CSE 31

Computer Organization

Lecture 9 – MIPS: Conditionals

Announcement

- Lab #4 this week
  - Due next week
- Project #1
  - Due Monday (10/22)
    - Don’t start late, you won’t have time!
- Midterm exam Wednesday (10/3, postponed)
  - Lectures 1 – 7
  - HW #1 and #2
  - Closed book
  - 1 sheet of note (8.5” x 11”)
  - Sample exam online
- Reading assignment
  - Chapter 2.1 – 2.9 of zyBooks (Reading Assignment #2)
    - Make sure to do the Participation Activities
    - Due Wednesday (9/26)

So Far...

- All instructions so far only manipulate data…we've built a calculator of sorts.

- In order to build a computer, we need ability to make decisions…

- C (and MIPS) provide labels to support "goto" jumps to places in code.
  - C: Horrible style;
  - MIPS: Necessary!

C Decisions: `if` Statements

- 2 kinds of if statements in C

  `if` *(condition) clause*

  `if` *(condition) clause1* `else` *clause2*

- Rearrange 2nd if into following:

```
 if  (condition) goto L1;
     clause2;
      goto L2;
 L1: clause1;
 L2:
```

- Not as elegant as if-else, but same meaning

MIPS Decision Instructions

- Decision instruction in MIPS:

  `beq   register1, register2, L1`

  `beq` is "Branch if (registers are) equal"

  Same meaning as (using C):

  `if  (register1==register2) goto L1`

- Complementary MIPS decision instruction

  `bne   register1, register2, L1`

  `bne` is "Branch if (registers are) not equal"

  Same meaning as (using C):

  `if  (register1!=register2) goto L1`

- Called conditional branches

MIPS Goto Instruction

- In addition to conditional branches, MIPS has an <u>unconditional branch</u>:

```
j label
```

- Called a Jump Instruction: jump (or branch) directly to the given label without needing to satisfy any condition
- Same meaning as (using C): `goto label`
- Technically, it's the same effect as:

```
beq $0,$0,label
```

since it always satisfies the condition.

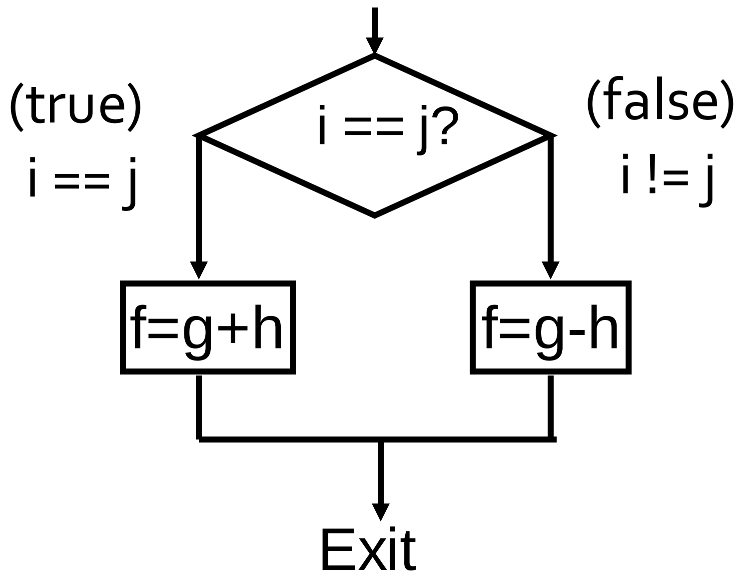
Compiling C `if` into MIPS (1/2)

- Compile by hand

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

- Use this mapping:

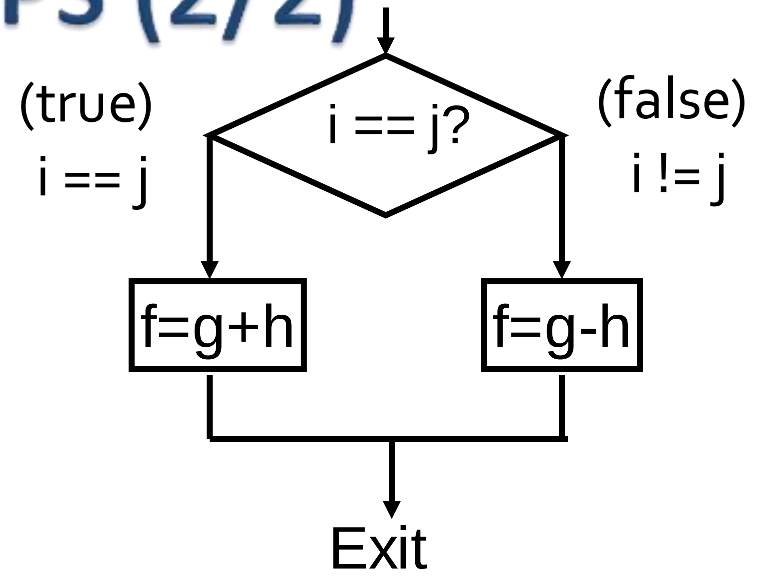
```
f: $s0
g: $s1
h: $s2
i: $s3
j: $s4
```



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

- Compile by hand

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



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

- 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.

