

Introductory Assembly Language

Thorne : Chapter 3, Sections 7.1, 13.1, Appendix V.A
(Irvine, Edition IV : 4.1, 4.2, 6.2 7.2, 7.3, 7.4)

Intel 8086 Assembly Instructions

- Assembly instructions are readable forms of **machine instructions**
 - They use **mnemonics** to specify operations in a human-oriented short form
 - Examples
 - MOV (move)
 - SUB (subtract)
 - JMP (jump)
- Instructions have two aspects : operation and operands
 - **Operation (Opcode)**: how to use state variable values
 - **operands**: which state variables to us
- Operands can be specified in a variety of ways that are called **addressing modes**
 - Simple modes: register, immediate, direct
 - More powerful: indirect

Sample Instructions

Syntax

Semantics

MOV	AX, BX	$AX := BX$	(2 ops)
ADD	DX, CX	$DX := DX + CX$	(2 ops)
SUB	DX, AX	$DX := DX - AX$	(2 ops)
INC	AX	$AX := AX + 1$	(1 op)
NOP			(0 op)

Instructions with two operands : **destination (dest)**, **source (src)**

MOV	AX,	BX
↑	↑	↑
Operation	operand (dest),	operand (src)
(Opcode)		

(Order of dest and src is important, Must know on exams)

Instruction Syntax : Operand Compatibility

- For all instructions with two operands, the two operands must be **compatible (same size)**.
 - In high level languages : type checking
 - In assembly : same size

- Examples :

MOV AH, CL


8-bit src and dest 😊

MOV AL, CX

????? ☹️

Example uses register mode, but compatibility is required for all addressing modes to come.

Addressing Modes

Syntax	Semantics	Addressing Mode
MOV AX, BX	$AX := BX$	Register, Register
ADD DX, 1	$DX := DX + 0001$	Register, Immediate
SUB DX, [1]	$DX := DX - m[DS:0001]$	Register, Direct Memory
SUB DX, var	$DX := DX - m[DS:var]$	Register, Direct Memory
	 A variable declared in data segment (more later)	
INC [BX]	$m[DS:BX] := m[DS:BX] + 1$	Register Indirect
MOV AX, [BX+1]	$AX := m[DS:BX+1]$	Based Indirect
MOV AX, [BX+SI]	$AX := m[DS:BX+SI]$	Based-Index Indirect
MOV AX, [BX+SI+1]	$AX := m[DS:BX+SI+1]$	Based-Index Indirect with Displacement

Addressing Mode : (1) Register

Register mode allows a register to be specified as an operand

As a source operand : Instruction will **copy** register value

As a destination: write value to register

Example : MOV AX, DX

register addressing mode for
both dest and src

AX := DX

Contents of DX is copied to AX

Addressing Mode : (2) Immediate

- **Immediate mode** allows a **constant** to be specified as source
 - Constant value is *encoded* as part of the instruction
- **Example** : MOV AL, 5
 - Because AL is an 8-bit destination, the instruction encoding includes 8-bit value 05h
- **Example** : MOV AX, 5
 - Because AX is a 16-bit destination, the instruction encoding includes the 16-bit value 0005h
- **Question** : Is this possible ? MOV 4, BH ????

Addressing Mode : (3) Direct Memory

- **Direct memory mode** allows the **address offset** of a memory variable to be specified as an operand
 - A **constant address offset** is encoded as part of the instruction
 - The address offset is **static** : It must be known at assembly-time and remains constant through execution ... but the **contents** of that address may be **dynamic**
 - During execution, the address offset is *implicitly* combined with **DS**
- **Example** : `MOV AL, [5]`
 - Reads contents of byte at address **DS:0005**
- **Example** : `MOV var, AX`
 - Assumes a variable is declared in data segment
 - Write contents of word at address **DS:var**

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

Addressing Mode : (4a) Register Indirect

- A register holds the address offset of the operand
- The register can only be : **BX, SI, DI, BP**
- **DS** is default segment for: **BX, SI, DI**
- **SS** is default segment for **BP** (later!)
- Syntax : [register]

- Indirect Example:

MOV AX, [BX]
= MOV AX, **DS**: [BX]
⇒ AX := m[DS:BX]

Value in BX is used as address offset to a memory operand

CPU loads AX with contents of contents of that memory

Indirect addressing mode use registers as a pointer, which is a convenient way to handle an array! (later)

Addressing Mode : (4b) Indirect Indexed or Based

- Like register indirect, except you also specify a constant
e.g. [BX + constant]
- **During execution**, the **processor** uses a temporary register to calculate BX + constant
 - It then accesses memory addressed by BX + constant
- Restriction: may only use **BP, BX, SI or DI** ← same as register indirect
- Example :

```
MOV AX, [ BX +2 ]  
= MOV  AX, 2[BX]  
= MOV  AX, [BX][2]  
  
MOV AX, [ BX +var ]  
= MOV  AX, var[BX]
```

In both cases :
CPU computes address = Value in BX+2
CPU loads AX with of that address

Addressing Mode (4c) : Indirect Based-Indexed

- It is like indexed, except you use a second register instead of a constant
e.g. [BX + SI]
- **During execution**, the **processor** uses a temporary register to calculate sum of register values
 - It then accesses memory addressed by sum
- Restrictions:
 - one must be base register: BX (or BP ← later!)
 - one must be index register: SI or DI
 - The only legal forms:

Default DS	[BX + SI]	[BX + DI]	base = BX
Default SS	[BP + SI]	[BP + DI]	base = BP

Addressing Mode (4c) : Indirect Based-Indexed with Displacement

- It is like based-indexed mode, except includes a constant too
e.g. [BX + SI + constant]
- **During execution**, the **processor** uses a temporary register to calculate sum of values
 - It then accesses memory addressed by sum
- Restrictions: same as based mode
- MOV AX, [BX + SI + 2]
= MOV AX, [BX][SI+2]
= MOV AX, 2[BX+SI]
- MOV AX, [BX + SI + var]
= MOV AX, var[BX][SI]

In both cases :
CPU computes address – Value in
BX+SI+2
CPU loads AX with of that
address

Loading Registers with Addresses

- Before most instructions that use indirect addressing, the registers have to be loaded with address.
- Two alternatives :

`MOV BX, OFFSET W`



`LEA BX, W`

Functionally
equivalent!

