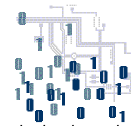
Assembly Language Fundamentals

Chapter 3

3

Numeric Constants

- Numeric constants are made of numerical digits with, possibly, a sign and a suffix. Ex:
  - -23 (a negative integer, base 10 is default)
  - 1011b (a binary number)
  - 1011 (a decimal number)
  - 0A7Ch (an hexadecimal number)
  - A7Ch (this is the name of a variable, *an hexadecimal number beginning with a letter must start with a decimal digit; assembler will take it has identifier*)



- Identifier: It is an user defined name to identify a variable, a constant, a procedure or a code label.
  - Not case sensitive
  - First character must be either a letter, _, @ or $.
  - Cannot be the same as an assembler reserve word

Character and String Constants

- Any sequence of characters enclosed either in single or double quotation marks. Embedded quotes are permitted. Ex:
  - ‘A’
  - ‘ABC’
  - “Hello World!”
  - “123” (this is a string, not a number)
  - “This isn’t a test”
  - ‘Say “hello” to him’ – Embedded quotes

Statements

- The general format is:
  - [label name:] mnemonic [operands] [;comment]
- A label is a name that appears in the code area. Must be followed by ‘:’
- Instruction Mnemonic is a short word that identifies the operation carried out by an instruction.
- Operand is generally a input to mnemonic.
  - 0 to 3 operands
  - Can be register, memory, constant or I/O port.

Statements

- Statements are either:
  - Instructions: executable statements -- translated into machine instructions. Ex:
    - call MySub ;transfer of control
    - mov ax,5 ;data transfer
  - Directives: tells the assembler how to generate machine code and allocate storage.
  - Directive set varies for different assemblers.
  - Ex:.data, .code, name PROC
  - .data

```
count db 50
        ;creates 1 byte of storage
        ;allocates 1 byte to variable count
        ;initialized to 50
```

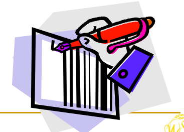
Names (cont.)

- A variable is a symbolic name for a location in memory that was allocated by a data allocation directive. Ex:

  count db 50 ; allocates 1 byte to variable *count*

Segment Directives

- A program normally consist of a:
  - code segment that holds the executable code
  - data segment that holds the variables
  - stack segment that holds the stack (used for calling and returning from procedures)
- Directives `.code`, `.data`, and `.stack` mark the beginning of the corresponding segments



ece.

Program Template (MASM)

```
;Program description
;author
INCLUDE <include files>
.data
   ; insert variables
. Code
start
   ;Insert instructions
end
```

Segment Directives

ece.

Program Template (NASM)

```
;Program description
;author
SEGMENT .data
   ; insert variables
SEGMENT . text
   ;Insert instructions
SEGMENT .bss
   ;insert uninitialised data
end
```

Segment Directives

ece.

Procedure

- The `proc` and `endp` directives denote the beginning and end of a procedure
- To return the control to DOS we use a software interrupt

  mov ah,4Ch

  int 21h
- The `end` directive marks the end of the program and specify the pgm's entry point

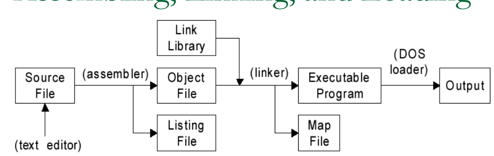
ece.

The Program Segment Prefix (PSP)

- When DOS loads a program in memory, it prefaces the program with a PSP of 256 bytes
  - the PSP contains info (about the pgm) used by DOS
- DS (and ES) gets loaded by DOS with the segment address of the PSP. To load DS with the segment address of the data we do:
  - mov ax,@data
  - mov ds,ax ;cannot move a constant into ds directly
- @data is the name of the data segment defined by .data (and gets translated by the assembler into the data's segment number)
- CS and SS are correctly loaded by DOS with the segment number of code and stack respectively

ece.

Assembling, Linking, and Loading



- The object file contains machine language code with some external and relocatable addresses that will be resolved by the linker
- Link library = file containing several object modules (compiled procedures)
- The loader loads the executable program in memory and transfers control to it

ece.

Review

- Numeric, character and string constants
- Basic statement syntax.
- Mnemonics
- Operands
- Directives
- Variables
- Label
- PSP
- Assembling, linking and executing.

ece. 15 WICHITA STATE UNIVERSITY

Simple Data Allocation Directives

- The DB (define byte) directive allocates storage for one or more byte values
  - [name] DB initval [,initval]
- Each initializer can be any constant. Ex:
  - a db 10, 32, 41h ;allocate 3 bytes
  - b BYTE 0Ah, 20h, ‘A’ ;same values as above
- A question mark (?) in the initializer leaves the initial value of the variable undefined.
- Ex:
  - c db ? ;the initial value for c is undefined

ece. 16 WICHITA STATE UNIVERSITY

Simple Data Allocation Directives (cont.)

