

# CS61C – Machine Structures

## Lecture 10 - MIPS Branch Instructions II

2/8/2006

John Wawrzynek

([www.cs.berkeley.edu/~johnw](http://www.cs.berkeley.edu/~johnw))

[www-inst.eecs.berkeley.edu/~cs61c/](http://www-inst.eecs.berkeley.edu/~cs61c/)

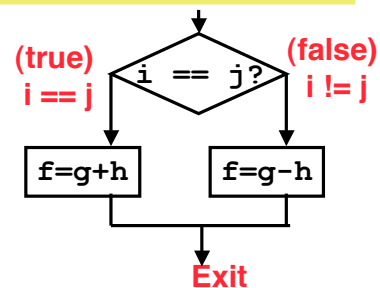
CS 61C L10 MIPS Branch II (1)

Wawrzynek Spring 2006 © UCB

### Compiling C `if` into MIPS (1/2)

#### ◦ Compile by hand

```
if (i == j) f=g+h;  
else f=g-h;
```



#### ◦ Use this mapping:

```
f: $s0  
g: $s1  
h: $s2  
i: $s3  
j: $s4
```

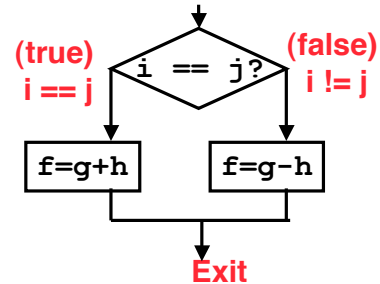
CS 61C L10 MIPS Branch II (2)

Wawrzynek Spring 2006 © UCB

## Compiling C if into MIPS (2/2)

- Compile by hand

```
if (i == j) f=g+h;
else f=g-h;
```



- Final compiled MIPS code:

```
        beq $s3,$s4,True  # branch i==j
        sub $s0,$s1,$s2  # f=g-h(false)
        j   Fin          # goto Fin
True:   add $s0,$s1,$s2  # f=g+h (true)
Fin:
```

**Note:** Compiler automatically creates labels to handle decisions (branches). Generally not found in HLL code.

## Review

- Memory is **byte**-addressable, but `lw` and `sw` access one **word** at a time.
- A pointer (used by `lw` and `sw`) is just a memory address, so we can add to it or subtract from it (using offset).
- A Decision allows us to decide what to execute at run-time rather than compile-time.
- C Decisions are made using **conditional statements** within `if`, `while`, `do while`, `for`.
- MIPS Decision making instructions are the **conditional branches**: `beq` and `bne`.
- New Instructions:  
`lw`, `sw`, `beq`, `bne`, `j`

## From last time: Loading, Storing bytes 1/2

- In addition to word data transfers (`lw`, `sw`), MIPS has byte data transfers:
- load byte: `lb`
- store byte: `sb`
- same format as `lw`, `sw`

CS 61C L10 MIPS Branch II (5)

Wawrzynek Spring 2006 © UCB

## Loading, Storing bytes 2/2

- What do with other 24 bits in the 32 bit register?

- `lb`: sign extends to fill upper 24 bits



- Normally don't want to sign extend chars
- MIPS instruction that doesn't sign extend when loading bytes:

load byte unsigned: `lbu`

CS 61C L10 MIPS Branch II (6)

Wawrzynek Spring 2006 © UCB

## Overflow in Arithmetic (1/2)

---

◦ **Reminder: Overflow occurs when there is a mistake in arithmetic due to the limited precision in computers.**

◦ **Example (4-bit unsigned numbers):**

+15	1111
<u>+3</u>	<u>0011</u>
+18	10010

- But we don't have room for 5-bit solution, so the solution would be 0010, which is +2, and wrong.

