Chapter 2

Instructions: Language of the Computer

Tutorial 4

- Character Data
- Byte / Halfword Operations
- String Copy Example
- Scanf and Printf Examples
- Linking and Loading

Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters

Character Data

- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, …
  - Most of the world's alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

Byte/Halfword Operations

- ARM byte load/store
  - String processing is a common case

```
LDRB r0, [sp,#0]  ; Read byte from source
STRB r0, [r10,#0] ; Write byte to destination
```

  - Sign extend to 32 bits

```
LDRSB ; Sign extends to fill leftmost 24 bits
```

Byte/Halfword Operations

- ARM halfword load/store

```
LDRH r0, [sp,#0]  ; Read halfword (16 bits) from source
STRH r0,[r12,#0] ; Write halfword (16 bits) to destination
```

  - Sign extend to 32 bits

```
LDRSH ; Sign extends to fill leftmost 16 bits
```

String Copy Example

- C code (naïve):
  - Null-terminated string

```
void strcpy (char x[], char y[]){
  int i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i++;
}
```

  - *Addresses of x, y in registers r0, r1*
  - *i in register r4*

String Copy Example

- ARM code:

```
strcpy:
    SUB sp,sp, #4          ; adjust stack for 1 item
    STR r4,[sp,#0]         ; save r4
    MOV r4,#0              ; i = 0
L1: ADD r2,r4,r1           ; addr of y[i] in r2
    LDRB  r3, [r2, #0]     ; r3 = y[i]
    ADD r12,r4,r0 ;        ; Addr of x[i] in r12
    STRB r3 [r12, #0]      ; x[i] = y[i]
    CMP r3,#0
    BEQ L2                 ; exit loop if y[i] == 0
    ADD r4,r4,#1           ; i = i + 1
    B L1                   ; next iteration of loop
L2: LDR r4, [sp,#0]        ; restore saved r4
    ADD sp,sp, #4          ; pop 1 item from stack
    MOV pc,lr              ; return
```

scanf and printf (example01.s)

- Read a number from **stdin** and print to the **stdout**

```
sub    sp, sp, #4          @allocate stack for input
ldr    r0, =formats        @scanf to get an integer
mov    r1, sp
bl     scanf               @scanf("%d",sp)
ldr    r1, [sp,#0]         @copy from stack to register
add    sp, sp, #4          @release stack
ldr    r0, =formatp        @format for printf
bl     printf              @printf
```

```
.data  @ data memory
formats: .asciz "%d“
formatp: .asciz "The number is %d\n“
```

