CS61C
Starting a Program

Lecture 10

February 19, 1999

Dave Patterson
(http.cs.berkeley.edu/~patterson)

www-inst.eecs.berkeley.edu/~cs61c/schedule.html
## Review: New MIPS arithmetic instructions

<table>
<thead>
<tr>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mult $2,$3</code></td>
<td>Hi, Lo = $2 \times $3</td>
<td>64-bit signed product</td>
</tr>
<tr>
<td><code>multu$2,$3</code></td>
<td>Hi, Lo = $2 \times $3</td>
<td>64-bit unsigned product</td>
</tr>
<tr>
<td><code>div $2,$3</code></td>
<td>Lo = $2 \div $3,</td>
<td>Lo = quotient, Hi = rem</td>
</tr>
<tr>
<td><code>divu $2,$3</code></td>
<td>Lo = $2 \div $3,</td>
<td>Unsigned quotient, rem.</td>
</tr>
<tr>
<td><code>mfhi $1</code></td>
<td>$1 = Hi</td>
<td>Used to get copy of Hi</td>
</tr>
<tr>
<td><code>mflo $1</code></td>
<td>$1 = Lo</td>
<td>Used to get copy of Lo</td>
</tr>
<tr>
<td><code>add.s $0,$1,$2</code></td>
<td>$f0=$f1+$f2</td>
<td>Fl. Pt. Add (single)</td>
</tr>
<tr>
<td><code>add.d $0,$2,$4</code></td>
<td>$f0=$f2+$f4</td>
<td>Fl. Pt. Add (double)</td>
</tr>
<tr>
<td><code>sub.s $0,$1,$2</code></td>
<td>$f0=$f1-$f2</td>
<td>Fl. Pt. Subtract (single)</td>
</tr>
<tr>
<td><code>sub.d $0,$2,$4</code></td>
<td>$f0=$f2-$f4</td>
<td>Fl. Pt. Subtract (double)</td>
</tr>
<tr>
<td><code>mul.s $0,$1,$2</code></td>
<td>$f0=$f1x$f2</td>
<td>Fl. Pt. Multiply (single)</td>
</tr>
<tr>
<td><code>mul.d $0,$2,$4</code></td>
<td>$f0=$f2x$f4</td>
<td>Fl. Pt. Multiply (double)</td>
</tr>
<tr>
<td><code>div.s $0,$1,$2</code></td>
<td>$f0=$f1÷$f2</td>
<td>Fl. Pt. Divide (single)</td>
</tr>
<tr>
<td><code>div.d $0,$2,$4</code></td>
<td>$f0=$f2÷$f4</td>
<td>Fl. Pt. Divide (double)</td>
</tr>
<tr>
<td><code>c.X.s $0,$1</code></td>
<td>flag1= $f0 X $f1</td>
<td>Fl. Pt. Compare (single)</td>
</tr>
<tr>
<td><code>c.X.d $0,$2</code></td>
<td>flag1= $f0 X $f2</td>
<td>Fl. Pt. Compare (double)</td>
</tr>
</tbody>
</table>

X is eq, lt, le; bc1t, bc1f tests flag
IEEE 754 Floating Point standard: accuracy first class citizen

Multiply product 2n bits; Divide produces both n-bit quotient and n-bit remainder

Computer numbers have limited size => limited precision

- underflow: too small for Fl. Pt. (bigger negative exponent than can represent)
- overflow: too big for Fl. Pt. or integer (bigger positive exponent than can represent, or bigger integer than fits in word)

Programmers beware!
Outline

° Floating Point Questions and Answers
° Compiling, Assembling a Program
° Administrivia, “Computers in the News”
° Linking, Loading a Program
° An Example
° Conclusion
What is result of this program?

```c
main() {
    float i, j;
    i = 1.0; j = 0.0;
    printf("%3.1f\n", i/j);
    printf("%3.1f\n", -i/j);
    printf("%3.1f\n", j/(-i/j));
    if (j/(-i/j) == 0.0)
        printf("0.0 == -0.0\n");
    else printf("0.0 != -0.0\n");
    printf("%3.1f\n", i/j-i/j);
    printf("%3.1f\n", (i/j-i/j)+i);
}
```

Output:
```
Inf
-Inf
-0.0
0.0 == -0.0
NaN
NaN
```
Big Idea: Levels of Abstraction (Unix/DOS)

C program: foo.c/foo.txt

Assembly program: foo.s/ foo.asm

Object (mach lang module): foo.o/foo.obj

Executable (mach lang pgm): a.out/foo.exe

Compiler

Assembler

Linker

Loader

Memory
Assembler

- Follow Directions
- Replace Pseudoinstructions
- Find list of labels, addresses
- Produce machine language
- Creates Object File
Example Assembly Program: Directives

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

Tell assembler how to translate program but do not produce machine instructions

.text (addr): Subsequent items put in user text segment (starting at addr)

