ICT 106 Fundamentals of Computer Systems
Topic 10

Basic Data Processing: Conditional Processing and Integer Arithmetic

Objectives
- To understand how HLL control constructs (selection and iteration) are implemented at the assembly level
- To understand how integer arithmetic (multiplication and division) is performed at the assembly level
- To understand the relationship between the arithmetic and conditional assembly instructions and the flags register
- To understand the miscellaneous assembly instructions LEA, LDS, LES, XLAT
- To understand the string instructions at assembly level

Writing control constructs (selection and iteration) in assembly language
- HLLs use high-level structures such as WHILE, IF-THEN-ELSE, SWITCH, … to control the flow of execution
- Algorithms are normally expressed in terms of these high-level structures
- Assembly languages (of processors) only provide conditional and unconditional jumps and loops. This is the case of Intel CPU, may also be called BRANCH in other processors.
- We need to convert the high-level control-flow structures into low-level ones.
- The words WHILE-WEND, DO-DOEND, IF-THEN-ELSE-ENDIF can be used as labels for jump instructions.
- Labels have to be unique, so for multiple WHILE statements, unique WHILE-WEND labels are needed.
- One way is to postfix each label with a 2 digit number. Eg, WHILEnn – WENDnn (WHILE01, WHILE02, WEND01, WEND02 etc.)
• **WHILE LOOP**

WHILEnn:

- ;test if loop should be performed. If test
- ;fails jump to WENDnn
- ;statements for body of loop
- ;change the test value - can be
- ;part of the loop-body.
  
  jmp WHILEnn

WENDnn:

---

**Example**

**High-Level code:**

```plaintext
while (op1 < op2)
  statements
endwhile
```

**Assembly Level code:**

```plaintext
WHILE00:
  cmp op1, op2 ; while (op1 < op2)
  jge WEND00 ; assume signed integers
               ; statements
  jmp WHILE00
WEND00: ; endwhile
```

---

**Example:** Write a while loop which reads a character with echo. Loop terminates when an “enter” key is pressed.

```plaintext
mov ah, 1 ; get a char and echo
int 21h ; DOS input function
WHILE01:
  cmp al, 13 ; while char <> enterkey
  je WEND01 ; jump out if equal
  mov ah, 1 ; get another char and
  int 21h ; echo
  jmp WHILE01 ; repeat
WEND01: ; endwhile
```
DO-WHILE:

- Same idea as above but test is done at the end. Loop continues if test succeeds.
- **Example:**
  Write a do-while loop which reads a character from the keyboard without echo. Loop continues while the character ‘Q’ is not pressed. The terminating character ‘Q’ is then displayed.

DO01: ; do label
mov ah, 8 ; input character w/o echo
int 21h ; DOS function
cmp al, 'Q' ; check equal?
jne DO01 ; if not, repeat
DOEND01: ; character <> 'Q'
mov ah, 2 ; to output char
mov dl, al ; move to dl for output
int 21h ; output character

Example: Read a character without echo.
Display the character if it is a ‘Z’

mov ah, 8
int 21h ; read w/o echo
IF01: cmp al, 'Z' ; if character = 'Z' then
jne ENDIF01
THEN01: 
mov ah, 2 ; print character
mov dl, al
int 21h
ENDIF01: ; endif
**IF-THEN-ELSE-ENDIF**

IFnn:
  ; test condition, if false jump to ELSEnn
THENnn:
  ; statements for true condition
  ; jmp ENDIFnn
ELSEnn:
  ; statements for false condition
ENDIFnn:

---

**Example:** Read a character from the keyboard without echo. If the character is 'Z' print 'T' else print 'F'

```plaintext
mov ah, 8 ; input char w/o echo
int 21h
mov ah, 2 ; get ready to print

IF01: cmp al, 'Z' ; if char = 'Z' then
  jne ELSE01

THEN01: mov dl, 'T'
  int 21h ; print 'T'
  jmp ENDIF01

ELSE01: mov dl, 'F' ; else
  int 21h ; print 'F'

ENDIF01: ; endif
```

