Warm-Up Problem

What is the Fetch-Execute Cycle? Try to remember the order of the steps without looking it up!

CS 241 Lecture 3

More on Machine Language

With thanks to Brad Lushman, Troy Vasiga, Kevin Lanctot, and Carmen Bruni

Putting Values In Registers

- This works well provided you already had values in registers!
- How to we put values in registers? In CS241 we give you a non-standard MIPS command:

```
lis $d: 0000 0000 0000 0000 dddd d000 0001 0100
```

- Load immediate and skip. This places the next value in RAM [an immediate] into $d and increments the program counter by 4 (it skips the next line which is usually not an instruction).
- OK, but how do we get the value we care about into the next location in RAM?

Putting Values In Registers

```
.word i: iiii iiii iiii iiii iiii iiii iiii iiii
```

- This is an assembler directive (not a MIPS instruction). The value *i*, as a two's complement integer, is placed in the correct memory location in RAM as it occurs in the code.
- Can also use hexadecimal values: `0xi`

Example

Write a MIPS program that adds together 11 and 13 and stores the result in register $3.

```
lis  $8          0000 0000 0000 0000 0100 0000 0001 0100
.word  11        0000 0000 0000 0000 0000 0000 0000 1011
lis  $9          0000 0000 0000 0000 0100 1000 0001 0100
.word  0xd       0000 0000 0000 0000 0000 0000 0000 1101
add  $3,$8,$9    0000 0001 0000 1001 0001 1000 0010 0000
```

The code on the left is what we call **Assembly Code**.

The code on the right is what we call **Machine Code**.

Example

Previous example revisited.
What it looks like in RAM with memory locations

0x00000000	0000 0000 0000 0000 0100 0000 0001 0100
0x00000004	0000 0000 0000 0000 0000 0000 0000 1011
0x00000008	0000 0000 0000 0000 0100 1000 0001 0100
0x0000000c	0000 0000 0000 0000 0000 0000 0000 1101
0x00000010	0000 0001 0000 1001 0001 1000 0010 0000

Incomplete examples

The two examples above are still incomplete. Recall that the fetch-execute cycle has a while loop that we still haven't exited yet. How to we return control back to the loader?

```
jr  $s
0000 00ss sss0 0000 0000 0000 0000 1000
```

Jump Register. Sets the `pc` to be `$s`.

For us, our return address will typically be in $31, so we will typically call

```
jr  $31
```

```
0000 0011 1110 0000 0000 0000 0000 1000
```

This command returns control to the loader.

Complete Example

Write a MIPS program that adds together 11 and 13 and stores the result in register $3.

```
lis $8          0000 0000 0000 0000 0100 0000 0001 0100
.word 11        0000 0000 0000 0000 0000 0000 0000 1011
lis $9          0000 0000 0000 0000 0100 1000 0001 0100
.word 0xd       0000 0000 0000 0000 0000 0000 0000 1101
add $3,$8,$9    0000 0001 0000 1001 0001 1000 0010 0000
jr $31          0000 0011 1110 0000 0000 0000 0000 1000
```

More Operations

- There is an issue with multiplying words
- Multiplying two words together might give a word that requires twice as much space! E.g., $2^{30} \cdot 2^{30} = 2^{60}$.
- To fix this, we use the two special registers hi and lo.

More Operations

```
mult $s, $t
0000 00ss ssst tttt 0000 0000 0001 1000
```

- Performs the multiplication and places the most significant word (largest 4 bytes) in hi and the least significant word in lo.

```
div $s, $t
0000 00ss ssst tttt 0000 0000 0001 1010
```

- Performs integer division and places the quotient $s / $t in lo [lo quo] and the remainder $s % $t in hi. Note the sign of the remainder matches the sign of the divisor stored in $s.

More Operations

There are also unsigned versions of these operations (check the reference sheet!)

Wait a Minute…

Multiplication and division happen on these special registers hi and lo. How can I access the data?

```
mfhi $d
0000 0000 0000 0000 dddd d000 0001 0000
```

- Move from register hi into register $d.

```
mflo $d
0000 0000 0000 0000 dddd d000 0001 0010
```

- Move from register lo into register $d.

RAM

- Large[r] amount of memory stored off the CPU.
- RAM access is slower than register access (but is larger, as a tradeoff).
- Data travels between RAM and the CPU via the bus.
- Modern day RAM consists of in the neighbourhood of 10^{10} bytes.
- Instructions occur in RAM starting with address 0 and increase by the word size (in our case 4).
  - But, this simplification will vanish later…

More on RAM

- Each memory block in RAM has an address; say from 0 to $n - 1$.
- Words occur every 4 bytes, starting with byte 0. Indexed by 0, 4, 8, ... $n - 4$.
- Words are formed from consecutive, aligned (usually) bytes.
- Cannot directly use the data in the RAM. Must transfer first to registers.

Operations on RAM

```
lw $t, i($s)
1000 11ss ssst tttt iiii iiii iiii iiii
```

- Load word. Takes a word from RAM and places it into a register. Specifically, load the word in MEM[$s + i] and store in $t.

```
sw $t, i($s)
1010 11ss ssst tttt iiii iiii iiii iiii
```

- Store word. Takes a word from a register and stores it into RAM. Specifically, load the word in $t and store it in MEM[$s + i].

Note that i must be an immediate, NOT another register!

Example

Suppose that $1 contains the address of an array and $2 takes the number of elements in this array (assume less than 220). Place the number 7 in the last possible spot in the array.

```
lis $8
.word 0x7
lis $9
.word 4
mult $2, $9
mflo $3
add $3, $3, $1
sw $8, -4($3)
jr $31
```

Branching

- MIPS also comes equipped with control statements.

```
beq $s, $t, i
0001 00ss ssst tttt iiii iiii iiii iiii
```

- Branch on equal. If $s == $t then pc += i * 4. That is, skip ahead i many instructions if $s and $t are equal.

```
bne $s, $t, i
0001 01ss ssst tttt iiii iiii iiii iiii
```

- Branch on not equal. If $s != $t then pc += i * 4. That is, skip ahead i many instructions if $s and $t are not equal.

Example

Write an assembly language MIPS program that places the value 3 in register $2 if the signed number in register $1 is odd and places the value 11 in register $2 if the number is even.

```
lis $8
.word 2
lis $9
.word 3
lis $2
.word 11
div $1, $8
mfhi $3
beq $3, $0, 1
add $2, $9, $0
jr $31
```