Lecture 23, Nov 1, 2022

RISC-V Instructions Continued

- Arithmetic operations can only access registers and immediates, not main memory
  - We need a memory instruction to first retrieve a value from memory before it can be used, and then write back a value if needed
  - This is known as a *load-store* architecture – we can only access memory via loads and stores
- Memory operations
  - Load word instruction reads a data word from memory into a register (reads 4 bytes at once)
    - e.g. `lw s0, 8(zero)` performs `a = mem[2];`
      - `8` is the *offset address*, `(zero)` is the *base address*
      - We are using the zero register to start at address 0 and offset by 8, so we're accessing address `0x00000008`
      - Note memory is byte addressable, so address 8 is the third word
    - Load word requires the address to be *word-aligned*, that is, a multiple of 4
      - Can't load a word that's split up into two places in memory
  - Store word instruction writes a data word from a register into memory (writes 4 bytes at once)
    - e.g. `sw s0, 12(zero)` performs `mem[3] = a;`

Basic Assembly Program

```
.data # Global data section - stores data used by the whole program

# LIST is a label, which we can use to refer to the data later
# These 4 words could be stored anywhere, but they are guaranteed to be contiguous
LIST: .word 1, 2, 3, 4 ; Declare 4 words, initialize to 1, 2, 3, 4

.text # Program instructions

_start: # The entry point of the program; another label
    la s1, LIST # Load address of LIST into s1
    lw s2, 0(s1) # s2 = mem[LIST + 0]; s2 is now 1
    lw s3, 4(s1) # s3 = mem[LIST + 4]; s3 is now 2
    add s2, s2, s3 # s2 = 1 + 2; s2 is now 3
    lw s3, 8(s1) # s3 = mem[LIST + 8]; s3 is now 3
    add s2, s2, s3 # s2 = 3 + 3; s2 is now 6
    lw s3, 12(s1) # s3 = mem[LIST + 12]; s3 is now 4
    add s2, s2, s3 # s2 = 6 + 4; s2 is now 10

END: ebreak # Transfer control over to the debugger
# Without the ebreak, the processor keeps executing whatever is in memory
```

- `.data`, `.global`, `.text` are *assembler directives* – not instructions, but tell the assembler about what it should do
  - `.data` declares the global data section
    - We can use this to store data used by the whole program
    - In this example, it's an array
  - `.text` declares the section for the program itself
  - `.global` declares something to be visible outside the file (for a multi-file program)
  - `.word` declares the things that come next should take up an entire word of memory
- `la` is the load-address psuedo-instruction, which loads the address of some global data into a register

More Instructions

- Logic instructions
  - Bitwise operations that operate on 2 source registers
  - `and s0, s1, s2` puts the bitwise AND of `s1` and `s2` into `s0`
  - Similarly for `or s0, s1, s2` and `xor s0, s1, s2`
  - `not s0, s1` puts the bitwise NOT of `s1` into `s0`
    * Actually a pseudo-instruction, compiles to `xori s0, s1, -1`
  - Also have immediate versions `andi`, `ori`, `xori`