Instructions

Class 29

Addition

- addition is a simple operation, easy to understand
- to begin our study of the MIPS instruction set, we'll use the `add` instruction
- adds two signed integers together

```
add a, b, c
```

  (style note: always a space after comma, never before)
- this is equivalent to the C statement

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

Addition

- assembly language instructions are rigid
- in C, we can do

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

- but there's no equivalent in assembly
- if we wish to accomplish this in assembly, we have to do:

```
add a, b, c    # add b + c, put the result in a
add a, a, d    # add (b + c) + d, put the result in a
```

A More Complex Example

- what if we wish to create the equivalent of this?

```
a = (b + c) - (d + e);
```

```
add t0, b, c    # store b + c into temporary variable t0
add t1, d, e    # store d + e into temporary variable t1
sub a, t0, t1   # final result: a = (b + c) - (d + e)
```

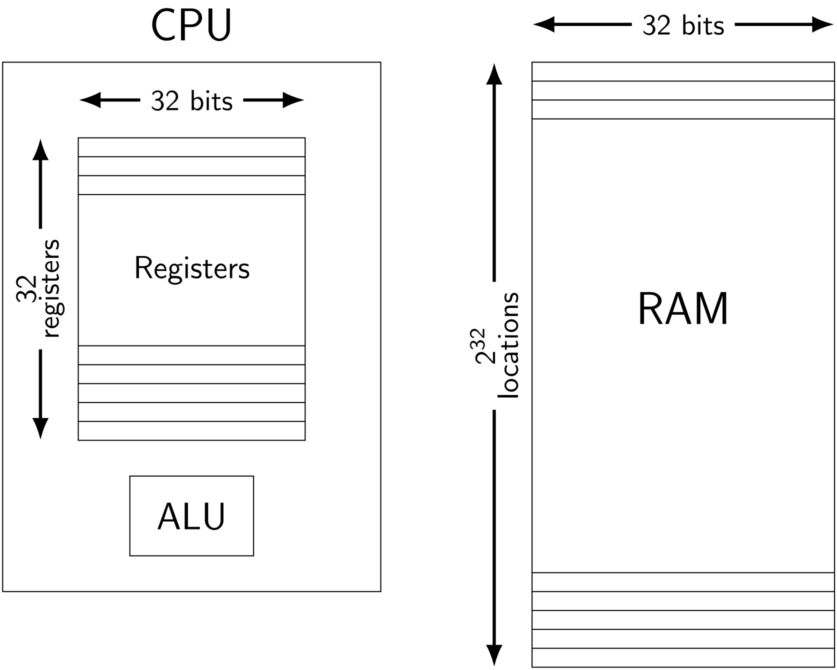
Operands

- so far, I've been talking about variables
- a variable is a storage location in memory
- but MIPS instructions like add can't operate on values in memory locations
- instead arithmetic instructions like add and sub can only operate on the contents of registers
- so our program from the previous slide really looks like this:

```
add $t0, $s1, $s2  # store b + c into register t0
add $t1, $s3, $s4  # store d + e into register t1
sub $s0, $t0, $t1  # final result: a = (b + c) - (d + e)
```

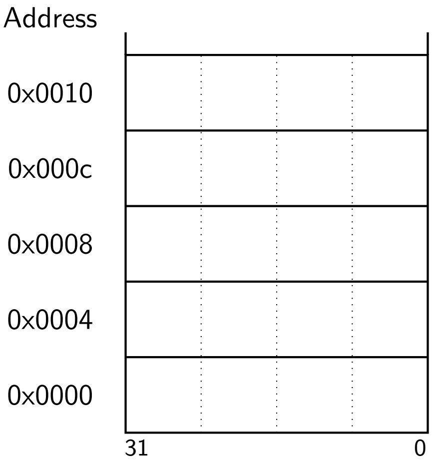
Registers

- now we have two big questions:
  1. how do values get into and out of registers
  2. we only have 32 registers total, and many of them are special-purpose; using the 18 free registers, what if we have an array of 1,000 elements?
- there are only a handful of free registers, so all data structures and variables with persistent data reside in memory
- remember that memory is just a huge linear space


CPU
32 bits
32 registers
Registers
ALU
32 bits
2^{32} locations
RAM

Memory

- memory is byte addressable
- every memory location is a word which is 4 bytes
- MIPS has a big-endian architecture
- 0x0000 is the address of the most significant byte of the word



Load Word

- the MIPS instruction that copies a word from a memory address into a register is `lw`
- immediately we have a problem:
- an instruction must specify:
  - which instruction it is
  - the destination, the register into which the word will go
  - the source, the memory location from which to get the word

- every MIPS machine instruction is exactly 32 bits long

- it takes 6 bits to specify which instruction it is (`lw`)
- it takes 5 bits to specify one of the 32 registers
- it takes 32 bits to specify one of the 2^{32} memory addresses
- we appear to need 43 bits, but we only have 32

Von Neumann Architecture

- remember, instructions and data are in the same memory space
- memory is just a bunch of bits — a bunch of numbers
- how does a number represent an instruction?

