CS61C - Machine Structures

Lecture 11 - Starting a Program

October 4, 2000

David Patterson

http://www-inst.eecs.berkeley.edu/~cs61c/

Review (1/2)

- IEEE 754 Floating Point Standard: Kahan pack as much in as could get away with
  - +/- infinity, Not-a-Number (Nan), Denorms
  - 4 rounding modes
- **Stored Program Concept**: Both data and actual code (instructions) are stored in the same memory.
- **Type is not associated with data**, bits have no meaning unless given in context

Things to Remember (1/2)

- **Machine Language Instruction**: 32 bits representing a single MIPS instruction

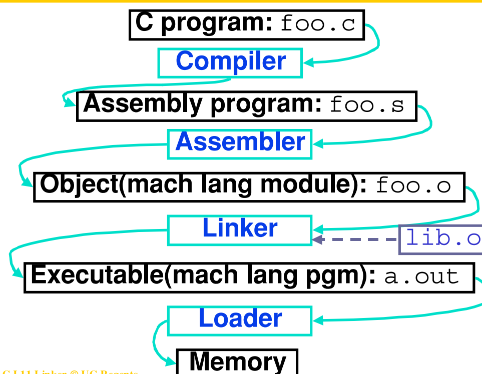
R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	immediate		
J	opcode	target address				

- Instructions formats kept similar
- Branches, Jumps optimized for greater branch distance and hence strange
- New Logical, Shift Instructions: `and, andi, or, ori,sll, srl, sra`

Outline

- Compiler
- Assembler
- Linker
- Loader
- Example

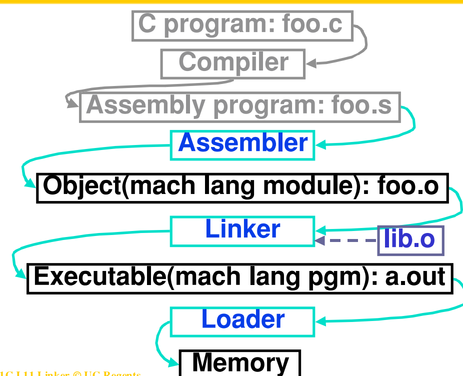
Steps to Starting a Program



Compiler

- Input: High-Level Language Code (e.g., C, Java)
- Output: Assembly Language Code (e.g., MIPS)
- Note: Output *may* contain pseudoinstructions
- **Pseudoinstructions**: instructions that assembler understands but not in machine (e.g., HW#4); For example:
- `mov $s1, $s2` = `or $s1, $s2, $zero`

Where Are We Now?



Assembler

° Reads and Uses **Directives**

° Replace Pseudoinstructions

° Produce Machine Language

° Creates **Object File**

Assembler Directives (p. A-51 to A-53)

° Give directions to assembler, but do not produce machine instructions

- `.text`: Subsequent items put in user text segment
- `.data`: Subsequent items put in user data segment
- `.globl sym`: declares `sym` global and can be referenced from other files
- `.asciiz str`: Store the string `str` in memory and null-terminate it
- `.word w1 wn`: Store the *n* 32-bit quantities in successive memory words

Pseudoinstruction Replacement

° Asm. treats convenient variations of machine language instructions as if real instructions

Pseudo:	Real:
`subu $sp,$sp,32`	`addiu $sp,$sp,-32`
`sd $a0, 32($sp)`	`sw $a0, 32($sp)` `sw $a1, 36($sp)`
`mul $t7,$t6,$t5`	`mul $t6,$t5` `mflo $t7`
`addu $t0,$t6,1`	`addiu $t0,$t6,1`
`ble $t0,100,loop`	`slti $at,$t0,101` `bne $at,$0,loop`
`la $a0, str`	`lui $at,left(str)` `ori $a0,$at,right(str)`

Absolute Addresses in MIPS

° Which instructions need relocation editing?

° J-format: jump, jump and link

j/jal	xxxxx

° Loads and stores to variables in static area, relative to global pointer

lw/sw	$gp	$x	address

° What about conditional branches?

beq/bne	$rs	$rt	address

° PC-relative addressing preserved even if code moves

Producing Machine Language (1/2)

° Simple Case

- Arithmetic, Logical, Shifts, and so on.
- All necessary info is within the instruction already.

° What about Branches?

- PC-Relative
- So once pseudoinstructions are replaced by real ones, we know by how many instructions to branch.

° So these can be handled easily.

