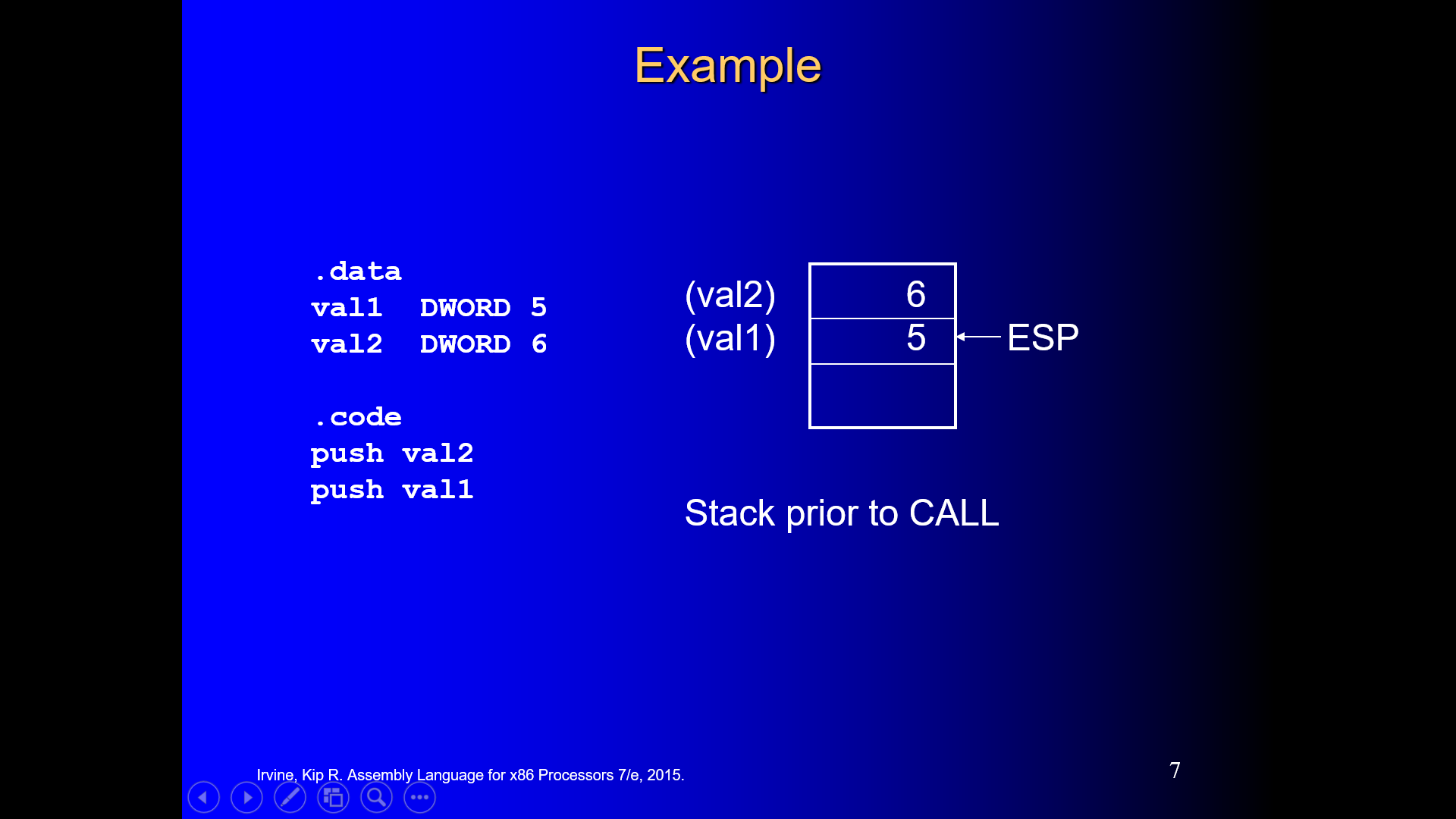
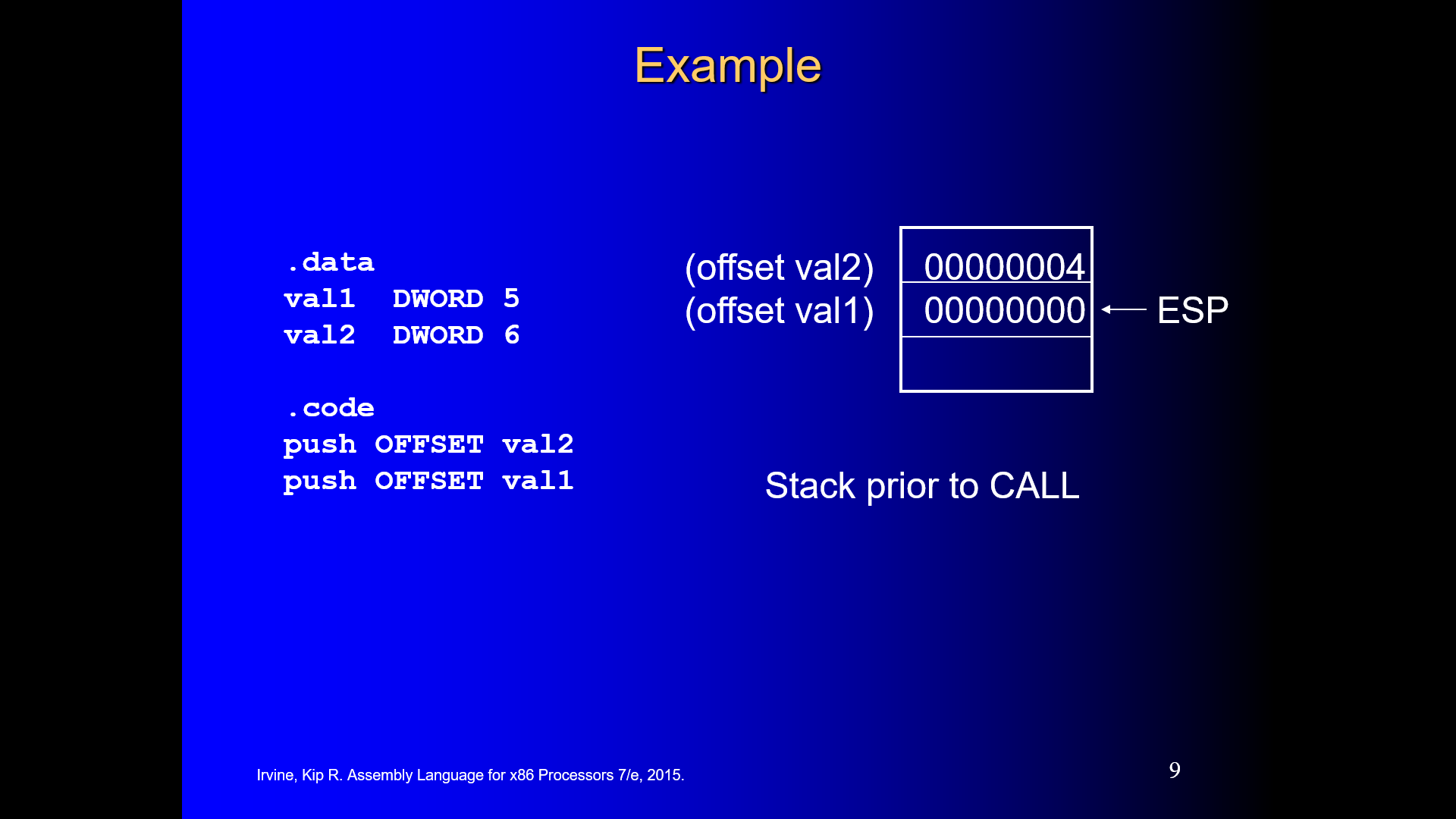
Please note that the final exam is comprehensive. But, there will be more emphasis on the material covered after mid-semester. Below is the review for the final exam that only includes the material covered after the second exam. The full review for the Final Exam also includes the review for Exam 1 and the review for Exam 2.

