Dr. Zbigniew Kalbarczyk

University of Illinois at Urbana- Champaign

- Memory access example
- Flags
- Logic instructions
- Addition and subtraction
- Conditional branches
- Basic conditional clauses

Memory Example -Declarations

```
SEGMENT code

myvar1  DW      1234h   ; define word variable
                        ; value=1234h
myvar2  DW      1234    ; define word variable
                        ; value=1234d=4D2h
myvar3  RESW            ; define word variable
                        ; value not specified
myvar4  DW      0ABCDh  ; define word variable
                        ; value = ABCDh
                        ; note the leading zero
ece390msg       DB “ECE390 is great!”
                        ; define strings as series of bytes
                        ; this is not one byte but 16!
..start                 ; denotes execution starting point
```


What it looks like to the computer

1F	??	??	1E	
1D	??	??	1C	
1B	??	??	1A	
19	??	??	18	← CS:IP
17	21 = ‘!’	74 = ‘t’	16	
15	61 = ‘a’	65 = ‘e’	14	
13	72 = ‘r’	67 = ‘g’	12	
11	‘ ’	73 = ‘s’	10	
F	69 = ‘i‘	20 = ‘ ’	E	
D	31 = ‘0’	39 = ‘9‘	C	
B	32 = ‘3’	45 = ‘E’	A	
9	43 = ‘C’	45 = ‘E’	8	← ece291msg
7	AB	CD	6	← myvar4
5	00	00	4	← myvar3
3	04	D2	2	← myvar2
1	12	34	0	← myvar1

CS – Code segment begins

Memory Example – MOVing (1)

```
mov ax, cs    ; make data seg same as code seg
mov ds, ax

mov ax, [myvar2]     ; move value of myvar2 to ax
                    ; same as mov ax, [2]
                    ; ax now contains 04D2h
mov si, myvar2       ; si now points to myvar2
                    ; it contains 0002h

mov ax, [si]  ; move value pointed to by si to ax
              ; ax again contains 04D2h
              ; this was an indirect reference
```


Memory Example – MOVing(2)

```
mov bx, ece390msg    ; bx now points to the beginning of
                     ; out "ECE390 is great" string.
                     ; bx points to the first 'E'
                     ; bx contains 0008h

dec byte [bx + 1]    ; new instruction!  arg = arg - 1
                     ; Just changed the C in "ECE" to a B

mov si, 1            ; now lets use SI as an index

inc byte [ece390msg + si]  ; new instruction! arg = arg + 1
                     ; evaluates to inc [8 + 1] = inc [9]
                     ; B becomes a C again!
```

Memory Example – MOVing (3)

```
; Memory can be addressed using four registers only!!!
;  SI -> offsets from DS
;  DI -> offsets from DS
;  BX -> offsets from DS
;  BP -> offsets from SS

mov ax, [bx]    ; ax = word in memory pointed to by bx
mov al, [bx]    ; al = byte in memory pointed to by bx
mov ah, [si]    ; ah = byte in memory pointed to by si
mov cx, [di]    ; cx = word in memory pointed to by di
mov ax, [bp]    ; ax = [SS:BP] Stack Operation!

; In addition, BX + SI, BX + DI, BP + SI, and BP + DI are allowed
mov ax, [bx + si]       ; ax = [ds:bx+si]
mov ch, [bp + di]       ; ch = [ss:bp+di]
```

