CPSC-240 Computer Organization and Assembly Language

Chapter 5

Tool Chain

Instructor: Yitsen Ku, Ph.D.
Department of Computer Science,
California State University, Fullerton, USA

CALIFORNIA STATE UNIVERSITY
FULLERTON™

Outline

- Assemble/Link/Load Overview
- Assembler
- Linker
- Assemble/Link Script
- Loader
- Debugger

Assemble/Link/Load Overview

Assemble/Link/Load Overview

- In broad terms, the assemble, link, and load process is how programmer written source files are converted into an executable program.
- The human readable source file is converted into an object file by the assembler. In the most basic form, the object file is converted into an executable file by the linker. The loader will load the executable file into memory.

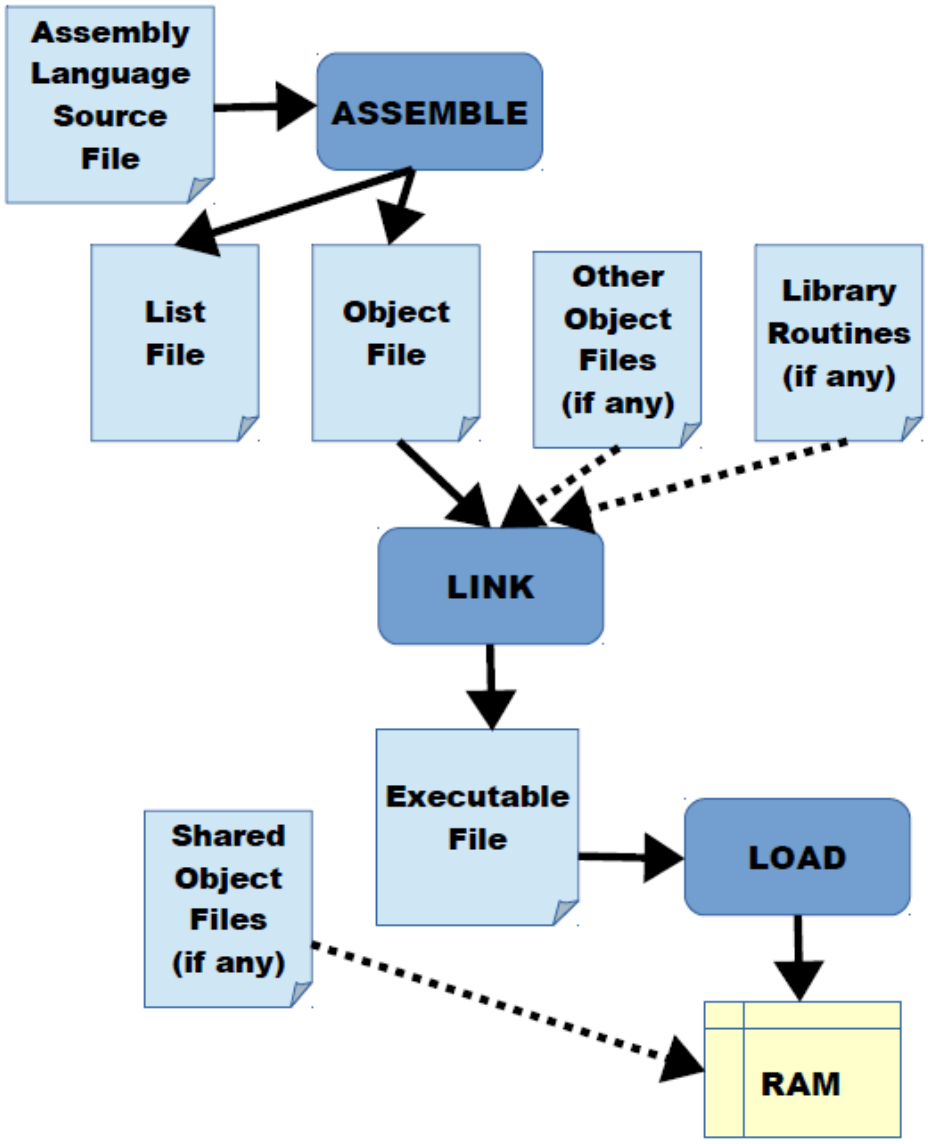

Illustration 6: Overview: Assemble, Link, Load

Assembler

Assembler

- The assembler is a program that will read an assembly language input file and convert the code into a machine language binary file. The input file is an assembly language source file containing assembly language instructions in human readable form. The machine language output is referred to as an object file. As part of this process, the comments are removed, and the variable names and labels are converted into appropriate addresses (as required by the CPU during execution).
- The assembler used in this text is the **yasm** assembler. Links to the **yasm** web site and documentation can be found in Chapter 1, Introduction

Assemble Commands

yasm -g dwarf2 -f elf64 example.asm -l example.lst

- The **-g dwarf226** option is used to inform the assembler to include debugging information in the final object file.
- The **-f elf64** informs the assembler to create the object file in the **ELF64** format which is appropriate for 64-bit, Linux-based systems.
- The **example.asm** is the name of the assembly language source file for input.
- The **–l example.lst** (dash lower-case letter L) informs the assembler to create a list file named example.lst.

List File

- In addition, the assembler is optionally capable of creating a list file.
- The list file shows the line number, the relative address, the machine language version of the instruction (including variable references), and the original source line. The list file can be useful when debugging.

Data section of List File

```
36 00000009 40660301 dVar1    dd 17000000 ; 0x40660301
37 0000000D 40548900 dVar2    dd 9000000
38 00000011 00000000 dResult dd 0
```

- On the first line, the **36** is the line number. The next number, **0x00000009**, is the relative address in the data area of where that variable will be stored. Since *dVar1* is a double-word, which requires four bytes, the address for the next variable is **0x0000000D**.
- The *dVar1* variable uses 4 bytes as addresses **0x00000009**, **0x0000000A**, **0x0000000B**, and **0x0000000C**.

