Ambo University

Hachalu Hundessa Campus
School of Informatics and Electrical Engineering

Department of Computer Science

Microprocessor & Assembly Language Programming (CoSc3025)

Kenesa B. (getkennyo@gmail.com)

CHAPTER FIVE

PROGRAM CONTROL INSTRUCTIONS

Introduction

❑ What is a program control instruction?

- Instructions are fetched from successive memory locations for processing and executing.
- The change in the content of the program counter can cause a break in the instruction execution.
- However, the program control instructions control the flow of program execution and can branch to different program segments.
  - Program control instructions modify or change the flow of a program.
- It is the instruction that alters the sequence of the program's execution
  - which means it changes the value of the program counter, due to which the execution of the program changes.

THE JUMP GROUP

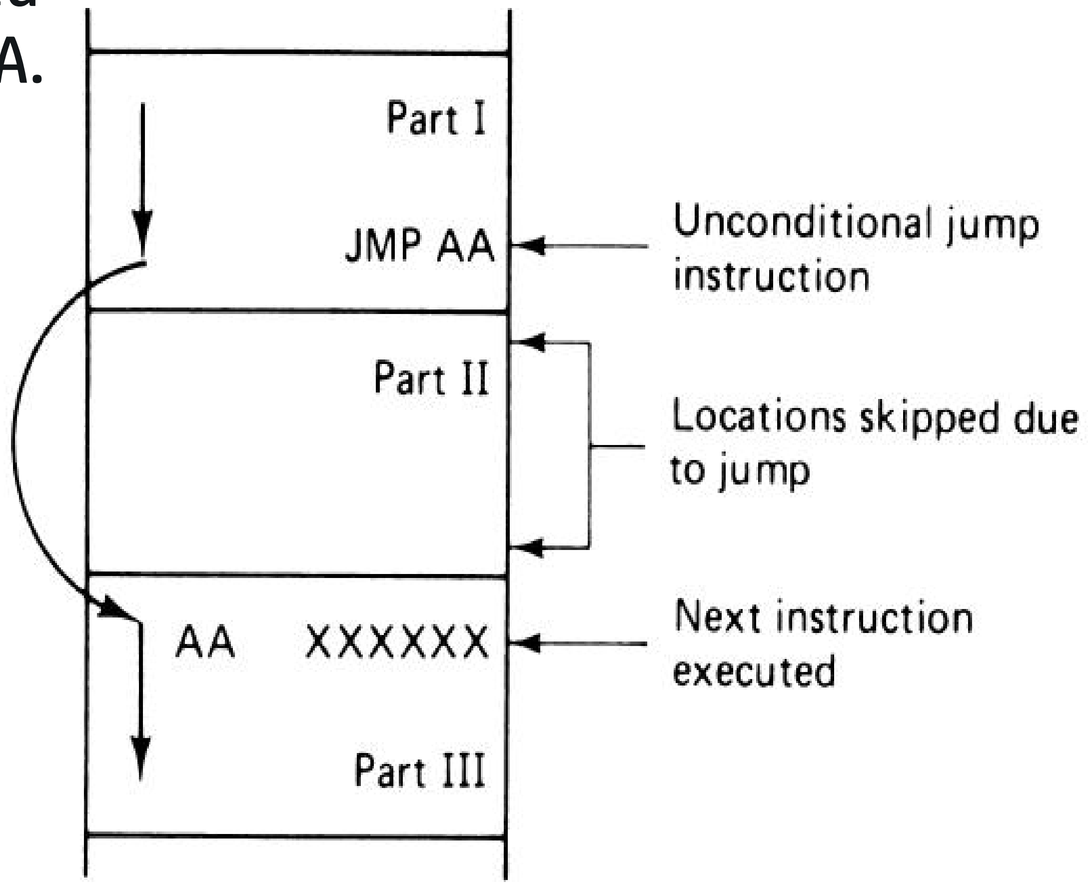
- Jump (JMP) instruction allows the programmer to skip sections of a program and branch to any part of the memory for the next instruction.
- Jump/branch are two types:
  - Unconditional Jump/branch
  - Conditional Jump/branch
- A conditional jump instruction allows decisions based upon numerical tests.
  - results are held in the flag bits, then tested by conditional jump instructions
- LOOP and conditional LOOP are also forms of the jump instruction.
- Branch/jump is usually an indication of a short change relative to the current program counter.
- Jump is usually an indication of a change in program counter that is not directly related to the current program counter, and is often free of distance limits from the current program counter.

Unconditional Jump (JMP)

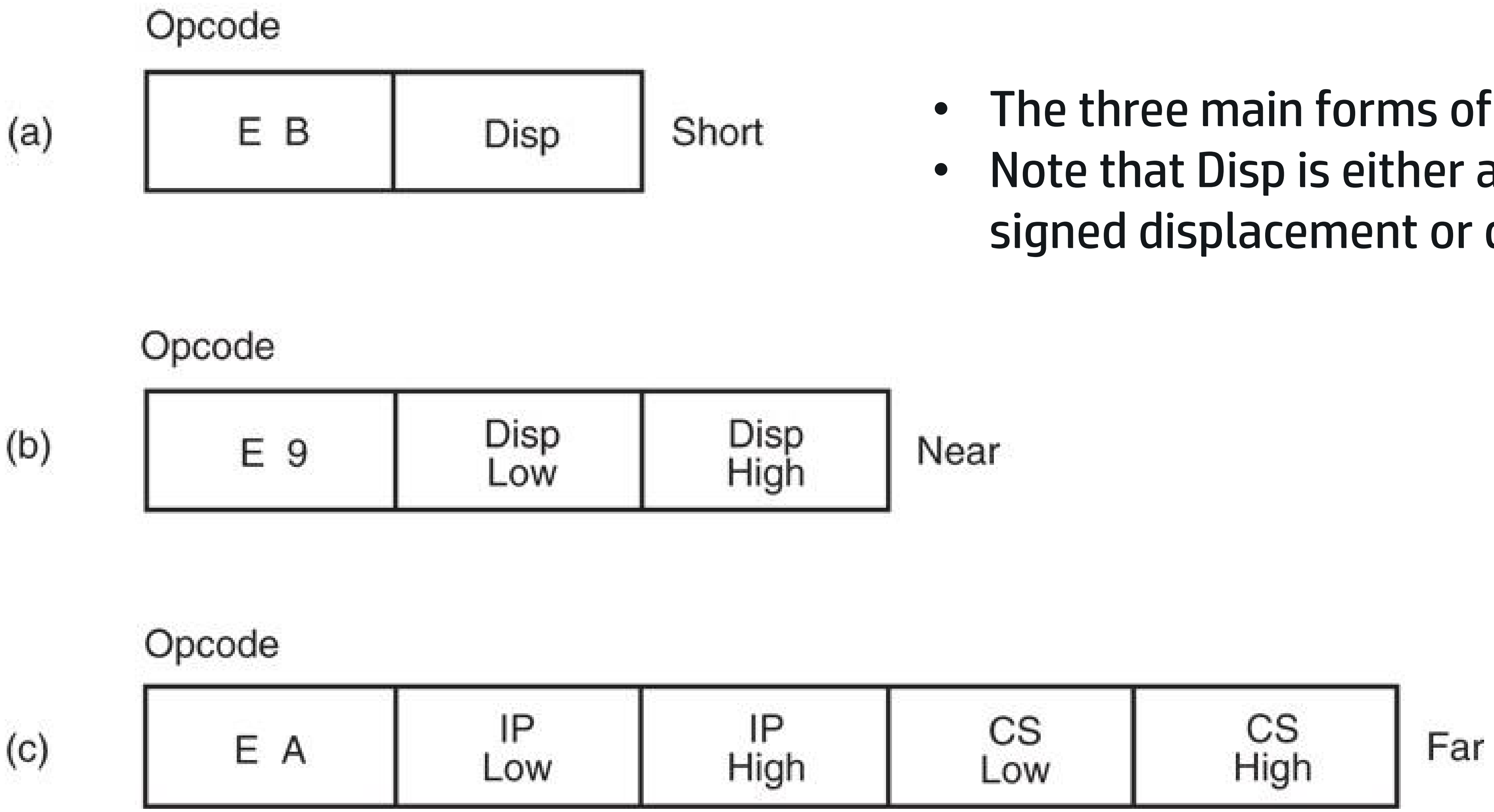
- In an unconditional jump, no status requirements are imposed for the jump to occur.
  - That is, as the instruction is executed, the jump always takes place to change the execution sequence.
- Three types: short jump, near jump, far jump.
- Short jump is a 2-byte instruction that allows jumps or branches to memory locations within +127 and −128 bytes.
  - from the address following the jump
- 3-byte near jump allows a branch or jump within ± 32K bytes from the instruction in the current code segment.
- 5-byte far jump allows a jump to any memory location within the real memory system.
  - The short and near jumps are often called intrasegment jumps.
  - Far jumps are called intersegment jumps.