- in fact, this is what machine language is: just numbers

Load Word

- the actual format of the lw machine language instruction is this:

$$\underbrace{\text{opcode}}_{\text{6 bits}} \quad \underbrace{\text{source reg}}_{\text{5 bits}} \quad \underbrace{\text{dest reg}}_{\text{5 bits}} \quad \underbrace{\text{literal offset}}_{\text{16 bits}}$$

- an example in assembly language looks like this
- \$t2 is the destination
- the source memory address is the contents of \$s4 plus 8

```
lw    $t2, 8($s4)
```

- \$s4 is called the base register, and the (8) is called the offset
- the offset is in bytes, not words, and can be positive or negative

Load Word

- the actual format of the lw machine language instruction is this:

$$\underbrace{\text{opcode}}_{\text{6 bits}}\quad\underbrace{\text{source reg}}_{\text{5 bits}}\quad\underbrace{\text{dest reg}}_{\text{5 bits}}\quad\underbrace{\text{literal offset}}_{\text{16 bits}}$$

- an example in assembly language looks like this
- $t2 is the destination
- the source memory address is the contents of $s4 plus 8

```
lw    $t2, 8($s4)
```

- $s4 is called the **base register**, and the (8) is called the **offset**
- the offset is in bytes, not words, and can be positive or negative

Offset

- for simple variables, the offset is usually zero
- if $s3 contains the address of variable a
- and we wish to load a into $t4
- all we have to do is

```
lw   $t4, 0($s3)
```

Offset

- the offset can be important when working with arrays
- often a register will have the starting address of an array
- we wish to access a specific element of the array
- assume A is an array of 10 words; $s2 holds the address of the first word of A
- then `A[0]` is at location `0($s2)`
- and `A[1]` is at location `4($s2)`
- and `A[9]` is at location `36($s2)`

- if `$s2` is pointing to `A[4]`, then `-4($s2)` refers to `A[3]`

Example

assume:

- A is an array of 100 words
- a is a variable currently associated with register $s1
- b is a variable currently associated with register $s2
- the address of the first word of A is currently stored in $s3

the C language statement:

```
a = b + A[8];
```

becomes the assembly language statements:

```
lw   $t0, 32($s3)
add  $s1, $s2, $t0
```

- remember: A is an array of words
- offset is always in bytes

Store

- the complement to `lw` is store word `sw`
- it has exactly the same format as load word
- consider the C language statement

```
a = b;
```

- assuming \$s2 has the address of a and \$s4 has b's address, this could be implemented in assembly language as

```
lw   $t0, 0($s4)  # copy b to $t0
sw   $t0, 0($s2)  # copy $t0 to a
```

- MIPS has no instruction to copy the contents of one memory address directly to another memory address

Immediate

- the computational instructions use registers for operands
- but many instructions require an immediate, or literal value
- e.g., `i++;` is the most common instruction in all of programming
- this instruction requires the literal 1

- the MIPS instruction set includes many instructions that have a register as one operand and a literal for the other

```
addi     $s3, $s3, 1   # $s3++
addi     $s1, $s1, 4   # $s1 += 4
```

MIPS Instruction Formats

- there are 3 main instruction formats.
  format R: `opcode rs rt rd sh_amt funct`
  format I: `opcode rs rt immediate`
  format J: `opcode address`

- opcode: 6 bits that determine which instruction this is
- rs: the address of a source register
- rt: a second register address
- rd: the address of a destination register
- sh_amt: a shift amount
- funct: a code that selects the variant of the operation
- address: a literal address in memory

Characters

- a character is 8 bits
- a MIPS word is 4 bytes: 32 bits
- to store characters in memory, there are two options
  - one character per word: simple, but wastes 75% of space
  - one character per byte: efficient, requires instructions to extract individual bytes from words

- to store 4 characters in a word, we need operations that can extract 8 bits out of 32
- we have already seen these in C
  - shift
  - bitwise and with a mask
  - bitwise or with a mask

A Single Byte

- we have a 32-bit word

  `xxxxxxxx 10101110 xxxxxxxx xxxxxxxx`

- we want just the 3rd byte
  1. shift the word 16 places to the right

     `00000000 00000000 xxxxxxxx 10101110`

  2. and the result with the mask 0xFF

```
00000000 00000000 xxxxxxxx 10101110
00000000 00000000 00000000 11111111
-----------------------------------
00000000 00000000 00000000 10101110
```