## Overflow in Arithmetic (2/2)

---

◦ **Some languages detect overflow (Ada), some don't (C)**

◦ **MIPS solution is 2 kinds of arithmetic instructions to recognize 2 choices:**

- add (add), add immediate (addi) and subtract (sub) **cause overflow to be detected**
- add unsigned (addu), add immediate unsigned (addiu) and subtract unsigned (subu) do **not** cause overflow detection

◦ **Compiler selects appropriate arithmetic**

- MIPS C compilers produce addu, addiu, subu

## Two Logic Instructions

- 2 lectures ago we saw add, addi, sub
- Here are 2 more new instructions
- Shift Left: `sll $s1,$s2,2` #s1=s2<<2
  - Store in \$s1 the value from \$s2 shifted 2 bits to the left, inserting 0's on right; << in C
  - Before: `0000 0002hex`  
`0000 0000 0000 0000 0000 0000 0000 0010two`
  - After: `0000 0008hex`  
`0000 0000 0000 0000 0000 0000 0000 1000two`
  - What arithmetic effect does shift left have?
- Shift Right: `srl` is opposite shift; >>

CS 61C L10 MIPS Branch II (9)

Wawrzynek Spring 2006 © UCB

## Loops in C/Assembly (1/3)

- Simple loop in C; A[] is an array of ints

```
do {  
    g = g + A[i];  
    i = i + j;  
} while (i != h);
```

- Rewrite this as:

```
Loop: g = g + A[i];  
      i = i + j;  
      if (i != h) goto Loop;
```

- Use this mapping:

`g`, `h`, `i`, `j`, base of A  
`$s1`, `$s2`, `$s3`, `$s4`, `$s5`

CS 61C L10 MIPS Branch II (10)

Wawrzynek Spring 2006 © UCB

## Loops in C/Assembly (2/3)

---

### ◦ Final compiled MIPS code:

```
Loop: sll $t1,$s3,2    # $t1 = 4*i
      add $t1,$t1,$s5  # $t1 = addr A
      lw  $t1,0($t1)   # $t1 = A[i]
      add $s1,$s1,$t1  # g = g + A[i]
      add $s3,$s3,$s4  # i = i + j
      bne $s3,$s2,Loop # goto Loop
                        # if i != h
```

### ◦ Original code:

```
Loop: g = g + A[i];
      i = i + j;
      if (i != h) goto Loop;
```

## Loops in C/Assembly (3/3)

---

### ◦ There are three types of loops in C:

- while
- do... while
- for

### ◦ Each can be rewritten as either of the other two, so the method used in the previous example can be applied to while and for loops as well.

### ◦ **Key Concept:** Though there are multiple ways of writing a loop in MIPS, the key to decision making is **conditional branch**

## Inequalities in MIPS (1/4)

◦ Until now, we've only tested equalities (== and != in C). General programs need to test < and > as well.

◦ Introduce MIPS Inequality Instruction:

- “Set on Less Than”

- Syntax: `slt reg1, reg2, reg3`

- Meaning: `reg1 = (reg2 < reg3);`

```
if (reg2 < reg3)
    reg1 = 1;
else reg1 = 0;
```

← Same thing...

“set” means “set to 1”,  
“reset” means “set to 0”.

## Inequalities in MIPS (2/4)

◦ How do we use this? Compile by hand:

```
if (g < h) goto Less; #g:$s0, h:$s1
```

◦ Answer: compiled MIPS code...

```
slt $t0, $s0, $s1 # $t0 = 1 if g<h
bne $t0, $0, Less # goto Less
# if $t0!=0
# (if (g<h)) Less:
```

◦ Branch if `$t0 != 0` → `(g < h)`

◦ Register `$0` always contains the value 0, so `bne` and `beq` often use it for comparison after an `slt` instruction.

◦ A `slt` → `bne` pair means `if (... < ...) goto...`

## Inequalities in MIPS (3/4)

---

- Now, we can implement  $<$ , but how do we implement  $>$ ,  $\leq$  and  $\geq$ ?
- We could add 3 more instructions, but:
  - MIPS goal: **Simpler is Better**
- Can we implement  $\leq$  in one or more instructions using just `slt` and the branches?
- What about  $>$ ?
- What about  $\geq$ ?

