Testing and Branching

CSE 410, Spring 2009
Computer Systems

http://www.cs.washington.edu/410
Reading and References

- *Computer Organization and Design*
  - Section 2.6, Logical Operations
  - Section 2.7, Instructions for Making Decisions
  - Section B.9, SPIM
  - Section B.10 through page B-50, MIPS R2000 Assembly Language
Control Flow

• All interesting programs have:
  » Loops (while, for, do-while)
    • With an occasional break or continue
  » Conditionals (if, switch)

• Machines have:
  » goto
  » conditional goto

• Have to synthesize what we want from that
goto considered harmful

• “Oh what a tangled web we weave, When first we practice to deceive!”
  » Sir Walter Scott

• Branching in assembly language can turn your program into a rat’s nest that cannot be debugged

• Keep control flow simple and logical

• Use comments describing the overall logic
  » (if, while, for, … pseudo-code is often great!)
Conditional Branch

A change in the program’s flow of control that depends on some condition.
Branch instructions

• Branch instructions are I-format instructions
  » op code field
  » two register fields
  » 16-bit offset field

• Simplest branches check for equality
  » beq $t0, $t1, address
  » bne $t0, $t1, address

• Meaning: if condition is true, set PC = address
  » i.e., fetch next instruction from address
if (i==j) then a=b;

• Assume all values are in registers
• Note that the test is inverted compared to if!

# $t0=i, \ $t1=j, \ $s0=a, \ $s1=b

    bne \ $t0, \ $t1, \ skip
    move \ $s0, \ $s1

skip:
while (s[i]==k) i = i+j;

# $s0=addr(s), $v1=i, $a0=k, $al=j

loop:
    sll     $v0,$v1,2    # v0 = 4*i
    addu    $v0,$s0,$v0  # v0 = addr(s[i])
    lw      $v0,0($v0)   # v0 = s[i]
    addu    $v1,$v1,$a1  # i = i+j
    beq     $v0,$a0,loop # loop if equal
    subu    $v1,$v1,$a1  # i = i-j
for (i=0; i<10; i++) s[i] = i;

# $s0=addr(s), $t1=i
move $t1,$zero # i = 0

loop:
  sll $t0,$t1,2 # t0 = i*4
  addu $t0 $s0 $t0 # t0 = addr(s[i])
  sw $t1,0($t0) # s[i] = i
  addu $t1,$t1,1 # i++
  slt $t0,$t1,10 # if (i<10) $t0=1
  bnez $t0,loop # loop if (i<10)
How do we encode the destination?

• Calculating the destination address
  » 4*(the 16-bit offset value) is added to the Program Counter (PC)
  » This is calculated with the *incremented* value of the PC *after* the branch instruction is fetched
• The offset is a **word** offset in this case
• The base register is always the PC, so we don’t need to specify it in the instruction
• Covers a range of $2^{16}$ words (64 KW)
Comparison instructions

• For comparisons other than equality
  » `slt` : set less than
  » `sltu` : set less than unsigned
  » `slti` : set less than constant value
  » `sltiu` : set less than unsigned constant

• set `t0` to 1 if `t1`<`t2`, otherwise set to 0
  `slt $t0, $t1, $t2`
Pseudo-instructions

• The assembler is your friend and will build instruction sequences for you

• Original code:

\[ \text{bge } \$a0,\$t1,\text{end} \quad \# \text{ if } a0\geq t1 \text{ jump} \]

• Pseudo-instruction; no such instruction in the real processor hardware

• Actual instructions:

\[ \text{slt } \$at,\$a0,\$t1 \quad \# \text{ if } a0<t1 \text{ at}=\text{true} \]
\[ \text{beq } \$at,\$0,\text{end} \quad \# \text{ jump if } \text{at}==\text{false} \]
Jump Instructions

• Jump instructions provide longer range than branch instructions

• 26-bit word offset in J-format instructions
  » j : jump
  » jal : jump and link (store return address)

• 32-bit address in register jumps
  » jr : jump through register
  » jalr : jump through register and link
J-format fields

<table>
<thead>
<tr>
<th>op code</th>
<th>word offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

- The word offset value is multiplied by 4 to create a byte offset
  - the result is 28 bits wide
- Then concatenated with top 4 bits of PC to make a 32 bit destination address
  - i.e., can’t jump outside a 256MB segment (not a problem in most real code)
Important Jumps

• Jump and link (jal)
  » call procedure and store return address in $ra

• Jump through register (jr)
  » return to caller using the address in $ra

• We will talk about procedure calls in excruciating detail shortly