Producing Machine Language (2/2)

- What about jumps (`j` and `jal`)?
  - Jumps require **absolute address**.
- What about references to data?
  - `la` gets broken up into `lui` and `ori`
  - These will require the full 32-bit address of the data.
- These can't be determined yet, so we create two tables…

Symbol Table

- List of "items" in this file that may be used by other files.
- What are they?
  - Labels: function calling
  - Data: anything in the `.data` section; variables which may be accessed across files
- First Pass: record label-address pairs
- Second Pass: produce machine code
  - Result: can jump to a later label without first declaring it

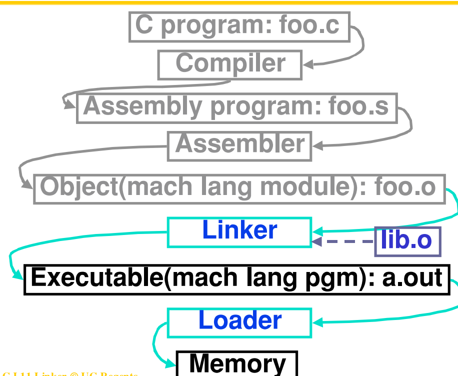
Relocation Table

- List of "items" for which this file needs the address.
- What are they?
  - Any label jumped to: `j` or `jal`
    - internal
    - external (including lib files)
  - Any piece of data
    - such as the `la` instruction

Object File Format

- <u>object file header</u>: size and position of the other pieces of the object file
- <u>text segment</u>: the machine code
- <u>data segment</u>: binary representation of the data in the source file
- <u>relocation information</u>: identifies lines of code that need to be "handled"
- <u>symbol table</u>: list of this file's labels and data that can be referenced
- <u>debugging information</u>

Where Are We Now?



Link Editor/Linker (1/2)

- What does it do?
- Combines several object (.o) files into a single executable ("<u>linking</u>")
- Enable Separate Compilation of files
  - Changes to one file do not require recompilation of whole program
    - Windows NT source is >30 M lines of code! And Growing!
  - Called a <u>module</u>
  - Link Editor name from editing the "links" in jump and link instructions

Link Editor/Linker (2/2)

- Step 1: Take text segment from each .o file and put them together.
- Step 2: Take data segment from each .o file, put them together, and concatenate this onto end of text segments.
- Step 3: Resolve References
  - Go through Relocation Table and handle each entry
  - That is, fill in all absolute addresses

Four Types of Addresses

- PC-Relative Addressing (`beq`, `bne`): never relocate
- Absolute Address (`j`, `jal`): always relocate
- External Reference (usually `jal`): always relocate
- Data Reference (often `lui` and `ori`): always relocate

Resolving References (1/2)

- Linker *assumes* first word of first text segment is at address 0x00000000.
- Linker knows:
  - length of each text and data segment
  - ordering of text and data segments
- Linker calculates:
  - absolute address of each label to be jumped to (internal or external) and each piece of data being referenced

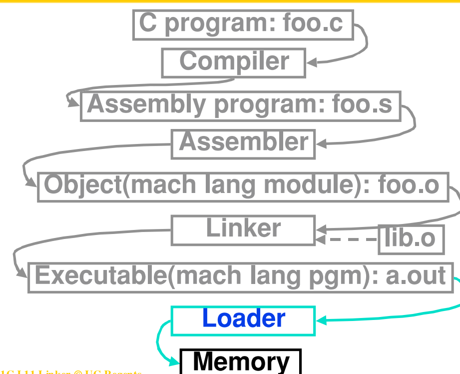
Resolving References (2/2)

- To resolve references:
  - search for reference (data or label) in all symbol tables
  - if not found, search library files (for example, for `printf`)
  - once absolute address is determined, fill in the machine code appropriately
- Output of linker: executable file containing text and data (plus header)

Administrivia

- Reading assignment:
  - P&H A.8, 8.1-8.4

Where Are We Now?



Loader (1/3)

- Executable files are stored on disk.
- When one is run, loader's job is to load it into memory and start it running.
- In reality, loader is the operating system (OS)
  - loading is one of the OS tasks

Loader (2/3)

- So what does a loader do?
- Reads executable file's header to determine size of text and data segments
- Creates new address space for program large enough to hold text and data segments, along with a stack segment
- Copies instructions and data from executable file into the new address space (this may be anywhere in memory)

Loader (3/3)

- Copies arguments passed to the program onto the stack
- Initializes machine registers
  - Most registers cleared, but stack pointer assigned address of 1st free stack location
- Jumps to start-up routine that copies program's arguments from stack to registers and sets the PC
  - If main routine returns, start-up routine terminates program with the exit system call

Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

```
#include <stdio.h>
int main (int argc, char *argv[]) {
 int i;
 int sum = 0;

 for (i = 0; i <= 100; i = i + 1)
     sum = sum + i * i;
 printf ("The sum from 0 .. 100 is %d\n",
     sum);
}
```


Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

```
 .text
 .align  2
 .globl  main
main:
 subu $sp,$sp,32
 sw  $ra, 20($sp)
 sd  $a0, 32($sp)
 sw  $0, 24($sp)
 sw  $0, 28($sp)
loop:
 lw  $t6, 28($sp)
 mul $t7, $t6,$t6
 lw  $t8, 24($sp)
 addu $t9,$t8,$t7
 sw  $t9, 24($sp)
 addu $t0, $t6, 1
 sw  $t0, 28($sp)
 ble $t0,100, loop
 la  $a0, str
 lw  $a1, 24($sp)
 jal printf
 move $v0, $0
 lw  $ra, 20($sp)
 addiu $sp,$sp,32
 j   $ra
 .data
 .align  0
str:
 .asciiz "The sum from 0 .. 100 is %d\n"
```


Symbol Table Entries

- **Label** **Address**

```
main:
loop:        ?
str:
printf:
```

Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

•Remove pseudoinstructions, assign addresses

```
00 addiu $29,$29,-32    30 addiu $8,$14, 1
04 sw    $31,20($29)    34 sw    $8,28($29)
08 sw    $4, 32($29)    38 slti  $1,$8, 101
0c sw    $5, 36($29)    3c bne   $1,$0, loop
10 sw     $0, 24($29)   40 lui   $4, l.str
14 sw     $0, 28($29)   44 ori   $4,$4,r.str
18 lw    $14, 28($29)   48 lw    $5,24($29)
1c multu $14, $14       4c jal   printf
20 mflo   $15           50 add   $2, $0, $0
24 lw    $24, 24($29)   54 lw    $31,20($29)
28 addu $25,$24,$15     58 addiu $29,$29,32
2c sw    $25, 24($29)   5c jr    $31
```


Symbol Table Entries

° **Symbol Table**

Label	Address
main:	0x00000000
loop:	0x00000018
str:	0x10000430
printf:	0x000003b0

° **Relocation Information**

Address	Instr. Type	Dependency
¥0x0000004c	jal	printf

Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

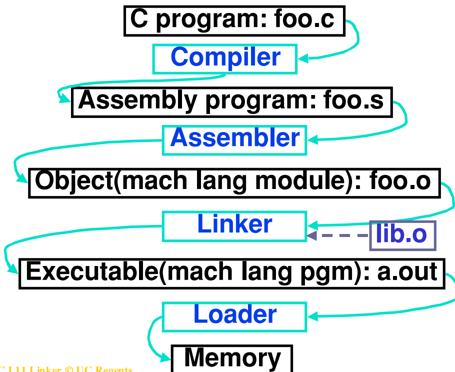
•Edit Addresses: start at 0x0040000

```
00 addiu $29,$29,-32    30 addiu $8,$14, 1
04 sw    $31,20($29)    34 sw    $8,28($29)
08 sw    $4, 32($29)    38 slti  $1,$8, 101
0c sw    $5, 36($29)    3c bne   $1,$0, -10
10 sw     $0, 24($29)   40 lui   $4, 4096
14 sw     $0, 28($29)   44 ori   $4,$4,1072
18 lw    $14, 28($29)   48 lw     $5,24($29)
1c multu $14, $14       4c jal    812
20 mflo   $15           50 add    $2, $0, $0
24 lw    $24, 24($29)   54 lw     $31,20($29)
28 addu $25,$24,$15     58 addiu $29,$29,32
2c sw    $25, 24($29)   5c jr     $31
```


Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

```
0x004000  00100111101111011111111111100000
0x004004  10101111101111110000000000010100
0x004008  10101111101001000000000000100000
0x00400c  10101111101001010000000000100100
0x004010  10101111101000000000000000011000
0x004014  10101111101000000000000000011100
0x004018  10001111101011100000000000011100
0x00401c  10001111101110000000000000011000
0x004020  00000001110011100000000000011001
0x004024  00100101110010000000000000000001
0x004028  00101001000000010000000001100101
0x00402c  10101111101010000000000000011100
0x004030  00000000000000000111100000010010
0x004034  00000011000011111100100000100001
0x004038  00010100001000001111111111110111
0x00403c  10101111101110010000000000011000
0x004040  00111100000001000001000000000000
0x004044  10001111101001010000000000011000
0x004048  00001100000100000000000011101100
0x00404c  00100100100001000000010000110000
0x004050  10001111101111110000000000010100
0x004054  00100111101111010000000000100000
0x004058  00000011111000000000000000001000
0x00405c  00000000000000000001000000100001
```


Things to Remember (1/3)



Things to Remember (2/3)

° Compiler converts a single HLL file into a single assembly language file.

° Assembler removes pseudos, converts what it can to machine language, and creates a checklist for the linker (relocation table). This changes each .s file into a .o file.

° Linker combines several .o files and resolves absolute addresses.

° Loader loads executable into memory and begins execution.

Things to Remember 3/3

- Stored Program concept mean instructions just like data, so can take data from storage, and keep transforming it until load registers and jump to routine to begin execution
  - Compiler ⇒ Assembler ⇒ Linker (⇒ Loader)
- Assembler does 2 passes to resolve addresses, handling internal forward references
- Linker enables separate compilation, libraries that need not be compiled, and resolves remaining addresses