Lecture 23

Assembly Operations

- Memory operands
  - arithmetic operations cannot directly access memory
- Memory instructions
  - 32 variables is too limiting
  - Store data in memory and move data to/from memory using loads and stores
- Byte addressable

Byte Address	Word Address	Data	Word Number
3 2 1 0	0x00000000	01 23 45 67	word 0
7 6 5 4	0x00000004	FF EE DD CC	word 1
B A 9 8	0x00000008	AA BB CC DD	word 2
F E D C	0x0000000C	FF FF FF FF	word 3
13 12 11 10	0x00000010	00 00 00 01	word 4

then `a = mem[2];` becomes `lw s7, 8(zero)`

- Specifies memory address using an offset added to a base register
  - In example, base = 0, offset = 8
  - lw addr should be word aligned (evenly divisible by 4)
- Store word instruction writes a data word from a register into memory
  - `mem[3] = 0x42;` becomes

```
addi t3, zero, 0x42
sw t3, 12(zero)
```

Assembly Example

```
.data
LIST: .word 1, 2, 3, 4

.global _start
.text

_start:
    la s1, LIST
    lw s2, 0(s1)
    lw s3, 4(s1)
    add s2, s2, s3
    lw s3, 8(s1)
    add s2, s2, s3
    lw s3, 12(s1)
```

```
    add s2, s2, s3
END: ebreak
```

- Assemble Directives
  - .data: global data section
  - .text: where instructions go
  - .global: label is visible to other files
  - guide assembler in allocating, initialising global variables, defining constants, differentiating between code and data
  - Don't actually become code; are remove by assembler
- `LIST`
  - a label to refer to stuff in code
  - Assembler would change into address
  - no need to figure out all the addresses
- `_start`
  - Also a label; like main in C
  - Where program begins
- `la`
  - pseudocode
  - Load address of global data
- `ebreak`
  - Transfers control to debugger
  - Stops processor from continuing to execute
  - Or else processor will continue executing regardless

Logic Instructions

- bitwise operation blt 2 source registers

```
# s1 = 0111 0101 1001 0010 0001 1111 0000 1010
# s2 = 1111 1111 1111 1111 0000 0000 0000 0000
and s3, s1, s2
# s3 = 0111 0101 1001 0010 0000 0000 0000 0000
or s4, s1, s2
# s4 = 1111 1111 1111 1111 0001 1111 0000 1010
xor s5, s1, s2
# s5 = 1000 1010 0110 1101 0001 1111 0000 1010
not s6, s1
# s6 = 1000 1010 0110 1101 1110 0000 1111 0101
```

Note that `not` is pseudocode, it is actually `xori s6, s1, -1` where -1 gives all 1s