Warm Up Problem

What is the Fetch-Execute Cycle?

CS 241 Lecture 3

More on Machine Language

With thanks to Brad Lushman, Troy Vasiga and Kevin Lanctot

Warm Up Problem

Write an assembly language MIPS program that can calculate when you were born (that is, it actually does the year subtraction to figure out your age). Put the answer in register 3.

Another question: What is the range of integers that a two's complement integer could express if you are allowed 4 bytes of memory?

CS 241 Lecture 3

More on Machine Language

With thanks to Brad Lushman, Troy Vasiga and Kevin Lanctot

Incomplete examples

The examples from the previous lecture were 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?

Incomplete examples

The examples from the previous lecture were 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
```

which is

```
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.

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 and division; what is this problem?

More Operations

There is an issue with multiplying and division; what is this problem?
Multiplying two words together might give a word that requires twice as much space! Eg. $2^{30} \cdot 2^{30} = 2^{60}$.

To fix this, we use the two special registers `hi` and `lo`.
`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`.

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?

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 of the CPU.
- RAM access is slower than register access (but is larger - tradeoff).
- Data travels between RAM and the CPU via the bus.
- Modern day RAM consists of in the neighbourhood of $n = 10^9$ bytes with $4 \mid n$ where here 4 is our word size.
- Instructions occur in RAM starting with address 0 and increase by the word size (in our case 4).

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 form consecutive bytes.
- Cannot use the data on 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 2^{20}). Place the number 7 in the last possible spot in the array.

Example

Suppose that \$1 contains the address of an array and \$2 takes the number of elements in this array (assume less than 2^{20}). 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.

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
```

Inequality Command

`slt $d, $s, $t`: 0000 00ss ssst tttt dddd d000 0010 1010

Set Less Than. Sets the value of register `$d` to be 1 provided the value in register `$s` is less than the value in register `$t` and sets it to be 0 otherwise.

Note: Again there is also an unsigned version of this command. See the reference sheet.

Example

Write an assembly language MIPS program that negates the value in register $1 provided it is positive.

Example

Write an assembly language MIPS program that negates the value in register $1 provided it is positive.

```
slt $2, $1, $0
bne $2, $0, 1
sub $1, $0, $1
jr $31
```

Idea: Register 2 is 0 if register 1 is non-negative. Branch if register 2 is not zero. Otherwise negate register 1.

Exercise (No solution in notes)

Write an assembly language MIPS program that places the absolute value of register $1 in register $2.

Looping

With branching, we can even do looping!!!

Write an assembly language MIPS program that adds together all even numbers from 1 to 20 inclusive. Store the answer in register $3.

Looping

With branching, we can even do looping!!!

Write an assembly language MIPS program that adds together all even numbers from 1 to 20 inclusive. Store the answer in register $3.

```
lis $2
.word 20
lis $1
.word 2
add $3, $0, $0
add $3, $3, $2 ; line -3
sub $2, $2, $1 ; line -2
bne $2, $0, -3 ; line -1 from here
jr $31
```

Note: The semi colon above denotes commented code.

It Works But...

... you're a good programmer right? It should instinctively bother you of our use of the hard coded -3 in the previous slide. If we were to say add a new instruction in between the lines specified by our branching, all of our numbers would be incorrect. How can we fix this?

It Works But...

... you're a good programmer right? It should instinctively bother you of our use of the hard coded -3 in the previous slide. If we were to say add a new instruction in between the lines specified by our branching, all of our numbers would be incorrect. How can we fix this?

Our answer will be to use a label:

```
label:  Operation commands
```

Note that labels do not get their own line number if followed by a white space, that is, a label needs to be followed by an operation and it is both the label and the operation that share the line number.

Note: A label at the end of code is allowed (it would be the address of the first line after your program).

Explicit Example

```
sub $3, $0, $0
sample:
  add $1, $0, $0
```

Notice that both `sample` and `add $1, $0, $0` would have the same memory address in RAM of `0x00000004`.

Note also that labels can occur after the last line of code (it would represent the address of the end of the program).

Looping Revisited

Edit the previously created assembly language MIPS program that added together all even numbers from 1 to 20 inclusive to use a label instead of hard coded numbers in branching.

Looping Revisited

Edit the previously created assembly language MIPS program that added together all even numbers from 1 to 20 inclusive to use a label instead of hard coded numbers in branching.

```
lis $2
.word 20
lis $1
.word 2
add $3, $0, $0
top:
  add $3, $3, $2 ; 0x14
  sub $2, $2, $1
  bne $2, $0, top
jr $31
```

Looping Revisited

```
lis $2
.word 20
lis $1
.word 2
add $3, $0, $0
top:
  add $3, $3, $2 ; 0x14
  sub $2, $2, $1
  bne $2, $0, top
jr $31
```

Note that top is computed by the assembler to be the difference between the program counter and `top`. That is, here it computes `(top-PC)/4` which is `(0x14 - 0x20)/4 = -3`
Recall the Fetch-Execute cycle. PC is the line number after the current line since the branch has been stored in the IR first, then the PC incremented then the IR executed.