Advanced Procedures

1. Stack Frame
   1. Also known as an *activation record*
   2. Area of the stack set aside for a procedure's return address, passed parameters, saved registers, and local variables
   3. Created by the following steps:
   4. Calling program pushes arguments on the stack and calls the procedure.
   5. The called procedure pushes EBP on the stack, and sets EBP to ESP.
   6. If local variables are needed, a constant is subtracted from ESP to make room on the stack.
2. Pass by value/pass by reference
   1. Value
      1. Push argument values on stack
      2. (Use only 32-bit values in protected mode to keep the stack aligned)
      3. Call the called-procedure
      4. Accept a return value in EAX, if any
      5. Remove arguments from the stack if the called-procedure did not remove them



* 1. Reference
     1. Push the offsets of arguments on the stack
     2. Call the procedure
     3. Accept a return value in EAX, if any
     4. Remove arguments from the stack if the called procedure did not remove them



1. Ret and ret n
   1. *Return from subroutine*
   2. Pops stack into the instruction pointer (EIP or IP). Control transfers to the target address.
   3. Syntax:
   4. **RET**
   5. **RET*****n***
   6. Optional operand *n* causes *n* bytes to be added to the stack pointer after EIP (or IP) is assigned a value.
2. USES

**MySub1 PROC USES ecx edx**

**ret**

**MySub1 ENDP**

USES operator generates code to save and restore registers:

**MySub1 PROC**

**push ecx**

**push edx**

**pop edx**

**pop ecx**

**ret**

1. LOCAL
   1. The LOCAL directive declares a list of local variables
   2. immediately follows the PROC directive
   3. each variable is assigned a type
   4. Syntax:

**LOCAL *varlist***

Example:

**MySub PROC**

**LOCAL var1:BYTE, var2:WORD, var3:SDWORD**

**BubbleSort PROC**

**LOCAL temp:DWORD, SwapFlag:BYTE**

**. . .**

**ret**

**BubbleSort ENDP**

**MASM generates the following code:**

**BubbleSort PROC**

**push ebp**

**mov ebp,esp**

**add esp,0FFFFFFF8h ; add -8 to ESP**

**. . .**

**mov esp,ebp**

**pop ebp**

**ret**

**BubbleSort ENDP**

1. LEA vs Offset
   1. LEA returns offsets of direct and indirect operands
   2. OFFSET operator only returns constant offsets
   3. LEA required when obtaining offsets of stack parameters & local variables
   4. Example

**CopyString PROC,**

**count:DWORD**

**LOCAL temp[20]:BYTE**

**mov edi,OFFSET count ; invalid operand**

**mov esi,OFFSET temp ; invalid operand**

**lea edi,count ; ok**

**lea esi,temp ; ok**

Suppose you have a Local variable at [ebp-8]

And you need the address of that local variable in ESI

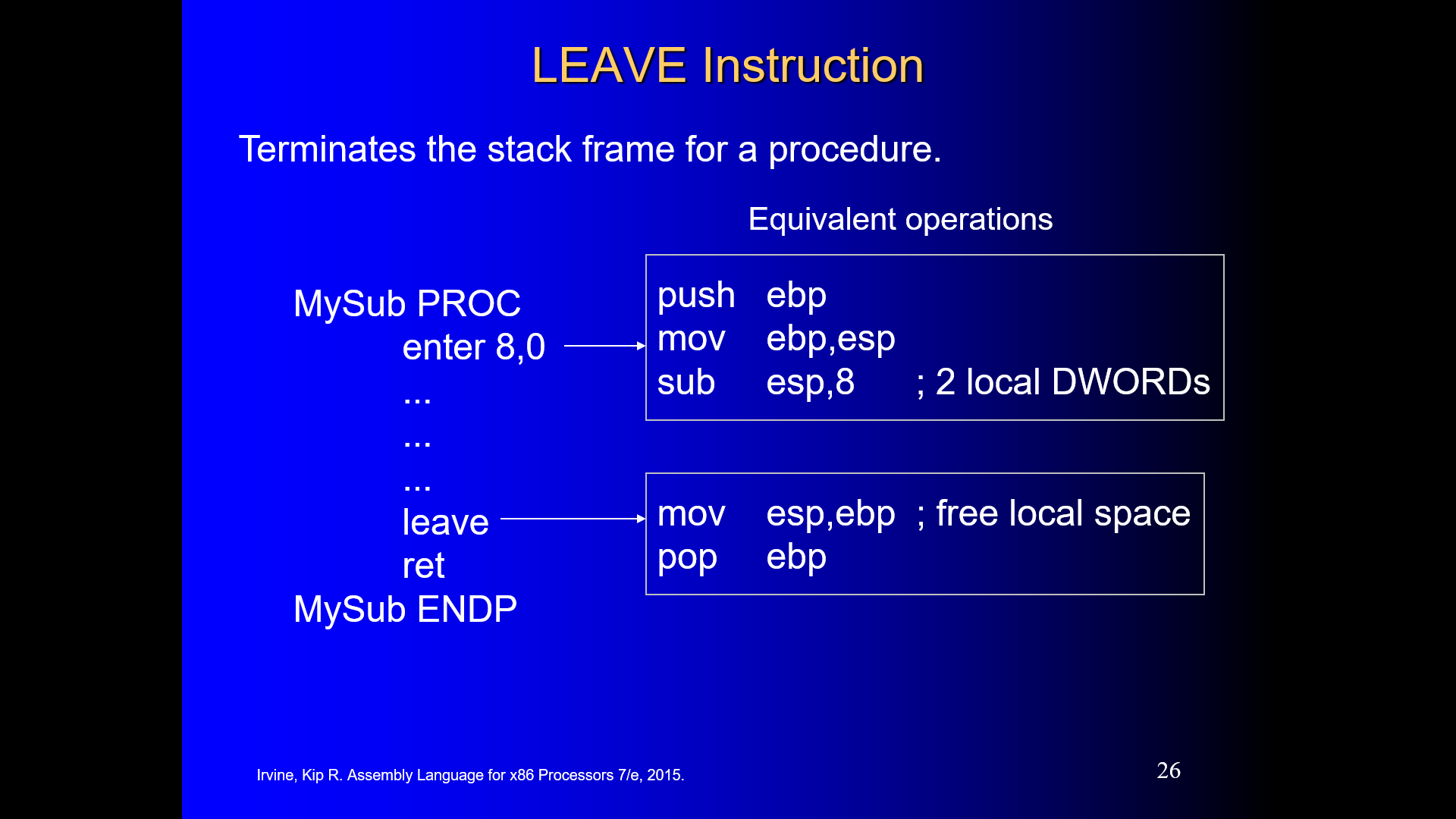
You cannot use this:

**mov esi, OFFSET [ebp-8] ; error**

Use this instead:

**lea esi,[ebp-8]**

1. Enter and Leave implementation
   1. Enter
      1. ENTER instruction creates stack frame for a called procedure
      2. pushes EBP on the stack
      3. sets EBP to the base of the stack frame
      4. reserves space for local variables
      5. Example:
      6. **MySub PROC**
      7. **enter 8,0**
      8. Equivalent to:
      9. **MySub PROC**
      10. **push ebp**
      11. **mov ebp,esp**
      12. **sub esp,8**
   2. Leave



1. Bubble Sort
   1. See Slides

Strings

1. Direction Flag
   1. The Direction flag controls the incrementing or decrementing of ESI and EDI.
   2. DF = clear (0): increment ESI and EDI
   3. DF = set (1): decrement ESI and EDI
   4. The Direction flag can be explicitly changed using the CLD and STD instructions:

**CLD ; clear Direction flag**

**STD ; set Direction flag**

1. MOVSB/W/D
   1. The MOVSB, MOVSW, and MOVSD instructions copy data from the memory location pointed to by ESI to the memory location pointed to by EDI.
   2. ESI and EDI are automatically incremented or decremented:
   3. MOVSB increments/decrements by 1
   4. MOVSW increments/decrements by 2
   5. MOVSD increments/decrements by 4

**.data**

**source DWORD 0FFFFFFFFh**

**target DWORD ?**

**.code**

**mov esi,OFFSET source**

**mov edi,OFFSET target**

**movsd**

* 1. REP (a repeat prefix) can be inserted just before MOVSB, MOVSW, or MOVSD.
  2. ECX controls the number of repetitions
  3. Example: Copy 20 doublewords from source to target

**.data**

**source DWORD 20 DUP(?)**

**target DWORD 20 DUP(?)**

**.code**

**cld ; direction = forward**

**mov ecx,LENGTHOF source ; set REP counter**

**mov esi,OFFSET source**

**mov edi,OFFSET target**

**rep movsd**

1. CMPSB/W/D
   1. The CMPSB, CMPSW, and CMPSD instructions each compare a memory operand pointed to by ESI to a memory operand pointed to by EDI.
   2. CMPSB compares bytes
   3. CMPSW compares words
   4. CMPSD compares doublewords
   5. Repeat prefix often used
   6. REPE (REPZ)
   7. REPNE (REPNZ)
2. SCASB/W/D
   1. The SCASB, SCASW, and SCASD instructions compare a value in AL/AX/EAX to a byte, word, or doubleword, respectively, addressed by EDI.
   2. Useful types of searches:
   3. Search for a specific element in a long string or array.
   4. Search for the first element that does not match a given value.