.align n: Align the next data on a \(2^n\) byte boundary; \(\text{align } 2 \Rightarrow \text{next word boundary}\)

.globl sym: declares sym global and can be referenced from other files

.data (addr): Subsequent items put in user data segment (starting at addr)

.ascii str: Store the string str in memory and null-terminate it
Example Assembly Program: PseudoInstr

.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)
    bl  $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"
Pseudoinstruction replacements

Asm. treats common variations of machine language instructions as if real instructions

- `subu $sp, $sp, 32`  `addiu $sp, $sp, -32`
- `sd $a0, 32($sp)`  `sw $a0, 32($sp)`
  `sw $a1, 36($sp)`
- `mul $t7, $t6, $t6`  `mul $t6, $t6`
  `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)`
- `move $v0, $0`  `add $v0, $0, $0`
Example Assembly Program: Labels

```assembly
.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)
bler $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:
.ascii "The sum from 0 .. 100 is %d\n"
```

str:
Addresses

Main job of Assembler:

1) Find list of labels and their addresses

2) Produce machine language

- Puts label and memory address of that instruction in Symbol Table

- Can use a name before it is defined; forward reference e.g., str in example
  
  • Note: Must declare before use in C

- Uses Symbol Table in 2nd pass to produce machine code, including addresses
Administrivia

° **Readings:** (3.9) A.2, A.3, A.4, 3.10, 3.11

° **5th homework:** Due 2/24 7PM
  • Exercises 4.21, 4.25, 4.28

° **3rd Project/5th Lab:** MIPS Simulator
  Due Wed. 3/3 7PM; deadline Thurs 8AM
  • Twice / semester 24-hour extension
  • Everything (but midterm) may be done in pairs now

° **Solutions:** ~cs61c/solutions directory. Access it using cs61c account. **Not** www!

° **Midterm conflict time:** Mon 3/15 6-9PM
“Computers in the News”

“Microsoft Denies Rumors That It Is Crafting a Language Like Java”, NY Times, 2/15/99

- “Microsoft Corp. is playing down a report in ‘PC Week’ that it has been briefing software developers on a new programming language (code-named ‘Cool’) that would be similar to Sun's Java software.” [Java SW not binaries]

- “.... Cross platform means that a developer can write one version of a program in Java, which can then run on Windows, Macintosh, Unix and other software operating systems.”

- “... A compiler is a programming tool that converts high-level languages, like C++ or Java, into the machine code of ones and zeros that computers understand.”
Big Idea: Levels of Abstraction (Unix/DOS)

C program: foo.c/foo.txt

Assembly program: foo.s/foo.asm

Object (mach lang module): foo.o/foo.obj

Library: lib.o/lib.obj

Executable (mach lang pgm): a.out/foo.exe

Linker

Loader

Memory

Compiler

Assembler
Object File Format (output of Assembler)

- **object file header**: size and position of the other pieces of the object file
- **text segment**: the machine code
- **data segment**: binary representation of the data in the source file
- **relocation information**: identifies instructions and data words that depend on absolute addresses
- **symbol table**: remaining labels not defined, such as external references
- **debugging information**: help debugger map source file to instructions, data
Link Editor or Linker

° 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 module
  • Link Editor name from editing the “links” in jump and link instructions

° Job of Linker: resolve external labels in object code and place modules and data into memory
Steps of Linker

1) Resolve references across files

2) If some labels unresolved, search for library routine references (e.g., printf in example)
   • If finds reference, it extracts routine’s code and inserts into program text

3) Determine memory address of code modules and adjust absolute addresses (using relocation information)

° Produces file in same format, but no relocation information or unresolved references
C memory allocation seen by the Program

Stack Segment

Stack

$sp →

Reserved

Heap data

Static data

Text

Data Segment

Text Segment

Reserved

0x1000 0000

0x0400 0000

0x0000 0000

0x7fff ffff
Absolute Addresses in MIPS

- Which instructions need relocation editing?

- J-format: jump, jump and link

  \[
  \text{j/jal} \quad \text{xxxxxx}
  \]

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

  \[
  \begin{array}{ccc}
  \text{lw/sw} & \$gp & \$x & \text{address} \\
  \end{array}
  \]

- What about conditional branches?

  \[
  \begin{array}{ccc}
  \text{beq/bne} & \$rs & \$rt & \text{address} \\
  \end{array}
  \]

- PC-relative addressing preserved even if code moves
Big Idea: Levels of Abstraction (Unix/DOS)

- **C program:** foo.c/ foo.txt
- **Assembly program:** foo.s/ foo.asm
- **Object (mach lang module):** foo.o/foo.obj
- **Executable (mach lang pgm):** a.out/foo.exe

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Assembler</th>
<th>Linker</th>
<th>Loader</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Loading

° If links without error can be run
° Before run, executable program stored on disk
° UNIX systems, Operating System Kernel loads the program from disks
Steps 1-5 (out of 6) for loading program

