CSE 410
Computer Systems

Hal Perkins

Spring 2010

Lecture 3 – MIPS Instructions

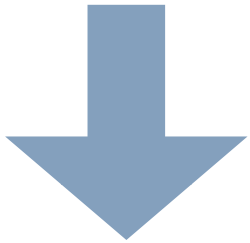
Reading

- Computer Organization and Design
  - Section 2.1, Introduction
  - Section 2.2, Operations of the Computer Hardware
  - Section 2.3, Operands of the Computer Hardware

From Java/C to Machine Language

High-level language

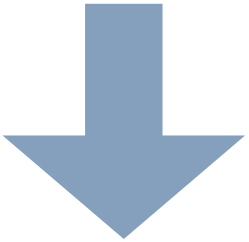
```
a = b + c;
```



Compiler

Assembly Language (MIPS)

```
add $16, $17, $18
```

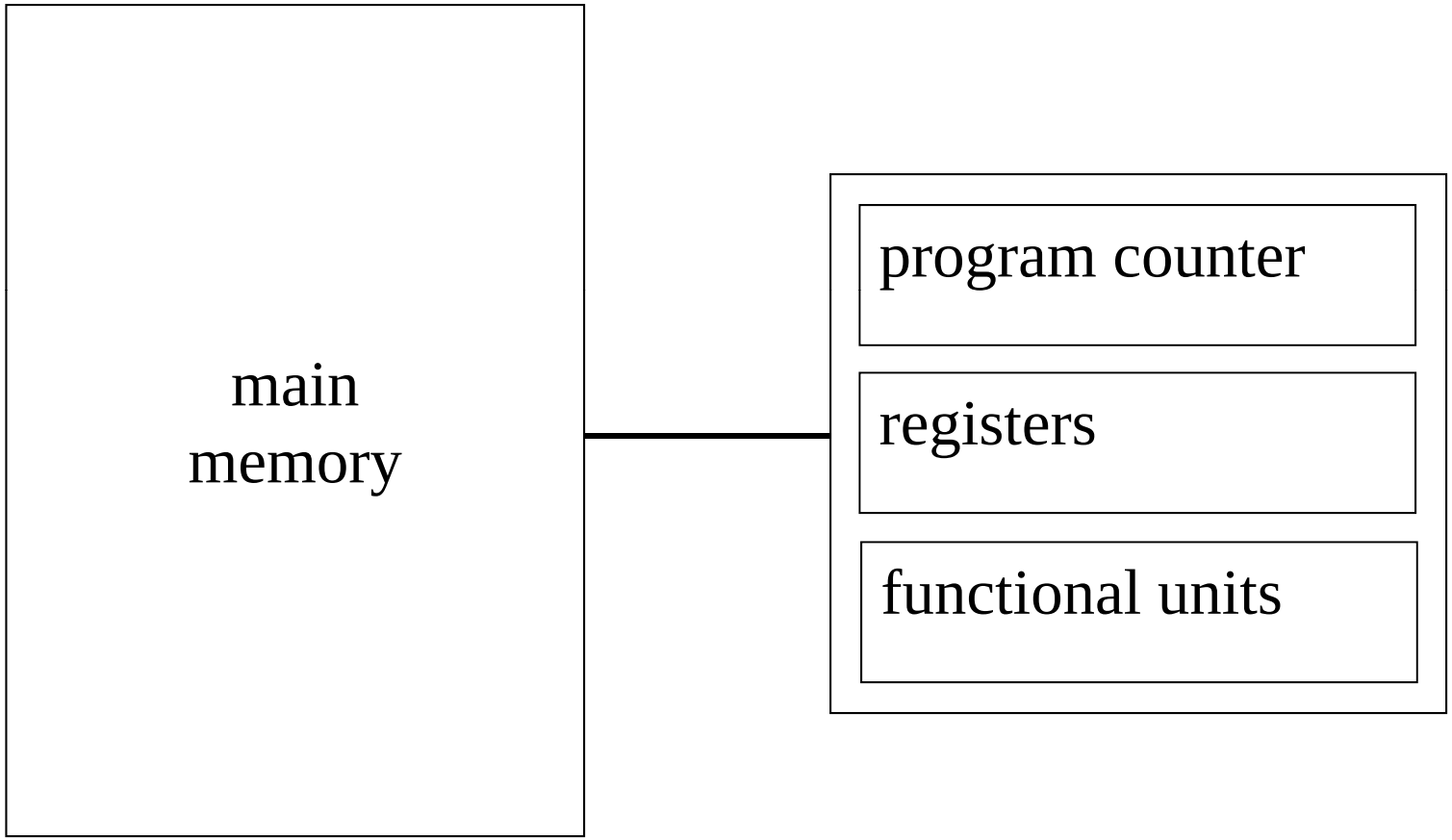


Assembler

Binary Machine Language (MIPS)

```
01010111010101101...
```

The Computer (for now)



Instructions in main memory

- Instructions are stored in main memory
- The program counter (PC) register points to (contains the address of) the next instruction
  - All MIPS instructions are 4 bytes long, and so instruction addresses are always multiples of 4

Fetch/Execute Cycle

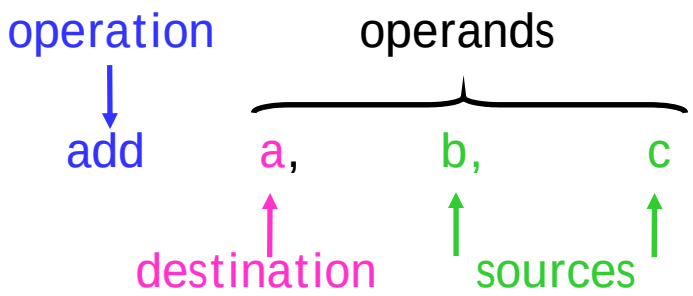
- What the processor does (programmer's view):

```
while (processor not halted) {
    fetch instruction at memory location given by PC;
    PC = PC + 4;   // increment to point to next instruction
    execute fetched instruction;
}
```

