Program Translation (Review)

- Suppose we want to execute the following statement in a *high-level programming language* (e.g.,C):
  - a = b + c;
- The C compiler is going to take that statement, and translate it into *assembly language* for a particular CPU architecture:

```
LOAD    20      ; get b (stored at 20)
ADD     21      ; add c (stored at 21)
STORE   22      ; store in a (at 22)
```

- The assembler will translate those assembly language statements into *machine language*:

```
100 10100
000 10101
101 10110
```


Programming in Assembly Language

- Common programming conventions:
  - One instruction per line, each containing:
    - Opcode
    - Comma-separated list of operands
  - A semicolon (";") denotes a comment, which lasts until the end of that line
  - A label is defined by an identifier followed by a colon (":") at the beginning of a line
  - A line may be empty, contain a label definition, contain an instruction, or contain a label followed by an instruction
  - Spacing can be done with spaces or tabs
  - Good style: labels, opcodes, operands, and comments should line up in columns
- Assembly language programs also contain *assembler directives* — instructions to the assembler

Equates

- Most assemblers allow the programmer to define symbolic constants

```
.equate     MAX,100
```

  - < *name*, *value* > added to symbol table
- After discussing equates, your book shows how they can be used to keep track of locations:

```
.equate     loc_x, 100
STORE       loc_x,R1
```

- This technique would make the programs we've seen up until now more readable
  - But — is this a good idea? What are the problems with this?

Reserving Space for Variables

- Most assemblers allow the programmer to reserve named memory locations for the purpose of storing variables

```
a:      .reserve  4     ; reserve 4 bytes
b:      .reserve  4     ; reserve 4 bytes

        LOAD  R1,a
        LOAD  R2,b
```

  - The assembler keeps track of the actual addresses, while the programmer simply refers to each by name
- There may also be a mechanism for reserving variables with initial values

```
counter:    .word   100
init_temp:  .word   40
```

- Read Section 5.1.3 in detail

Worksheet

- What is wrong with the following code segment? (There may be more than one type of error.)

```
start:
y:    .reserve  4        ; y needs 4 bytes
      LOAD      R2,y     ; store y in R2 (temp)
z:    .reserve  4        ; z needs 4 bytes
      LOAD      R3,z     ; store z in R3 (z)
      CLEAR     R4       ; clear R4 (x)
      JUMP      test     ; goto test
top:  ADD       R4,R4,#1  ; x = x + 1
test: SUB       R2,R2,R3  ; temp = temp–x
      BRGE      top      ; if (temp>=0) goto…
x:    .reserve  4        ; x needs 4 bytes
      STORE     x,R4     ; store the result
```


Assembler Segments

- A assembler program is typically organized into three segments:
  - *Text segment* — holds instructions
  - *Data segment* — holds initialized data (data reserved using .word, .byte, etc.)
  - *Bss segment* — holds uninitialized data (data reserved using .reserve)
- Text and data segments are present in object file, but only bss header is there
  - All segments are present when file is loaded into memory
- It is usually up to the programmer (!) to identify these segments, and to put the appropriate items in the proper segment

```
.text
.data
.bss
```


Translating an Assembly Language Program

- The *assembler* performs initial translation of an assembly language module into machine language
  - The *source* module is translated into an *object* module
- The *linker* links a set of assembled modules and libraries together to form a complete program (an executable file)
  - Resolves *external references* — symbols defined in one module and used in another
- The *loader* loads the completed program into memory where it can be executed
  - Must usually be capable of loading the program at an arbitrary location in memory (*relocation*)
    - Must adjust all addresses in the program

Building the Symbol Table

- As the assembler translates a program, it maintains a *symbol table*
  - < *label*, *address* >
  - When a label is defined, that label, along with the current value of the location counter, is stored in the symbol table
  - When a label is used, the assembler looks in the symbol table to find the corresponding address
- Location counter — keeps track of address of current instruction during assembly process
  - The location counter is not the Program Counter
    - Location counter is a variable used when the program is translated
    - Program Counter is a register used when the program is executed