- Both calculate and load the 16-bit effective address of a memory operand.

Segment Override

Required for exam : Restricted uses of registers

MOV [DX], AX

Marks will be deducted.

Recall : **DS** is default segment for: **BX, SI, DI**

SS is default segment for **BP** (later!)

MOV [BX], AL == MOV DS:[BX], AL

MOV [BP], AL == MOV SS:[BP], AL

At times, you may run out of registers and need to use either the index registers or the segment registers outside of their assigned default roles (eg. duplicating data structures),

MOV SS:[BX], AL

MOV ES:[BX], AL

MOV DS:[BP], AL

Operand Compatibility with Memory Operands

Clear and **unambiguous** Examples

MOV [0BCh], AX

MOV [BX], AL

- Why ? Because the other **REGISTER** operand determines **size**

Ambiguous Examples :

MOV [0BCh], 1

MOV [BX], 0

- Why ? The 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** [BX], 0

16-bit destination, no ambiguity

Assembler Tip About Operand Compatibility

W DW 0AA33h

16-bit memory src operand

...

MOV AL, W

8-bit register dest operand

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

Programs to do simple arithmetic

Problem : Write a **code fragment** to add the values of memory locations at DS:0, DS:01, and DS:02, and save the result at DS:10h.

Solution:

Step 1

$AL \leftarrow m[DS:00]$

Step 2

$AL \leftarrow AL + m[DS:01]$

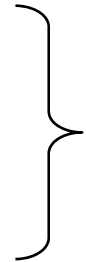
Step 3

$AL \leftarrow AL + m[DS:02]$

Step 4

$DS:10 \leftarrow AL$

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



DS is default!

Processor

AL=10h/30h/44h

Memory

DS:00	10h
DS:01	20h
DS:02	14h
DS:10h	FFh 44h

Learning how to read a reference manual on assembly instructions

We've seen that instructions often have restrictions – registers, addressing mode

For each instruction – whether in textbook or in processor's programming manual - the permitted operands and the side-effects are given

Thorne text, Appendix V.A “Instruction set summary”

ADD

Instruction Formats :

ADD reg, reg	ADD reg, immed
ADD mem, reg	ADD mem, immed
ADD reg, mem	ADD accum, immed

Is this permitted : ADD X, Y ?

Flag status affected: AF, PF, CF, SF, OF, ZF

Learning how to read a reference manual on assembly instructions

MOV

Instruction Formats :

MOV reg, reg

MOV mem, reg

MOV reg, mem

MOV reg16, segreg

MOV segreg, reg16

MOV reg,immed

MOV mem, immed

MOV mem16, segreg

MOV segreg, mem16

Flag status affected:
None

Segment registers (CS, DS, SS, ES) are 16-bit!

Question: Suppose we want to initialize DS with a constant value 45DFh ?

Understanding High-Level Control Flow at Machine Level

- Execution of data transfer/manipulation instruction advances **CS:IP** to **next** sequential instruction.
- Execution of control flow instructions changes address for fetch of next instruction.
- For example :
 - If condition is true, continue sequentially then **skip** to **next_statements**
 - If condition is false, **skip** to **false_statements**, then continue sequentially
 - Skip = control flow or jumps
- Conditions depend on the **status flags** (zero, carry, overflow, sign, parity) (ZF, CF, OF, SF, PF)

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 is also seen 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” are dependent on the status flags

Control Flow Implications of Segmented Memory Model

- The address of the **next** instruction is determined by **CS:IP**
- **Intra-segment** control flow : control stays **in** current code segment
 - Only need to modify **IP**
 - Need only supply (up to) **16-bit** of information
- **Inter-segment** control flow : control passes to an address **outside** of current code segment
 - Must modify both **CS and IP**
 - Must supply **32**-bits of information.
- To begin : We will only be concerned with **intrasegment** control flow (only modify **IP**)

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 0247H
0037H		
....
0247H		CS:247

Not
E9 47 02!

start of fetch: IP = 0034H IR = ????????

after fetch: IP = 0037H IR = E9 10 02

after execute: IP = 0247H IR = E9 10 02

(Little endian=0210h)

Simple Conditional Jumps

- Specify condition in terms of FLAG values set by the execution of the previous instruction

JZ Jump Zero: Jump if ZF = 1 else continue

JC Jump Carry : Jump if C=1 else continue

JO Jump Overflow: Jump if O=1 else continue

JS Jump Signed : Jump if S=1 else continue

JP Jump Parity : Jump if P=1 else continue

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

e.g. JNZ Jump Not Zero

Loop Example: MOV CX, 5

DoLoop:

. . .

SUB CX, 1

JNZ DoLoop

Comparison Instructions **CMP**

- Comparison instructions are used to simply set the flags

CMP dest, src

- Performs `dest - src` and sets FLAGS (but does **not** store result)

- **CMP Example :**

```
CMP    AL, 10
```

```
JZ     EqualToTen ; or JE !
```

```
...           ; Code for Not Equal
```

EqualToTen:

- It is often useful to think of combination as:

```
CMP dest, src
```

```
J*
```

Where the jump is taken if “dest - src” meets condition *

- in above example, jump is taken if `AL == 10`