**.data**

**alpha BYTE "ABCDEFGH",0**

**.code**

**mov edi,OFFSET alpha**

**mov al,'F' ; search for 'F'**

**mov ecx,LENGTHOF alpha**

**cld**

**repne scasb ; repeat while not equal**

**jnz quit**

**dec edi ; EDI points to 'F'**

1. STOSB/W/D
   1. The STOSB, STOSW, and STOSD instructions store the contents of AL/AX/EAX, respectively, in memory at the offset pointed to by EDI.
   2. Example: fill an array with 0FFh

**.data**

**Count = 100**

**string1 BYTE Count DUP(?)**

**.code**

**mov al,0FFh ; value to be stored**

**mov edi,OFFSET string1 ; ES:DI points to target**

**mov ecx,Count ; character count**

**cld ; direction = forward**

**rep stosb ; fill with contents of AL**

1. LODSB
   1. LODSB, LODSW, and LODSD load a byte or word from memory at ESI into AL/AX/EAX, respectively.
   2. Example:

**.data**

**array BYTE 1,2,3,4,5,6,7,8,9**

**.code**

**mov esi,OFFSET array**

**mov ecx,LENGTHOF array**

**cld**

**L1: lodsb ; load byte into AL**

**or al,30h ; convert to ASCII**

**call WriteChar ; display it**

**loop L1**

1. Base IDX
   1. A base-index operand adds the values of two registers (called base and index), producing an effective address. Any two 32-bit general-purpose registers may be used. *(Note: esp is not a general-purpose register)*
   2. Base-index operands are great for accessing arrays of structures. (A structure groups together data under a single name. )
2. Base IDX, DISPL
   1. A base-index-displacement operand adds base and index registers to a constant, producing an effective address. Any two 32-bit general-purpose register can be used.
   2. Common formats:

[ *base* + *index* + *displacement* ]

*displacement* [ *base* + *index* ]

Interrupts

1. Reset
   1. Not maskable
2. Trap
   1. Stops the code to allow for line by line execution during debugging.
3. INT21
   1. Software Interrupt
   2. Handles reading and writing to the screen
4. NMI
   1. Non Maskable Interrupt
5. INTR
   1. Maskable
   2. The INTR is a maskable interrupt because the microprocessor will be interrupted only if interrupts are enabled using set interrupt flag instruction. It should not be enabled using clear interrupt Flag instruction.
   3. The INTR interrupt is activated by an I/O port. If the interrupt is enabled and NMI is disabled, then the microprocessor first completes the current execution and sends ‘0’ on INTA pin twice. The first ‘0’ means INTA informs the external device to get ready and during the second ‘0’ the microprocessor receives the 8 bit, say X, from the programmable interrupt controller.
6. NMI is a non-maskable interrupt and INTR is a maskable interrupt having lower priority.
7. INTA
   1. One more interrupt pin associated is INTA called interrupt acknowledge.
8. Vector-table



The CPU processes the INT instruction using the interrupt vector table, which, as we’ve mentioned,

is a table of addresses in the lowest 1024 bytes of memory. Each entry in this table is a

32-bit segment-offset address that points to an interrupt handler. The actual addresses in this

table vary from one machine to another. Figure 14-2 illustrates the steps taken by the CPU when

the INT instruction is invoked by a program:

• Step 1: The operand of the INT instruction is multiplied by 4 to locate the matching interrupt

vector table entry.

• Step 2: The CPU pushes the flags and a 32-bit segment/offset return address on the stack, disables

hardware interrupts, and executes a far call to the address stored at location (10h \* 4) in

the interrupt vector table (F000:F065).

• Step 3: The interrupt handler at F000:F065 executes until it reaches an IRET (interrupt return)

instruction.

• Step 4: The IRET instruction pops the flags and the return address off the stack, causing the

processor to resume execution immediately following the INT 10h instruction in the calling

program.

1. Software interrupts vs hardware interrupts

Hardware interrupt

Caused by any peripheral device by sending a signal through a specified pin to the microprocessor.

One more interrupt pin associated is INTA called interrupt acknowledge.

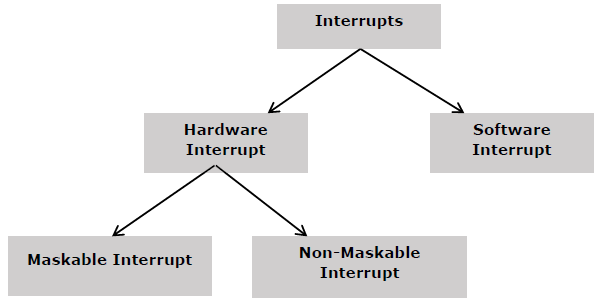
Software Interrupts

Some instructions are inserted at the desired position into the program to create interrupts. These interrupt instructions can be used to test the working of various interrupt handlers. It includes −

INTO - Interrupt on overflow instruction

INT- Interrupt instruction with type number

INT 3-Break Point Interrupt Instruction



Computer Organization

1. Instruction Execution Stages
2. Caches
3. Memory Read Cycle