Example 1 of List File

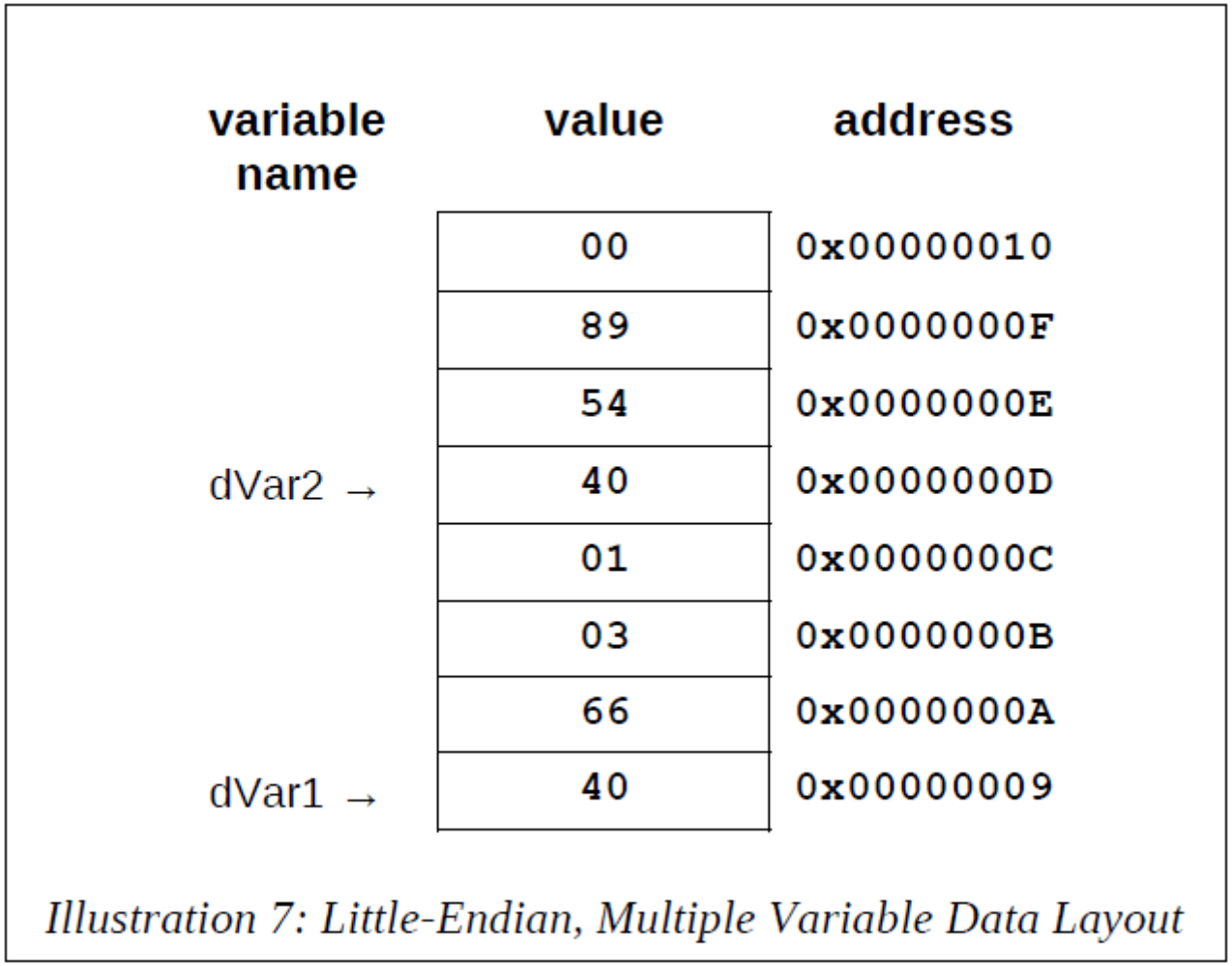

Illustration 7: Little-Endian, Multiple Variable Data Layout

Code section of List File

```
95                                      last:
96 0000005A 48C7C03C000000 mov rax, SYS_exit
97 00000061 48C7C300000000  mov rdi, EXIT_SUCCESS
98 00000068 0F05 syscall
```

- The numbers to the left are the line numbers. The next number, **0x0000005A**, is the relative address of the code.
- The next number, **0x48C7C03C000000**, is the machine language version of the instruction, in hex, that the CPU reads and understands. The rest of the line is the original assembly language source instruction.
- The label, **last:**, does not have a machine language instruction since the label is used to reference a specific address and is not an executable instruction.

Two-Pass Assembler

```
        mov rax, 0
        jmp skipRest
        ...
        ...
skipRest:
```

- This is referred to as a forward reference. If the assembler reads the assembly file one line at a time, it has not read the line where *skipRest* is defined. In fact, it does not even know for sure if *skipRest* is defined at all.
- This situation can be resolved by reading the assembly source file twice. The entire process is referred to as a two-pass assembler.

First Pass

- Some of the basic operations performed on the first pass include the following:
  - Create symbol table
  - Expand macros
  - Evaluate constant expressions
- A macro is a program element that is expanded into a set of programmer predefined instructions.
- A constant expression is an expression composed entirely of constants. Since the expression is constants only, it can be fully evaluated at assemble-time.

First Pass Example

- Assuming the constant BUFF is defined, the following instruction contains a constant expression;

mov rax, BUFF+5

- This type of constant expression is used commonly in large or complex programs.
- Addresses are assigned to all statements in the program. The symbol table is a listing or table of all the program symbols, variable names and program labels, and their respective addresses in the program.
- Some assembler directives are processed in the first pass.

