Advanced Instructions

Addressing Mode : (3) Direct Memory

- Direct memory mode: address of **memory variable** specified as operand
  - constant address **offset: encoded as part of the instruction**
  - **address offset static**: must be known at assembly-time. Remains constant through execution… (contents of the address may be dynamic)
  - During execution, address offset ***implicitly*** **combined with DS**

- **Example** : MOV AL, [5]
  - Reads contents of byte at address DS:[0005]

- **Example** : MOV X, AX
  - Assumes variable declared in data segment
  - Write contents of word at address DS:X

BEWARE : Compare To Immediate Mode!!
MOV AL, 5

Programs to do simple arithmetic

Problem : Write a code fragment to add values of memory locations at DS:0000, DS:0001, and DS:0002, and save the result at DS:0010h.

Solution:

Step 1 AL ⇐ m[DS:0000]

Step 2 AL ⇐ AL + m[DS:0001]

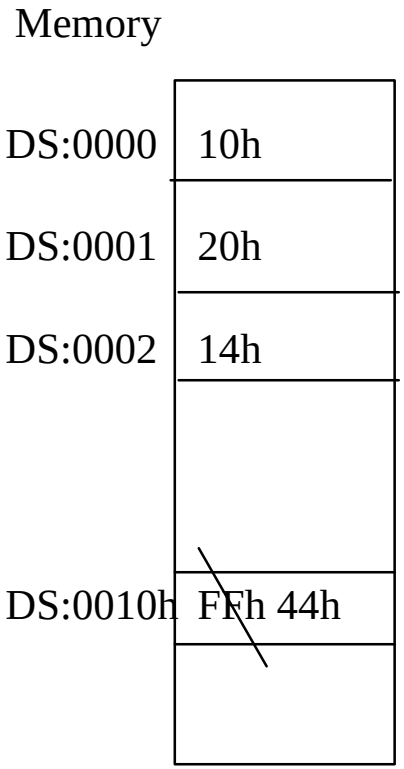
Step 3 AL ⇐ AL + m[DS:0002]

Step 4 DS:0010h ⇐ AL

MOV AL, [0000]
ADD AL, [0001]
ADD AL, [0002]
MOV [0010h], AL

Processor

AL=10h/30h/44h



Problem : Write a program to perform the following operation :
z = x + y where x = 55667788h and y = 99669988h

Solution:

```
.data
x DW  7788h
  DW  5566h
y DW  9988h
  DW  9966h
z DW  ?
  DW  ?

.code
  MOV AX, x
  MOV BX, x+2
  ADD AX, y
  ADD BX, y+2
  MOV z, AX
  MOV z+2, BX
END
```

```
    1  111
 5566  7788
+9966  9988
-----------
 EECD 1110
```

Operand Compatibility with Memory Operands

Clear and unambiguous Examples

MOV [BC], AX

MOV [1234h], AL

– Why? REGISTER operand determinates size

Ambiguous Examples :

MOV [BC], 1

MOV [1234h], 0

– Why? Immediate operand could be 8 bits or 16 bits.

– How does the assembler decide ?

Operand Compatibility with Memory Operands

- Memory Access **Qualifiers**

  WORD PTR **word pointer – 16-bit operand**
  BYTE PTR **byte pointer – 8-bit operand**

- Example :

  MOV BYTE PTR [0FF3E], 1
  8 bit destination, no ambiguity

  MOV WORD PTR [1234h], 0
  16-bit destination, no ambiguity

Assembler Tip About Operand Compatibility

```
    W       DW
. . .
    MOV   AL,  W
```

16-bit **memory src operand**

8-bit **register dest operand**

- The assembler will generate an error
  - Basic “type checking”

Intel 8086 Assembly Language – Memory Declarations

- Multiple data declarations on one line:
  - Separate by a comma
  - Allocated to successive locations
- Examples:

```
                DB      3, 6, 9
Array1          DW      -1, -1, -1, 0
Array2          DB      5 dup(0)
Array3          DW      3 dup(?)
```

Intel 8086 Assembly Language – Memory Declarations

- To declare a string variable
  - enclose in quotes
  - ASCII chars stored in consecutive bytes

```
Message             DB    ‘Hi Mom!’
MessageNullT        DB    ‘Hi Mom!’, 0
DOSMessage          DB    ‘Hi Mom!’, ‘$’
```

Any string to be printed out by DOS functions must be terminated by ‘$’

Understanding High-Level Control Flow at Machine Level

- Data transfer/manipulation instructions: **CS:IP to next sequential instruction**.

