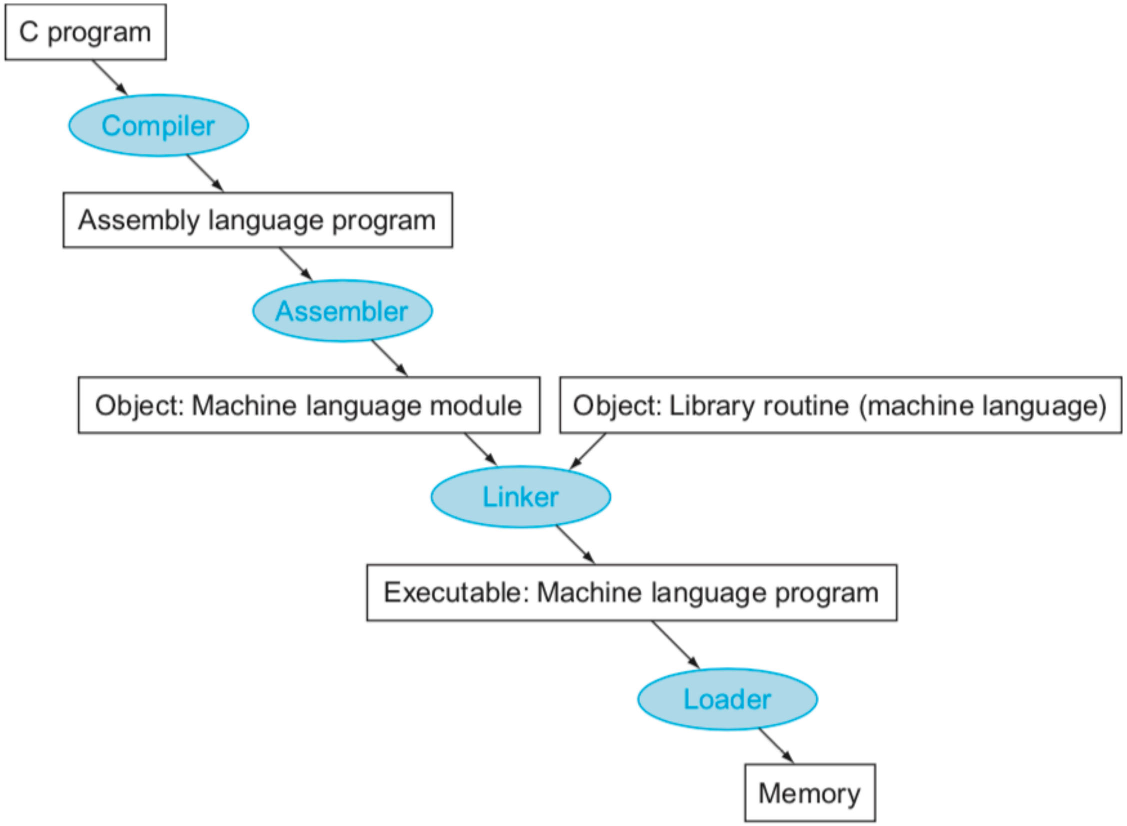
Chapter 2

Instructions: Language of the Computer

A translation hierarchy for C



Compiler

- **Transforms the C program into an assembly language program**, a symbolic form of what the machine understands
- High level language programs take many fewer lines of code than assembly language, so <u>programer productivity is much higher</u>

Assembler

- Since assembly language is an interface to higher-level software, the **assembler can also treat common variations of machine language instructions** as if they were instructions in their own right. (pseudoinstruction)
- **MIPS assembler accepts this instruction** even though it is **not found in the MIPS architecture**

```
move $t0,$t1       # register $t0 gets register $t1
```

- The assembler converts this assembly language instruction into the machine language equivalent of the following instruction:

```
add $t0,$zero,$t1 # register $t0 gets 0 + register $t1
```

- **Assemblers will also accept numbers in a variety of bases. MIPS assemblers use hexadecimal.**

Assembler

- **Primary task of assembler: Transfer assembly code to machine code**
- The assembler turns the assembly language program into an *object file*, which is a combination of **machine language instructions**, **data**, and **information needed to place instructions properly in memory**.