Second Pass

- Some of the basic operations performed on the second pass include the following:
  - Final generation of code
  - Creation of list file (if requested)
  - Create object file
- The term code generation refers to the conversion of the programmer provided assembly language instruction into the CPU executable machine language instruction.
- Due to the one-to-one correspondence, this can be done for instructions that do not use symbols on either the first or second pass.

Assembler Directives

- Assembler directives are instructions to the assembler that direct the assembler to do something. This might be formatting or layout.
- These directives are not translated into instructions for the CPU.

Linker

Linker

- The linker will combine one or more object files into a single executable file. Additionally, any routines from user or system libraries are included as necessary.
- The appropriate linker command for the example program from the previous chapter is as follows:

ld -g -o example example.o

  - the **-o** is a dash lower-case letter O.
  - The **-g** option is used to inform the linker to include debugging information in the final executable file.
  - The **example** specifies to create the executable file named *example* (with no extension). If **the -o <fileName>** option is omitted, the output file is named a.out (by default).
  - The *example.o* is the name of the input object file read by the linker.

Linking Multiple Files

- In programming, large problems are typically solved by breaking them into smaller problems.
- Additional input object files, if any, would be listed, in order, separated with a space.

  ld -g -o example main.o funcs.o

- When using functions located in a different, external source file, any function or functions not in the current source file must be declared as **extern**.
- Variables, such as global variables, in other source files can be accessed by using the extern statement as well, however data is typically transferred as arguments of the function call.