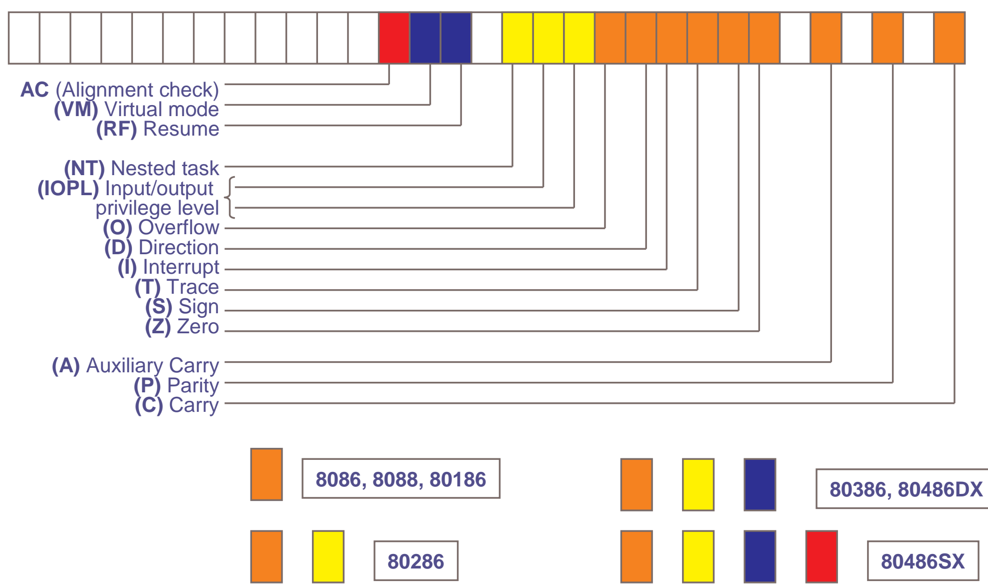

Memory Example – MOVing (4)

```
; Furthermore, a fixed 8- or 16-bit displacement can be
  added

mov ax, [23h]               ; ax = word at [DS:23h]
mov ah, [bx + 5]            ; ah = byte at [DS:bx+5]
mov ax, [bx + si + 107]     ; ax = word at [DS:bx+si+107]
mov ax, [bp + di + 42]      ; ax = word at [SS:bp+di+42]

; Remember that memory to memory moves are illegal!
mov [bx], [si]        ; BAD BAD BAD
mov [di], [si]        ; BAD BAD BAD

; Special case with stack operations
pop word [myvar]      ; moves data from top of stack to
                      ; memory variable myvar
```

The flag register

- Flags indicate the condition of the CPU as well as its operation
- Flag bits change after many arithmetic and logic instructions complete execution
- Example flags:
  - C(carry) indicates carry after addition or borrow after subtraction in most significant bit
  - O(overflow) is a condition that can occur when signed numbers are added or subtracted
  - Z(zero) indicates that the result of an operation is zero

The flag register



Logic instructions

- Logic instructions operate on a bit-by-bit basis:

Assembly		C
NOT	A	A = ~A
AND	A, B	A &= B
OR	A, B	A |= B
XOR	A, B	A ^= B

- Except for NOT these instructions affect the flags as follows:
  - clear the carry (C)
  - clear the overflow (0)
  - set the zero flag (Z) if the result is zero, or clear it otherwise
  - copy the high order bit of the result into the sign flag (S)
  - set the parity bit (P) according to the parity (number of 1's) in the result
  - scrambles the auxiliary carry flag (A)
- The NOT instruction does not affect any flags

Logic examples

AL	1100 1010
NOT AL	
AL	0011 0101

AL	0011 0101
BL	0110 1101
AND AL, BL	
AL	0010 0101

AL	0011 0101
BL	0000 1111
AND AL, BL	
AL	0000 0101

AL	0011 0101
BL	0110 1101
OR AL, BL	
AL	0111 1101

AL	0011 0101
BL	0000 1111
OR AL, BL	
AL	0011 1111

AL	0011 0101
BL	0110 1101
XOR AL, BL	
AL	0101 1000

AND/OR applications

- AND and OR instructions are often used to mask out data
  - a mask value is used to force certain bits to zero or one within some other value
  - a mask typically affects certain bits and leaves other bits unaffected
    - AND forces selected bits to zero   AND   CL, 0Fh
    - OR forces selected bits to one   OR   CL, 0Fh

Logic instructions

- TEST is a non-destructive AND
  - logically ands two operands just as AND would
  - sets flags the same as AND
  - doesn't modify contents of destination
  - TEST AL, 1 sets flags same as AND AL, 1 but doesn't change AL
  - useful when setting up conditional jumps

Addition

- Syntax: ADD A, B
  - A and B can be register or memory (not both memory)
  - B can also be immediate data
- Function: A = A + B
- Examples:
  - ADD AX, BX ; register addition
  - ADD AX, 5h ; immediate addition
  - ADD [BX], AX ; addition to memory location
  - ADD AX, [BX] ; memory location added to register
  - ADD DI, MYVAR ; memory offset added to register
- Flags modified: S, C, A, Z, P, O

