Lecture 2

Sept 12

Midterm Oct. 25 7pm - 8:50pm

Encoding integers with 32 bits - unsigned 0 to $2^32 - 1$ - signed 2^31 to $2^31 - 1$

2's compliment Instead of regular binary numbers

$$2^n * a_n, 2^n - 1, a_{n-1}...2^0 n_0$$

Flip the sign of the highest power of 2 like:

$$-2^n * a_n, 2^n - 1, a_{n-1}...2^0 n_0$$

Benefits of 2's compliment - each number has a unique bit representation - can add to positive numbers regularly

- Hardware implements arithmetic mod 2^32
- Theorem: if $x \equiv x` mod n$ $y \equiv y` mod n$ then $x + y \equiv x` + y` mod n$
- same to add, subtract, multiply
- Need separate hardware for division
- Memory, 32 bits with 2^22 addresses
- CPU 34 registers with 32 bits per register
- Next to the CPU is the Control Unit
  - Circuit that changes the bits based
  - implements a function 'Step'
    - a function from State_1 -> State_2 where States are the set of all bit strings
- A stored program computer
  - Includes a program as part of the input
  - Rather than hardcoding operations, use a piece of the memory to encode the program
  - So the hardware step function must be general to read these instructions

Registers and Memory Locations - PC (Program Counter) - **Note memory addresses differ by increments of 4** -

```
Step(state) = {
  // store instruction
  instr = state.mem[state.reg[PC]]
  state2 = state.setReg(PC, state.reg[PC] + 4)
```

```
  // execute instruction returns new state
  instr match {
    ...
  }
}
```

- different instructions for signed/unsigned inputs
- MIPS instruction sheet https://www.student.cs.uwaterloo.ca/~cs241e/current/mipsref.pdf
- magic value has to be written to PC to make marmoset stop running your code. Stored in reg 31?

Lecture 3

Sept 12

opcode a word that represents a machine language instruction *assembly lang.* is a language for writing machine language programs using opcodes instead of bits *assembler* is a program that transofrms assembly language to machine language

Find the absolute value of whatever is in register 1

```
SLT $2 $1 $0
BEQ $2 $0
SUB $1, $0, $1
```

Sept 19th Lecture

Linking and Relocation

Def. **Relocation**: the process of adjusting addresses in macine lang code so it can be loaded at different memory addresses.

Def. An **object file** contains:

- machine lang code
- metadata describing what the labels used to be

Assembly (a.s, b.s) -> *Assembler* -> Object Files (machine lang + metadata) (a.o, b.o) -> *Linker* -> Linked Machine Lang Program

Def. **Linking** is the process of combining object files into one program or library or object file

Example we want to export the labels from the following in our object file so we can Link

```
(b.s)
Define(a)
lis $1
Use(b)
jalr $1
jr $31
Define(b)
jr $31
```

Gets assembled to this which includes label info

```
(b.o)
0 lis $1
4 16
8 jalr $1
12 jalr $31
16 jr $31
meta:
export label a = 0
       label b = 16
```

Another Example for Assembling Multiple Files with Dependent Labels

```
(a.s)
lis $1
Use(a)
jalr $1
jr $31
```

the label a is not defined so when we assemble we need to add metadata for the label a. So our object file would look something like

```
(a.o)
0 lis $1
4 ???? 0
8 jalr $1
12 jr $31
meta: use label a at address 4
```

Now when we Link a.o and b.o we resolve the missing Labels by using the metadata. Note our assembled addresses in b.o would be wrong if we naively linked a.o and b.o so we must **Relocate** the addresses in b.o (add a offset to each offset in this case by adding 16) **We also must update the exported labels with this offset**

The Linked version:

```
0 lis $1
4 20
8 jalr $1
```

```
12 jr $31
16 lis $1
20 32
24 jalr $1
28 jalr $31
32 jr $31
```

Linking should give the same output as concating the assembled code (as long as we don't have constants that rely on offsets)

Variables

Def. A **variable** is an abstraction of a storage location that has a value we translate variable into machine concepts

Def. the **extent** of a variable **instance** is the time interval in which its value can be accessed

e.g. global variable -> entire execution of the program procedure

local variable -> one execution of procedure

e.g.

```
def fact(x:Int): Int = if(x < 2) 1 else x * fact(x - 1)
fact(3) = 3 * fact(2) = 3 * 2 * fact(1) = 3 * 2 * 1
```

Where is variable x? There is only one variable x but there are three instances at runtime

We can implement this with a stack. We'll put the start of the stack at the end of memory. When we push we'll use lower addresses and when we pop we'll use higher.

Stack:

- designate storage register to store address of top of stack (**stack pointer**)
- push a word
  - decrease stack pointer by 4
- pop a word
  - increase stack pointer by 4
- by convention Reg(30) stores the stack pointer (sp)

Reference

Mips Reference Sheet