Lab no 06: MIPS Assembly

The purpose of this Lab is to learn:

1) Translate a program from a high-level language into machine language.
2) Simulate and verify MIPS assembly programs and track MIPS registers.

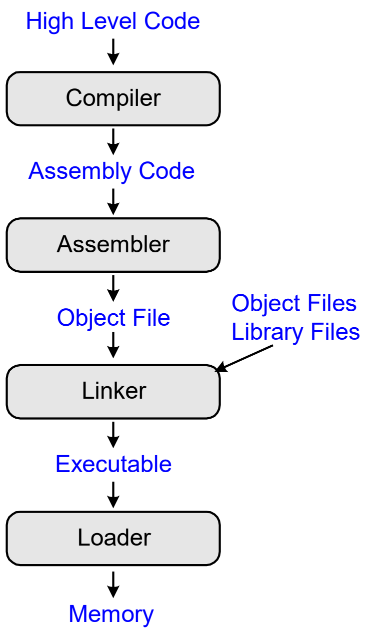
Parts: -

1. Introduction to translating and starting a program.
2. Case-study from high-level C language to machine code.
3. Hands-on MIPS assembly and Simulation of MIPS program.

Part 1. Introduction to translating and starting a program

Refer to section 6. 6. 2, 2nd Edition of "Digital Design and Computer Architecture" By David and Sarah Harris

The figure below shows the steps to translate a program from a high-level language into machine language and to start executing that program. First, the high-level code is compiled into assembly code. The assembly code is assembled into machine code in an object file. The linker combines the machine code with object code from libraries and other files to produce an entire executable program.



In practice, most compilers perform all three steps of compiling, assembling, and linking. Finally, the loader loads the program into memory and starts execution.

Step 1: Compilation

A compiler translates high-level code into assembly language. Code Example 6.30 shows a simple high-level program with three global variables and two functions, along with the assembly code produced by a typical compiler. The .data and .text keywords are assembler directives that indicate where the text and data segments begin. Labels are used for global variables f, g, and y. Their storage location will be determined by the assembler; for now, they are left as symbols in the code.

Code Example 6.30 COMPILING A HIGH-LEVEL PROGRAM

High-Level Code

```
int f, g, y; // global variables

int main(void)
{
  f = 2;
  g = 3;
  y = sum(f, g);
  return y;
}

int sum(int a, int b) {
  return (a + b);
}
```

MIPS Assembly Code

```
.data
f:
g:
y:

.text
main:
  addi $sp, $sp, −4   # make stack frame
  sw   $ra, 0($sp)    # store $ra on stack
  addi $a0, $0, 2     # $a0 = 2
  sw   $a0, f         # f = 2
  addi $a1, $0, 3     # $a1 = 3
  sw   $a1, g         # g = 3
  jal  sum            # call sum function
  sw   $v0, y         # y = sum(f, g)
  lw   $ra, 0($sp)    # restore $ra from stack
  addi $sp, $sp, 4    # restore stack pointer
  jr   $ra            # return to operating system

sum:
  add  $v0, $a0, $a1  # $v0 = a + b
  jr   $ra            # return to caller
```

Step 2: Assembling

The assembler turns the assembly language code into an object file containing machine language code. The assembler makes **two passes** through the assembly code.

On the first pass, the assembler assigns instruction addresses and finds all the symbols, such as labels and global variable names. The code after the first assembler pass is shown here.

```
0x00400000 main: addi $sp, $sp, −4
0x00400004       sw   $ra, 0($sp)
0x00400008       addi $a0, $0, 2
0x0040000C       sw   $a0, f
0x00400010       addi $a1, $0, 3
0x00400014       sw   $a1, g
0x00400018       jal  sum
0x0040001C       sw   $v0, y
0x00400020       lw   $ra, 0($sp)
0x00400024       addi $sp, $sp, 4
0x00400028       jr   $ra
0x0040002C sum:  add  $v0, $a0, $a1
0x00400030       jr   $ra
```

The names and addresses of the symbols are kept in a symbol table, as shown in Table 6.4 for this code. The symbol addresses are filled in after the first pass, when the addresses of labels are known. Global variables are assigned storage locations in the global data segment of memory, starting at memory address 0x10000000.

Table 6.4 Symbol table

Symbol	Address
f	0x10000000
g	0x10000004
y	0x10000008
main	0x00400000
sum	0x0040002C

On the second pass, the assembler produces the machine language code. Addresses for the global variables and labels are taken from the symbol table. The machine language code and symbol table are stored in the object file.

Step 2: Linking

Most large programs contain more than one file. In our example, there is only one object file, so no relocation is necessary. Figure 6.33 shows the executable file. It has three sections:

- The executable file header, the text segment, and the data segment. The executable file header reports the text size (code size) and data size (amount of globally declared data). Both are given in units of bytes.
- The text segment gives the instructions in the order that they are stored in memory.
- The data segment gives the address of each global variable.