° 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

° 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
Steps 6 (out of 6) for loading program

- Jumps to start-up routine that copies program’s arguments from stack to registers and calls program’s main routine
  - If main routine returns, start-up routine terminates program with the exit system call
#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

<table>
<thead>
<tr>
<th>Label</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>main:</td>
<td></td>
</tr>
<tr>
<td>loop:</td>
<td></td>
</tr>
<tr>
<td>str:</td>
<td></td>
</tr>
</tbody>
</table>
Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

- Remove pseudoinstructions, assign addresses

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

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>04</th>
<th>08</th>
<th>0c</th>
<th>10</th>
<th>14</th>
<th>18</th>
<th>1c</th>
<th>20</th>
<th>24</th>
<th>28</th>
<th>2c</th>
<th>30</th>
<th>34</th>
<th>38</th>
<th>3c</th>
<th>40</th>
<th>44</th>
<th>48</th>
<th>4c</th>
<th>50</th>
<th>54</th>
<th>58</th>
<th>5c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>addiu</td>
<td>sw</td>
<td>sw</td>
<td>sw</td>
<td>sw</td>
<td>sw</td>
<td>lw</td>
<td>multu</td>
<td>mflo</td>
<td>lw</td>
<td>addu</td>
<td>sw</td>
<td>addiu</td>
<td>sw</td>
<td>slti</td>
<td>bne</td>
<td>lui</td>
<td>addiu</td>
<td>lw</td>
<td>jal</td>
<td>add</td>
<td>lw</td>
<td>addiu</td>
<td>jr</td>
</tr>
<tr>
<td></td>
<td>$29,</td>
<td>$31,</td>
<td>$4,</td>
<td>$5,</td>
<td>$0,</td>
<td>$0,</td>
<td>$14,</td>
<td>$14,</td>
<td>$15</td>
<td>$24,</td>
<td>$25,</td>
<td>$25,</td>
<td>$8,</td>
<td>$8,</td>
<td>$1,</td>
<td>$1,</td>
<td>$4,</td>
<td>$4,</td>
<td>$5,</td>
<td>$4,</td>
<td>$29,</td>
<td>$29,</td>
<td>$31</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$29,</td>
<td>-32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Example: C \(\Rightarrow\) Asm \(\Rightarrow\) Obj \(\Rightarrow\) **Exe** \(\Rightarrow\) Run**

<table>
<thead>
<tr>
<th>Address</th>
<th>Binary Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x004000</td>
<td>00100111101111011111111111110000</td>
</tr>
<tr>
<td>0x004004</td>
<td>1010111110111111000000000010100</td>
</tr>
<tr>
<td>0x004008</td>
<td>1010111111010010000000000100000</td>
</tr>
<tr>
<td>0x00400C</td>
<td>1010111110100101000000000100100</td>
</tr>
<tr>
<td>0x004010</td>
<td>1010111110100000000000000011000</td>
</tr>
<tr>
<td>0x004014</td>
<td>1010111110100000000000000011100</td>
</tr>
<tr>
<td>0x004018</td>
<td>1000111110101111000000000011100</td>
</tr>
<tr>
<td>0x00401C</td>
<td>1000111110111100000000000111100</td>
</tr>
<tr>
<td>0x004020</td>
<td>0000000111001110000000000011001</td>
</tr>
<tr>
<td>0x004024</td>
<td>0010010111001000000000000010001</td>
</tr>
<tr>
<td>0x004028</td>
<td>0010100100000000100000000110010</td>
</tr>
<tr>
<td>0x00402C</td>
<td>1010111110101000000000000111100</td>
</tr>
<tr>
<td>0x004030</td>
<td>00000000000000000000111100000100</td>
</tr>
<tr>
<td>0x004034</td>
<td>00000001110000111111100100000100</td>
</tr>
<tr>
<td>0x004038</td>
<td>0010100000100000011111111111011</td>
</tr>
<tr>
<td>0x00403C</td>
<td>0101011110111100100000000111000</td>
</tr>
<tr>
<td>0x004040</td>
<td>0011110000000100000100000000000</td>
</tr>
<tr>
<td>0x004044</td>
<td>1000111110101001000000000111000</td>
</tr>
<tr>
<td>0x004048</td>
<td>0001110000001000000000000111011</td>
</tr>
<tr>
<td>0x00404C</td>
<td>00100100010000010000000100011100</td>
</tr>
<tr>
<td>0x004050</td>
<td>1000111110111111111100000010100</td>
</tr>
<tr>
<td>0x004054</td>
<td>00100111101110100000000001101100</td>
</tr>
<tr>
<td>0x004058</td>
<td>0000000111111000000000000100010</td>
</tr>
<tr>
<td>0x00405C</td>
<td>00000000000000000000011000000001</td>
</tr>
</tbody>
</table>
“And in Conclusion..” 1/1

° Stored Program concept means 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

° Next: Reviewing C, Pointers, Procedures