- To produce the binary version of each instruction in the assembly language program, the **assembler must determine the addresses corresponding to all labels.**
- **Assemblers keep track of labels** used in branches and data transfer instructions in a **symbol table**.
- **Symbol table:** A table that matches names of labels to the addresses of the memory words that instructions occupy

Assembler

- The object file for UNIX systems typically contains six distinct pieces:

1. **The *object file header*** describes the size and position of the other pieces of the object file.
2. The ***text segment*** contains the machine language code.
3. The ***static data segment*** contains data allocated for the life of the program.
4. The ***relocation information* identifies** instructions and data words that depend on absolute addresses when the program is loaded into memory.
5. The ***symbol table*** contains the remaining labels that are not defined, such as external references.

Linker

- **Link editor** or **linker**, which takes **all the independently assembled machine language programs** and **“stitches” them together.**

- There are three steps for the linker:

1. Place code and data modules symbolically in memory.
2. Determine the addresses of data and instruction labels.
3. Patch both the internal and external references.

The linker produces an executable file that can be run on a computer.

Example- Linking two Object Files below

Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	`lw $a0, 0($gp)`	
	4	`jal 0`	
	…	…	
Data segment	0	(X)	
	…	…	
Relocation information	Address	Instruction type	Dependency
	0	`lw`	X
	4	`jal`	B
Symbol table	Label	Address	
	X	–	
	B	–	
Object file header			
	Name	Procedure B	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	`sw $a1, 0($gp)`	
	4	`jal 0`	
	…	…	
Data segment	0	(Y)	
	…	…	
Relocation information	Address	Instruction type	Dependency
	0	`sw`	Y
	4	`jal`	A
Symbol table	Label	Address	
	Y	–	
	A	–	

Example- Linking two Object Files below

Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	`lw $a0, 0($gp)`	
	4	`jal 0`	
	…	…	
Data segment	0	(X)	
	…	…	
Relocation information	Address	Instruction type	Dependency
	0	`lw`	X
	4	`jal`	B
Symbol table	Label	Address	
	X	–	
	B	–	
Object file header			
	Name	Procedure B	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	`sw $a1, 0($gp)`	
	4	`jal 0`	
	…	…	
Data segment	0	(Y)	
	…	…	
Relocation information	Address	Instruction type	Dependency
	0	`sw`	Y
	4	`jal`	A
Symbol table	Label	Address	
	Y	–	
	A	–	

Note that in the object files we have highlighted the **addresses and symbols** that **must be updated in the link process**

The instructions that refer to the addresses of procedures A and B **and** the instructions that refer to the addresses of data words X and Y.

Linking Object Files

- Procedure A needs to find the address for the variable labeled X to put in the load instruction and to find the address of procedure B to place in the jal instruction.

- Procedure B needs the address of the variable labeled Y for the store instruction and the address of procedure A for its jal instruction.

Linking Object Files

- we know that the **text segment starts at address 40 0000$_{hex}$** and **the data segment at 1000 0000$_{hex}$**
- The text of procedure A is placed at the first address and its data at the second. The object file header for procedure A says that its text is 100$_{hex}$ bytes and its data is 20$_{hex}$ bytes, so the starting address for procedure B text is 40 0100$_{hex}$, and its data starts at 1000 0020$_{hex}$.

Executable file header		
	Text size	300$_{hex}$
	Data size	50$_{hex}$
Text segment	Address	Instruction
	0040 0000$_{hex}$	lw \$a0, 8000$_{hex}$(\$gp)
	0040 0004$_{hex}$	jal 40 0100$_{hex}$
	...	...
	0040 0100$_{hex}$	sw \$a1, 8020$_{hex}$(\$gp)
	0040 0104$_{hex}$	jal 40 0000$_{hex}$
	...	...
Data segment	Address	
	1000 0000$_{hex}$	(X)
	...	...
	1000 0020$_{hex}$	(Y)
	...	...

Linking Object Files

- **Now the linker updates the address fields of the instructions.**
- **The jals are easy because they use pseudo direct addressing.**
  - The jals are easy because they use pseudodirect addressing. The jal at address 40 0004_{hex} gets 40 0100_{hex} (the address of procedure B) in its address field
  - the jal at 400104_{hex} gets 400000_{hex} (the address of procedure A) in its address field.
- **The load and store addresses are harder because they are relative to a base register.**
  - $gp is initialized to 1000 8000_{hex}.
  - To get the address 1000 0000_{hex} (the address of word X), we place 8000_{hex} in the address field of lw at address 40 0000_{hex}.
  - Similarly, we place 8020_{hex} in the address field of sw at address 400100_{hex} to get the address10000020_{hex} (the address of word Y).