- Control flow instructions: **changes address** for fetch next instruction.
- Example:
  - condition true:
    - continue sequentially
    - skip to `next_statements`
  - condition false:
    - skip to `false_statements`
    - continue sequentially

- Conditions depend on flags (zero, carry, overflow)

Conditional Statements :

```
if ( condition ) {
      true_statements;
}

else {
      false_statements;
}

next_statements;
```

Skip == Change CS:IP of next fetched instruction

Understanding High-Level Control Flow at Machine Level

- Control Flow instructions also needed in Program Loops

```
for (i = n1 to n2)    for (i= n2 downto n1)
{ do statements S }   { do Statements S }

while (condition C) do { statements S }

repeat {  statements S } until (condition C)
```

"condition" dependent on the status flags

Control Flow Implications of Segmented Memory Model

- Address of next instruction determined by CS:IP

- Intra-segment control flow: stay in current code segment
  - **Only modify IP**
  - Supply 16-bit of information

- Inter-segment control flow: control passes to address outside of current code segment
  - **Modify both CS and IP**
  - Supply 32-bits of information.

- To begin: only concerned with intrasegment control flow (modify IP)

Control Flow : JMP instructions

Four types of JUMP instructions:

Unary (unconditional) jumps: **always** execute

```
JMP     target
```

Conditional Jumps

Simple jumps: taken when a **specific flag** is set

```
JC      target
```

Unsigned jumps: comparison of **unsigned numbers** results in specific combination of status flag

```
JA      target
```
(Jump if above)

Implication : Preceded by an instruction that alters the appropriate flags

Signed jumps: comparison of **signed quantities** results in specific combination of status flags

```
JG      target
```
(Jump if greater than)

Specifying Control Flow Targets (Intra-segment)

- Jump instructions: target must supply a value used to modify IP

1. **Absolute addressing**: 16-bit constant value to replace the IP
   Execution Semantics: IP := new value

2. **Relative addressing**: value to be added to IP (after fetch!)
   Execution Semantics : IP := IP + value
   Value positive: jump “forward”
   Value negative: jump “backward”

3. **Register/memory indirect addressing**: a register or memory location contains the value to be used to replace IP
   Execution Semantics : IP := mem[addrs]
   IP := register

Specifying Control Flow Targets (Intra-segment)

Question : What addressing modes are used below ?

```
JMP     1000h

JMP     here
```

JMP target Unconditional JUMP

- Control is always transferred to specified (relative) target.

Relative Addressing Example: .LST file fragment

address	machine instruction (memory contents)	ASM instruction
0034H	E9 10 02	JMP here
….	….	….
0247H		here:

```
start of fetch:     IP = 0034H   IR =  ????????
after fetch:        IP = 0037H   IR =  E9 02 10
after execute:      IP = 0247H   IR =
```

(Little endian=0210h)

Simple Conditional JMPs

- Specify condition in terms of FLAG values

  JZ Jump Zero: if ZF = 1 else continue

  JC Jump Carry: if C=1 else continue

  JO Jump Overflow: if O=1 else continue

  JS Jump Sign: if S=1 else continue

  JP Jump Parity: if P=1 else continue

- For each case, there is a “not” condition

  e.g. JNZ Jump Not Zero

Loop Example:

```
DoLoop:
        . . .
        SUB   CX, 1
        JNZ   DoLoop
```

Assembly language label translated to relative offset by assembler

Comparison Instructions - CMP

- Used to simply set the flags

```
CMP   dest, src
```

- Performs **dest – src**. Sets FLAGS (but does not store result)

- Example :

```
    CMP    AL, 10
    JZ     EqualToTen    ; or JE !
    …                    ; Code for Not Equal
EqualToTen:
```

- Often useful to think of combination of:

```
CMP dest, src
J*
```

Jump taken **if "dest * src" condition holds**

  – in above example, jump is taken if AL == 10

Signed and Unsigned Conditional Instructions

- Processor provides status flags to reflect results of (binary) manipulation under both signed and unsigned interpretations

- Separate conditional jump instructions for signed and unsigned
  - implementation tests flags appropriate to the data type.

