Review

• Memory is byte-addressable, but **lw** and **sw** access one word at a time. These instructions transfer the contents of memory to/from a register.
  - lw $s3, X

• Load address instruction: **la** loads the memory address of X. A memory address is a pointer.
  - la $s2, X

• A pointer (used by **lw** and **sw**) is just a memory address, so we can do this: (assume $s2 has address of X)
  - Base/displacement: lw $s3, 32($s2)
  - Indexing by adding: addi $s2, $s2, 32
  - lw $s3, 0($s2)

Lecture 4: Load, Logic, Loops

• Loading bytes and halfwords
• A little more about arithmetic
• Two logical operations (shift left, shift right)
• Loops

Review

• 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:
  - native MIPS instructions **beq** and **bne**
  - pseudo instructions **blt**, **ble**, **bgt**, **bge**.
• MIPS conditional branch: **j** (jump)

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**

• Ib moves one byte of data into a register which holds one word.

Loading, Storing bytes 2/2

• What to do with the 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**

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 \text{ 1111}
  - +3 \text{ 0011}
  - +18 \text{ 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
• Shift Left: \texttt{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 0002 hex
  0000 0000 0000 0000 0000 0000 0000 10 two
  • After: 0000 0008 hex
  0000 0000 0000 0000 0000 0000 0000 10000 two
• What arithmetic effect does shift left have?

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)
(This code uses a trick to multiply by 4 using logical shift. Just accept this trick for now.)
Loop: \texttt{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
# if $i!=h$

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

MIPS Loop Examples
• These are excerpts of code from the examples posted on the class website.
• Example 1: a simple loop
• Example 2: print out Fibonacci numbers
• Example 3: array indexing version 1
• Example 4: array indexing version 2
Loops Example 1: print integers 1 to 10

```c
for (i=1; i<=10; i++) printf(" %d",i)
```

```assembly
li $s0,1          # $s0 holds index of loop
loop:
    # print this element
    li $v0,1         # load code for print integer
    syscall
    li $v0,10,loop  # print it
    # set up for next iteration
    add $s0,$s0,1   # get next in list
    blz $s0,10,loop # finished whole list? if not, go back around
    # done
out: li sv0,10    # syscalls
```

Loops Example 2: Print Fibonacci numbers

```assembly
li $v0,1           # load code for print integer
syscall
```

```assembly
loop:
    # print this element
    move $a0,$s0         # load value to print with
    syscall
    li $v0,1           # load code for print integer
    syscall
    # print it
    # set up for next iteration
    add $s0,$s0,1       # get next in list
    ble $s0,10,loop     # finished whole list? if not, go back around
    # done
out: li sv0,10     # syscalls
```

Loops Example 2: (Fibonacci - cont.)

```assembly
li $t1,$v0         # save required number of iterations in t1
li $t2,0           # number of this iteration
li $s1,1          # initialize s1 and s2
li $s2,1
```

```assembly
LOOP:
    bge $t2,$t1,DONE # not done, print next one (code for printing has been omitted)
    # not done, print next one (code for printing has been omitted)
    #Increment for next iteration
    add $s0,$s1,$s2   #get next Fibonacci number
    move $s1,$s2     #update s1 and s2
    move $s2,$s0     #Initialization for Fibonacci numbers
    addi $t2,$t2,1   #increment iteration count
    j LOOP
DONE:
    # end program
```

Loops Example 3: array indexing 1

```assembly
la $s0,List        # $s0 holds current address in array
add $s1,$s0,36      # $s1 holds address of last element
li $s2,0           # initialize sum
```

```assembly
loop:
    bgt $s0,$s1,out  # summed entire array?
    # no, get this element
    lw $t0,0($s0)   # load element value
    add $s2,$s2,$t0  # add into sum
    # set up for next iteration
    add $s0,$s0,4   # get address of next element
    j loop          # go back around
    # sum completed, print it (code omitted)
out: <blah blah blah>
```

Loops Example 3: array indexing 2

```assembly
li $s0,0           # $s0 holds current offset in array
li $s1,36          # $s1 holds last offset in array
li $s2,0           # initialize sum
```

```assembly
loop:
    bgt $s0,$s1,out  # summed entire array?
    # no, get this element
    lw $t0,List($s0) # load element value
    add $s2,$s2,$t0  # add into sum
    # set up for next iteration
    add $s0,$s0,4   # get address of next element
    j loop          # go back around
    # sum completed, print it (code omitted)
out: <blah blah blah>
```

Inequalities in MIPS (1/3)

- **Pseudo MIPS inequality instructions:**
  - `blt`, `bgt`, `ble`, `bge`
- **Native MIPS inequality instructions:**
  - “Set on Less Than”
  - Syntax: `slt reg1,reg2,reg3`
  - Meaning: `reg1 = 1;` if `(reg2 < reg3)`; `reg1 = 0;` otherwise
  - In `compreeeze`, “set” means “set to 1”, “reset” means “set to 0”.
More decisions and logic (2/3)

Inequalities in MIPS (2/3)

• 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 \(\Rightarrow (g < h)\)
• 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 (3/3)

• 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 the
  branches?
• What about >?
• What about ≥?

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  goto Loop
  P  beq $t0,$0,Loop
  S  # if $t0==0
  A slt \(\Rightarrow\) beq pair means if(… ≥ …) goto...

“And in conclusion…”

“And in conclusion…”

• In order to help the conditional branches
  make decisions concerning inequalities,
  we introduce a single instruction: “Set
  on Less Than” called slt, slti
• One can store and load (signed and
  unsigned) bytes as well as words
  • Unsigned add/sub don’t cause overflow
• New MIPS Instructions:
  sll, sr1
  slt, slti
  addu, addiu, subu

“And in conclusion…”

• You have all the basics to write loops
  and to manipulate arrays of data.