CSE 31
Computer Organization

Lecture 9 – MIPS: Conditionals

Announcement

- Project #1
  - Due at 11:59pm on 3/18, Monday
  - You must demo your submission to your TA within 7 days after due date
- No new lab this week
  - Your attendance is required
  - Use lab this week to finish your Lab #3/#4
  - Use lab this week to kick start Project #1
- HW #1 in CatCourses
  - Due Monday (2/25) at 11:59pm
- Reading assignment #2
  - Chapter 2.1 – 2.9 of zyBook
    - Due Wednesday (2/27) at 11:59pm

Announcement

- Midterm exam Wednesday (3/6, not 2/27)
  - Lectures 1 – 7
  - Lab #1 - #4
  - HW #1
  - Closed book
  - 1 sheet of note (8.5” x 11”), both sides
  - Sample exam in CatCourses
  - Review session by PALS tutors on Wednesday during lecture

Review

- In MIPS Assembly Language:
  - Registers replace variables
  - One Instruction (simple operation) per line
  - Simpler is Better, Smaller is Faster
- New Instructions:

  `add, addi, sub`
- New Registers:

  C Variables: `$s0` - `$s7`

  Temporary Variables: `$t0` - `$t7`

  Zero: `$zero`

Data Transfer: Memory to Reg (4/4)

Data flow

Example: `lw $t0,12($s0)`

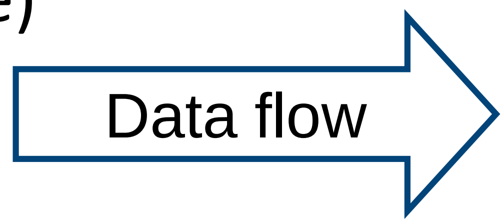
This instruction will take the pointer stored in $s0, add 12 bytes to it, and then load the value from the memory pointed to by this calculated sum into register $t0

- Notes:
  - `$s0` is called the <u>base register</u>
  - 12 is called the <u>offset</u>
  - offset is generally used in accessing elements of array or structure: base reg points to beginning of array or structure (note offset must be a constant known at assembly time)

Data Transfer: Reg to Memory

- Also want to store from register into memory
  - Store instruction syntax is identical to Load’s
- MIPS Instruction Name:

  `sw` (meaning Store Word, so 32 bits or one word is stored at a time)



- Example: `sw $t0,12($s0)`

  This instruction will take the pointer in `$s0`, add 12 bytes to it, and then store the value from register `$t0` into that memory address
- Remember: “Store INTO memory”

Pointers vs. Values

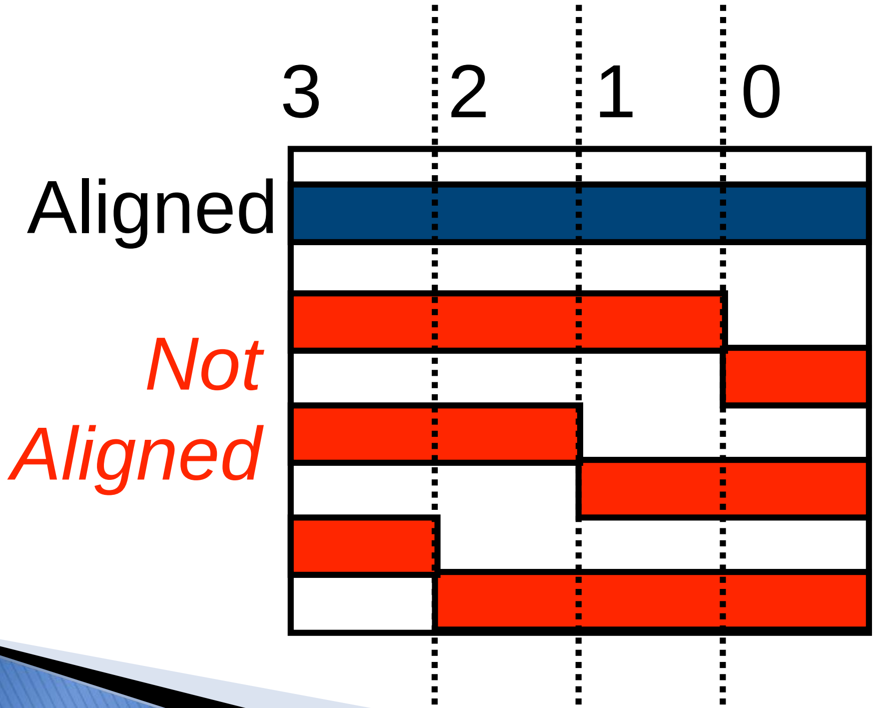
- Key Concept: A register can hold any 32-bit value. That value can be a (signed) int, an unsigned int, a pointer (memory addr), and so on
  - E.g., If you write: `add $t2,$t1,$t0 # c = b + A;` then `$t0` and `$t1` better contain values that can be added
  - E.g., If you write:

    ```
    lw $t2, 0($t0) # c = A[0];
    add $t2, $t2, $t1 #c=A[0]+b
    ```

    then `$t0` better contains a pointer
- Don't mix these up!

More Notes about Memory: Alignment

- MIPS requires that all words start at byte addresses that are multiples of 4 bytes
- Called Alignment: objects fall on address that is multiple of their size



Last hex digit of address is:

0, 4, 8, or C_{hex}

1, 5, 9, or D_{hex}

2, 6, A, or E_{hex}

3, 7, B, or F_{hex}

Notes about Memory

- Pitfall: Forgetting that sequential word addresses in machines with byte addressing do not differ by 1.
  - Many assembly language programmers have toiled over errors made by assuming that the address of the next word can be found by incrementing the address in a register by 1 instead of by the word size in bytes.
  - Also, remember that for both `lw` and `sw`, the sum of the base address and the offset must be a multiple of 4 (to be word aligned)

Role of Registers vs. Memory

- What if more variables than registers?
  - Compiler tries to keep most frequently used variable in registers
  - Less common variables in memory: spilling
- Why not keep all variables in memory?
  - Smaller is faster:
    registers are faster than memory
  - Registers more versatile:
    - MIPS arithmetic instructions can read 2, operate on them, and write 1 per instruction
    - MIPS data transfer only read or write 1 operand per instruction, and no operation

Compilation with Memory

- What offset in `lw` to select `A[5]` in C?
  - 4x5=20 to select `A[5]`: byte vs. word
- Compile by hand using registers:

```
g = h + A[5];
```

  `g`: `$s1`, `h`: `$s2`, `$s3`: base address of `A`

- 1st transfer from memory to register:

```
lw $t0,20($s3)  # $t0 gets A[5]
```

  - Add 20 to `$s3` to select `A[5]`, put into `$t0`
- Next add it to h and place in `g`

```
add $s1,$s2,$t0   # $s1 = h+A[5]
```

Quiz

We want to translate `*x = *y` into MIPS
(`x`, `y` ptrs stored in: `$s0 $s1`)

```
1: add $s0,    $s1, zero
2: add $s1,    $s0, zero
3: lw  $s0,  0($s1)
4: lw  $s1,  0($s0)
5: lw  $t0,  0($s1)
6: sw  $t0,  0($s0)
7: lw  $s0,  0($t0)
8: sw  $s1,  0($t0)
```

```
a) 1 or 2
b) 3 or 4
c) 5 → 6
d) 6 → 5
e) 7 → 8
```

Quiz

We want to translate `*x = *y` into MIPS
(`x`, `y` ptrs stored in: `$s0 $s1`)

```
1: add $s0,    $s1, zero
2: add $s1,    $s0, zero
3: lw  $s0,  0($s1)
4: lw  $s1,  0($s0)
5: lw  $t0,  0($s1)
6: sw  $t0,  0($s0)
7: lw  $s0,  0($t0)
8: sw  $s1,  0($t0)
```

a) 1 or 2
b) 3 or 4
c) 5 → 6
d) 6 → 5
e) 7 → 8

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 <u>conditional branches</u>

MIPS Goto Instruction

- In addition to conditional branches, MIPS has an unconditional branch:

  `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.