---

**Nested control constructs**

- Example:
- Write code which reads characters until the Enter key is pressed. If the character read is greater than the blank character (20h), print it, otherwise don’t print.

```plaintext
mov ah, 8 ; get char w/o echo
int 21h

WHILE02: cmp al, 13 ; while char <> 'n'
  je WEND02

IF04: cmp al, ' ' ; if char > ' ' then
  jb ENDIF04 ; note: use jb and not jl

THEN04: mov ah, 2
  mov dl, al
  int 21h ; print char

ENDIF04: ; endif

ENDIF04: mov ah, 8
  int 21h
  jmp WHILE02

WEND02: ; endwhile
```
Compound test conditions

• Example:
• Write code which will read in characters until the enter key is pressed. Characters ‘A’ to ‘D’ are to be displayed. Other characters are not displayed.

```asm
mov ah, 8 ; get ch w/o echo
int 21h

WHILE03: cmp al, 13 ; while ch <> ‘n’
    je WEND03

IF05: cmp al, ‘A’ ; if ch >= ‘A’
    jb ENDIF05

AND05: cmp al, ‘D’ ; and ch <= ‘D’ then
    ja ENDIF05

THEN05: mov ah, 2
    mov dl, al
    int 21h ; print ch

ENDIF05: ; endif

mov ah, 8 ; get ch w/o echo
int 21h
jmp WHILE03

WEND03: ; endwhile
```

Multiplication and Division

• The 80x86 instruction set includes instructions that can perform integer multiplication and division on 8, 16, and 32-bit integers
• Floating-point operations are handled either by a floating-point unit or by software emulation (library)
• The MUL (multiply) and DIV (divide) instructions are used for unsigned binary numbers
• IMUL (integer multiply) and IDIV (integer divide) are used for signed binary numbers

Multiplication and division of unsigned numbers in assembly language

• The following registers are required in multiplication or division:
  One of: al, ax, or eax, and one of: dx or edx (eg, ax and dx for 16-bit operands)
• The MUL instruction multiplies an 8, 16, or 32-bit operand by AL, AX, or EAX.
  Format: MUL source where source (multiplier) must be either a register or a memory operand, but not an immediate value
• Source is multiplied by:
  – AL if source is of type byte (i.e., 8-bit)
  – AX if source is of type word
  – EAX if source is of type dword

• The result of MUL/IMUL is stored in:
  – AX if source is of type byte (i.e., 8-bit)
  – DX:AX if source is of type word
  – EDX:EAX if source is of type dword

• Thus there is always enough space to hold the result.

Examples

• Multiplication: byte x byte
  – one value (operand) must be in AL,
  – the other value (source) in either register or memory
  – result in AX

Examples

• Eg, multiply 10 by 20 and store the result in a memory location called product.

```assembly
product dw ? ; defined in data segment
num1 db 10 ; for use later
num2 db 20
mov al, 10 ; in the code segment
mov bl, 20 ; any register can be used
mul bl ; ax = 200
mov product, ax ; save the result
```

Alternatives

```assembly
mov al, num1 ; data is in memory
mov bl, num2
mul bl ; register addressing
    ; ax = al * bl
mov product, ax ; save the result
OR
mov al, num1
mul num2 ; direct addressing mode
mov product, ax ; save result
```
mov   al, num1
mov   si, offset num2 ; can use bx or di too
mul   byte ptr [si]  ; reg indirect addressing
         ; byte ptr needed to indicate
         ; byte at location pointed by si

---

Multiplication: word x word

- one value must be in AX
- other value in a register or memory
- result in AX and DX
  - DX <- high word
  - AX <- low word

---

Eg,  104ah x 11abh

product dw 2 dup (?); defined in data seg.
num1   dw 104ah  ; hex in word
num2   dw 11ab

mov   ax, num1 ; in code segment
mul   num2
mov   product, ax ; save lower word
mov   product+2, dx ; save the higher word

---

Multiplication: word x byte

- same as word x word - AX is used with AH = 0 and AL contains the byte value
- other value in a register or memory
- result in AX and DX
  - DX <- high word
  - AX <- low word
• Eg. 2ah x 123dh
  product dw 2 dup (?) ; defined in data seg.
  num1 db 2ah ; hex in byte
  num2 dw 123dh ; hex in word

    mov al, num1 ; byte
    xor ah, ah ; ah = 0
    mul num2 ; al x num2
    mov bx, offset product ; pointer to product
    mov [bx], ax ; low word
    mov [bx]+2, dx ; high word