Unconditional Jump (JMP)

- Note that when the instruction JMP AA in part- I is executed, program control is passed to a point in part III, identified by the label AA.
- Execution resumes with the instruction corresponding to AA.
- In this way, the instructions in part II of the program are bypassed—that is, they are jumped over.
- Some high-level languages have a GOTO statement.
  - This is an example of a high-level language program construct that performs an unconditional jump operation.



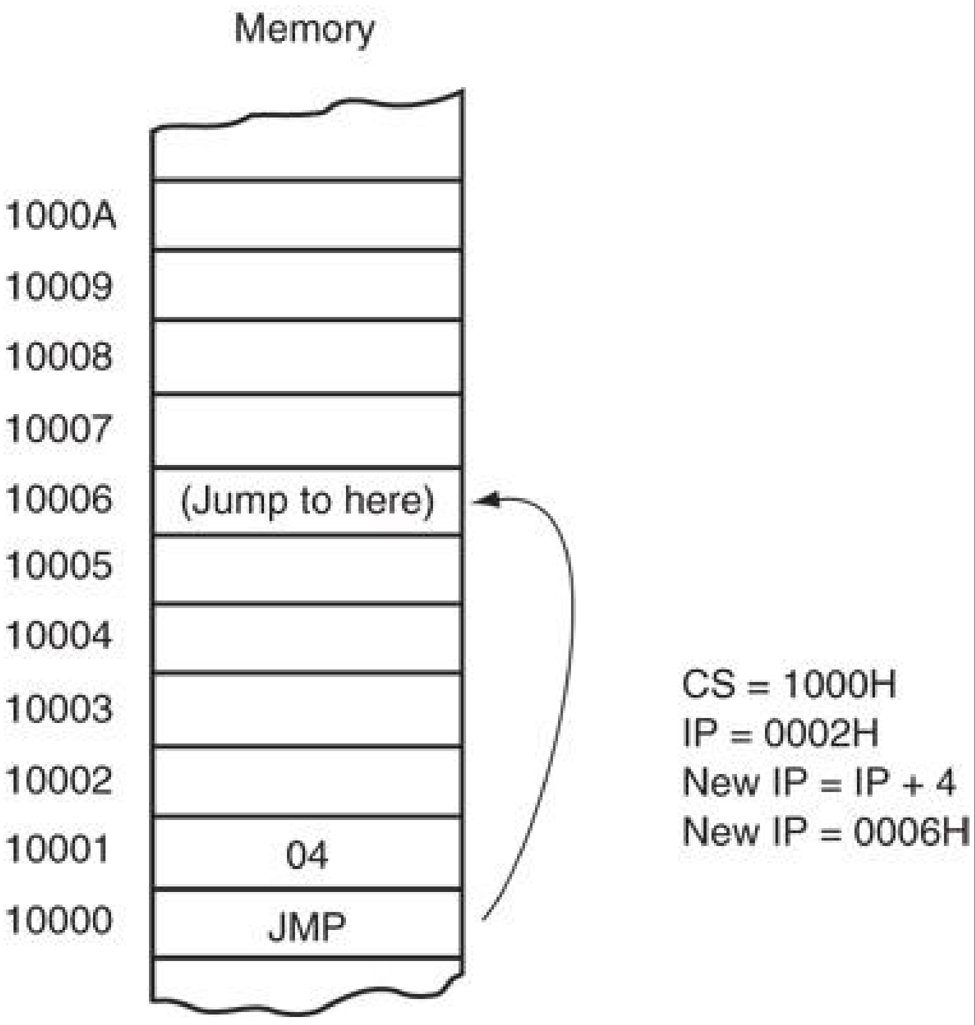
Unconditional Jump (JMP)



- The three main forms of the JMP instruction.
- Note that Disp is either an 8- or 16-bit signed displacement or distance.