- A string is stored as a sequence of characters. Ex:
  - aString db “ABCD”
- The offset of a variable is the distance from the beginning of the segment to the first byte of the variable. Ex. If Var1 is at the beginning of the data segment:

```
.data
Var1 db “ABC”
Var2 BYTE “DEFG”
```

- The offset of Var1 is 0 = the offset of ‘A’
  - The offset of ‘B’ is 1
  - The offset of ‘C’ is 2 ...
- The offset of Var2 is 3

ece. 17 WICHITA STATE UNIVERSITY

Simple Data Allocation Directives (cont.)

- Define Word (DW) allocates a sequence of words. Ex:
  - A dw 1234h, 5678h ; allocates 2 words
  - B WORD 3245h
- Intel’s x86 are little endian processors: the lowest order byte (of a word or double word) is always stored at the lowest address. Ex: if the offset of variable A (above) is 0, we have:
  - offset: 0 1 2 3
  - value: 34h 12h 78h 56h

ece. 18 WICHITA STATE UNIVERSITY

Little Endian Order

- Little endian order refers to the way Intel stores integers in memory.
- Multi-byte integers are stored in reverse order, with the least significant byte stored at the lowest address
- For example, the doubleword 12345678h would be stored as:

byte	offset
78	0000
56	0001
34	0002
12	0003

When integers are loaded from memory into registers, the bytes are automatically re-reversed into their correct positions.

ece. 19 WICHITA STATE UNIVERSITY

Simple Data Allocation Directives (cont.)

- Define Double Word (DD) allocates a sequence of double words. Ex:
  - B dd 12345678h ; allocates one double word
  - C DWORD ?
- If this variable has an offset of 0, we have:
  - offset: 0 1 2 3
  - value: 78h 56h 34h 12h
- Sign ?

ece. 20 WICHITA STATE UNIVERSITY

Simple Data Allocation Directives

- If a value fits into a byte, it will be stored in the lowest ordered one available. Ex:
  - V dw 'A'
- the value will be stored as:
  - offset: 0 1
  - value: 41h 00h
- The value of a variable B will be the address of a variable A whenever B's initializer is the name of variable A. Ex:
  - A dw 'This is a string'
  - B dw A ; B points to A (B contains A's offset)

ece. 21

Simple Data Allocation Directives (cont.)-(MASM)

- The DUP operator enables us to repeat values when allocating storage. Ex:
  - a db 100 dup(?) ; 100 bytes uninitialized
  - b db 3 dup("Ho") ; 6 bytes: "HoHoHo"
- DUP can be nested:
  - c db 2 dup('a', 2 dup('b')) ; 6 bytes: 'abbabb'
- DUP must be used with data allocation directives

ece. 22

Symbolic constants

- We can use the equal-sign (=) directive to give a name to a constant. Ex:
  - one = 1; this is a (numeric) symbolic constant
- The assembler does not allocate storage to a symbolic constant (in contrast with data allocation directives)
  - it merely substitutes, at assembly time, the value of the constant at each occurrence of the symbolic constant

ece. 23

Symbolic constants (cont.)

- In place of a constant, we can use a *constant expression* involving the standard operators used in HLLs: +, -, *, /
- Ex: the following constant expression is evaluated at assembly time and given a name at assembly time:
  - A = (-3 * 8) + 2
- A symbolic constant can be defined in terms of another symbolic constant:
  - B = (A+2)/2

ece. 24

Symbolic constants (cont.)

- To make use of it, a symbolic constant must evaluate to a numerical value that can fit into 16 bits or 32 bits (when the .386 directive is used...) Ex:
  - prod = 5 * 10 ; fits into 16 bits
  - string = 'xy' ; fits into 16 bits
  - string2 = 'xyxy' ; when using the .386 directive
- The equate (EQU) directive is almost identical to the equal-sign directive
  - except that a symbolic constant defined with EQU cannot be redefined again in the pgm

ece. 25

Data Transfer Instructions

- The MOV instruction transfers the content of the source operand to the destination operand
  - mov destination, source
- Both operands must be of the same size.
- An operand can be either direct or indirect
- Direct operands (this chapter):
  - immediate (imm) (constant or constant expression)
  - register (reg)
  - memory variable (mem) (with displacement)
- Indirect operands are used for indirect addressing (next chapter)

ece. 26

Data Transfer Instructions (cont.)

- Some restrictions on MOV:
  - imm cannot be the destination operand...
  - IP cannot be an operand
  - the source operand cannot be imm when the destination is a segment register (segreg)
    - mov ds, @data ; illegal
    - mov ax, @data ; legal
    - mov ds, ax ; legal
  - source and destination cannot both be mem (direct memory-to-memory data transfer is forbidden!)
    - mov wordVar1,wordVar2; illegal

ece.

27

Data Transfer Instructions (cont.)

- The type of an operand is given by its size (byte, word, doubleword…)
  - both operands of MOV must be of the same type
  - type check is done by the assembler
  - the type assigned to a mem operand is given by its data allocation directive (DB, DW…)
  - the type assigned to a register is given by its size
  - an imm source operand of MOV must fit into the size of the destination operand

ece.

