## CS378: Machine Organization and Assembly Language

Fall 2007

- Topics
  - — Memory review
  - — Memory & Loads/Stores in MIPS
    - write MIPS programs that use more than just registers
  - — Control flow MIPS
    - loops
    - if, if/else and case/switch



## Announcements

- Homework 1 will be posted today
  - — due 1 week from today, Friday, Oct 5th, 5pm
  - — remember the late policy. Use your late days wisely.

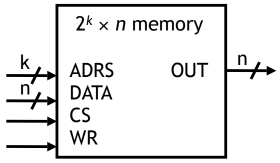
- Find a lab partner soon

- Use feedback form on the webpage to tell us how we are doing

- We might need to shuffle the lecture schedule to accommodate the material you will need for the Lab0 assignment sooner.
  - — the goal is to give you more time for the lab, which is a good thing

## Memory review

- Memory sizes are specified much like register files; here is a $2^k \times n$ RAM.

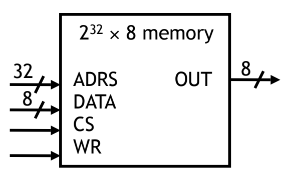



| CS | WR | Operation |
|---|---|---|
| 0 | x | None |
| 1 | 0 | Read selected address |
| 1 | 1 | Write selected address |

- A chip select input CS enables or "disables" the RAM.
- ADRS specifies the memory location to access.
- WR selects between reading from or writing to the memory.
  - To read from memory, WR should be set to 0. OUT will be the n-bit value stored at ADRS.
  - To write to memory, we set WR = 1. DATA is the n-bit value to store in memory.



## MIPS memory




- MIPS memory is byte-addressable, which means that each memory address references an 8-bit quantity.
- The MIPS architecture can support up to 32 address lines.
  - This results in a $2^{32} \times 8$ RAM, which would be 4 GB of memory.
  - Not all actual MIPS machines will have this much!

## Loading and storing bytes

- The MIPS instruction set includes dedicated load and store instructions for accessing memory
- The main difference is that MIPS uses indexed addressing.
  - The address operand specifies a signed constant and a register.
  - These values are added to generate the effective address.
- The MIPS "load byte" instruction lb transfers one byte of data from main memory to a register.

```
lb $t0, 20($a0)     # $t0 = Memory[$a0 + 20]
```

- The "store byte" instruction sb transfers the lowest byte of data from a register into main memory.

```
sb $t0, 20($a0)     # Memory[$a0 + 20] = $t0
```



## Loading and storing words

- You can also load or store 32-bit quantities—a complete word instead of just a byte—with the lw and sw instructions.

```
lw $t0, 20($a0)        # $t0 = Memory[$a0 + 20]
sw $t0, 20($a0)        # Memory[$a0 + 20] = $t0
```

- Most programming languages support several 32-bit data types.
  - Integers
  - Single-precision floating-point numbers
  - Memory addresses, or pointers
- Unless otherwise stated, we'll assume words are the basic unit of data.

## Computing with memory

- So, to compute with memory-based data, you must:
  1. Load the data from memory to the register file.
  2. Do the computation, leaving the result in a register.
  3. Store that value back to memory if needed.
- For example, let's say that you wanted to do the same addition, but the values were in memory. How can we do the following using MIPS assembly language?

```
char A[4] = {1, 2, 3, 4};
int result;

result = A[0] + A[1] + A[2] + A[3];
```



## An array of words

- Remember to be careful with memory addresses when accessing words.
- For instance, assume an array of words begins at address 2000.
  - The first array element is at address 2000.
  - The second word is at address *2004*, not 2001.
- Example, if $a0 contains 2000, then

```
lw $t0, 0($a0)
```

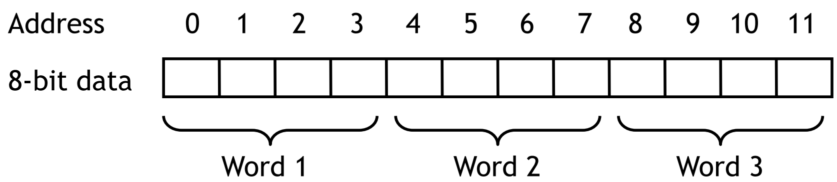
accesses the first word of the array, but

```
lw $t0, 8($a0)
```

would access the *third* word of the array, at address 2008.

