CS378: Machine Organization and Assembly Language

Lecture 2 – Spring 2010

What is an instruction? And a register?
What does register-to-register mean?
In what order is a program executed?

Why do we need memory?

Where are the instructions stored?

What is the C equivalent to: **sub $t0, $t1, $t2** ?

1

Announcements

- Homework 0 posted – *not* graded, just for your benefit
  - — on your own, explore SPIM.
- Homework 1 (for a grade, to be done individually) will be posted Monday.
  - — write a couple of functions in MIPS assembly
  - — due about a week later
- Lab 1 (to be done in partners) posted
  - — Please find a lab partner soon
  - — Or we will find one for you ☺

2

A more complete assembly example

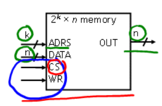
- How would you write code in MIPS assembly to compute:
  - — 1 + 2 + 3 * 4

```
addi $t0, $0, 1
addi $t0, $t0, 2
addi $t1, $0, 3
addi $t2, $0, 4
mult $t1, $t1, $t2
add $t0, $t0, $t1
```

3

Memory review

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

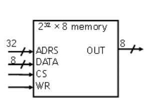


CS	WR	Operation
0	x	None
1	0	Read selected address
1	1	Write selected address

- 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.

4

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} x 8 RAM, which would be 4 GB of memory.
  - – Not all actual MIPS machines will have this much!

5

Loading and storing bytes

- The MIPS instruction set includes dedicated load and store instructions for accessing memory
- MIPS uses indexed addressing to reference memory.
  - – 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
```

6

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? (A's address is in $a0, result's address is in $a1)

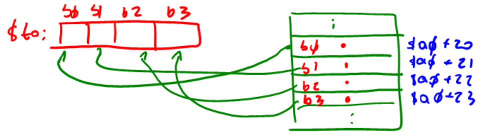
```
char A[4] = {1, 2, 3, 4};
int result;

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


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 words

- Same example, but with 4-byte ints instead of 1-byte chars. What changes? (As before, A's address is in $a0, result's address is in $a1)

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

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


An Array of Words From Memory of Bytes

Use care 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

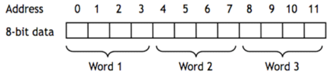
lw $t0, 8($a0)

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

Memory is byte addressed but usually word referenced

Memory Alignment

- Picture words of data stored in byte-addressable memory as follows



- 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 giving the illusion of a more expressive instruction set by translating into one or more simpler, "real" instructions
- For example, li and move are 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 more pseudo-instructions this semester.
  - A complete list of instructions is given in Appendix B
  - Unless otherwise stated, you can always use pseudo-instructions in your assignments and on exams
  - But remember that these do not really exist in the hardware – they are conveniences provided by the assembler