28

MOV syntax

- Mov r8, r8
- Mov m8, r8
- Mov r8, m8
- Mov r8, i8
- Mov r16, r16
- Mov m16, r16
- Mov r16, m16
- Mov m16, i16
- Mov r16, i16
- Mov sr, r16
- Mov sr, m16

ece.

29

MOV examples

- Examples of MOV usage:
  - mov bh, 55; 8-bit operands
  - mov al, 256;
    - error: size too large
  - mov bx, AwordVar; 16-bit operands
  - mov bx, AbyteVar;
    - error: size mismatch
  - mov edx, AdoublewordVar; 32-bit operands
  - mov cx, bl ;
    - error: operand not of same size
  - mov wordVar1, wordVar2;
    - error: mem-to-mem

ece.

30

Data Transfer Instructions (cont.)

- We can add a displacement to a memory operand to access a memory value without a name Ex:

```
.data
arrB db 10h, 20h
arrW dw 1234h, 5678h
```

- arrB+1 refers to the location one byte beyond the beginning of arrB and arrW+2 refers to the location two bytes beyond the beginning of arrW.

```
mov al,arrB      ; AL = 10h
mov al,arrB+1    ; AL = 20h (mem with displacement)
mov ax,arrW+2    ; AX = 5678h
mov ax,arrW+1    ; AX = 7812h (little endian convention!!)
```

ece.

31

Data Transfer Instructions (cont.)

- The XCHG instruction exchanges the content of the source and destination operands:

  XCHG destination, source
- Only mem and reg operands are permitted (and must be of the same size)
- both operands cannot be mem (direct mem-to-mem exchange is forbidden).
- To exchange the content of word1 and word2, we have to do:

```
mov ax,word1
xchg word2,ax
mov word1,ax
```

ece.

32

Simple arithmetic instructions

- The ADD instruction adds the source to the destination and stores the result in the destination (source remains unchanged)
  - ADD destination, source
- The SUB instruction subtracts the source from the destination and stores the result in the destination (source remains unchanged)
  - SUB destination, source
- Both operands must be of the same size and they cannot be both mem operands
- Recall that to perform A - B the CPU in fact performs A + NEG(B)

ece.

Simple arithmetic instructions (cont.)

- ADD and SUB affect all the status flags according to the result of the operation
  - ZF (zero flag) = 1 iff the result is zero
  - SF (sign flag) = 1 iff the msb of the result is one
  - OF (overflow flag) = 1 iff there is a signed overflow
  - CF (carry flag) = 1 iff there is an unsigned overflow
- Signed overflow: when the operation generates an out-of-range (erroneous) signed value
- Unsigned overflow: when the operation generates an out-of-range (erroneous) unsigned value

ece.

Simple arithmetic instructions (cont.)

- Both types of overflow occur independently and are signaled separately by CF and OF
  - mov al, 0FFh
  - add al,1 ; AL=00h, OF=0, CF=1
  - mov al,7Fh
  - add al, 1 ; AL=80h, OF=1, CF=0
  - mov al,80h
  - add al,80h ; AL=00h, OF=1, CF=1
- Hence: we can have either type of overflow or both of them at the same time

ece.

Simple arithmetic instructions (cont.)

- The INC (increment) and DEC (decrement) instructions add 1 or subtracts 1 from a single operand (mem or reg operand)
  - INC destination
  - DEC destination
- They affect all status flags, except CF. Say that initially we have, CF=OF=0
  - mov bh,0FFh ; CF=0, OF=0
  - inc bh ; bh=00h, CF=0, OF=0
  - mov bh,7Fh ; CF=0, OF=0
  - inc bh ; bh=80h, CF=0, OF=1

ece.

Simple I/O Instructions

- We can perform simple I/O by calling DOS functions with the INT 21h instruction
- The I/O operation performed (on execution of INT 21h) depends on the content of AH
- When AH=2: the ASCII code contained in DL will be displayed on the screen. Ex:
  - mov dl, 'A'
  - int 21h ; displays 'A' on screen at cursor position
- Also, just after displaying the character:
  - the cursor advance one position
  - AL is loaded with the ASCII code
- When the ASCII code is a control code like 0Dh (CR), or 0Ah (LF): the corresponding function is performed

ece.

Reading a single char from the keyboard

- When we strike a key, a word is sent to the *keyboard buffer* (in the BIOS data area)
  - low byte = ASCII code of the char
  - high byte = Scan Code of key (more in chap 5)
- When AH=1, the INT 21h instruction:
  - loads AL with the next char in the keyb. buff.
  - echoes the char on the screen
  - if the keyboard buffer is empty, the processor busy waits until one key gets entered
    - mov ah,1
    - int 21h ; input char is now in AL

ece.

Displaying a String

- When AH=9, INT 21h displays the string pointed by DX. To load DX with the offset address of the desired string we can use the OFFSET operator:
  - .data
  - message db 'Hello', 0Dh, 0Ah, 'world!', '$'
  - .code
  - mov dx, offset message
  - mov ah,9 ;prepare for writing string on stdout
  - INT 21h ;DOS system call to perform the operation
- This instruction will display the string until the first occurrence of '$'.
- The sequence 0Dh, 0Ah will move the cursor to the beginning of the next line.