Short Jump

- Called relative jumps because they can be moved, with related software, to any location in the current code segment without a change.
  - jump address is not stored with the opcode
  - a distance, or displacement, follows the opcode
- The short jump displacement is a distance represented by a 1-byte signed number whose value ranges between +127 and −128.
- when the microprocessor executes a short jump, the displacement is sign-extended and added to the instruction pointer (IP/EIP) to generate the jump address within the current code segment



- A short jump to four memory locations beyond the address of the next instruction.
- The instruction branches to this new address for the next instruction in the program

Short Jump

- When a jump references an address, a label normally identifies the address.
- The JMP NEXT instruction is an example.
  - it jumps to label NEXT for the next instruction
- The label NEXT must be followed by a colon (NEXT:) to allow an instruction to reference it
  - if a colon does not follow, you cannot jump to it

```
       XOR  BX,BX
START: MOV  AX,1
       ADD  AX,BX
       JMP  SHORT NEXT
       <skipped memory locations>
NEXT:  MOV  BX,AX
       JMP  START
```

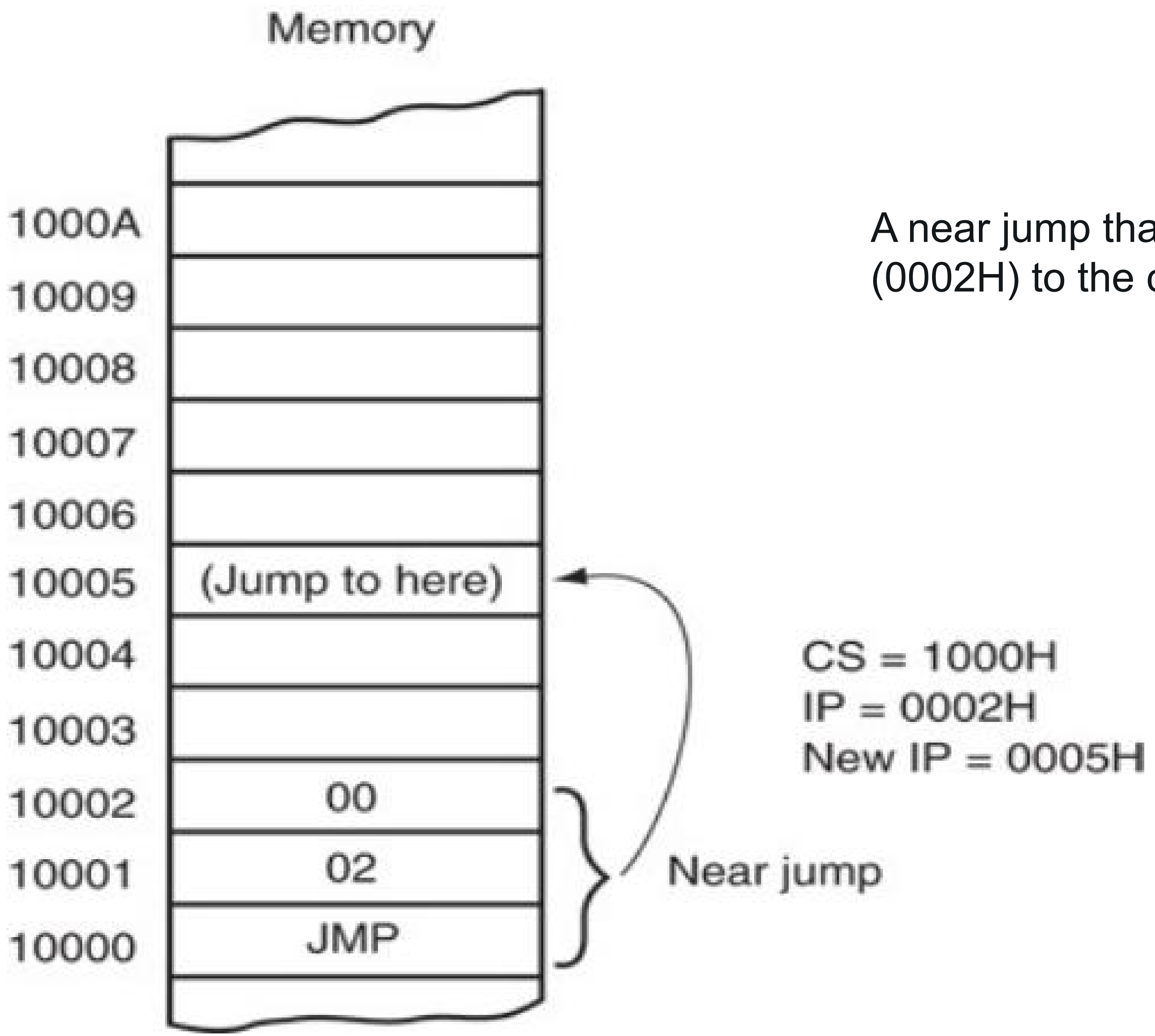
Short Jump

- Notice how one jump (JMP SHORT NEXT) uses the SHORT directive to forcea short jump, while the other does not.
- Most assembler programs choose the best form of the jump instruction so the second jump instruction (JMP START) also assembles as a short jump.
  - If the address of the next instruction (0009H) is added to the sign-extended displacement (0017H) of the first jump, the address of NEXT is at location 0017H + 0009H or 0020H.
- The only time a colon is used is when the label is used with a jump or call instruction.
  - It is very rare to use an actual hexadecimal address with any jump instruction
- but the assembler supports addressing in relation to the instruction pointer by using the **$+a** displacement.
- For example: JMP $+2
  - this instruction jumps over the next two memory locations (bytes) following the JMP instruction.

Near Jump

- A near jump passes control to an instruction in the current code segment located within ± 32K bytes from the near jump instruction.
- Near jump is a 3-byte instruction with opcode followed by a signed 16-bit displacement.
- Signed displacement adds to the instruction pointer (IP) to generate the jump address.
  - because signed displacement is ± 32Kb, a near jump can jump to any memory location within the current real mode code segment
- The near jump is also relocatable because it is also a relative jump.
- This feature, along with the relocatable data segments, Intel microprocessors ideal for use in a general-purpose computer system.
- Software can be written and loaded anywhere in the memory and function without modification because of the relative jumps and relocatable data segments.