Multiplication: dword x dword

• one value must be in EAX
• other value in a register or memory
• result in EAX and EDX
  EDX <- high dword
  EAX <- low dword

Integer Division

• Produces two results:
  – quotient
  – remainder
• Notation for integer division:
  – dividend / divisor = quotient + remainder
  – Or, numerator / denominator = quotient + remainder
  – Eg, 22 / 4 = 5 + 2

• There are 2 instructions for division:
  – DIV divisor (denominator) ; unsigned division
  – IDIV divisor (denominator) ; signed division
• The denominator must be register or mem
• In case of IDIV, the remainder has always the same sign as the denominator
  – Eg, -7/2 = -3 + -1 (not -4 + 1)
• The size of the denominator determines which registers will hold the numerator, the quotient, and the remainder
### Denominator Numerator Quotient Remainder

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>AX</td>
<td>AL</td>
<td>AH</td>
</tr>
<tr>
<td>word</td>
<td>DX:AX</td>
<td>AX</td>
<td>DX</td>
</tr>
<tr>
<td>dword</td>
<td>EDX:EAX</td>
<td>EAX</td>
<td>EDX</td>
</tr>
</tbody>
</table>

- The effect on the flags is undefined
- We get a divide overflow error whenever the quotient cannot be contained in its destination (AL if denominator is byte...)
  - execution then traps into INT 0h which displays a message on screen and returns control to DOS

### Examples

#### Divide byte / byte
- AL = numerator
- AH = 0
- denominator is register or memory - cannot be immediate
- result (quotient) is in AL and the remainder is in AH

```assembly
num1     db   87 ; data segment
t
num2     db   10
quout     db    ?
t
remain   db    ?

mov al, num1 ; numerator	xor ah, ah ; clear ah to 0
t
mov bh, num2 ; use bh
div bh ; divide ax by bh
t
mov quout, al ; the quotient
t
mov remain, ah ; the remainder
```
Divide word / word

- AX = numerator
- denominator in register or memory
- DX = 0
- result (quotient) in AX, remainder in DX

Eg,

```
mov ax, 102Ch ; numerator
mov bx, 10h ; a reg used for denom
sub dx, dx ; zero dx
div bx
mov quout, ax ; the quotient
mov remain, dx ; remainder
```

Divide word / byte

- AX = numerator
- denominator in register or memory
- result (quotient) in AL, remainder in AH
- eg,

```
mov ax, 2000 ; ax has numerator
mov cl, 120 ; a reg has denominator
div cl
mov quout, al ; save quotient
mov remain,ah ; save remainder
```

In some cases, the CPU cannot perform the division. An interrupt (exception) is activated (int 0). The computer may display an error message “divide error”

- A divide by zero is one such case and the other is when the result (quotient) is too large to fit into the register used for the quotient.
- Note that signed multiplication and division will require interpreting the Flags register

Specialised Addressing instructions

- **LEA** - load effective address
  
  Format:
  
  ```
  LEA destination, source
  ```

  - where **destination** is any general word-sized register and **source** is any addressed memory location. The address of the memory location is placed in the register.
• LEA is similar to the OFFSET operator but is much more flexible.
  LEA BX, Sum ; BX = address of Sum
  has the same effect as
  MOV BX, OFFSET Sum
• If base or index registers are not used for addressing:
  LEA DI, buffer
  is the same as
  MOV DI, OFFSET buffer

• Note: the MOV instruction with OFFSET cannot be used with indirect addressing. Instead, LEA should be used
  LEA SI, buffer[SI][BX]+4
• Note: it is the address of memory location which is put into SI, not the contents.