Unsigned		Signed	
JA	Above	JG	Greater
JAE	Above or Equal	JGE	Greater or Equal
JB	Below	JL	Less
JBE	Below or Equal	JLE	Less or Equal

- There are also instructions for Not conditions too!

Example : Conditional Branches

Suppose AL = 7FH:

Unsigned Scenario	Signed Scenario
SUB AL, 80h	SUB AL,80h
JA Bigger	JG Bigger

In each scenario, is the jump taken? Why?

Programmer MUST know how binary values are to be interpreted!
(e.g. value in AX above)

Limitation of J* Instructions

- Conditional jump restricted to 8-bit signed relative offset!
  - IP := IP + (offset sign-extended to 16-bits)
  - Can't jump very far! – 128 ←→ +127 bytes

- Example:

```
        JL      Less
        ADD     AX, 1
        . . .
Less:   MOV     . . .
```

maximum possible distance = 127 bytes

- One possible workaround if distance is greater than 127 bytes (but not the only one!):

```
        JNL     Continue
        JMP     Less
Continue:
        ADD     AX, 1
        . . .
Less:   MOV     . . .
```

JMP can have 16-bit relative offset

distance can now be > 127

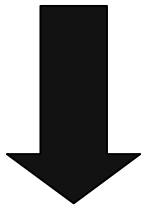
LOOP Instruction

- Useful when you have an action repeated a given number of times
- C++ analogy

  for (int i=max; i > 0; i--)

```
        MOV   CX, max
DoLoop:     .  .  .
        SUB   CX, 1
        JNZ   DoLoop
```

Functionally equivalent



Different performance & code size

```
        MOV CX, max
DoLoop:     .  .  .
        LOOP  DoLoop
```

- LOOP automatically decrements CX
- only works with CX

Example Write a code fragment showing how you would implement the following pseudocode

```
boolean  done = FALSE;

while ( ! done )

{       . . . }
```

Solution:

```
TRUE equ 1

FALSE equ 0

.code

          MOV AL, FALSE    ; AL= register variable done

notDone:  CMP AL, TRUE

          JE amDone

          ; . . .

          JMP    notDone

amDone:   . . .
```

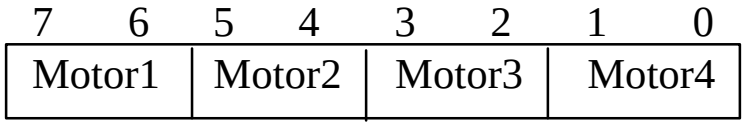
Example: Write a code fragment to test whether a variable is divisible by 4, leaving the boolean result in AX.

Solution: A number divisible by 4 would have the least significant two bits equal 0s.

```
FALSE       equ       0
TRUE        equ       1
.data
variable    dw        1922h
.code
    MOV     AX, variable
    AND     AX, 03h
    JZ      yes
    MOV     AX, FALSE
    JMP     continue
yes:MOV     AX, TRUE
continue:
    . . .
```

Alternative :
TEST variable, 03h

Example: Suppose a robot has four motors, each of which can be off, in forward direction or in reverse direction. Status of motors is stored by the robot into a status word, called “motors” in the following bitmap formation.



where the two bits for each motor are set according

01	forward
10	reverse
11	off

Write a code fragment that waits until motor1 is off before continuing on.

…

Solution:

```
            .data
            motors db  ?

            .code
waiting:    MOV AL, motors
            AND AL, 0C0h
            CMP AL, 0C0h
            JNZ waiting
            …
```

Basic Coding Conventions

- As in high-level languages, coding conventions make programs easier to read, debug and maintain.

```
varName      DB ?
MAX_LIMIT    EQU   5

label:
   MOV AX, BX
   CMP AX, varName
```

Indentation :

1.Label left-justified

2.Instructions lined up one tab in.

3.Next instruction follows label.

Naming Convention :

4. Labels lower case, upper case for joined words

5. EQU symbols are UPPER_CASE

6. Keywords (mnemonics and registers) upper-case

Basic Coding Conventions

```
varName     DB ?                ; Counter
MAX_LIMIT   EQU   5             ; Limit for counter

; Test for equality
label:
  MOV AX, BX          ; AX is current
  CMP AX,varName      ; If current < varName
```

Comments
1. Use them
2. Comments on the side to explain instruction
3. Comments on the left for highlight major sections of code.

