Basic Assembly

Program Development

- Problem: convert ideas into executing program (binary image in memory)

- Program Development Process: tools to provide people-friendly way to do it.

- Tool chain:

1. Programming Language
   - **Syntax**: symbols + grammar for constructing statements (C=A+B)
   - **Semantics**: what is meant by statements → what happens upon execution (add A plus B and store the result in C)
   - **Assembly Language**: simplest readable language. One-to-one mapping to machine instructions.

Program Development

2. **Assembler**: Program to **convert** assembly language to object format
   - Object Code: program in machine format (i.e. binary)
   - May contain unresolved references (variables or functions)

3. **Linker**: program to **combine** object files into a single executable file
   – All references resolved

4. **Loader**: program to **load** executable files **into memory**. May initialize registers (e.g. IP) and starts it going.

5. **Debugger**: program that **loads** and **controls** execution of the program
   · start/stop execution, view and modify state variables

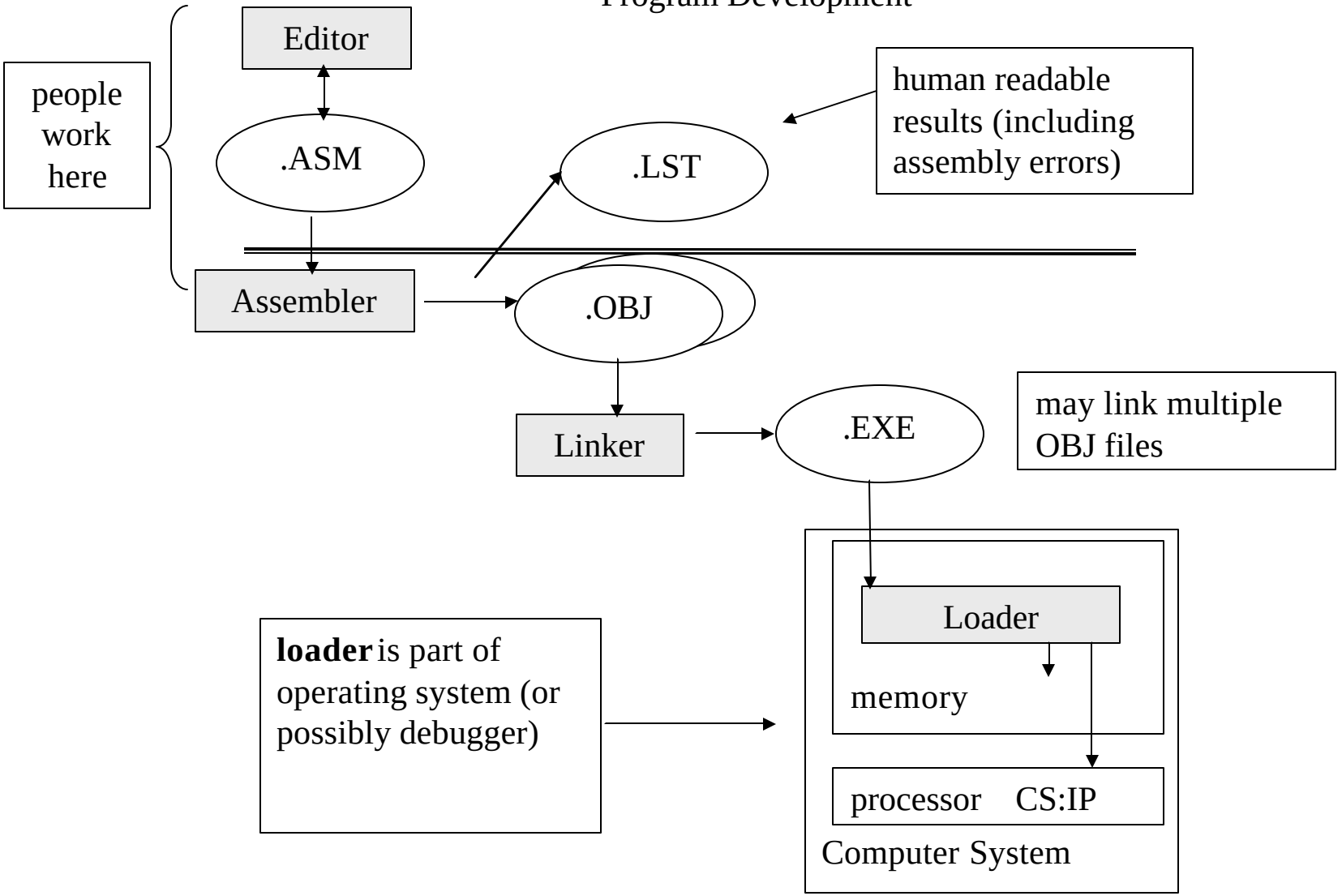
Program Development

- **Source Code**
  - Program written in assembly or high-level language

- **Object Code**
  - Output of assembler or compiler
  - Executable program in binary format (machine instructions)
  - Unsolved external references (Linker: solves these references and creates executable file)

- **Executable Code**
  - The complete executable program in binary format.