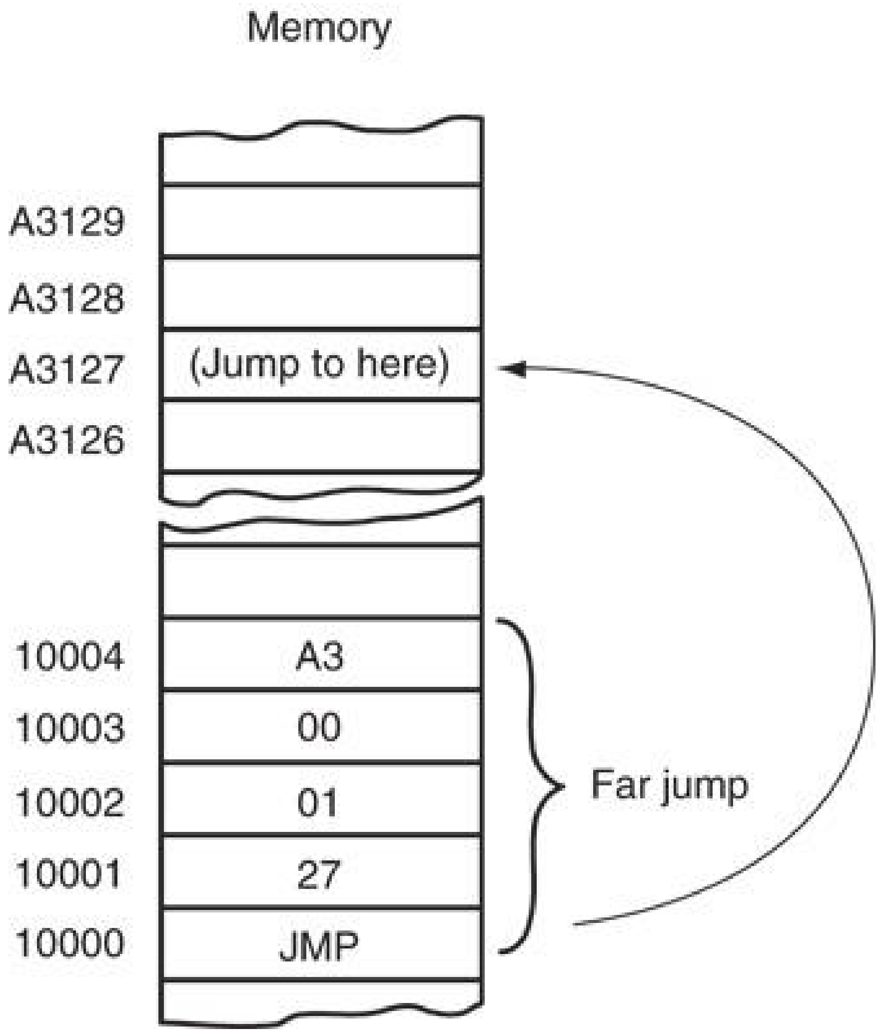
Near Jump



A near jump that adds the displacement (0002H) to the contents of IP.

Far Jump

- Obtains a new segment and offset address to accomplish the jump:
  - bytes 2 and 3 of this 5-byte instruction contain the new offset address
  - bytes 4 and 5 contain the new segment address
  - in protected mode, the segment address accesses a descriptor with the base address of the far jump segment
  - offset address, either 16 or 32 bits, contains the offset address within the new code segment



A far jump instruction replaces the contents of both CS and IP with 4 bytes following the opcode.

Far Jump

- The far jump instruction sometimes appears with the FAR PTR directive
- Another way to obtain a far jump is to define a label as a far label.
- A label is far only if it is external to the current code segment or procedure.
- The JMP UP instruction in the example references a far label.
  - The label UP is defined as a far label by the EXTRN UP:FAR directive.
  - External labels appear in programs that contain more than one program file.

```
        EXTRN UP:FAR
        XOR BX,BX
START:  ADD AX,1
        JMP NEXT
        ;<skipped memory locations>
NEXT:   MOV BX,AX
        JMP FAR PTR START
        JMP UP
```

Jumps with Register Operands

- Jump can also use a 16- or 32-bit register as an operand.
  - automatically sets up as an indirect jump
  - address of the jump is in the register specified by the jump instruction
- Unlike displacement associated with the near jump, register contents are transferred directly into the instruction pointer.
- An indirect jump does not add to the instruction pointer.
- **JMP AX**, for example, copies the contents of the AX register into the IP.
  - allows a jump to any location within the current code segment
- In 80386 and above, JMP EAX also jumps to any location within the current code segment;
  - in protected mode the code segment can be 4G bytes long, so a 32-bit offset address is needed

Indirect Jumps Using an Index

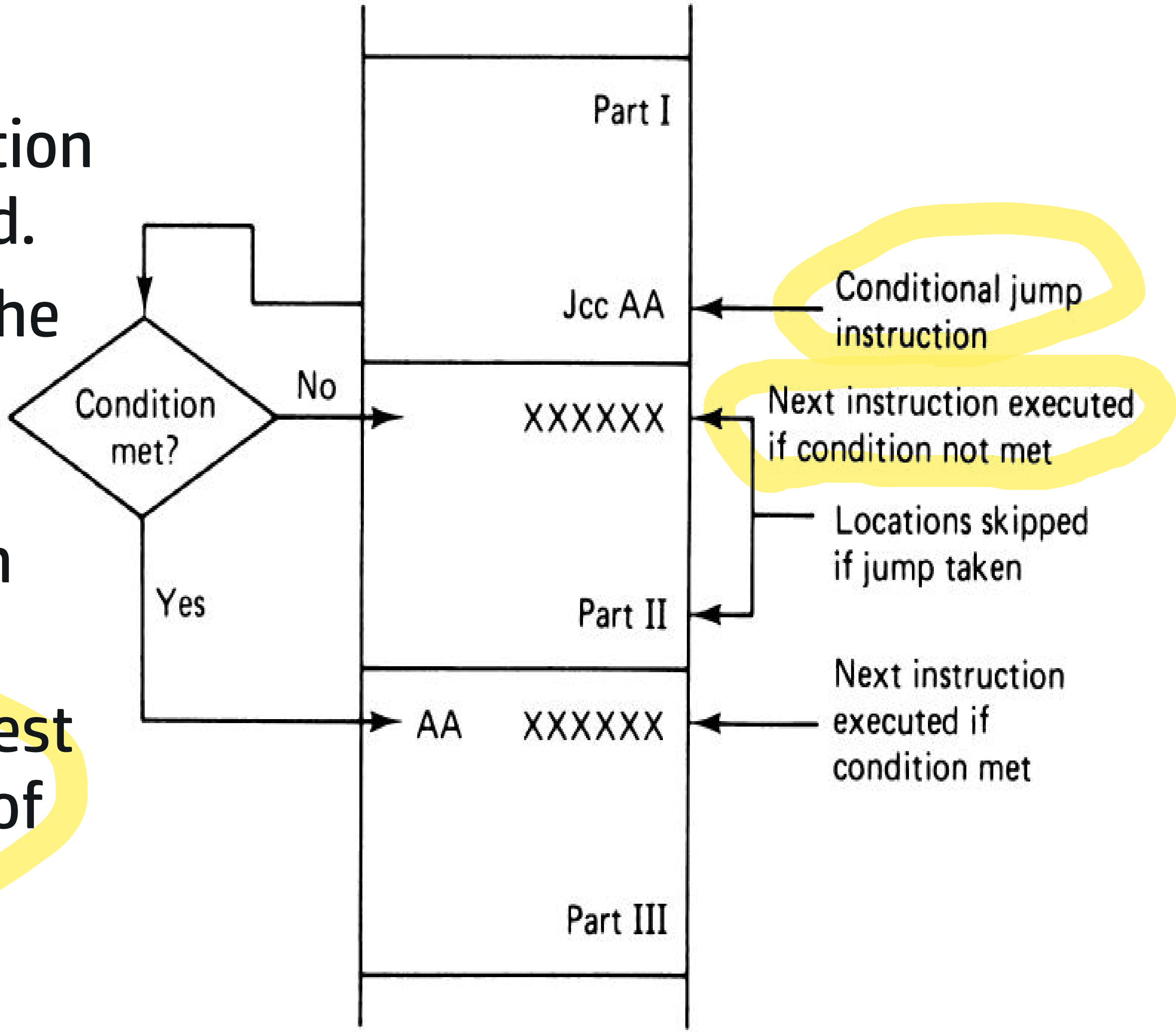
- Jump instruction may also use the [ ] form of addressing to directly access the jump table.
- The jump table can contain offset addresses for near indirect jumps, or segment and offset addresses for far indirect jumps.
  - also known as a **double-indirect jump** if the register jump is called an indirect jump
- The assembler assumes that the jump is near unless the FAR PTR directive indicates a far jump instruction.
- Mechanism used to access the jump table is identical with a normal memory reference.
  - JMP TABLE [SI] instruction points to a jump address stored at the code segment offset location addressed by SI
- Both the register and indirect indexed jump instructions usually address a 16-bit offset - both types of jumps are near jumps
- If JMP FAR PTR [SI] or JMP TABLE [SI], with TABLE data defined with the DD directive:
  - microprocessor assumes the jump table contains doubleword, 32-bit addresses (IP and CS)