LDS AND LES

• LDS is used to load the DS register
• LES is used to load the ES register
• Format:
  LDS destination, source
  LES destination, source
• where destination is any general register and source is an addressed memory location that contains two adjacent words.
• The first word goes into the destination register and the second word goes into DS (in case of LDS) or ES (in case of LES)

• Assuming the data segment as in the following table,
  Memory addr. | Contents
  ds:0100     | 27h
  ds:0101     | 10h
  ds:0102     | 59h
  ds:0103     | 23h

  the instruction LDS SI, [0100h]
  has the following effect
  SI <- 1027h
  DS <- 2359h
  The data segment is now changed
  If LES was used, ES is changed
  Once the instruction has been executed, the register is used as an offset into the appropriate segment.
Some Arithmetic flags (Flags register)

- Flags Register:
  - 15 14 13 12 11 10 9 8
  - x x x x O D I T
  - 7 6 5 4 3 2 1 0
  - S Z x A x P x C

Two types of flags:

- Status flags
  - O = overflow
  - D = direction
  - S = Sign
  - I = interrupt
  - Z = Zero
  - T = Trap

- Control flags
  - A = Auxiliary Carry
  - P = Parity
  - C = Carry

of - overflow flag - set when the result of signed numbers exceeds -128 to 127 for bytes, or -32768 to 32767 for 16-bit words

cf - carry flag - set when the result of arithmetic operation on unsigned numbers becomes less than 0 or exceeds 255 (0ffh) for bytes or 65535 (0ffffh) for word instructions

sf - sign flag - is set if the result of some operation is negative and cleared if it is positive. The sf is a copy of MSB of the signed result.

zf - zero flag - is set to 1 if the result of some operation is 0, and it is set to 0 if the result of the operation is not zero (careful when reading this)

Conditional Jump instructions and the flags register

- Typically, a conditional jump can be made after an operation changes one or more of the flags in the Flags register.
- A conditional jump is usually made after a compare instruction (CMP).
  - CMP destination, source
- It is equivalent to:
  - SUB destination, source
- ie. (destination - source) except that destination is not changed when CMP is used.
• **zf, sf, cf, of** - are some of the flags which change as a result of the compare instruction

• Different types of conditional jumps can be made depending on the settings of these flags. In addition, jumps can be made depending on the type of operands - ie signed or unsigned

• The following table summarizes the jump instructions used after a compare (CMP) instruction:

<table>
<thead>
<tr>
<th>Desired result Signed operands instruction</th>
<th>Jump if Signed operands instruction</th>
<th>Jump if Unsigned operands instruction</th>
<th>Jump if Unsigned operands instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>je/jz</td>
<td>zf = 1</td>
<td>je/jz</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>jne/jnz</td>
<td>zf = 0</td>
<td>jne/jnz</td>
</tr>
<tr>
<td>&gt;</td>
<td>jg/jnie</td>
<td>zf = 0 and sf = of</td>
<td>ja/jnbe</td>
</tr>
<tr>
<td>&gt;=</td>
<td>jge/jnil</td>
<td>sf = of</td>
<td>jae/jnb</td>
</tr>
<tr>
<td>&lt;</td>
<td>jl/jnge</td>
<td>sf &lt;&gt; of</td>
<td>jb/jnae</td>
</tr>
<tr>
<td>&lt;=</td>
<td>jle/jng</td>
<td>zf = 1 and sf &lt;&gt; of</td>
<td>jbe/jna</td>
</tr>
</tbody>
</table>

a = above (unsigned cmp)  
b = below (unsigned cmp)  
g = greater (signed cmp)  
e = equal  
l = less (signed cmp)  
n = not

• There are other instructions that execute a jump after testing the parity flag.

• The parity flag is used in data communications. This flag is set (to 1) for even parity. Even parity means that there are an even number of bits with 1.
String Instructions

- A string is an array or series of bytes.
- There are 5 very useful instructions provided for string manipulation:
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lods</td>
<td>load string</td>
</tr>
<tr>
<td>stos</td>
<td>store string</td>
</tr>
<tr>
<td>scas</td>
<td>scan string</td>
</tr>
<tr>
<td>movs</td>
<td>mov string</td>
</tr>
<tr>
<td>cmps</td>
<td>compare string</td>
</tr>
</tbody>
</table>

  Each of these strings can operate on a byte or a word at a time. Eg,
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lodsb</td>
<td>load string byte</td>
</tr>
<tr>
<td>stosb</td>
<td>store string byte</td>
</tr>
<tr>
<td>lodsw</td>
<td>load string word</td>
</tr>
<tr>
<td>stosw</td>
<td>store string word</td>
</tr>
<tr>
<td>movsb</td>
<td>copy string byte</td>
</tr>
</tbody>
</table>

- These instructions use either the SI and/or DI registers. The SI register is associated with the Source and the DI register is associated with the Destination.
- The SI register holds an offset into the data segment and the DI register holds the offset into the extra segment. (The source is by default in the data segment and the destination in the extra segment). This means that the extra segment register, ES, must be initialised.