Intel 8086 Assembly Language – Memory Declarations

- When using constants to initialize a memory declaration

1. Beware an assembler quirk

```
DW  8000h  ; 16-bit value of 8000h is loaded into word
DW  0FFFFh ; 16-bit value of FFFFh is loaded into word
           ; Zero does not mean 20-bit value
           ; Zero is needed by assembler to distinguish
           ; a HEX number from a label reference
```

2. Which one is easier to read?

```
DW    -1
DW    0FFFFH
DW    1111111111111111B
```

In all three cases, the same Binary value is assigned.

Intel 8086 Assembly Language – Directives

Directives are statements that are intended for other tools

- **not assembled** directly into instructions or memory declarations

END Directive:

- used by 2 tools: assembler & loader
- Assembler: **stop reading from .ASM file**
  - Any statements after END are ignored.
- Optional operand: a **control flow label** reference.
  - Loader uses this as the address of **first instruction** to be executed:

Syntax : END [label-reference] [ ] means optional

Intel 8086 Assembly Language – Directives

- Tell the tools what type of machine the program will be running on
  - Diferent members of 80x86 family have diferent instructions
  - Also, diferent address spaces (sizes/configurations of programs to run)

- **Example**: a program requires 20 bytes for data, 137 bytes for instructions (code), 100 bytes for stack
  - could all fit in **one segment** !
  - optimal organization: CS, DS and SS could all overlap

- **Example** : a program requires 80K bytes for data, 47K bytes for instructions (code), 10K bytes for stack
  - need **diferent non-overlapping segments** for each one, and two data segments!
  - Segment management is more complicated.

- In this course, we will be writing **small** 8086 programs.
  - amount of memory reserved for code: less than 64K
  - amount reserved for data: less than 64K
  - amount of memory used for run-time stack: will not exceed 1K bytes.

Intel 8086 Assembly Language – Directives

.8086

- limits assembler 8086 processor instruction set

.model

- Allows tools to make simplifying assumptions when organizing data
- We will use .model small
- At most: program will use **one code** and **one data** segment
- No intersegment control flow needed
- Never need to modify DS or CS once initialized

Intel 8086 Assembly Language – Directives

Only when working with .model small

.code

- Identifies the start of the code segment
- Tools will ensure that enough memory is reserved for the encodings of the instructions

.data

- Identifies the start of the data segment

.stack size

- Reserves size bytes of memory for the run-time stack
- More on stack later

A question of style !

Intel 8086 Assembly Language – Directives

. **EQU <u>directive</u>**

- allows you to define a **symbolic name** for a number
- symbolic name can be used anywhere you want to use that number
- assembler will replace symbolic name with the actual number **before assembling**

- A EQU directive does NOT result in any memory being declared or initialized!

```
VAL  EQU   0FFFFh
X    DW     VAL
V    DW     VAL
```

What is the advantage of defining a symbol with EQU ?

```
MOV BX, VAL
CMP AX, VAL
MOV DX, X
```

Loading a Program

- Our program must be loaded into memory to be executed
  - Done by the **loader** (a part of the operating system)
  - decides which **physical memory segments** to be used
  - initializes SS:SP (for stack use – later!) and CS:IP (to point to first instruction to be executed)
  - initializes DS but NOT to the data segment for the program !
  - Loader “knows’ which segment it has loaded as the data segment
  - As the program is loaded, loader replaces every occurrence of “@data” with the data segment number

- What does this mean for our program ?

Loading a Program

- Recall : processor uses contents of DS as 16-bit segment value to access **memory variables**
  - programmer only needs to supply 16-bit offset in instructions

- DS must be initialized **before** ANY access to the data segment
  - Before any reference to a memory label

- Because the loader does not initialize DS, DS is **initialized dynamically** (at run time)
  - It should be the first thing program does !
  - Specifically, no variable defined in the data segment can be referenced until DS is initialized.

Loading a Program

- How do we initialize DS ?
  - Wrong way (due to limited addressing modes, later)

```
MOV DS, @data
```

  - Correct way

```
MOV AX,  @data
MOV DS, AX
```