Loader

1. Reads the executable file header to determine size of the text and data segments.
2. Creates an address space large enough for the text and data.
3. Copies the instructions and data from the executable file into memory.
4. Copies the parameters (if any) to the main program onto the stack.
5. Initializes the machine registers and sets the stack pointer to the first free location.
6. Jumps to a start-up routine that copies the parameters into the argument registers and calls the main routine of the program. When the main routine returns, the start-up routine terminates the program with an exit system call.

Example- bubble sort

```
void swap(int v[], int k)
 {
   int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
 }
```

A C procedure that swaps two locations in memory. This subsection uses this procedure in a sorting example.

Procedure body

```
swap: sll    $t1, $a1, 2          # reg $t1 = k * 4
      add    $t1, $a0, $t1        # reg $t1 = v + (k * 4)
                                  # reg $t1 has the address of v[k]
      lw     $t0, 0($t1)          # reg $t0 (temp) = v[k]
      lw     $t2, 4($t1)          # reg $t2 = v[k + 1]
                                  # refers to next element of v
      sw     $t2, 0($t1)          # v[k] = reg $t2
      sw     $t0, 4($t1)          # v[k+1] = reg $t0 (temp)
```

Procedure return

```
      jr     $ra                  # return to calling routine
```

MIPS assembly code of the procedure swap

```
void sort (int v[], int n)
{
   int i, j;
   for (i = 0; i < n; i += 1) {
        for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j =1) {
            swap(v,j);
        }
   }
}
```

A C procedure that performs a sort on the array v.

Example- bubble sort

Saving registers			
	`sort:`	`addi`	`$sp,$sp, -20 # make room on stack for 5 registers`
		`sw`	`$ra, 16($sp)# save $ra on stack`
		`sw`	`$s3,12($sp) # save $s3 on stack`
		`sw`	`$s2, 8($sp)# save $s2 on stack`
		`sw`	`$s1, 4($sp)# save $s1 on stack`
		`sw`	`$s0, 0($sp)# save $s0 on stack`
Procedure body			
Move parameters		`move`	`$s2, $a0 # copy parameter $a0 into $s2 (save $a0)`
		`move`	`$s3, $a1 # copy parameter $a1 into $s3 (save $a1)`
Outer loop		`move`	`$s0, $zero# i = 0`
	`for1tst:`	`slt`	`$t0, $s0,$s3 #reg$t0=0if$s0Š$s3(iŠn)`
		`beq`	`$t0, $zero, exit1# go to exit1 if $s0 Š $s3 (i Š n)`
Inner loop		`addi`	`$s1, $s0, -1# j = i - 1`
	`for2tst:`	`slti`	`$t0, $s1,0 #reg$t0=1if$s1<0(j<0)`
		`bne`	`$t0, $zero, exit2# go to exit2 if $s1 < 0 (j < 0)`
		`sll`	`$t1, $s1, 2# reg $t1 = j * 4`
		`add`	`$t2, $s2, $t1# reg $t2 = v + (j * 4)`
		`lw`	`$t3, 0($t2)# reg $t3 = v[j]`
		`lw`	`$t4, 4($t2)# reg $t4 = v[j + 1]`
		`slt`	`$t0, $t4, $t3 # reg $t0 = 0 if $t4 Š $t3`
		`beq`	`$t0, $zero, exit2# go to exit2 if $t4 Š $t3`
Pass parameters and call		`move`	`$a0, $s2 # 1st parameter of swap is v (old $a0)`
		`move`	`$a1, $s1 # 2nd parameter of swap is j`
		`jal`	`swap # swap code shown in Figure 2.25`
Inner loop		`addi`	`$s1, $s1, -1# j -= 1`
		`j`	`for2tst # jump to test of inner loop`
Outer loop	`exit2:`	`addi`	`$s0, $s0, 1 # i += 1`
		`j`	`for1tst # jump to test of outer loop`
Restoring registers			
	`exit1:`	`lw`	`$s0, 0($sp) # restore $s0 from stack`
		`lw`	`$s1, 4($sp)# restore $s1 from stack`
		`lw`	`$s2, 8($sp)# restore $s2 from stack`
		`lw`	`$s3,12($sp) # restore $s3 from stack`
		`lw`	`$ra,16($sp) # restore $ra from stack`
		`addi`	`$sp,$sp, 20 # restore stack pointer`
Procedure return			
		`jr`	`$ra # return to calling routine`

MIPS assembly version of procedure `sort`

Reading assignment

- Read 2.11,2.12 and 2.13 of the text book