## Memory alignment

- Keep in mind that memory is byte-addressable, so a 32-bit word actually occupies four contiguous locations (bytes) of main memory.




- The MIPS architecture requires words to be aligned in memory; 32-bit words must start at an address that is divisible by 4.
  – 0, 4, 8 and 12 are valid word addresses.
  – 1, 2, 3, 5, 6, 7, 9, 10 and 11 are *not* valid word addresses.
  – Unaligned memory accesses result in a bus error, which you may have unfortunately seen before.
- This restriction has relatively little effect on high-level languages and compilers, but it makes things easier and faster for the processor.



## Pseudo-instructions

- MIPS assemblers support pseudo-instructions that give the illusion of a more expressive instruction set, but are actually translated into one or more simpler, "real" instructions.
- For example, you can use the li and move pseudo-instructions:

```
li    $a0, 2000        # Load immediate 2000 into $a0
move  $a1, $t0         # Copy $t0 into $a1
```

- They are probably clearer than their corresponding MIPS instructions:

```
addi  $a0, $0, 2000    # Initialize $a0 to 2000
add   $a1, $t0, $0     # Copy $t0 into $a1
```

- We'll see lots more pseudo-instructions this semester.
  – A complete list of instructions is given in Appendix A of the text.
  – Unless otherwise stated, you can always use pseudo-instructions in your assignments and on exams.

## Control flow in high-level languages

- The instructions in a program usually execute one after another, but it's often necessary to alter the normal control flow.
- Conditional statements execute only if some test expression is true.

```
// Find the absolute value of a0
v0 = a0;
if (v0 < 0)
    v0 = -v0;           // This might not be executed
v1 = v0 + v0;
```

- Loops cause some statements to be executed many times.

```
// Sum the elements of a five-element array a0
v0 = 0;
t0 = 0;
while (t0 < 5) {
    v0 = v0 + a0[t0];   // These statements will
    t0++;               // be executed five times
}
```



## Control-flow graphs

```
// Find the absolute value of a0
v0 = a0;
if (v0 < 0)
    v0 = -v0;
v1 = v0 + v0;
```

```
// Sum the elements of
v0 = 0;
t0 = 0;
while (t0 < 5) {
    v0 = v0 + a0[t0];
    t0++;
}
```

## MIPS control instructions

- MIPS's control-flow instructions

| | |
|---|---|
| j | // for unconditional jumps |
| bne and beq | // for conditional branches |
| slt and slti | // set if less than (w/o and w an immediate) |

- Now we'll talk about
  - MIPS's pseudo branches
  - if/else
  - case/switch



## Pseudo-branches

- The MIPS processor only supports two branch instructions, beq and bne, but to simplify your life the assembler provides the following other branches:

```
blt  $t0, $t1, L1        // Branch if $t0 < $t1
ble  $t0, $t1, L2        // Branch if $t0 <= $t1
bgt  $t0, $t1, L3        // Branch if $t0 > $t1
bge  $t0, $t1, L4        // Branch if $t0 >= $t1
```

- There are also immediate versions of these branches, where the second source is a constant instead of a register.

- Later this quarter we'll see how supporting just beq and bne simplifies the processor design.

