If S=1, then jump.
JO Label	Short jump if overflow	If O=1, then jump.

Iterations and Looping

A **loop** is a block of statements that are repeatedly executed until a specific condition is satisfied. In assembly language, the JMP instruction is commonly used to implement loops. However, the processor instruction set includes a group of loop instructions for implementing iterations conveniently (Naeem, n.d.).

Common x86 iteration and looping instructions:

Instruction	Description	Algorithm
LOOP Label	Decreases the CX; Jump to label if CX is not zero	• CX=CX-1 • If CX<>0 then ○ jump else ○ no jump, continue
LOOPE/LOOPZ Label	Decrease the CX; Jump to label if CX is not zero and the value of Z is one (equal)	• CX=CX-1 • If CX<>0 and Z=1 then ○ jump else ○ no jump, continue
LOOPNE/LOOPNZ Label	Decrease the CX; Jump to label if CX is not zero and the value of Z is zero (not equal)	• CX=CX-1 • If CX<>0 and Z=0 then, ○ jump else ○ no jump, continue
JCXZ Label	Short jump if CX register is 0	If CX=0, then jump.

Programs typically contains a number of iterative loops and subroutines. Once a loop or a subroutine is entered, there are repeated referencing to a small set of instructions.

References:

JavaTpoint (n.d.). *Branch instruction in computer organization*. Retrieved on November 4, 2021 from https://www.javatpoint.com/branch-instruction-in-computer-organization

Ledin, J. (2020). *Modern computer architecture and organization*. Packt Publishing

Naeem, A. (n.d.). What are loops in assembly language?. Retrieved on November 6, 2021 from https://www.educative.io/edpresso/what-are-loops-in-assembly-language

Sharma, M. (2019, July 26). Jump instructions in 8086 microprocessor. Retrieved on November 5, 2021 from https://www.includehelp.com/embedded-system/jump-instructions-in-8086-microprocessor.aspx

Sharma, M. (2019, July 30). *The CALL and RET instruction in the 8086 microprocessor*. Retrieved on November 4, 2021 from https://www.includehelp.com/embedded-system/the-call-and-ret-instruction-in-the-8086-microprocessor.aspx

Stallings, W. (2019). *Computer organization and architecture: Designing for performance* (11th ed.). Pearson Education, Inc.