Conditional Jumps

- Conditional jump instructions tests for the presence or absence of certain status conditions.
- The status conditions that exist at the time the jump instruction is executed decide whether or not the jump will occur.
  - If the condition or conditions are met, the jump takes place;
  - Otherwise, execution continues with the next sequential instruction of the program.
- The conditions that can be referenced by a conditional jump instruction are status flags such as carry (CF), zero (ZF), sign (SF) flags…
- Always short jumps in 8086 - 80286.
  - – limits range to within +127 and – 128 bytes from the location following the conditional jump

Conditional Jumps

- From the following example, we see that execution of the conditional jump instruction Jcc AA in part I causes a test to be initiated.
- If the conditions of the test are not met, the NO path is taken and execution continues with the next sequential instruction.
- This corresponds to the first instruction in part II.
- However, if the result of the conditional test is YES, a jump is initiated to the segment of program identified as part III, and the instructions in part II are bypassed.



Conditional Jumps

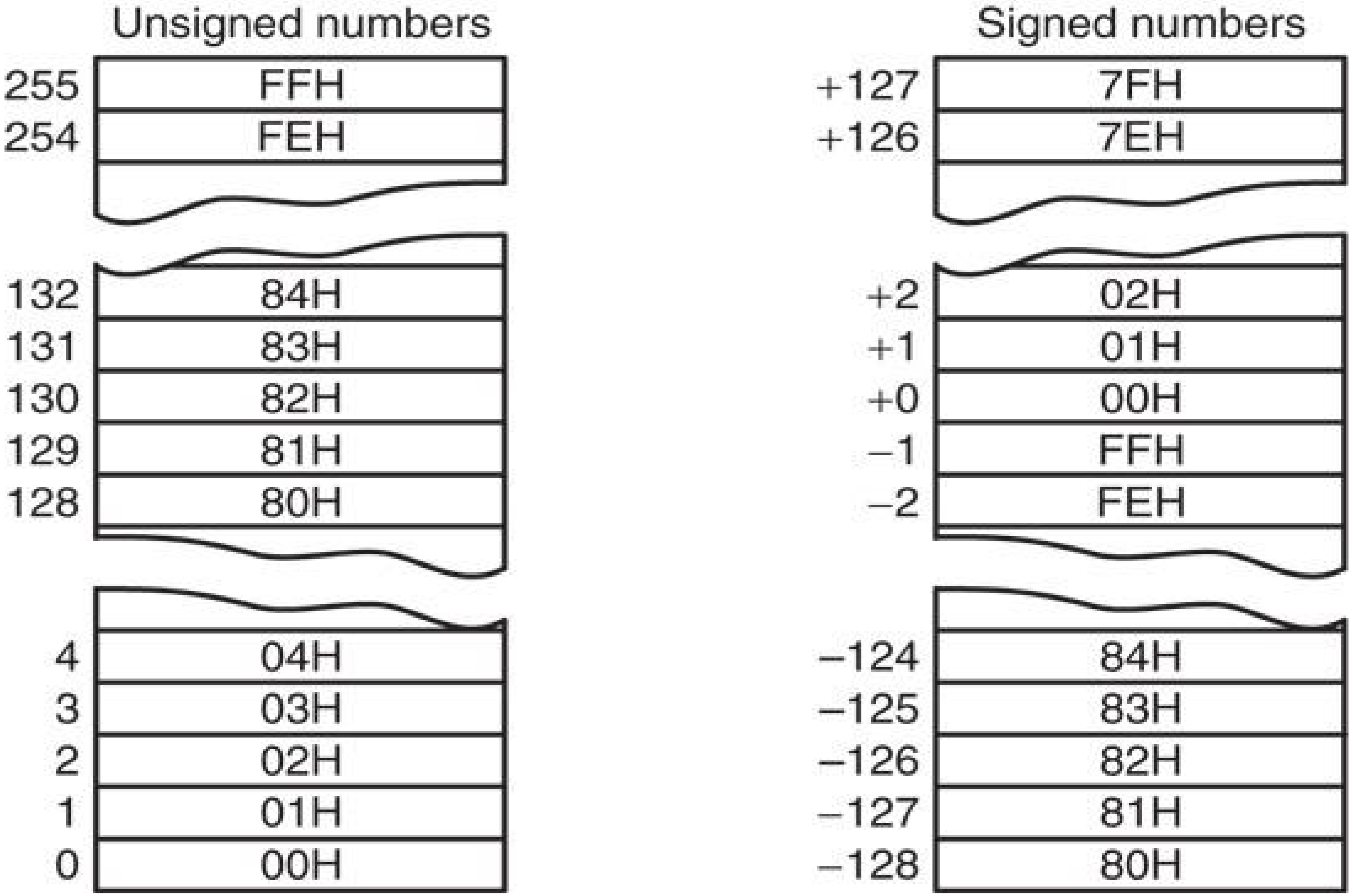
- In 80386 and above, conditional jumps are either short or near jumps (± 32K).
  - in 64-bit mode of the Pentium 4, the near jump distance is ± 2G for the conditional jumps
- Allows a conditional jump to any location within the current code segment.
- Conditional jump instructions test flag bits:
  - sign (S), zero (Z), carry (C), parity (P), overflow (O)
- If the condition under test is true, a branch to the label associated with the jump instruction occurs.
  - if false, next sequential step in program executes
  - for example, a **JC** will jump if the carry bit is set
- Most conditional jump instructions are straightforward as they often test one flag bit.
  - although some test more than one

Conditional Jumps

- There are two sets of conditional jump instructions for magnitude comparisons.
  - because both signed and unsigned numbers are used in programming and the order of these numbers is different
- 16- and 32-bit numbers follow the same order as 8-bit numbers, except that they are larger.
- When signed numbers are compared, use the JG, JL, JGE, JLE, JE, and JNE instructions.
  - terms greater than and less than refer to signed numbers
- When unsigned numbers are compared, use the JA, JB, JAE, JBE, JE, and JNE instructions.
  - terms above and below refer to unsigned numbers
- Remaining conditional jumps test individual flag bits, such as overflow and parity.
  - notice that JE has an alternative opcode JZ

Conditional Jumps

- All instructions have alternates, but many aren't used in programming because they don't usually fit the condition under test.



