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 unconditional branch: `j` (jump)
In addition to word data transfers (lw, sw), MIPS has byte data transfers:

- load byte: lb
- store byte: sb
- same format as lw, sw

lb 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?
  • $\text{lb}$: sign extends to fill upper 24 bits
  
  \[
  \begin{array}{cccccccccccccccccccccccc}
  xxxx & xxxx & xxxx & xxxx & xxxx & xxxx & xxxx & xxxx & \text{xzzz} & \text{zzzz} \\
  \end{array}
  \]

  ...is copied to “sign-extend”

• Normally don't want to sign extend chars
• MIPS instruction that doesn’t sign extend when loading bytes:
  load byte unsigned: $\text{lbu}$
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):

  \[
  \begin{array}{c}
  +15 \\
  +3 \\
  +18
  \end{array}
  \]

  \[
  \begin{array}{c}
  1111 \\
  0011 \\
  10010
  \end{array}
  \]

• 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: `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: \(0000\ 0002_{\text{hex}}\)
      \(0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{\text{two}}\)
  • After: \(0000\ 0008_{\text{hex}}\)
      \(0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1000_{\text{two}}\)
  • What arithmetic effect does shift left have?

• Shift Right: `srl` is opposite shift; `>>`
Loops in C/Assembly (1/3)

• Simple loop in C; $A[\cdot]$ is an array of ints

\[
\begin{align*}
do & \{ \\
g & = g + A[i] \\
i & = i + j \\
\} \text{ while } (i \neq h)
\end{align*}
\]

• Rewrite this as:

Loop: $g = g + A[i]$
\[
\begin{align*}
i & = i + j \\
\text{if } (i \neq h) \text{ goto Loop;}
\end{align*}
\]

• Use this mapping:

$g, h, i, j, \text{ base of } A$
$s1, s2, s3, s4, s5$
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:  

```
    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;
```
A more efficient version

lw $t1, $t2($t3) not allowed
offset needs to be a constant

sll $t2,$s4,2 #4*j
sll $t1,$s3,2 #4*i
add $t1,$t1,$s5 #add A[i]

L: lw $t7,0($t1) #$t1=A[i]
add $s1,$s1,$t7 #g=g+A[i]
add $t1,$t1,$t2 # add A[i]
# +=4*j
bne $s3,$s2,L # goto Loop
# if i!=h

In the loop
Original:
1 sll, 3 add, 1 lw, 1 bne
Improved:
2 add, 1 lw, 1 bne
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 code would be: for (i=1; i<= 10; i++) printf(" %d",i)

li $s0,1           # $s0 holds index of loop
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
    addi $s0,$s0,1       # get next in list
    ble $s0,10,loop     # finished whole list?

    # if not, go back around

# done
out:    li $v0,10
syscall
Loops Example 2: Print Fibonacci #s

# t1 is required number of iterations
# t2 is number of iterations so far
# s1 holds current Fibonacci number
# s2 holds next Fibonacci number
# v0 has the user’s input (how many Fibonacci to print out)

# print desired number of Fibonacci numbers
# initialize for while loop
move    $t1,$v0   # save required number of iterations in t1
li       $t2,0    # number of this iteration
li       $s1,1
li       $s2,1
Loops Example 2: (Fibonacci - cont.)

# Check for more to print?
LOOP:   bge  $t2,$t1,DONE
# not done, print next one (code for printing has been omitted)
   blah, blah, blah, …
# increment for next iteration
   add    $s0,$s1,$s2    # get next Fibonacci number
   move   $s1,$s2       # update s1 and s2
   move   $s2,$s0
   addi   $t2,$t2,1     # increment iteration count
   j       LOOP
DONE:
# end program

Modify it to fill up an array with Fibonacci numbers.
Loops Example 3: array indexing 1

# start with array values already there to shorten example
List: .word 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

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

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 4: array indexing 2

# start with array values already there to shorten example
List: .word 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
# initialize for loop
    li  $s0,0         # $s0 holds current offset in array
    li  $s1,36       # $s1 holds last offset in array
    li  $s2,0        # initialize sum

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
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 = (reg2 < reg3);`

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

• In computereese, “set” means “set to 1”, “reset” means “set to 0”.

Same thing...
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
  ```

• 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 (3/3)

• 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 $\text{slt}$ and the branches?

• What about $>$?

• What about $\geq$?
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

Loop:  ... 

MIPS

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

A `slt` ➔ `beq` pair means if(...) goto...
“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, srl`
  - `slt, slti`
  - `addu, addiu, subu`
“And in conclusion…”

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