Increment

- Syntax: INC A
  - A can be a register or memory location
- Function: A = A + 1
- Examples
  - INC AX ; AX = AX + 1
  - INC byte [BX] ; memory location increased by 1
  - INC word [MYVAR] ; memory location increased by 1
- Flags modified: S, A, Z, P, O
  - notice INC does **not** affect the carry flag

Add with carry

- Syntax: ADC A, B
  - A and B can be register or memory (not both memory)
  - B can also be immediate data
- Function: A = A + B + carry flag
  - used mainly to add numbers that are wider than 16 bits (when in 16-bit mode) or wider than 32 bits (when in 32-bit instruction mode)
- Examples:
  - ```
    add  ax, cx        ; this produces the 32 bit sum of
    adc  bx, dx        ; bx:ax + dx:cx
    ```
- Flags modified: S, C, A, Z, P, O

Subtraction

- Syntax: SUB A, B
  - A and B can be register or memory (not both memory)
  - B can also be immediate data
- Function: A = A – B
  - carry flag interpreted as borrow request in H.O. bit
- Examples:
  - ```
    mov  ch, 22h
    sub  ch, 34h
    ```
- Flags modified: S, C, A, Z, P, O

Result is –12h (1110 1110)
Flags change:
Z = 0 (result not zero)
C = 1 (borrow in H.O. bit)
S = 1 (result negative)
P = 0 (even parity)
O = 0 (no overflow)

Decrement

- Syntax: DEC A
  - A can be a register or memory location
- Function: A = A - 1
- Examples
  - DEC AX ; AX = AX - 1
  - DEC byte [BX] ; memory location decreased by 1
  - DEC word [MYVAR] ; memory location decreased by 1
- Flags modified: S, A, Z, P, O
  - notice DEC does **not** affect the carry flag

Subtract with borrow

- Syntax: SBB A, B
  - A and B can be register or memory (not both memory)
  - B can also be immediate data
- Function: A = A - B - carry flag
  - used mainly to subtract numbers that are wider than 16 bits (when in 16-bit mode) or wider than 32 bits (when in 32-bit instruction mode)
- Examples:
  - sub ax, di ; this produces the 32 bit difference of
    sbb bx, si ; bx:ax - si:di à bx:ax
- Flags modified: S, C, A, Z, P, O

Numerical comparison

- **CMP A, B** compares A to B
  - a subtraction that *only changes the flag bits*
  - useful for checking the entire contents of a register or a memory location against another value
  - usually followed by a conditional jump instruction

    CMP AL, 10h ;compare with 10h (contents of AL do not change)

    JAE foo ;if 10h or above then jump to memory location foo
- **SUB** A,B calculates difference A - B
  - saves result to A and sets flags
- CMP, SUB instructions always set flags the same way regardless if you are interpreting the operands as signed or unsigned.
- There are different flags you look at depending on what kind of data you're comparing

The carry flag

- **Carry**
  - indicates a carry in the H.O. bit after addition or a borrow in the H.O. bit after subtraction
- Carry is set when an unsigned number goes out of range (this is an unsigned arithmetic overflow)
- 8-bit example
  - FFh + 1h = 00h with carry (carry = 1; overflow = 0)
  - 02h – 03h = FFh, with carry (carry = 1; overflow = 0)

The overflow flag

- **Overflow**
  - condition that occurs when signed numbers go out of range
  - For 8-bit, that range is from -128 to + 127 decimal
- Overflow is set when ***signed*** number goes out of range (denotes a signed arithmetic overflow)
- 8-bit overflow example
  - 7Fh + 1h = 80h, wanted 127 + 1 = 128 (got –128) (carry = 0; overflow = 1)
  - 80h – 1h = 7F, we wanted -128 -1 = -129 (got +127) (carry = 0; overflow = 1)
  - FFh + FFh = FEh {(-1) + (-1)} = -2; (carry = 1; overflow = 0)