This figure shows the order of both signed and unsigned 8-bit numbers.

Conditional Jump instructions

Assembly Language	*Tested Condition*	*Operation*
JA	Z = 0 and C = 0	Jump if above
JAE	C = 0	Jump if above or equal
JB	C = 1	Jump if below
JBE	Z = 1 or C = 1	Jump if below or equal
JC	C = 1	Jump if carry
JE or JZ	Z = 1	Jump if equal or jump if zero
JG	Z = 0 and S = 0	Jump if greater than
JGE	S = 0	Jump if greater than or equal
JL	S != O	Jump if less than
JLE	Z = 1 or S != O	Jump if less than or equal
JNC	C = 0	Jump if no carry
JNE or JNZ	Z = 0	Jump if not equal or jump if not zero
JNO	O = 0	Jump if no overflow
JNS	S = 0	Jump if no sign (positive)
JNP or JPO	P = 0	Jump if no parity or jump if parity odd
JO	O = 1	Jump if overflow
JP or JPE	P = 1	Jump if parity or jump if parity even
JS	S = 1	Jump if sign (negative)
JCXZ	CX = 0	Jump if CX is zero
JECXZ	ECX = 0	Jump if ECX equals zero
JRCXZ	RCX = 0	Jump if RCX equals zero (64-bit mode)

LOOP

- The 8086/88 microprocessor has three instructions specifically designed for implementing loop operations.
- These instructions can be used in place of certain conditional jump instructions and give the programmer a simpler way of writing loop sequences.
- A combination of a decrement CX and the JNZ conditional jump.
- The first instruction, loop (LOOP), works with respect to the contents of the CX register.
- CX must be preloaded with a count that represents the number of times the loop is to repeat.
- In 8086 - 80286 LOOP decrements CX.
  - if CX != 0, it jumps to the address indicated by the label
  - if CX becomes 0, the next sequential instruction executes

LOOP

- Whenever LOOP is executed, the contents of CX are first decremented by one and then checked to determine if they are equal to zero.
  - If equal to zero, the loop is complete and the instruction following LOOP is executed; otherwise, control is returned to the instruction at the label specified in the loop instruction.
- In this way, we see that LOOP is a single instruction that functions the same as a decrement CX instruction followed by a JNZ instruction.
- In 16-bit instruction mode, LOOP uses CX; in the 32-bit mode, LOOP uses ECX.
  - default is changed by the LOOPW (using CX) and LOOPD (using ECX) instructions 80386 - Core2
- In 64-bit mode, the loop counter is in RCX.
  - and is 64 bits wide

Conditional LOOPs

- LOOP instruction also has conditional forms: LOOPE and LOOPNE
- **LOOPE** (loop while equal) instruction jumps if CX != 0 while an equal condition exists.
  - will exit loop if the condition is not equal or the CX register decrements to 0
- **LOOPNE** (loop while not equal) jumps if CX != 0 while a not-equal condition exists.
  - will exit loop if the condition is equal or the CX register decrements to 0
- In 80386 - Core2 processors, conditional LOOP can use CX or ECX as the counter.
  - LOOPEW/LOOPED or LOOPNEW/LOOPNED override the instruction mode if needed
- Alternates exist for LOOPE and LOOPNE.
  - LOOPE same as LOOPZ
  - LOOPNE instruction is the same as LOOPNZ
- In most programs, only the LOOPE and LOOPNE apply.

CONTROLLING THE FLOW OF THE PROGRAM

- Easier to use assembly language statements .IF, .ELSE, .ELSEIF, and .ENDIF to control the flow of the program than to use the correct conditional jump statement.
- Control flow assembly language statements beginning with a period (available to MASM version 6.xx, and not to earlier versions)
- Other statements developed include .REPEAT – .UNTIL and .WHILE – .ENDW.
  - the dot commands do not function using the Visual C++ inline assembler
- Never use uppercase for assembly language commands with the inline assembler.
  - some of them are reserved by C++ and will cause problems

PROCEDURES

- A procedure is a group of instructions that usually performs one task.
  - subroutine, method, or function is an important part of any system’ s architecture
- A procedure is a reusable section of the software stored in memory once, used as often as necessary.
  - saves memory space and makes it easier to develop software
- Disadvantage of procedure is time it takes the computer to link to, and return from it.
  - CALL links to the procedure; the RET (return) instruction returns from the procedure
- CALL pushes the address of the instruction following the CALL (return address) on the stack.
  - the stack stores the return address when a procedure is called during a program
- RET instruction removes an address from the stack so the program returns to the instruction following the CALL.

PROCEDURES

- A procedure begins with the PROC directive and ends with the ENDP directive.
  - each directive appears with the procedure name
- PROC is followed by the type of procedure:
  - NEAR or FAR
- In MASM version 6.x, the NEAR or FAR type can be followed by the USES statement.
  - USES allows any number of registers to be automatically pushed to the stack and popped from the stack within the procedure
- Procedures that are to be used by all software (global) should be written as far procedures.
- Procedures that are used by a given task (local) are normally defined as near procedures.
  - Most procedures are near procedures.

