Basic Assembly

Thorne : Section 3.5, Chapter 4, Chapter 5

(Irvine, Edition IV : Chapter 3)

Program Development

- Problem: must convert ideas (human thoughts) into an executing program (binary image in memory)

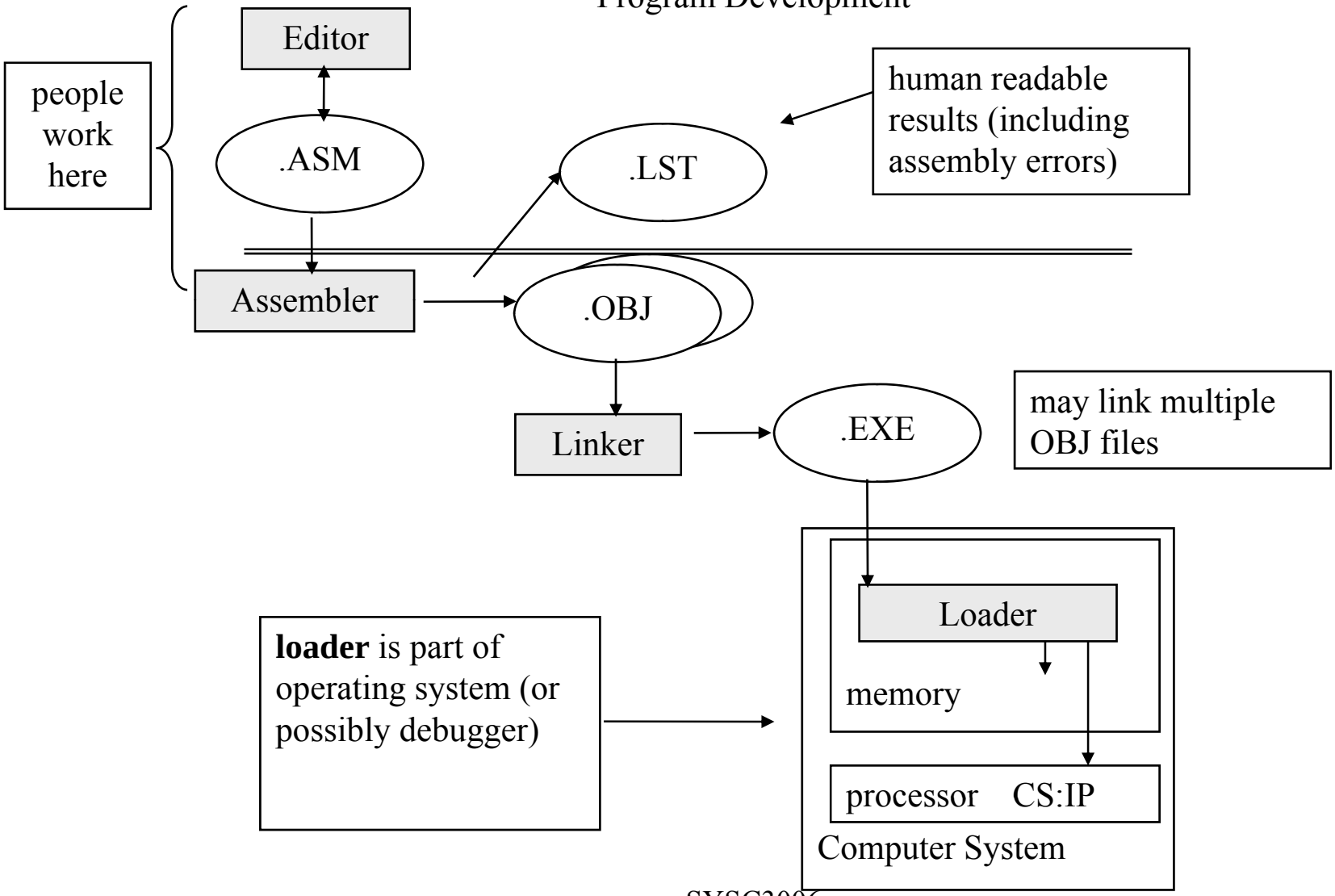
- The Program Development Process uses a set of tools to provide a people-friendly way to write programs and then convert them to the binary image.

1. Programming Language
   - Syntax: set of symbols + grammar rules for constructing statements using symbols
   - Semantics: what is meant by statements → ultimately, what happens upon execution
   - Assembly Language : A readable language that maps one-to-one to the machine instructions, to what operations are supported by the CPU.