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`
- E.g., **`lb $s0, 3($s1)`**
  - contents of memory location with address = sum of "3" + contents of register ***s1*** is copied to the **low byte position** of register ***s0***.

Loading, Storing bytes 2/2

- What to do with other 24 bits in the 32 bit register?
  - lb: **sign extends** to fill upper 24 bits

xxxx xxxx xxxx xxxx xxxx xxxx x zzz zzzz

byte loaded

…is copied to "sign-extend"

This bit

- Normally don't want to sign extend chars
- MIPS instruction that doesn't sign extend when loading bytes:
  - load byte unsigned: **lbu**

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
+ 3	+ 0011
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 instructs:
  - These cause overflow to be detected
    - add (**add**)
    - add immediate (**addi**)
    - subtract (**sub**)
  - These do not cause overflow detection
    - add unsigned (**addu**)
    - add immediate unsigned (**addiu**)
    - subtract unsigned (**subu**)
- Compiler selects appropriate arithmetic
  - MIPS C compilers produce **addu, addiu, subu**

Two "Logic" Instructions

- 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 (they fall off end), inserting 0's on right; << in C.
  - Before: $\texttt{0000 0002}_{hex}$
    $\texttt{0000 0000 0000 0000 0000 0000 0000 0010}_{two}$
  - After: $\texttt{0000 000}\underline{\texttt{8}}_{hex}$
    $\texttt{0000 0000 0000 0000 0000 0000 0000 10}\underline{\texttt{00}}_{two}$
  - What arithmetic effect does shift left have?
    - $n \times 2^i$
- Shift Right: `srl` is opposite shift; >>

Loops in C/Assembly (1/3)

- Simple loop in C; **A[]** is an array of `int`

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

How to write this in MIPS using what we have learned so far?

- 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
```

Loops in C/Assembly (2/3)

- Final compiled MIPS code:

Why???

```
Loop: sll  $t1,$s3,2      # $t1= 4*i
      addu $t1,$t1,$s5    # $t1=addr A+4i
      lw   $t1,0($t1)     # $t1=A[i]
      addu $s1,$s1,$t1    # g=g+A[i]
      addu $s3,$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;
```

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

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 these 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 "change to 1",
"reset" means "change 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 $t5,$s0,$s1 # $t0 = 1 if g<h
bne $t5,$0,Less # goto Less
                # if $t0!=0
                # (if (g<h)) Less:
```

- 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 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:
```

How about **>** and **<=**?

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
```

MIPS

```
Loop:     . . .
slti $t0,$s0,1       # $t0 = 1 if
                     # $s0<1 (g<1)
beq  $t0,$0,Loop     # goto Loop
                     # if $t0==0
                       # (if (g>=1))
```

An `slt` ➔ `beq` pair means `if (… ≥ …) goto…`

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`**?
- (**`$s0 = FFFF FFFA`**$_{hex}$, **`$s1 = 0000 FFFA`**$_{hex}$)

```
slt $t0, $s0, $s1    1
sltu $t1, $s0, $s1   0
```

MIPS Signed vs. Unsigned – diff meanings!

- MIPS terms Signed/Unsigned “overloaded”:
  - Do/Don't sign extend
    - `(lb, lbu)`
  - Do/Don't overflow
    - `(add, addi, sub, mult, div)`
    - `(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:
```

Always compared with $0!

Quiz

```
Loop:addi $s0,$s0,-1    # i = i - 1
     slti $t0,$s1,2     # $t0 = (j < 2)
     beq  $t0,$0 ,Loop  # goto Loop if $t0 == 0
     slt  $t0,$s1,$s0   # $t0 = (j < i)
     bne  $t0,$0 ,Loop  # goto Loop if $t0 != 0
```

($s0=i, $s1=j)

What C code properly fills in the blank in loop below?

do {i--;} while(__);

1) j < 2 && j < i
2) j ≥ 2 && j < i
3) j < 2 && j ≥ i
4) j ≥ 2 && j ≥ i
5) j > 2 && j < i
6) j < 2 || j < i
7) j ≥ 2 || j < i
8) j < 2 || j ≥ i
9) j ≥ 2 || j ≥ i
10) j > 2 || j < i

Quiz