- .asciz are for string literals. Assembler inserts \0 after the string

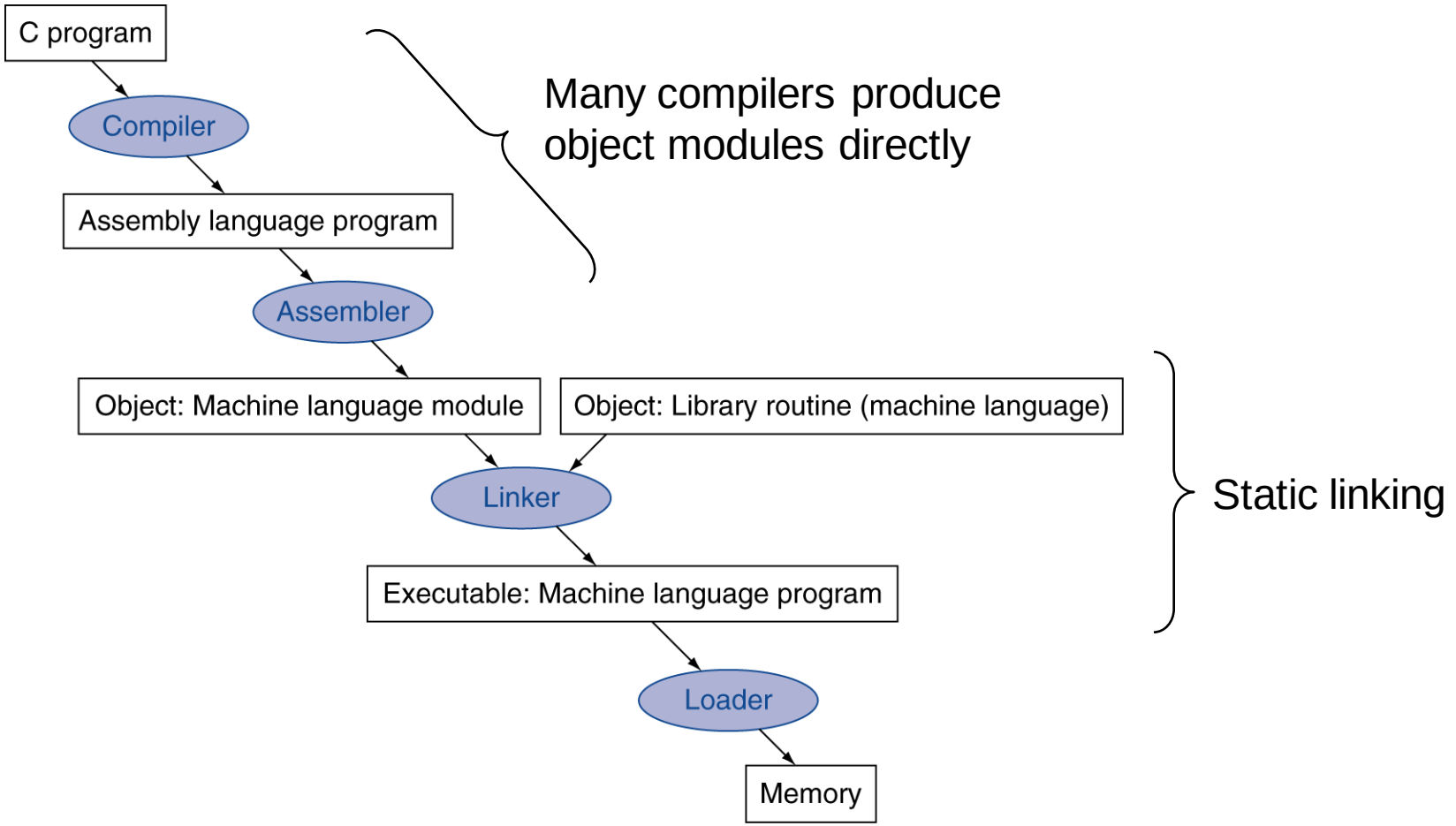
Some Examples to try

- Read two integers x and y, then Print x+y
  - Example02.s
- Read two integers x and y, thenPrint $x * 2^y$
  - Example03.s
- Write an ARM Assembly program to read two numbers and print whether they are equal or not
  - Example04.s

Some Examples

- Write an ARM Assembly program to read a number (N) and print numbers from 1 to N
  - Example05.s
- Write a function to find string length and call it from main
  - Example06.s

Translation and Startup



Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
  - Header: described contents of object module
  - Text segment: translated instructions
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code

Linking Object Modules

- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs
- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space

Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
     - Or set page table entries so they can be faulted in
  4. Set up arguments on stack
  5. Initialize registers
  6. Jump to startup routine
     - Copies arguments to r0, … and calls main
     - When main returns, startup terminates with exit system call

Dynamic Linking

- Only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions

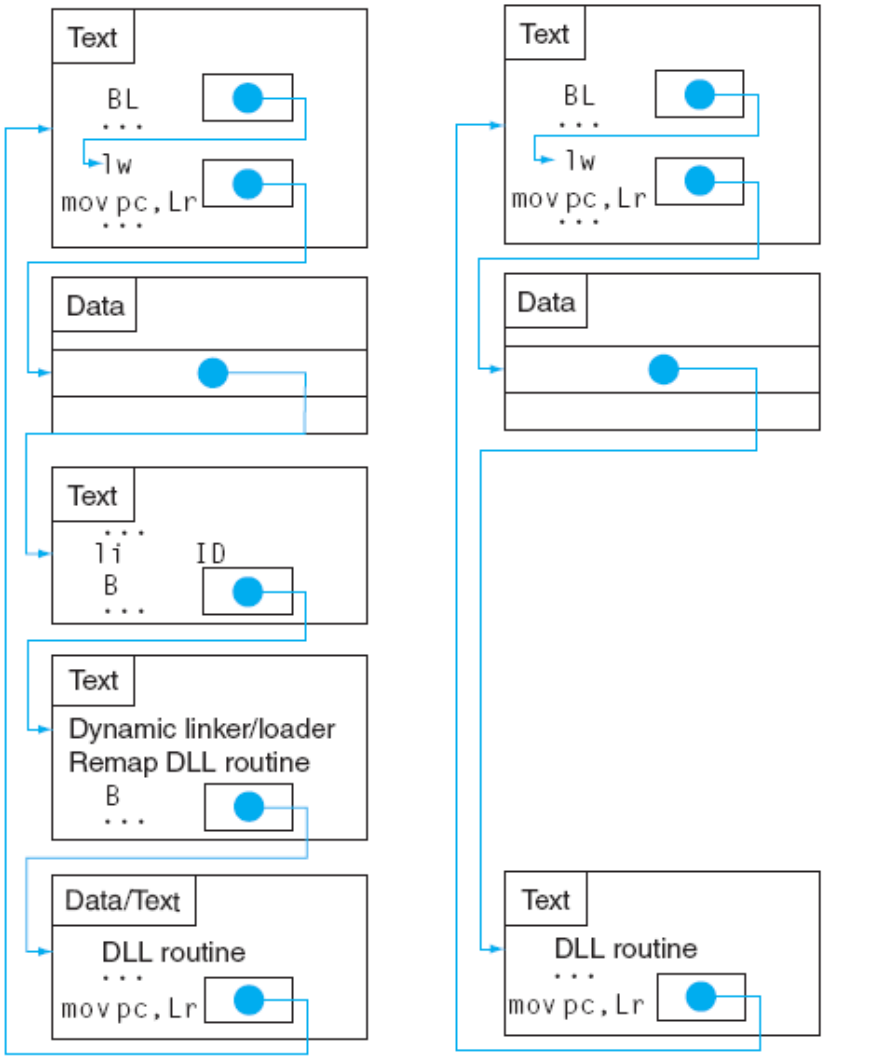
Lazy Linkage

Indirection table

Stub: Loads routine ID, Jump to linker/loader

Linker/loader code

Dynamically mapped code



a. First call to DLL routine

b. Subsequent calls to DLL routine