Program Development



Using TASM

A short Demo

Intel 8086 Assembly Language

- Assembly instructions: **readable** machine instructions (not binary)
  - **Mnemonic** encoding of instructions in a human-oriented short form
  - Examples

MOV	(move)
SUB	(subtract)
JMP	(jump)

- Instructions have two components:
  - operation (what is being done)
  - operands (data for operation), including varied addressing modes

- Translated instruction (in binary): encode **operation** and **operand** information

- Not only instructions: all aspects of a program
  - **constant** values
  - reserve memory to use for **variables**
  - **directives** to tools in development process

Intel 8086 Addressing Modes

- Variety of mechanisms to specify Operands
  - Simple modes: **immediate, register, direct**
  - More powerful: **indirect**

Addressing Mode : Immediate

- Immediate mode**: constant** specified as source

- **Example** : MOV AL, 5
  - AL: 8-bit destination; instruction encoding includes 8-bit value 05h

- **Example** : MOV AX, 5
  - AX: 16-bit destination; instruction encoding includes 16-bit value 0005h

- constant value **assembled into instruction** (hard-coded; **static value**)
- constant value **loaded into IR** as part of instruction
- constant value **obtained from IR** as instruction executed

Addressing Modes : 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

Instruction Syntax : Operand Compatibility

- For all instructions with two operands, the two operands must be compatible
  - 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.

Intel 8086 Instruction Set

1. **Data transfer**: copy data among variables (registers, memory and I/O ports)
   - Do not modify FLAGS

2. **Data manipulation**: modify variable values
   - Executed within the ALU data path
   - Modify the FLAGS

3. **Control-flow**: determine “next” instruction to execute
   - Allow non-sequential execution

Learning how to read a reference manual on assembly instructions

Instructions have restrictions – registers, addressing mode

Each instruction: permitted operands and the side-effects are given

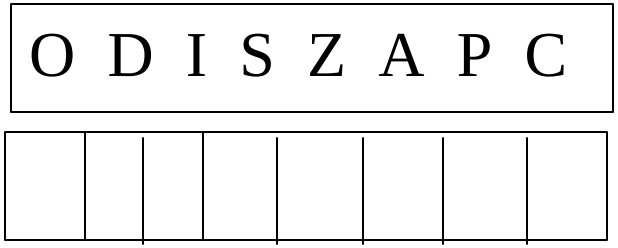
ADD

O	D	I	S	Z	A	P	C
*			*	*	*	*	*

Instruction Formats :

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

Learning how to read a reference manual on assembly instructions



MOV

Instruction Formats :

MOV reg, reg	MOV reg, immed
MOV mem, reg	MOV mem, immed
MOV reg, mem	MOV mem16, segreg
MOV reg16, segreg	MOV segreg, mem16
MOV segreg, reg16	

Data Transfer Instruction

- MOV (Move) Instruction

- Syntax: MOV dest , src
- Semantics: dest := src
  - Copy src value to dest state variable
  - register and memory operands only (I/O ??)

Data Manipulation Instructions

Use data to compute new values

- Modify variables to hold results
- Modify flags during on the results

set = 1, clear = 0

ZF = zero flag — set if **result = 0**

CF = carry flag — reflect **carry** value

SF = sign flag — set if **result < 0**

assumes 2's complement encoding!

OF = overflow flag

set if **signed overflow**

specific use of "**overflow**" – not the same as the general concept!

What about unsigned overflow ?