Program Development

2. Assembler : A Program that converts the assembly language to object format
   - Object Code is the program in machine format (ie. binary)
   - May contain unresolved references (ie. file contains some or all of complete program)
3. Linker : A program that combines object files to create an single "executable" file
   – Major functional difference is that all references are resolved. (ie. Program contains all parts needed to run)
4. Loader : A program that loads executable files into memory, and may initialize some registers (e.g. IP) and starts it going.
5. Debugger : A program that loads but controls the execution of the program · To start/stop execution, to view and modify state variables

Program Development



Program Development

- Source Code
  - A program written in assembly or high-level language
- Object Code
  - The output of an assembler or compiler
  - The 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.

Intel 8086 Assembly Language

- Two assemblers are available for 80x86 Assembly Language
  - Free with copy of textbook or from web : Microsoft's assembler MASM
  - In the lab : Turbo assembler TASM

- Assembly language syntax must account for all aspects of a program and development process:
  - constant values
  - reserve memory to use for variables
  - write instructions: operations & operands
  - specify addressing modes
  - directives to tools in development process

Intel 8086 Assembly Language - Constants

- **Binary** values: consist of only 0's and 1's
  - ends with '**B**' or '**b**'
  - e.g. 10101110**b**
- **Hexadecimal** value: 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. **0**FFH (8-bit hex value)
- **Decimal** value:
  - default format – no "qualifier" extension
  - consists of digits in 0 . . 9
  - · e.g. 12345
- **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

- Labels are user-defined names that represent addresses
  - Labels let the programmer refer to addresses using logical names – no need for concern with exact hexadecimal values
  - Leave it to the assembler to decide exact addresses to use and to deal with hexadecimal addresses

- Labels are used to identify addresses for:
  - Control flow – identify address of target
  - Memory variables – identify address where data is stored

- To be specific, labels identify the address *offset*
  - For control flow labels, will be combined with **CS**
  - For memory variable labels, will be combined with **DS** (usually)
- Labels appear in 2 roles: definition & reference

Intel 8086 Assembly Language – Label Definition

- The label represents address offset of first allocated byte that follows definition
- Used by assembler to decide exact address
- must be first non-blank text on a line
- name must start with alpha A .. Z a ..z and then may contain: alpha, numeric, '_'
- If the label is a control flow target (other than procedure name – later) then must append ":"
- Cannot redefine reserved words, eg. MOV (an assembly instruction)
- Examples of **Control flow** label definitions :

  Continue:

  L8R:

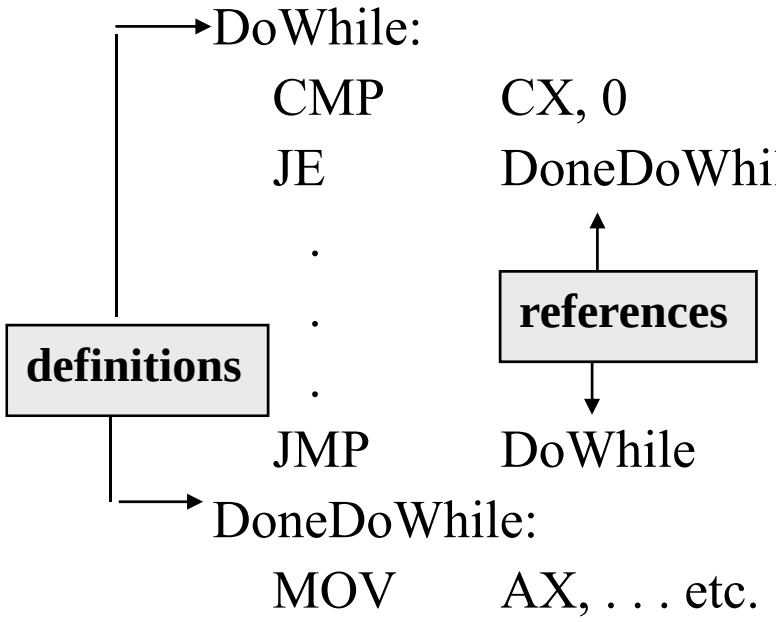
  Out_2_Lunch:

Example : DoThis: MOV AX, BX

DoThis represents address of first byte of the MOV instruction

Intel 8086 Assembly Language – Label Reference

- The reference is the use of label in an operand (as part of an instruction)
- Will refer to address assigned by assembler (by the label definition)
- Does not include " : "
- Control flow example: Assume CX contains loop counter



- specify target using label
- assembler assigns addresses **AND** calculates relative 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

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 – 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 '$'

Intel 8086 Assembly Language – Directives

Directives (pseudo-ops) are statements that are intended for other tools

- They are not assembled directly into instructions or memory declarations.
- No machine codes are created.
- No memories are allocated.