  If both the source and destination strings are in the same segment, both the DS and ES registers are set to the same value.

Direction Flag - DF

- This flag determines the direction in which the strings are processed:
  
<table>
<thead>
<tr>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ascending</td>
<td>strings are processed from the lowest address of the string to the highest address of the string. To do this, the direction flag must be cleared (set to zero) using the CLD (clear direction flag) instruction. This means that the pointers to the string (SI and DI) are incremented automatically.</td>
</tr>
<tr>
<td>Descending</td>
<td>strings are processed from the highest address of the string to the lowest address of the string. To do this, the direction flag must be set (set to 1) using the STD (set direction flag) instruction. This means that the pointers to the string (SI and DI) are decremented automatically.</td>
</tr>
</tbody>
</table>

REP (repeat) prefixes

- The REP prefix can be applied to the 5 string instructions (as well to the INS - input string from port and OUTS - output string to port - instructions) to repeat these instructions a number of times.
- The number of times to be repeated will be specified by the count register CX. Eg,
  
  MOV CX, 50
  REP MOVSB ; repeats MOVSB 50x
  ; SI and DI adjusted automatically depending on the direction flag
• Si, Di and ES must have been set before this. DF must have been cleared or set.
• CX is decremented by one each time. No flags are modified.
• The are other forms of the REP prefix. These other forms depend on the setting of the zero flag.
• REPE (repeat if equal) and REPZ (repeat if zero) are the same - repeat if zero flag indicates zero.
• REPNE (repeat if not equal) and REPNZ (repeat if not zero) repeats if zero flag indicates non-zero result.

Move String (MOVS)
• Used to move a string of bytes or words from one location in memory to another.
• moves a byte or word from the address pointed to by DS:SI to the address pointed to by ES:DI.
• After the byte or word is moved SI or DI is adjusted (by 1 for byte; by 2 for word). If DF is clear --> increment if DF is set --> decrement.
• MOVs can be repeated a number of times by setting CX and using the REP prefix.

• Eg, 2 strings - firstStr and secStr of size 100 bytes. ES and DS have been properly set. Then:
  cld ; clear DF
  lea si, firstStr ; si points to source
  lea di, secStr ; di points to dest
  mov cx, 100 ; count is 100
  rep movsb ; mov 100 bytes
• The same effect can be produced by setting cx to 50 and using movsw.

• Movs (without the sb or sw ) accepts 2 operands in memory. This means the last instruction in the above example could have been replaced by:
  rep movs secStr, firstStr
• In this case both strings must be defined as either db or dw. The assembler will move byte or word depending on the definition.
• If indirect addressing is to be used and the assembler cannot determine to move byte or word, then the following format is to be used.
  
  ```assembly
  movs size ptr [di], size ptr seg_reg:[si]
  ```

  where size is either byte or word and seg_reg can be cs, ds, es or ss (ie using segment override for source; no segment override for destination - ie it must be es). If size is not specified, then mov is carried out a byte at a time.

---

**XLAT - translate**

Format:

**XLAT source-table**

- The contents of AL is translated to any other desired byte according to the source-table. BX holds the address of source. The contents in AL is considered to be an offset into table and the byte at the offset is moved into AL after XLAT.

---

**Eg,**

```assembly
  table db "0123456789"
  mov al, 02h
  mov bx, offset table
  xlat table

  AL now will contain '2' or 32h
```

---

• XLAT is equivalent to the following instructions where XLAT is not available (on RISC processors)

  ```assembly
  sub ah, ah
  mov si, ax ; al is the offset
  mov al, [bx+si]
  ```