CS 241, Lecture 2 - Characters and Machine Language

1 Characters

- ASCII:
  - 0-31: control
  - 48-57: 0-9
  - 65-90: A-Z
  - 97-122: a-z
  - $|\text{A-a}| = 32$

2 Bitwise Operations

- In C, we have bitwise operations.
- ~ (not) flips all bits.
- & (and) will give the AND of two binary strings.
- | (or) will give the OR of two binary strings.
- ^ (xor) will give the XOR of two binary strings.
- $>>$ and $<<$ (right and left shift, respectively) will do a logical shift right and left by the given integer, respectively. A left shift is equal to multiplying by 2.
  - Logical shift: Always shift zeroes for both left and right.
  - Arithmetic shift: If left, shift zero. If right, shift MSB.
  - We get undefined behaviour if we right shift a signed integer.
  - For our case, we will ALWAYS use logical shifts!

3 Instructions

- Programs reside in the same place on the data they operate on - they are also data (von Neumann architecture).
- Thus, we cannot really distinguish, without more information, whether a data is a program (set of instructions) or if it just data!
- For this course, we will work with 32-bit MIPS.
- Our MIPS CPU contains 32 registers, $0, ... ,$31, and hi, lo.
- Our control unit contains our PC and instruction register (IR).
- We have two memory registers - memory data register (MDR) and memory address register (MAR).
- We also have an ALU.
- We use a bus to transfer data from CPU to memory.
- Wew have a few types of memory that we use in computers (from fastest to slowest):
  - CPU/registers
  - L1 cache (SRAM)
  - L2 cache (SRAM)
  - RAM (DRAM)
  - disk
  - network memory
- Each register can be represented in 5 bits.
- The registers we get to control with MIPS are our 32 registers (and hi and lo). Note some registers are special and we do not need to touch:
  - $0 is ALWAYS 0
  - $31 is used for return addresses
  - $30 is our stack pointer

- $29 are frame pointers

- Note that in most MIPS standards, $29 and $30's roles are reversed. Why? Who knows.
- MIPS takes instructions loaded from RAM and executes them.
- To solve our earlier problem with confusing data and programs, we will state that anything with memory address 0 in RAM is an instruction.
- After we execute an instruction, we go to the NEXT instruction labelled by our PC.
- We use a program called a loader to put our program into memory, and then sets the PC to be the first address.
- CPUs basically follow a fetch-execute cycle, in which the following occurs:

```
PC = 0
while true do
        IR = MEM[PC]
        PC += 4
        Decode and execute instruction in IR
end while
```

- Eventually some instruction will break out of the infinite loop.

4 Adding, lis, and returning

- Let us write an example program that takes in registers $8 and $9 and stores it in $3:

```
; add $3 $8 $9 is what we want - binary equiv. is 00011, 01000, 01001, respectively.
; we see that our binary equivalent is 0000 00ss ssst tttt dddd d000 0010 0000
; now let us insert our value, and get 0000 0001 0000 1001 0001 1000 0010 0000
; so we thus reduce this to a 32-bit word in hexadecimal: 0x01091820
```

```
.word 0x01091820
.word 0x03e00008 ; jr $31, return
```

- Note we HAVE to jump to register \$31 to end our program - we do so with `jr $31`.
- But this assumes you had values **in** the register - how do we put immediate values in registers? We use the `lis $d` MIPS command.
- Note that `lis` is **non-standard**, normally we would use addi to add immediate values.
- This will place the next word, immediately after, into register \$d. It will increment the PC by 4 and skip the next line as it is (usually) NOT an instruction.
- Note this value is a two's complement integer.
- Let us write a MIPS program that adds 11 and 13:

```
.word 0x00000814 ; lis $1
.word 0x0000000b ; ivalue 11
.word 0x00001014 ; lis $2
.word 0x0000000d ; ivalue 13
.word 0x00221820 ; add $1 and $2, store in $3
.word 0x03e00008 ; jr $31
```

5 Multiplication and Division

- We get a problem with multiplying and division - we may need more space when multiplying (ie: $2^{30} \times 2^{30} = 2^{60}$), and when dividing, we want the quotient and remainder!
- For multiplication, we COMBINE the hi and lo registers to get a 64-bit register. The most significant word is placed in hi, and least significant word in lo (most sig word is largest 4 bytes).
- For division, we put the quotient \$s $\div$ \$t in lo and the remainder \$s % \$t in hi.

- To access data from hi and lo, we use the mfhi \$d and mflo \$d instructions to move from the hi/lo register to the register we state.