- Instructions execute sequentially unless a jump or branch changes the PC to cause the next instruction to be fetched from somewhere else

MIPS: register-to-register, three address

- MIPS is a register-to-register, or load/store, architecture.
  - The destination and sources must all be registers.
  - Special instructions, which we'll see soon, are needed to access main memory.
- MIPS uses three-address instructions for data manipulation.
  - Each instruction contains a destination and two sources.
  - For example, an addition instruction (a = b + c) has the form:



MIPS register file

- MIPS processors have 32 registers, each of which holds a 32-bit value
  - Register addresses (numbers) are 5 bits long
- More registers might seem better, but there is a limit to the goodness
  - It's more expensive, because of both the registers themselves as well as hardware needed to access individual registers
  - Instruction lengths may be affected, as we'll see in the future

MIPS register names

- MIPS register names begin with a $. There are two naming conventions:
  - By number:

    $0 $1 $2 … $31

  - By (mostly) two-character names, such as:

    $a0-$a3 $s0-$s7 $t0-$t9 $sp $ra

MIPS register usage

- Not all of the registers are equivalent:
  - E.g., register $0 or $zero always contains the value 0

    (go ahead, try to change it)
- Other registers have special uses, by convention:
  - E.g., register $sp is used to hold the “stack pointer”
- You have to be a little careful in picking registers for your programs.
  - More about this later

Basic arithmetic and logic operations

- The basic integer arithmetic operations include the following:

  add sub mul div

- And here are a few logical operations:

  and or xor

- Remember that these all require three register operands; for example:

```
add  $t0, $t1, $t2   # $t0 = $t1 + $t2
mul  $s1, $s1, $a0   # $s1 = $s1 x $a0
```

Larger expressions

- More complex arithmetic expressions may require multiple operations at the instruction set level

$$t0 = (t1 + t2) \times (t3 - t4)$$

```
add  $t0, $t1, $t2    # $t0 contains $t1 + $t2
sub  $s0, $t3, $t4    # Temporary value $s0 = $t3-$t4
mul  $t0, $t0, $s0    # $t0 contains the final product
```

- Temporary registers may be necessary, since each MIPS instructions can access only two source registers and one destination
  - In this example, we could re-use $t3 instead of using $s0
  - But be careful not to modify registers that are needed again later

Immediate operands

- The instructions we've seen so far expect register operands. How do you get data into registers in the first place?
  - Some MIPS instructions allow you to specify a signed constant, or "immediate" value, for the second source instead of a register. For example, here is the immediate add instruction, addi:

```
addi $t0, $t1, 4   # $t0 = $t1 + 4
```

  - Immediate operands can be used in conjunction with the $zero register to write constants into registers:

```
addi $t0, $0, 4    # $t0 = 4
```

We need more space!

- Registers are fast and convenient, but we have only 32 of them, and each one is just 32-bits wide
  - That's not enough to hold data structures like large arrays
  - We also can't access data that is wider than 32 bits
- We need to add some main memory to the system!
  - RAM is cheaper and denser than registers, so we can add lots of it
  - But memory is also significantly slower, so registers should be used whenever possible
- In the past, using registers wisely was the programmer's job
  - For example, C has a keyword "register" to mark commonly-used variables which should be kept in a register if possible
  - However, modern compilers do a good job of using registers intelligently and minimizing RAM accesses

MIPS memory

- MIPS memory is byte-addressable, which means that each memory address references an 8-bit quantity
- The MIPS architecture supports up to 32 address bits
  - That means up to 2^{32} bytes, or 4 GB of memory.
  - Not all actual MIPS machines will have this much!
- The MIPS instruction set includes dedicated load and store instructions for accessing memory

Loading and storing bytes

- 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]
```

  - Question: What happens to the other 24 bits of the register?
    - How can we find out?

- 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
```

Memory Addressing

- 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 address of the byte to be loaded or stored

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.

Computing with memory - example

- Let's say that we want to add the numbers in a byte array stored are 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];
```

Word Arrays in Byte Memories

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

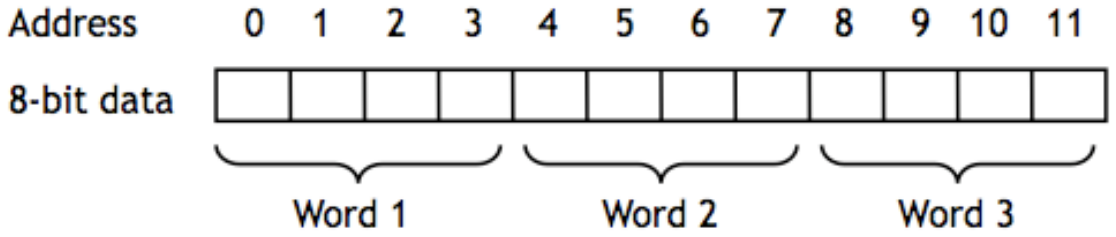
```
lw $t0, 8($a0)
```

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

Memory is byte addressed but usually word referenced

Memory Alignment (reminder)

- Picture words of data stored in byte-addressable memory like this



- 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 quarter.
  - 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