The figure shows the instructions in human-readable format next to the machine code for ease of interpretation, but the executable file includes only machine instructions.

Executable file header	Text Size	Data Size	
	0x34 (52 bytes)	0xC (12 bytes)	
Text segment	**Address**	**Instruction**	
	0x00400000	0x23BDFFFC	addi $sp, $sp, –4
	0x00400004	0xAFBF0000	sw $ra, 0($sp)
	0x00400008	0x20040002	addi $a0, $0, 2
	0x0040000C	0xAF848000	sw $a0, 0x8000($gp)
	0x00400010	0x20050003	addi $a1, $0, 3
	0x00400014	0xAF858004	sw $a1, 0x8004($gp)
	0x00400018	0x0C10000B	jal 0x0040002C
	0x0040001C	0xAF828008	sw $v0, 0x8008($gp)
	0x00400020	0x8FBF0000	lw $ra, 0($sp)
	0x00400024	0x23BD0004	addi $sp, $sp, –4
	0x00400028	0x03E00008	jr $ra
	0x0040002C	0x00851020	add $v0, $a0, $a1
	0x00400030	0x03E00008	jr $ra
Data segment	**Address**	**Data**	
	0x10000000	f	
	0x10000004	g	
	0x10000008	y	

Figure 1. Executable file

Step 4: Loading

The operating system loads a program by reading the text segment of the executable file from a storage device (usually the hard disk) into the text segment of memory. Figure 6.34 shows the memory map at the beginning of program execution.

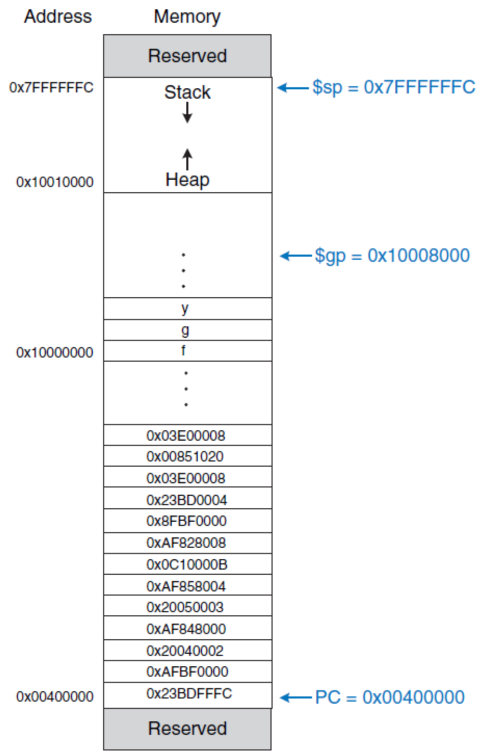


Figure 6.34 Executable loaded in memory

Part 2. Case-study from high-level C language to machine code

Exercise 6.11 Each number in the Fibonacci series is the sum of the previous two numbers. Table 6.16 lists the first few numbers in the series, fib(n).

Table 6.16 Fibonacci series

n	1	2	3	4	5	6	7	8	9	10	11	...
$fib(n)$	1	1	2	3	5	8	13	21	34	55	89	...

(1) Write a function called fib in a high-level language that returns the Fibonacci number for any nonnegative value of n. Hint: You probably will want to use a loop. Clearly comment your code.

(2) Convert the high-level function of the part (2) into MIPS assembly code. Add comments after every line of code that explain clearly what it does.

(3) Convert the MIPS assembly code of the part (3) into machine code.

(4) generate the executable file, as shown in ***Figure 1***.

Part 3. Hands-on MIPS assembly

In the lecture, we study the basics of C language including the conditions statements like if and if/else, the loops statements like while/for, and arrays. Refer to the for-loop example in the lecture below,

```
// C Code
// add the powers of 2 from 1 to 100
int sum = 0;
int i;
for (i=1; i < 101; i = i*2) {
  sum = sum + i;
}
```

The Translation of C code above into MIPS assembly is as follow:

```
# MIPS assembly code
# $s0 = i, $s1 = sum
       addi $s1, $0, 0
       addi $s0, $0, 1
       addi $t0, $0, 101
loop:  slt  $t1, $s0, $t0
       beq  $t1, $0, done
       add  $s1, $s1, $s0
       sll  $s0, $s0, 1
       j    loop
done:
```

Simulate the MIPS assembly code using the WeMIPS simulator:

- Go to: http://rivoire.cs.sonoma.edu/cs351/wemips/
- Copy the above assembly program into the code window on the left.
- Debug the program step by step and check the value of the registers.

Note: Replace $0 with $zero