Directives used for

1. Identifying segments to the assembler
2. Terminating the assembler
3. Defining symbols for the assembler

A Skeleton Program

```
.8086
.model small
MYSTACK SEGMENT STACK
    db 100h dup(?)
MYSTACK ENDS
DATA SEGMENT
message db "Hello, world!",0dh,0ah,'$‘
DATA ENDS
CODE SEGMENT
main PROC                    ; procedure start
    ASSUME DS:DATA, SS:MYSTACK, CS:CODE
   mov  ax,@data             ; Initalize DS
   mov  ds,ax
   …
   mov  ax,4C00h             ; DOS Function call to exit
   int  21h
main endp                    ; procedure end
CODE ENDS                    ; code segment end
END main
```

proc – like a function definition.code

Comments after ;

END Directive

- It is a directive that is used by 2 tools: assembler & loader
- Assembler : Uses it to know when to stop reading from .ASM file
  - Any statements after END are ignored.
- It has an optional operand which if present must be a control flow label reference.
  - Loader will uses this label as the address of the first instruction to be executed:

Syntax : END [label-reference]

[ ] means optional

Program Directives

- The following directives tell the tools what type of machine the program will be running on
  - Different members of the 80x86 family have different instructions, although they all have the basic 8086 instruction set
  - Different members of the 80x86 family have different address spaces, allowing different sizes and configurations of programs to run

.8086 limits assembler 8086 processor instruction set

.model

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

SEGMENT, ENDS and ASSUME

- **Example**: Suppose that a program requires 20 bytes for data, 137 bytes for instructions (code), 100 bytes for stack
  - This could all fit in one segment (< 64K bytes)!
  - For optimal organization, the CS, DS and SS could all overlap
- **Example** : Suppose that a program requires 80K bytes for data, 47K bytes for instructions (code), 10K bytes for stack
  - The segments may partially overlap, … and we need two data segments!
  - Segment management is more complicated and more execution time.
- In this course, we will be writing **small** 8086 programs.
  - the actual amount of memory reserved for code will be less than 64K
    → One code segment
  - the actual amount reserved for data will be less than 64K
    → One date segment

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

Assembly Program : Hello World

```
.8086
.model small
.stack 100h
.data
message db "Hello, world!",0dh,0ah,'$'
.code
main proc
    mov  ax,@data                    ; Initialize DS
    mov  ds,ax
    mov  ah,9                        ; DOS Function call to print a message
    mov  dx,offset message
    int  21h
    mov  ax,4C00h                    ; DOS Function call to exit back to DOS
    int  21h
main endp
end main
```

A question of style !

Intel 8086 Assembly Language – Directives

. **EQU directive**

- EQU is a directive that allows you to define a symbolic name for a number (constant)
- That symbolic name can be used anywhere you want to use that number
- The assembler will replace all occurrences of the 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
```

```
MOV BX, VAL    ; Immediate
CMP AX, VAL    ; Immediate
MOV DX, X      ; Memory direct
```

Loading a Program

- Our program must be loaded into memory to be executed
  - Done by the loader (tool that is part of the operating system)
  - Loader decides which actual segments to be used
  - Loader initializes SS:SP (for stack use – later!) and CS:IP (to point to first instruction to be executed
  - Loader initializes DS but NOT to the data segment for the program at run time !
  - Instead, 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 : The processor uses contents of DS as the 16-bit segment value whenever access memory variables (including fetching operands during an instruction fetch)
  - The programmer only needs to supply the 16-bit offset in instructions
- DS must be initialized before ANY access to the data segment
  - Before any reference to a memory label.
- 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)

```
MOV DS, @data          ; MOV segreg, immed (NG)
```

  - Correct way

```
MOV     AX,  @data     ; Immediate mode
MOV     DS, AX         ; Register mode
```

Basic Coding Conventions

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

```
varName         DW      ?
MAX_LIMIT       EQU     5

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

Indentation :

1.Label is left-justified

2.Instructions are lined up one tab in.

3.Next instruction follows label.

Naming Convention :

4. Labels are lower case, with upper case for joined words
5. EQU symbols are UPPER_CASE
6. Keywords (mneumonics and registers) are upper-case

Basic Coding Conventions

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

; Test for equality
next:
    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.

Understanding Program Development

```
Microsoft (R) Macro Assembler Version 6.15.8803
                                        ; This program displays "Hello World"
                                .model small
                                .stack 100h
                                .data
0000                            
0000 48 65 6C 6C 6F 2C          message db "Hello, world!", 0dh, 0ah, '$'
       20 77 6F 72 6C 64
       21 0D 0A 24
0000                            
0000                            .code
0000                            main PROC
0000 B8 ---- R                          MOV     AX, @data
0003 8E D8                              MOV     DS, AX

0005 B4 09                              MOV     AH, 9
0007 BA 0000 R                          MOV     DX, OFFSET message
000A CD 21                              INT     21h

000C B8 4C00                            MOV     AX, 4C00h
000F CD 21                              INT     21h
0011                            main ENDP
                                END main
```

Offset Address

Machine Encoding

Assembly Programming

Initialized DS **MUST!**

DOS function to display message

Return to DOS control **MUST!**