```
Loop:addi $s0,$s0,-1    # i = i - 1
     slti $t0,$s1,2     # $t0 = (j < 2)
     beq  $t0,$0 ,Loop  # goto Loop if $t0 == 0
     slt  $t0,$s1,$s0   # $t0 = (j < i)
     bne  $t0,$0 ,Loop  # goto Loop if $t0 != 0
```

($s0=i, $s1=j)

What C code properly fills in the blank in loop below?

do {i--;} while(__);

1) j < 2 && j < i
2) j ≥ 2 && j < i
3) j < 2 && j ≥ i
4) j ≥ 2 && j ≥ i
5) j > 2 && j < i
6) j < 2 || j < i
7) j ≥ 2 || j < i
8) j < 2 || j ≥ i
9) j ≥ 2 || j ≥ i
10) j > 2 || j < i

Summary

- To help the conditional branches make decisions concerning inequalities, we introduce: “Set on Less Than” called
  `slt`, **`slti`**, **`sltu`**, **`sltiu`**
- One can store and load (signed and unsigned) bytes as well as words with `lb`, `lbu`
- Unsigned add/sub don’t cause overflow
- New MIPS Instructions:

```
sll, srl, lb, lbu
slt, slti, sltu, sltiu
addu, addiu, subu
```

C functions

```
main() {
  int i,j,k,m;
  ...
  i = mult(j,k); ...
  m = mult(i,i); ...
}
```

What information must compiler/programmer keep track of?

```
/* really dumb mult function */
int mult (int mcand, int mlier){
  int product = 0;
  while (mlier > 0)   {
    product = product + mcand;
    mlier = mlier -1; }
  return product;
}
```

What instructions can accomplish this?

Function Call Bookkeeping

- Registers play a major role in keeping track of information for function calls.
- Register conventions:
  - Return address **`$ra`**
  - Arguments **`$a0, $a1, $a2, $a3`**
  - Return value **`$v0, $v1`**
  - Local variables **`$s0, $s1, … , $s7`**
- The stack is also used; more later.

Instruction Support for Functions (1/6)

C

```
... sum(a,b);...   /* a,b:$s0,$s1 */

int sum(int x, int y) {
    return x+y;
}
```

MIPS

address (shown in Hexadecimal)

```
1000
1004
1008
100c
1010
…
2000
2004
```

In MIPS, all instructions are 4 bytes, and stored in memory just like data. So here we show the addresses of where the programs are stored.

Instruction Support for Functions (2/6)

C

```
... sum(a,b);...   /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
    return x+y;
}
```

address (shown in hexadecimal)

MIPS

```
1000 add   $a0,$s0,$zero        # x = a
1004 add   $a1,$s1,$zero        # y = b
1008 addi  $ra,$zero,1010       #$ra=1010
100c j     sum                  #jump to sum
1010
…
2000 sum: add $v0,$a0,$a1
2004 jr    $ra              # new instruction
```

Instruction Support for Functions (3/6)

C

```
... sum(a,b);...   /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
    return x+y;
}
```

MIPS

- Question: Why use `jr` here? Why not use `j`?
- Answer: `sum` might be called by many places, so we can't return to a fixed place. The calling proc to `sum` must be able to say "return here" somehow.

```
2000 sum: add $v0,$a0,$a1
2004 jr   $ra        # new instruction
```

Instruction Support for Functions (4/6)

- Single instruction to jump and save return address: jump and link (`jal`)
- Before:

```
1008 addi $ra,$zero,1010 #$ra=1010
100c j sum               #goto sum
```

- After:

```
100c jal sum  # $ra=1010,goto sum
```

- Why have a **`jal`**?
  - Make the common case fast: function calls very common.
  - Don’t have to know where code is in memory with **`jal`**!

Instruction Support for Functions (5/6)

- Syntax for **`jal`** (jump and link) is same as for **`j`** (jump):

  `jal   label`

- **`jal`** should really be called **`laj`** for "link and jump":
  - Step 1 (link): Save address of next instruction into `$ra`
    - Why next instruction? Why not current one?
  - Step 2 (jump): Jump to the given label

Instruction Support for Functions (6/6)

- Syntax for **`jr`** (jump register):

  `jr register`

- Instead of providing a label to jump to, the **`jr`** instruction provides a register which contains an address to jump to.
- Very useful for function calls:
  - **`jal`** stores return address in register (`$ra`)
  - **`jr $ra`** jumps back to that address