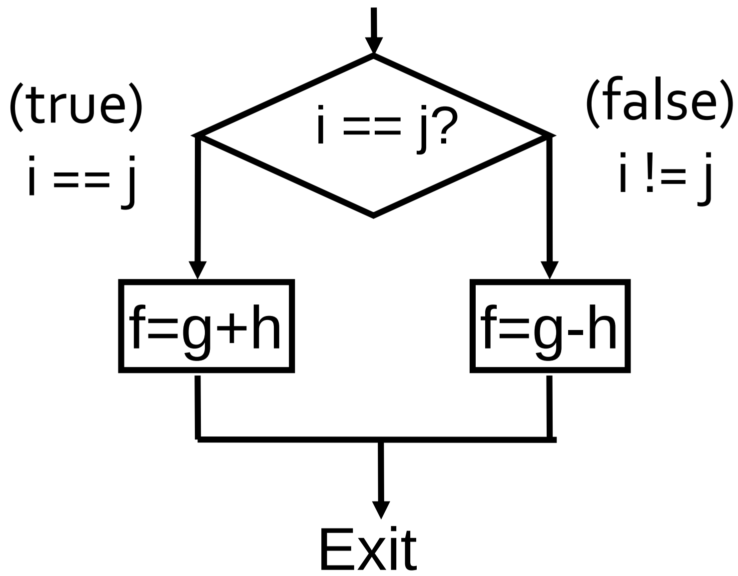
Compiling C `if` into MIPS (1/2)

- Compile by hand

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

- Use this mapping:

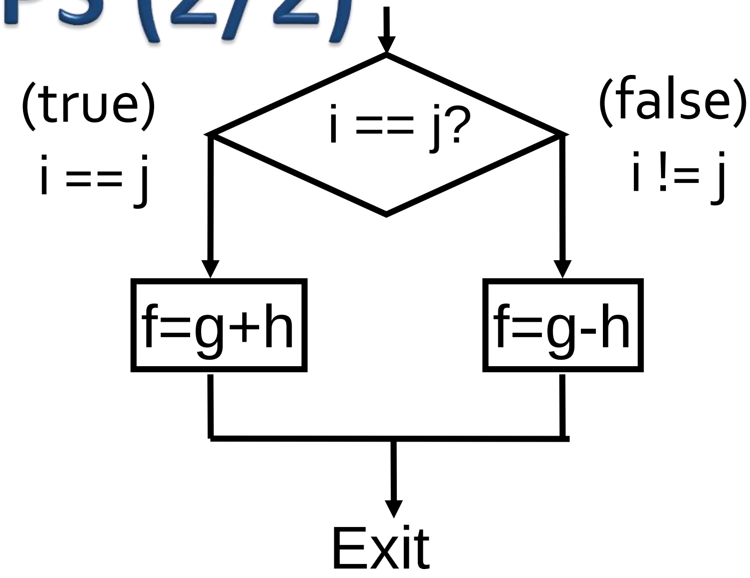
```
f: $s0
g: $s1
h: $s2
i: $s3
j: $s4
```



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: `0000 0002`$_{hex}$
    `0000 0000 0000 0000 0000 0000 0000 0010`$_{two}$
  - After: `0000 0008`$_{hex}$
    `0000 0000 0000 0000 0000 0000 0000 1000`$_{two}$
  - 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:
```

Why not **beq $t5, 1, 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 >, ≤ and ≥ ?
- We could add 3 more instructions, but:
  - MIPS goal: Simpler is Better
- Can we implement ≤ in one or more instructions using just `slt` and branches?
  - What about >?
  - What about ≥?

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)`