Lecture 5

We begin our descent into assembly language :D

Assembly Language Specifics

The MIPS architecture is simple, so we will use it

- compare to ARM 8 (smartphones), x86
- MIPS is still used in some embedded systems (e.g. routers)
- RISC type architecture (reduced instruction set computer)
- **RISC** - ideology, smaller set of simpler insitructions
- ARM is a newer example of RISC

Architecture - Specifies hardware (specifications that must be obeyed), and CPU instruction set

- Each instruction set has a corresponding assembly language

- We will use a MIPS simulator / emulator to simulate a MIPS computer (specifically, MARS)
- Architectures are of a certain size -- e.g. 32bit, 64bit, ...
  - This bit value is the natural unit of data used by the processor, and we call this unit a **word**
  - Every register is 1 word wide, all data channels are 1 word wide, and so on

- Assembly language instructions are mnemonics for CPU instructions (which are in binary)
  - The exception being pseudoinstructions, which don't have a direct mapping to CPU instructions

- All data manipulations happen from CPU registers
  - Small / fast memory, built into the CPU.
  - As MIPS is 32-bit, there are 32 registers
  - Registers are split into categories: general purpose registers (`a0 - a3, s0 - s7, ...`), special registers (program counters, instruction register, etc, which we don't touch directly)
  - They're all the same from the hardware point of view, but we make additional agreements about how we treat them (e.g. t registers vs s registers)

- Example instruction
  - c = a + b
  - ADD c, a, b
  - ADD $s5, $s0, $s1
  - ADD *destination , num1 , num2*

[diagram of RAM memory, separated into blocks, each 1 byte large]

- We can load more than 1 byte from memory at a time without any compromise of speed, since our data channels are wide enough to fit multiple bytes (4 in 32bit architectures)

- Simplest load instruction (to load data from memory) is LW (load word)
  - LW *destination* , *Memory Address*
  - The second field is a problem (writing the memory address directly), because memory addresses are 32 bits long (too much to write out, and it won't fit in the instruction call, which itself is 32 bits long)
  - Solution: indirect addressing
    - LW *$t3* , *Register w/ Memory Address*
    - Put memory address in register
    - Add offset to memory address
    - LW *$t3* , *â($s7)*
    - "That many bytes from the address in $s7"
    - Lends itself well to arrays

Aliasing

- When loading data, there are certain rules that must be followed (this is to simplify computer architecture)
- 4 byte large data must begin at addresses that are divisible by 4, 2 byte large data must begin at addresses that are divisible by 2, and so on.
  - E.g. you can't LW into address *1* , but you can into address *0* or *4*

- A consequence of this is that sometimes you may have to introduce "padding" between data in order to maintain aliasing rules
  - meaning, if you load 2 bytes into register *0* , and then decide to load 4 bytes, you'd have to start is at register *4*. This means you're filling addresses *0* and *1* , leaving addresses *2* and *3* empty, and then filling addresses 4 - 7 (inclusive)

- Aliasing rules apply to C++ as well
  - The compiler would probably optimize your bad decisions away, but for the sake of example:
  - ```
    struct s1 {          struct s2 {

        char c;              int i;

        int i;               short s;

        short s; };          char c; };
    ```

  - Struct s1 ends up taking 10 bytes (1 + 3 + 4 + 2), while struct s2 takes up 7 bytes (4 + 2 + 1), due to accounting for aliasing. In struct s1, after *char c* is declared, there are 3 bytes padded out before *int i* is loaded into memory, because its memory address must be a multiple of 4.
    ```

Some assembly instructions

- LW - load word
- LH - load half word
- LB - load byte
- SW - store word
  - SW *register (take from) , memory (store at)*
  - SW *$t3 , 4($s7)*