CALL

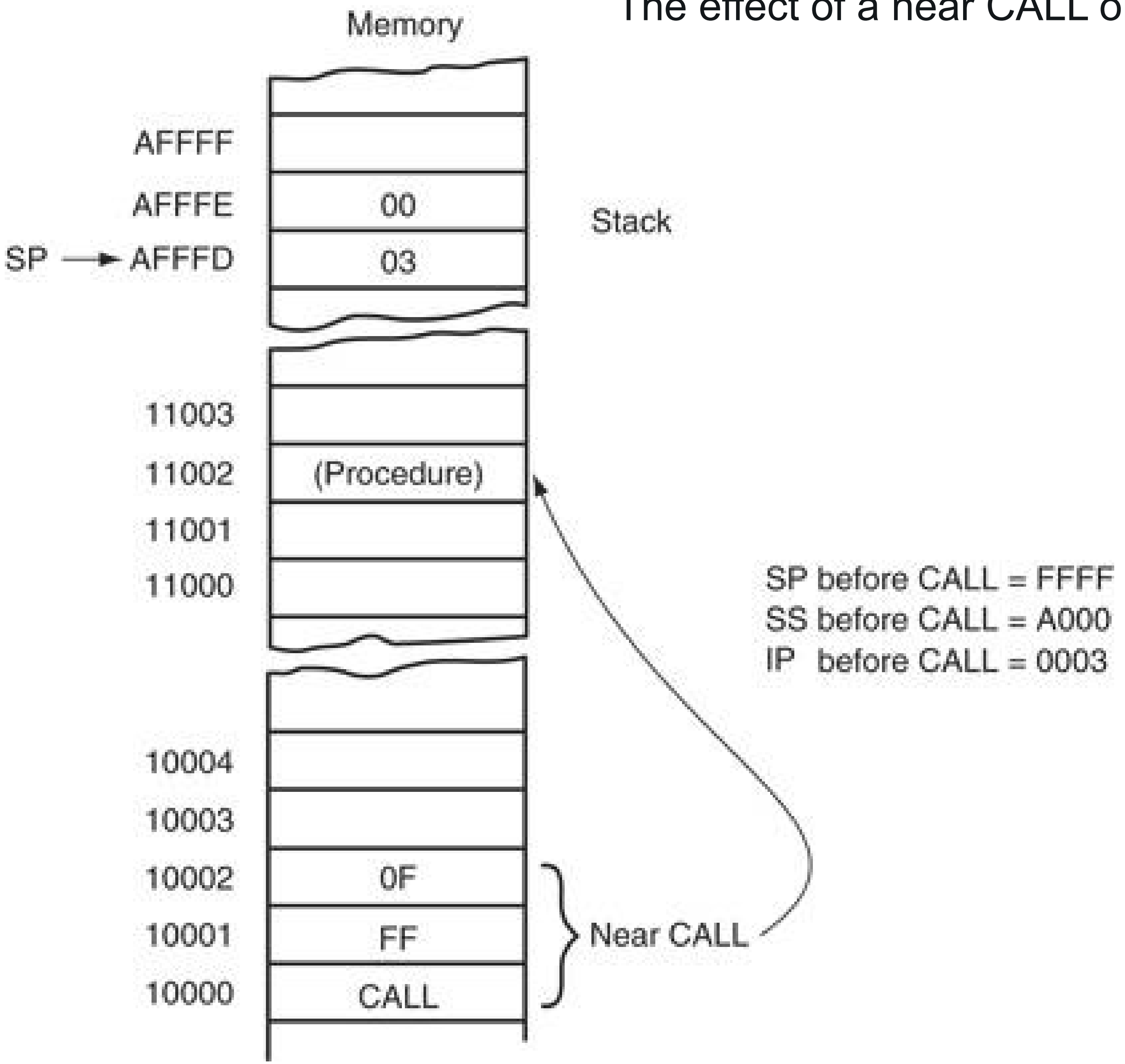
- Transfers the flow of the program to the procedure.
- CALL instruction differs from the jump instruction because a CALL saves a return address on the stack.
- The return address returns control to the instruction that immediately follows the CALL in a program when a RET instruction executes.
- Two types:
  - Near and
  - Far CALL

Near CALL

- 3 bytes long.
  - the first byte contains the opcode; the second and third bytes contain the displacement
- When the near CALL executes, it first pushes the offset address of the next instruction onto the stack.
  - offset address of the next instruction appears in the instruction pointer (IP or EIP)
- It then adds displacement from bytes 2 & 3 to the IP to transfer control to the procedure.
- Why save the IP or EIP on the stack?
  - the instruction pointer always points to the next instruction in the program
- For the CALL instruction, the contents of IP/EIP are pushed onto the stack.
  - program control passes to the instruction following the CALL after a procedure ends
- The following figure shows the return address (IP) stored on the stack and the call to the procedure.

Near CALL

The effect of a near CALL on the stack and the instruction pointer.