## Implementing pseudo-branches

- Most pseudo-branches are implemented using slt. For example, a branch-if-less-than instruction `blt $a0, $a1, Label` is translated into the following.

```
slt  $at, $a0, $a1     // $at = 1 if $a0 < $a1
bne  $at, $0, Label    // Branch if $at != 0
```

- This supports immediate branches, which are also pseudo-instructions. For example, `blti $a0, 5, Label` is translated into two instructions.

```
slti $at, $a0, 5       // $at = 1if $a0 < 5
bne  $at, $0, Label    // Branch if $a0 < 5
```

- All of the pseudo-branches need a register to save the result of slt, even though it's not needed afterwards.
  - MIPS assemblers use register $1, or $at, for temporary storage.
  - You should be careful in using $at in your own programs, as it may be overwritten by assembler-generated code.



## Translating an if-then statement

- We can use branch instructions to translate if-then statements into MIPS assembly code.

```
v0 = a0;
if (v0 < 0)
    v0 = -v0;
v1 = v0 + v0;
```

→

```
move $v0 $a0
bge $v0, $0  Label
sub $v0, 0, $v0
Label: add $v1, $v0, $v0
```

- Sometimes it's easier to *invert* the original condition.
  - In this case, we changed "continue if v0 < 0" to "skip if v0 >= 0".
  - This saves a few instructions in the resulting assembly code.

## What does this code do?

```
label:  sub   $a0, $a0, 1
        bne   $a0, $zero, label
```



## Loops

```
Loop:   j    Loop        # goto Loop
```

---

```
for (i = 0; i < 4; i++) {
  // stuff
}
```

```
        add    $t0, $zero, $zero   # i is initialized to 0, $t0 = 0
Loop:   // stuff
        addi   $t0, $t0, 1         # i ++
        slti   $t1, $t0, 4         # $t1 = 1 if i < 4
        bne    $t1, $zero, Loop    # go to Loop if i < 4
```

## Control-flow Example

- Let's write a program to count how many bits are set in a 32-bit word.

```
int count = 0;
for (int i = 0 ; i < 32 ; i ++) {
   int bit = input & 1;
   if (bit != 0) {
     count ++;
   }
   input = input >> 1;
}
```

```
.text
main:

        li      $a0, 0x1234             ## input = 0x1234
        li      $t0, 0                  ## int count = 0;
        li      $t1, 0                  ## for (int i = 0

main_loop:
        bge     $t1, 32, main_exit      ## exit loop if i >= 32

        andi    $t2, $a0, 1                       ## bit = input & 1
        beq     $t2, $0, main_skip      ## skip if bit == 0

        addi    $t0, $t0, 1             ## count ++

main_skip:
        srl     $a0, $a0, 1             ## input = input >> 1
        add     $t1, $t1, 1             ## i ++

        j       main_loop

main_exit:
        jr      $ra
```



## Translating an if-then-else statements

- If there is an else clause, it is the target of the conditional branch
  - And the then clause needs a jump over the else clause

```
// increase the magnitude of v0 by one
if (v0 < 0)
    v0 --;

else
    v0 ++;
v1 = v0;
```

```
     bge  $v0, $0, E
     sub  $v0, $v0, 1
     j    L

E:   add  $v0, $v0, 1
L:   move $v1, $v0
```

  - Drawing the control-flow graph can help you out.

## Case/Switch Statement

- Many high-level languages support multi-way branches, e.g.

```
switch (two_bits) {
   case 0:    break;
   case 1:    /* fall through */
   case 2:    count ++;     break;
   case 3:    count += 2;   break;
}
```

- We could just translate the code to if, thens, and elses:

```
if ((two_bits == 1) || (two_bits == 2)) {
   count ++;
} else if (two_bits == 3) {
   count += 2;
}
```

- This isn't very efficient if there are many, many cases.



## Case/Switch Statement

```
switch (two_bits) {
   case 0:    break;
   case 1:    /* fall through */
   case 2:    count ++;     break;
   case 3:    count += 2;   break;
}
```

- Alternatively, we can:
  1. Create an array of jump targets
  2. Load the entry indexed by the variable two_bits
  3. Jump to that address using the jump register, or jr, instruction

- This is much easier to show than to tell.