Overflows and carries examples

```
mov ax,0fffeh        ; 65534 interpreted as unsigned
add ax,3             ; C = 1, O = 0 --- Unsigned Overflow Condition

mov ax,0FFFEh        ; -2 interpreted as signed
add ax,3             ; C = 1, O = 0 --- Okay as signed

mov bx,07FFFh        ; 32767 interpreted as signed
add bx,1             ; C = 0, O = 1 ---   Signed Overflow Condition

mov bx,07FFFh        ; 32767 interpreted as unsigned
add bx,1             ; C = 0, O = 1 ---   Okay as unsigned

mov ax,07h           ; 7 interpreted as either signed or unsigned
add ax,03h           ; C = 0, O = 0 ---   Okay as either
```

Flag Setting

FLAG	Name	Description	Notes
ZF	Zero	1:ZR:Zero 0:NZ: Non-zero	1 indicates that the result was zero
CF	Carry	1:CY` 0:NC	**Unsigned Math** Needed a carry or borrow
OF	Overflow	1:OV 0:NV	**Signed Math** Also (+) or (-) to be represented as a valid two's complement number
SF	Sign Flag	1:NG: - 0:PL: +	MSB of result

Conditional branches

- Using flags we can perform conditional jumping, i.e., transfer program execution to some different place within the program
- if condition is true
  - jump back or forward in a code to the location specified
  - instruction pointer (IP) gets updated to point to the instruction to which the program will jump
- if condition is false
  - continue execution at the following instruction
  - IP gets incremented as usual
- Conditional jumps test: sign (S), zero (Z), carry (C), parity (P), and overflow (O)
  - An FFh is **above** 00h in the set of unsigned numbers
  - An FFh (-1) is **less** than 00h for signed numbers

Numerical comparison

CMP A, B

Unsigned operands

Z: equality/inequality

C: A < B (C = 1)

A > B (C = 0)

S: No meaning

O: No meaning

Signed operands

Z: equality/inequality

C: No meaning

S and O together:

if (S xor O = 1) then
A < B

else

A >= B

Signed comparisons

- Consider CMP AX,BX computed by the CPU
  - The Sign bit (S) will be set if the result of AX-BX has a 1 at the most significant bit of the result
  - The Overflow flag (O) will be set if the result of AX-BX produced a number that was out of range [-32768, 32767].
- **Case 1:** Sign = 1 and Overflow = 0
  - AX-BX is negative, and
  - No overflow so result was within range and is valid
  - Therefore, AX must be **less** than BX

Signed comparisons

- **Case 2:** Sign = 0 and Overflow = 1
  - AX-BX looks positive, but
  - Overflow is set, so result is out of range so the sign bit is incorrect
  - Therefore AX must still be **less** than BX
- **Case 3:** Sign = 0 and Overflow = 0
  - AX-BX is positive, and
  - There was no overflow, meaning sign bit is correct
  - So AX must be **greater** than BX
- **Case 4:** Sign = 1 and Overflow = 1
  - AX-BX looks negative, but
  - There was an overflow so the sign is incorrect
  - AX is actually **greater** than BX

Signed comparisons

- Difference in JS (jump on sign) and JL (jump less than)
  - JS looks at the sign bit (S) of the last compare (or subtraction) only. If S = 1 then jump.
  - JL looks at S *XOR* O of the last compare
    - REGARDLESS of the value AX-BX, i.e., even if AX-BX causes overflow, the JL will be correctly executed

Signed comparisons

- JL is true if the condition: S xor O is met
- JL is true for two conditions:
- S=1, O=0: (AX-BX) was negative and (AX-BX) did not overflow
  - Example (8-bit): CMP -5, 2

    (-5) - (2) = (-7)

    Result (-7) has the sign bit set

    Thus (-5) is less than (2).

Signed comparisons

- S=0, O=1: **Overflow!, Sign bit of the result is wrong!**
- Consider the following case:
  - AX is a large negative number (-)
  - BX is a positive number (+).
  - The subtraction of (-) and (+) is the same as the addition of (-) and (-)
  - The result causes negative overflow, and thus cannot be represented as a signed integer correctly (O=1).
  - The result of AX-BX appears positive (S=0).
- Example (8-bit): (-128) - (1) = (+127)
  - Result (+127) overflowed. Answer should have been -129.
  - Result appears positive, but overflow occurred
  - Thus (-128) is less than (1), the condition is **TRUE for JL**

Conditional jumps

- Terminology used to differentiate between jump instructions that use the carry flag and the overflow flag
  - Above/Below unsigned compare
  - Greater/Less signed (+/-) compare
- Names of jump instructions
  - J => Jump
  - N => Not
  - A/B G/L => Above/Below Greater/Less
  - E => Equal