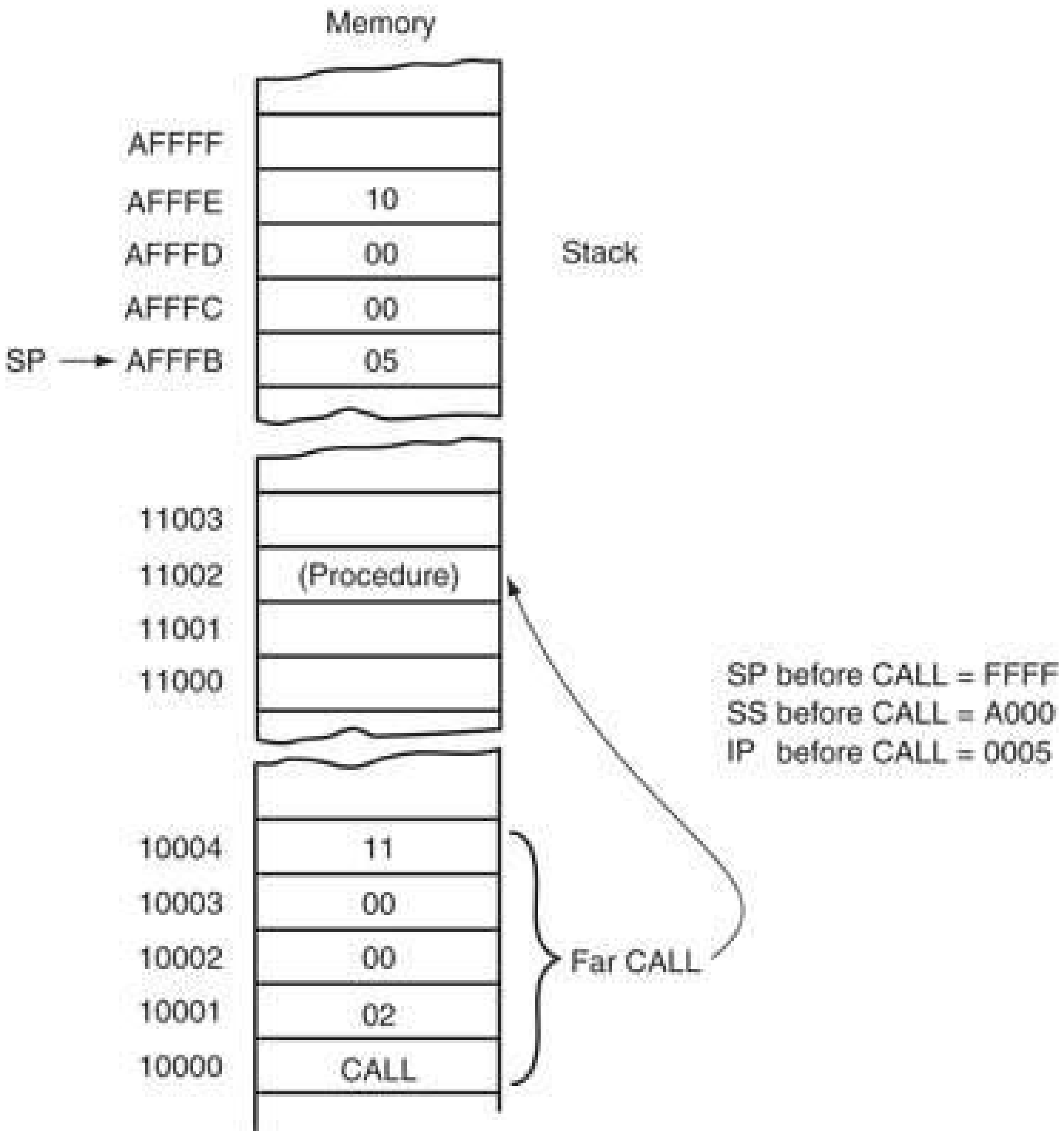


Far CALL

- 5-byte instruction contains an opcode followed by the next value for the IP and CS registers.
  - bytes 2 and 3 contain new contents of the IP
  - bytes 4 and 5 contain the new contents for CS
- Far CALL places the contents of both IP and CS on the stack before jumping to the address indicated by bytes 2 through 5.
- This allows far CALL to call a procedure located anywhere in the memory and return from that procedure.
  - contents of IP and CS are pushed onto the stack
- The program branches to the procedure.
  - A variant of far call exists as CALLF, but should be avoided in favor of defining the type of call instruction with the PROC statement

Far CALL

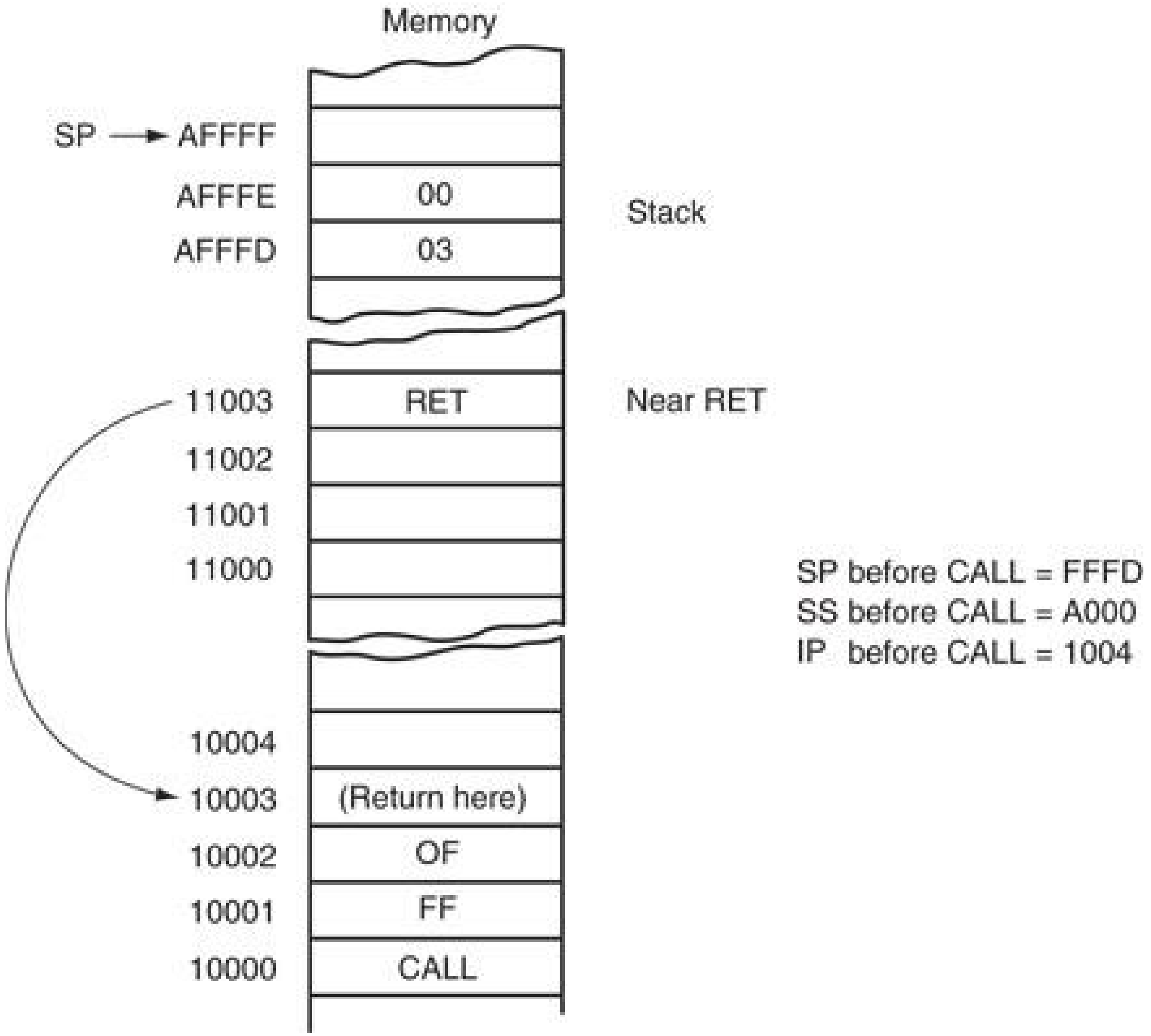
- ❑ the far return instruction retrieves an 8-byte return address from the stack and places it into RIP
- ❑ The following figure shows how far CALL calls a far procedure.
  - The effect of a far CALL instruction.



RET

- Removes a 16-bit number (near return) from the stack placing it in IP, or removes a 32-bit number (far return) and places it in IP & CS.
- The near and far return instructions are both defined in the procedure's PROC directive, which automatically selects the proper return instruction.
  - With the 80386 through the Pentium 4 processors operating in the protected mode, the far return removes 6 bytes from the stack.
  - The first 4 bytes contain the new value for EIP and the last 2 contain the new value for CS.
  - In the 80386 and above, a pro- tected mode near return removes 4 bytes from the stack and places them into EIP.
- When IP/EIP or IP/EIP and CS are changed, the address of the next instruction is at a new memory location.
- This new location is the address of the instruction that immediately follows the most recent CALL to a procedure.

RET



The effect of a near return instruction on the stack and instruction pointer.

THANKS !

Any questions?

You can find me at

getkennyo@gmail.com