Data Manipulation : ADD

Syntax : ADD dest, src

Semantics : dest := dest + src (bitwise add)

- dest is both a source and destination operand

- FLAGS

  ZF := 1 if result = 0

  SF := 1 if msbit of result = 1 (sign = negative)

  CF := 1 if carry out of msbit

  OF := 1 if result overflowed signed capacity

Data Manipulation : ADD

Example: AL = 73H, then we execute:

ADD AL, 40H

73 H + **40** H = **B3**H carry?

results: **AL** := **B3**H (= 1011 0011 B)

ZF := 0 result ≠ 0

SF := 1 result is negative (signed)

CF := 0 (no carry out of msbit)

OF := 1 +v + +v = –v

Data Manipulation : SUB and CMP

Syntax : **SUB** dest, src

Semantics : dest := dest - src (bitwise subtract)

- ZF := 1 if result = 0
  SF := 1 if msbit of result = 1 (sign = negative)
  CF := 1 if borrow into msbit
  OF := 1 if result overflowed signed capacity

Syntax : **CMP** dest, src (Compare)

Semantics : Modifies FLAGS only to reflect dest - src

Data Manipulation : Logical Operations

Syntax : BOOLEAN dest, src

Semantics : dest = dest BOOLEAN src

where BOOLEAN = { AND, OR, XOR }

Example : AND AL, 80h

Example : OR Control, BH

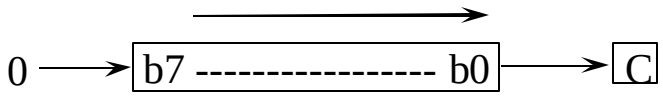
Example : XOR AX, AX

XOR AH, 0FFh

Data Manipulation : Shift

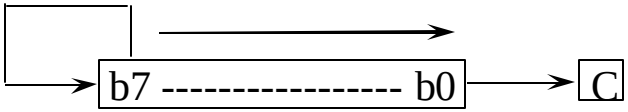
- Versions for : Left/Right and Arithmetic/Logical

Logical Shift Right



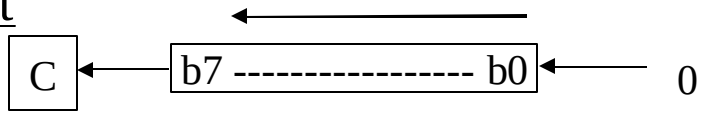

```
SHR     AL, 1
```

Arithmetic Shift Right




```
MOV     CL, 2
SAR     AL, CL
```

Logical or Arithmetic Shift Left



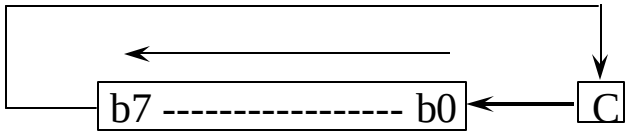

```
SHL     AL, 1
SAL     AL, 1
```

Data Manipulation : Rotate

- Versions for : Left/Right and with/out carry

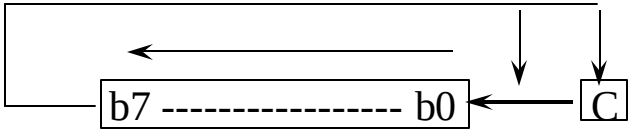
Rotate-Carry-Left

RCL AL, 1



Rotate Left

MOV CL, 4
ROL AL, CL



Data Manipulation : DIV

Unsigned Integer Division

- Syntax: **DIV src**
- Semantics: **accumulator / src** (integer division)
  - **src**: register, direct or indirect mode (**not** immediate)
  - 8-bit and 16-bit division depending on *src*

- 8-bit division: if src = 8-bit
  - **divide 16-bit value in AX by *src***
  - AL := AX ÷ src (unsigned divide)
    AH := AX mod src (unsigned modulus)
  - Flags **undefined** after DIV

16-bit dividend
8-bit divisor

Two 8-bit results

Data Manipulation: DIV

32-bit dividend
16-bit divisor

- 16-bit division : if src = 16-bit operand
  - **divide 32-bit value in DX:AX by *src***

Two 16-bit results

AX := DX:AX ÷ src (unsigned divide)
DX := DX:AX mod src (unsigned modulus)

  - **flags undefined after DIV**

Question: What if the result is too big to fit in destination?

- e.g.: AX ÷ 1 ?? AL = ??
- overflow trap – more later!

