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 offsets).
- 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**

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

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):
  
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>1111</td>
</tr>
<tr>
<td>+ 3</td>
<td>+ 0011</td>
</tr>
<tr>
<td>18</td>
<td>10010</td>
</tr>
</tbody>
</table>

  - 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:
  - 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...
  - After: 0000 0002...
  - What arithmetic effect does shift left have?
- Shift Right: srl is opposite shift; >>

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

Loops in C/Assembly (2/3)

- Final compiled MIPS code:
  Loop: sll $t1,$s3,2   # $t1= 4*I
    addu $t1,$t1,$s5    # $t1=addr A+i
    lw $t1,0($t1)      # $t1=A[i]
    addu $s1,$s1,$t1    # g=g+A[i]
    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 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:
    
    ```
    if (reg2 < reg3)
    reg1 = 1;
    else reg1 = 0;
    ```
    
    “Set” means “change to 1”,
    “reset” means “change to 0”.

Administrivia

- Any administrivia?
Inequalities in MIPS (2/4)

- How do we use this? Compile by hand:
  \[
  \text{if} \ (g < h) \ \text{goto} \ \text{Less}; \ \#g:s0, h:s1
  \]
- Answer: compiled MIPS code...
  \[
  \text{slt} \ \text{t0}, \text{s0}, \text{s1} \ \# \text{t0} = 1 \ \text{if} \ g < h
  \]
  \[
  \text{bne} \ \text{t0}, \text{t0}, \text{Less} \ \# \ \text{if} \ \text{t0}! = 0
  \]
  \[
  \# (g<h) \ \text{Less}:
  \]
- Register $0$ always contains the value 0, so $bne$ and $beq$ often use it for comparison after an $slt$ instruction.
- A $slt \Rightarrow bne$ pair means $if(\ ... < \ ...)$ $goto$...

Inequalities in MIPS (3/4)

- Now we can implement $<$, but how do we implement $>$, $s$ and $\geq$?
- We could add 3 more instructions, but:
  \[
  \begin{array}{l}
  \text{MIPS goal: Simpler is Better}
  \\
  \text{Can we implement } s \text{ in one or more instructions using just } slt \text{ and branches?}
  \\
  \quad \text{What about }>?
  \\
  \quad \text{What about } \geq?
  \end{array}
  \]

Inequalities in MIPS (4/4)

\[
\begin{align*}
\# \ a:s0, \ b:s1 \\
\text{slt} \ \text{t0}, \text{s0}, \text{s1} \ \# \text{t0} = 1 \ \text{if} \ a < b \\
\text{beq} \ \text{t0}, \text{t0}, \text{skip} \ \# \ \text{skip if} \ a \geq b \\
\text{\langle stuff\rangle} \quad \# \ \text{do if} \ a < b \\
\text{skip:}
\end{align*}
\]

Two independent variations possible:

- Use $\text{slt} \ \text{t0}, \text{s1}, \text{s0}$ instead of $\text{slt} \ \text{t0}, \text{s0}, \text{s1}$
- Use $\text{bne}$ instead of $\text{beq}$

Immediates in Inequalities

- There is also an immediate version of $\text{slt}$ to test against constants: $\text{slti}$
  \[
  \begin{array}{l}
  \text{Helpful in for loops}
  \\
  \text{C} \ \text{if} \ (g \geq 1) \ \text{goto} \ \text{Loop}
  \end{array}
  \]

\[
\begin{align*}
\text{Loop:} \quad \ldots & \\
\text{M} & \text{slti} \ \text{t0}, \text{s0}, 1 \ \# \ \text{t0} = 1 \ \text{if} \\
\text{P} & \text{\langle s0<1 \ (g<1)\rangle} \\
\text{S} & \text{beq} \ \text{t0}, \text{t0}, \text{Loop} \ \# \ \text{goto} \ \text{Loop} \\
& \text{if} \ \text{t0}==0 \\
& \text{\langle (g>1) \rangle}
\end{align*}
\]

An $\text{slt} \Rightarrow \text{beq}$ pair means $if(\ ... \geq \ ...)$ $\text{goto}$...

What about unsigned numbers?

- Also unsigned inequality instructions:
  \[
  \text{sltu, sltiu}
  \]
  \[
  \ldots \text{which sets result to 1 or 0 depending on unsigned comparisons}
  \]
- What is value of $\text{t0}, \text{t1}$?
  \[
  \begin{align*}
  \text{ss0} &= \text{FFFF FFFFA}_\text{hex} \ \text{ss1} = \text{0000 FFFFA}_\text{hex}
  \\
  \text{slt} \ \text{t0}, \text{ss0}, \text{ss1} \\
  \text{sltu} \ \text{t1}, \text{ss0}, \text{ss1}
  \end{align*}
  \]

MIPS Signed vs. Unsigned – diff meanings!

- MIPS terms Signed/Unsigned “overloaded”:
  \[
  \begin{array}{l}
  \text{Do/Don’t sign extend}
  \\
  \quad \langle \text{lb, lbu}\rangle
  \\
  \text{Do/Don’t overflow}
  \\
  \quad \langle \text{add, addi, sub, mul, div}\rangle
  \\
  \quad \langle \text{addu, addiu, subu, multu, divu}\rangle
  \\
  \text{Do signed/unsigned compare}
  \\
  \quad \langle \text{slt, slti/sltau, sltiu}\rangle
  \end{array}
  \]
Peer Instruction

Loop:
1. addi $s0,$s0,-1  # i = i - 1
2. slti $t0,$s1,2   # $t0 = (j < 2)
3. beq $t0,$s0,Loop # goto Loop if $t0 == 0
4. sl $t0,$s1,$s0   # $t0 = (j < i)
5. bne $t0,$s0,Loop # goto Loop if $t0 != 0
6. ($s0=i, $s1=j)

What C code properly fills in the blank in loop below?
do {i--;} while(__);

“And in conclusion…”

- 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

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:

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

Example: The C Switch Statement (3/3)

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