Linking Process

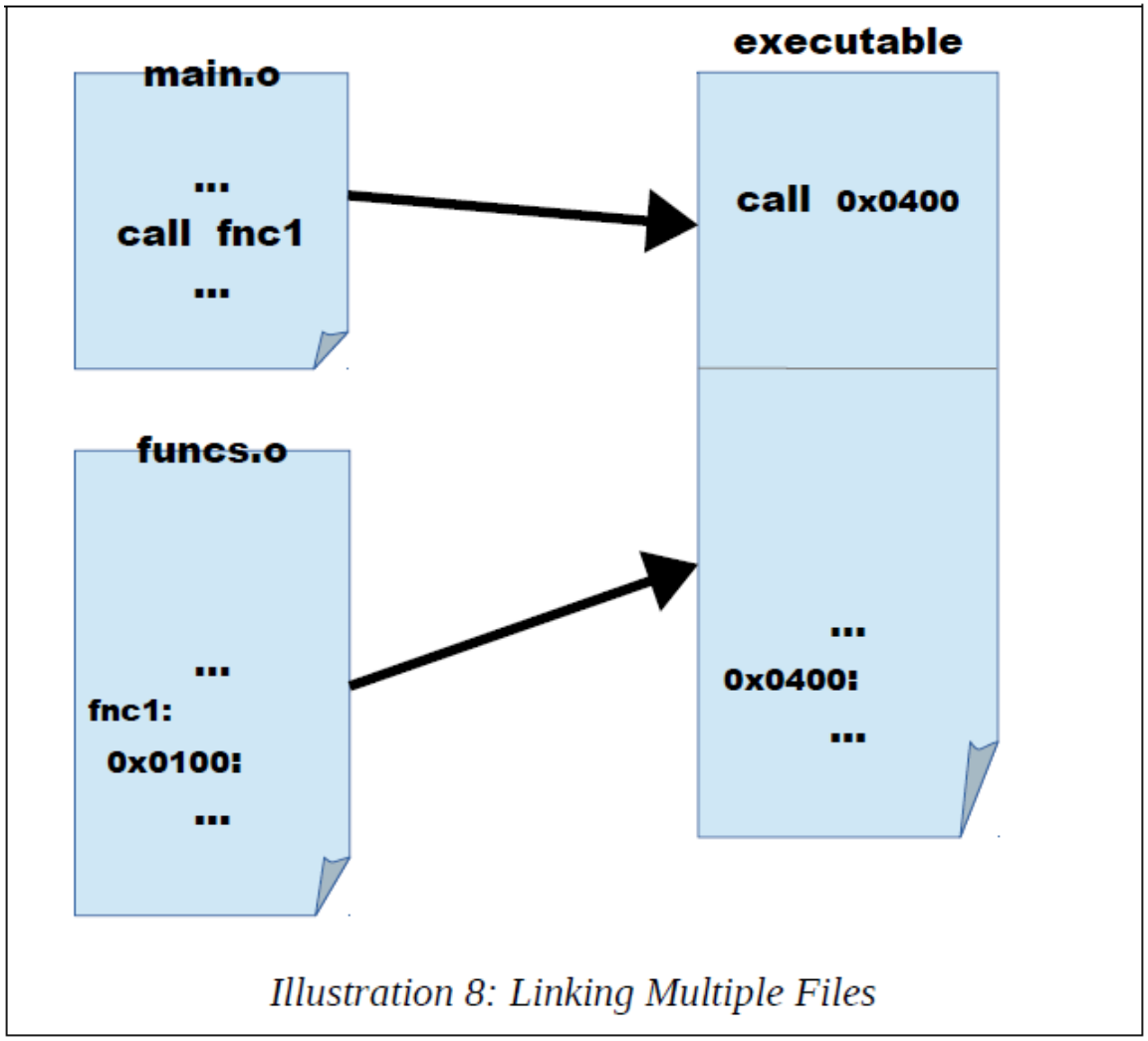

Illustration 8: Linking Multiple Files

Dynamic Linking

- The Linux operating system supports dynamic linking, which allows for postponing the resolution of some symbols until a program is being executed. The actual instructions are not placed in executable file and instead, if needed, resolved and accessed at run-time.
- In Linux/Unix, the dynamically linked object files typically a have **.so** (shared object) extension. In Windows, they have a **.dll** (dynamically linked library) extension. Further details of dynamic linking are outside the scope of this text.

Assemble/Link Script

Assemble/Link Script

```
#!/bin/bash
# Simple assemble/link script.
if [ -z $1 ]; then
echo "Usage: ./asm64 <asmMainFile> (no extension)"
exit
fi
# Verify no extensions were entered
if [ ! -e "$1.asm" ]; then
echo "Error, $1.asm not found."
echo "Note, do not enter file extensions."
exit
fi
# Compile, assemble, and link.
yasm -Worphan-labels -g dwarf2 -f elf64 $1.asm -l $1.lst
ld -g -o $1 $1.o
```

Assemble/Link Script

- When programming, it is often necessary to type the assemble and link commands many times with various different programs. Instead of typing the assemble (**yasm**) and link (**ld**) commands each time, it is possible to place them in a file, called a script file.
- The above script should be placed in a file (**asm64**).
- execute privilege will need to be added to the script file

chmod +x asm64

- use the script file to assemble the example

./asm64 example

Loader

Loader

- The loader is implicitly invoked by typing the program name. For example, on the previous example program, named *example*, the Linux command would be:

./example

- which will execute the file named *example* created via the previous steps (assemble and link).

Debugger

Debugger

- In the previous example, the program computed a series of calculations, but did not output any of the results. The debugger can be used to check the results. The executable file is created with the assemble and link command previously described and must include the -g option.
- The debugger used is the GNU DDD debugger which provides a visual front-end for the GNU command line debugger, **gdb**. The DDD web site and documentation are noted in the references section of Chapter 1, Introduction.

Lab Activity

Lab Activity

Given the following variable declarations and code fragment:

```
section .data
        text db "hello, World!", 10
section .text
        global _start
_start:
        mov rax, 1
        mov rdi, 1
        mov rsi, text
        mov rdx, 14
        syscall
        mov rax, 60
        mov rdi, 0
        syscall
```

Use YASM assembler to generate object code, use LD to generate execute file, and run the execute file to display the output result.

Thanks