MASM/TASM Assembly Language Syntax - Constants

- **Decimal** value: default format – no "qualifier". Digits in 0 . . 9 (e.g. 12345)
- **Binary**: only 0's and 1's, **ends with 'B' or 'b' (**e.g. 10101110**b**)
- **Hexadecimal**:
  - starts with 0 .. 9; may include 0 . . 9, A .. F (a . . f)
  - **ends with 'H' or 'h'**
  - Requires **leading zero** if the first digit is A..F
  - e.g. 0FF**H**
- **String**: sequence of characters encoded as ASCII bytes:
  - enclose characters in **single quotes**
  - e.g. 'Hi Mom' – 6 bytes
  - character: **string with length = 1**
  - DOS Strings MUST ALWAYS end with '$'

Intel 8086 Assembly Language - Labels

- User-defined names. **Represent addresses**
  - programmer uses **logical names** (not addresses)
  - Assembler: translates names into binary addresses

- Used to identify addresses for:
  - **Control flow** – address of target
  - Memory **variables** – address where data is stored

- Identify the address ***offset***
  - Control flow: **combined with CS** (default)
  - Variables: **combined with DS** (default)

- Appear in 2 roles: definition & reference

Intel 8086 Assembly Language – Label Definition

- Represents **offset** of first allocated byte after definition
- Assembler: translates into exact address
- First **non-blank text** on a line
- Must **start with alpha** (A .. Z/a ..z). Then, alpha, numeric, '_'
- Careful with **reserved words** (e.g. MOV and other instructions)
- **Control flow** target: **must append ":"**

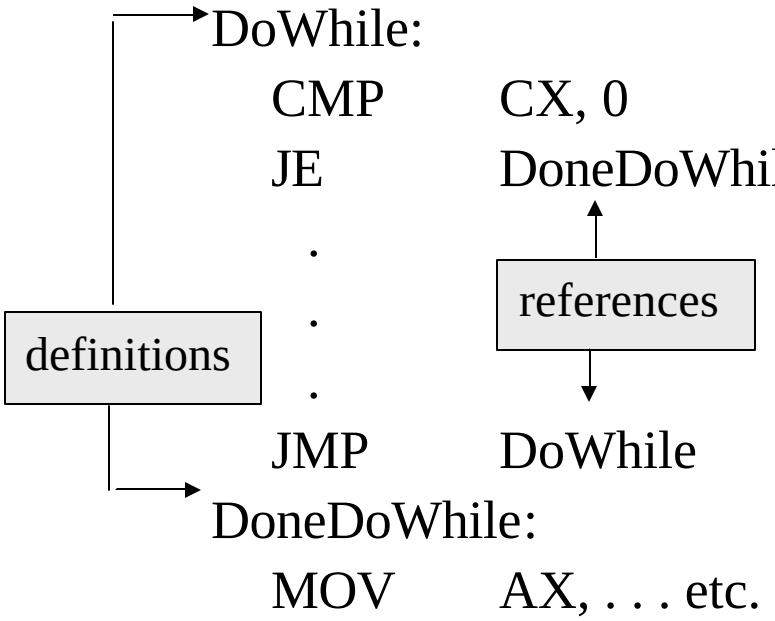
- Examples (Control Flow):

```
Continue:
L8R:
Out_2_Lunch:
 DoThis:        MOV    AX,   BX
```

DoThis represents address of first byte of the MOV instruction

Intel 8086 Assembly Language – Label Reference

- Used as **operand** (part of an instruction)
- Translated into address assigned by during the label definition
- Syntax: do not include " : "
- Control flow example: Assume CX contains loop counter



- target: labels
- assembler assigns addresses **AND** calculates offsets

Intel 8086 Assembly Language – Memory Declarations

- Memory Declarations
  - Reserves memory for variables
  - 2 common sizes on Intel 8086:

DB reserves a **byte of memory**

DW reserves a **word** (2 consecutive bytes) of memory

  - May also provide an (optional) initialization value as an operand

no “:” on variable name definitions

Intel 8086 Assembly Language – Memory Declarations

```
        DB          ; reserves one byte
X       DB          ; reserves one byte – label X
        ;           ; X represents the address of the byte
Y       DB    3     ; reserve one byte – label Y etc.
        ;           ; and initialize the byte to 3
        DW          ; reserve 2 consecutive bytes
Z       DW          ; reserves 2 bytes
W       DW  256     ; reserve 2 bytes – label W etc. - &
        ;           ; initialize the bytes to 256 (little endian)

HUH     DW  W       ; reserve 2 bytes – label etc.
        ;           ;       and initialize the bytes to
        ;           ;       contain the address of the
        ;           ;       variable W above
        DB    ‘C’   ; reserves 1 byte – initializes
        ;           ;   the byte to 43H
```

label Z represents the address of the first byte

Label definition

Label Reference

Understanding Program Development