Conditional jumps

Command	Description	Condition
JA=JNBE	Jump if above Jump if not below or equal	C=0 & Z=0
JBE=JNA	Jump if below or equal	C=1 | Z=1
JAE=JNB=JNC	Jump if above or equal Jump if not below Jump if no carry	C=0
JB=JNA=JC	Jump if below Jump if carry	C=1
JE=JZ	Jump if equal Jump if Zero	Z=1
JNE=JNZ	Jump if not equal Jump if not zero	Z=0
JS	Jump Sign (MSB=1)	S=1

Conditional jumps

Command	Description	Condition
JNS	Jump Not Sign (MSB=0)	S=0
JO	Jump if overflow set	O=1
JNO	Jump if no overflow	O=0
JG=JNLE	Jump if greater	
	Jump if not less or equal	S=O & Z=0
JGE=JNL	Jump if greater or equal	S=O
	Jump if not less	
JL=JNGE	Jump if less	S^O
	Jump if not greater or equal	
JLE=JNG	Jump if less or equal	S^O | Z=1
	Jump if not greater	
JCXZ	Jump if register CX=zero	CX=0

Unconditional jumps

- Basically an instruction that causes execution to transfer to some point other than the next sequential instruction in your code.
- Essentially a “goto” statement.
- Assembly language couldn’t exist without unconditional jumps.

  JMP *label*

If…then…else clauses

```
‘ Visual Basic Example
if ax > bx then
        ‘true instructions
else
        ‘false instructions
end if
```

```
      cmp ax, bx
      ja .true
             ;false instructions
        jmp .done
.true
             ;true instructions
.done
```


Switch…case clauses

```
/* C code example */
switch (choice) {

      case ‘a’:
                /* a instructions */
                break;

      case ‘b’:
                /* b instructions */
                break;

      default:
                /* default instructions */
}
```

```
      cmp   al, ‘a’
      jne .b
                 ; a instructions
      jmp .done

.b
      cmp al, ‘b’
      jne .default
                 ; b instructions
      jmp .done

.default
      ; default instructions
 .done
```

Do…while clause

```
do {
    // instructions
while (A=B);
```

```
.begin
    ; instructions
mov ax, [A]
cmp ax, [B]
je .begin
```


While…do clause

```
while (A=B) {
    // instructions
}
```

```
.begin
    mov ax, [A]
    cmp ax, [B]
    jne .done
    ; instructions
jmp .begin
```

Looping

- The LOOP instruction is a combination of the DEC and JNZ instructions
- LOOP decrements CX and if CX has not become zero jumps to the specified label
- If CX is zero, the instruction following the LOOP is executed

Loop example

```
ADDS
      mov   cx, 100           ;load count
      mov   si, BLOCK1
      mov   di, BLOCK2
.Again
      lodsw             ;get Block1 data
                        ; AX = [SI]; SI = SI + 2
      add   AX, [ES:DI] ;add Block2 data
      stosw             ;store in Block2
                        ; [DI] = AX; DI = DI + 2
      loop  .Again      ;repeat 100 times

      ret
```

LOOPNE

- The LOOPNE instruction is useful for controlling loops that stop on some condition or when the loop exceeds some number of iterations
- Consider **String1** that contains a sequence of characters that end with the byte containing zero
- we want to convert those characters to upper case and copy them to **String2**

```
…..
String1  DB  'This string contains lower case characters', 0
String2  times 128 DB 0
……..
```


Example (LOOPNE)

```
   mov si, String1            ;si points to beginning of String1
   lea di, String2            ;di points to beginning of String2
   mov cx, 127                ;Max 127 chars to String2
.StrLoop:
   lodsb              ;get char from String1; AL=[DS:SI]; SI= SI + 1
   cmp al, 'a'                ;see if lower case
   jb .NotLower               ;chars are unsigned
   cmp al, 'z'
   ja .NotLower
   and al, 5Fh                ;convert lower -> upper case
                              ;bit 6 must be 0
.NotLower:
   stosb                      ;[ES:DI] = AL; DI = DI + 1
   cmp al, 0                  ;see if zero terminator
   loopne .StrLoop            ;quit if AL or CX = 0
```