## Inequalities in MIPS (4/4)

---

```
# a:$s0, b:$s1
    slt $t0,$s0,$s1 # $t0 = 1 if a<b
    beq $t0,$0,skip # skip if a >= b
    <stuff>        # do if a<b

skip:
```

**Two independent variations possible:**

Use `slt $t0,$s1,$s0` instead of

```
slt $t0,$s0,$s1
```

Use `bne` instead of `beq`

## Immediates in Inequalities

◦ There is also an immediate version of `slt` to test against constants: `slti`

- Helpful in `for` loops

```
C    if (g >= 1) goto Loop
```

```
M    Loop: . . .
```

```
I    slti $t0,$s0,1      # $t0 = 1 if
P    # $s0<1 (g<1)
S    beq  $t0,$0,Loop   # goto Loop
                        # if $t0==0
                        # (if (g>=1))
```

An `slt` → `beq` pair means `if(... ≥ ...) goto...`

CS 61C L10 MIPS Branch II (17)

Wawrzynek Spring 2006 © UCB

## What about unsigned numbers?

◦ Also **unsigned** inequality instructions:

```
sltu, sltiu
```

...which sets result to 1 or 0 depending on unsigned comparisons

◦ What is value of `$t0`, `$t1`?

```
($s0 = FFFF FFFAhex, $s1 = 0000 FFFAhex)
```

```
slt $t0, $s0, $s1
```

```
sltu $t1, $s0, $s1
```

CS 61C L10 MIPS Branch II (18)

Wawrzynek Spring 2006 © UCB

## MIPS Signed vs. Unsigned – diff meanings!

◦ MIPS terms *Signed/Unsigned* are “overloaded”:

- **Do/Don't sign extend**  
(lb, lbu)
- **Don't overflow**  
(addu, addiu, subu, multu, divu)
- **Do signed/unsigned compare**  
(slt, slti/sltu, sltiu)

## Example: The C Switch Statement (1/3)

◦ Choose among four alternatives depending on whether *k* has the value 0, 1, 2 or 3.

Compile this C code:

```
switch (k) {  
  case 0: f=i+j; break; /* k=0 */  
  case 1: f=g+h; break; /* k=1 */  
  case 2: f=g-h; break; /* k=2 */  
  case 3: f=i-j; break; /* k=3 */  
}
```

## Example: The C Switch Statement (2/3)

- This is complicated, so **simplify**.
- Rewrite it as a chain of if-else statements, which we already know how to compile:

```
if(k==0) f=i+j;
  else if(k==1) f=g+h;
    else if(k==2) f=g-h;
      else if(k==3) f=i-j;
```

- Use this mapping:

```
f:$s0, g:$s1, h:$s2,
i:$s3, j:$s4, k:$s5
```

## Example: The C Switch Statement (3/3)

- Final compiled MIPS code:

```
      bne $s5,$0,L1      # branch k!=0
      add $s0,$s3,$s4   #k==0 so f=i+j
      j   Exit          # end of case so Exit
L1:   addi $t0,$s5,-1   # $t0=k-1
      bne $t0,$0,L2     # branch k!=1
      add $s0,$s1,$s2   #k==1 so f=g+h
      j   Exit          # end of case so Exit
L2:   addi $t0,$s5,-2   # $t0=k-2
      bne $t0,$0,L3     # branch k!=2
      sub $s0,$s1,$s2   #k==2 so f=g-h
      j   Exit          # end of case so Exit
L3:   addi $t0,$s5,-3   # $t0=k-3
      bne $t0,$0,Exit   # branch k!=3
      sub $s0,$s3,$s4   #k==3 so f=i-j
Exit:
```

