CSE 120
Day 5 Notes

Elijah Hantman

RISC-V User State

- 32 general purpose registers
- x0 is always set to zero
- In addition to the general purpose registers there is a PC register which stores the address of the current instruction
- Every register is 64 bit in the RV64I variant

```
add x3, x2, x1
```

add is the opcode, x3 is the destination register, x2 is the source register, and x1 is the second source register.

- Instructions are fetched from memory
- instructions are then fed directly into the cpu, with a specific binary format.

The Instruction cycle

1. First the PC increments and fetches the instruction
2. Processor then decodes the instruction
3. Logically execute the instruction
4. Access memory to write results if required
5. Write back to register if required

The cycle is usually abbreiviated $F \to D \to X \to M \to W$ The important thing to note is that memory writes always happen before register writes.

RV64I Names

During execution you are free to use general registers however you want.

However registers generally have conventions for how they are usually used.

The ABI defines how data structures or routines are accessed in machine code.

Some examples are

1. x1 is the ra register for a return address
2. x2 is the sp register for the stack pointer
3. x3 is gp, or the global pointer for global data
4. etc.

Registers are a finite resource that needs to be managed

The Goal is to keep as much data in the registers for as long as possible. Eventually data will be spilled from registers to memory. Arrays are generally too large to be kept in memory.

Assembly programmers need to balance between fully utilizing registers and taking advantage of the space memory provides.

Compiling

```
a = b + c;
```

Could be compiled to

```
add x1, x2, x3  # a = b + c
```

Most basic operations like, and, or, nor, xor, etc. can be compiled to a single instruction.

```
a = b + c + d - e;
```

More complicated code is broken apart into multiple instructions.

```
add x1, x2, x3 # a = b + c
add x1, x1, x4 # a = a + d
sub x1, x1, x5 # a = a - e
```

A smart programmer or compiler can pick fast instructions or optimize for size and pack multiple operations into single instructions depending on the goal and the architecture.

Numbers Representation

- Unsigned: 0 and absolute values only, no negatives.
- Signed: 0, positive and negative values
- In RISC-V all numbers are signed unless specified otherwise
- Signing convention is in 2's complement
- The word size is defined to be 32 bits.
- You can load a byte, half word, word, and double word directly.
- RISC-V is byte addressable, the smallest unit of data that can be loaded is a single byte.

RISC-V constants

- constant == immediate == literal == offset
- there are immediate versions of many instructions
- RISC-V only uses a 12 bit representation for immediates
  This means that the total range is limited

Memory Transfer

- Load always pulls from memory
- store always pushes data to memory
- all memory interactions in RISC-V happen through load and store instructions.
- floating-point loads and stores work with specialized floating point registers.

```
ld dst, offset(base)
```

When loading from memory, effective addresses are computed. An effective address is an aligned byte address.

The offset and base address are used for array operations, allowing for indexing into regions of memory.

For a given load operation, it must be aligned to that memory load size. lb must be byte aligned, lh must be half-word aligned, lw must be word aligned, ld must be double word aligned.

Misaligned data usually causes an exception on most RISC-V processors.

For caching, most systems will force alignement for speed and efficiency. This divides memory into cache-lines. If memory is misaligned it could be split among multiple cache lines.

C Data load example

```
a = b + *c;
```

C is a memory access so it always requires a load.

```
ld x5, 0(x3)      # x5 = *c
add x1, x2, x5    # a = b + x5
```

Memory Storing

```
sd src, offset(base)
```

Offset value is 12-bit signed constant.
Works identically to loading data but in reverse

C Data store example

```
*a = b + c;

add x1, x2, x5    # x1 = b + c
sd x1, 0(x3)      # *a = x1
```

RISC-V Binary Format

R type, non immediate

- first 7 bits encode additional opcode info
- next 10 bits encode the two source operands
- 3 bits are used for additional op code field
- next 5 bits encode the destination operand
- next 5 bits are for the destination operand
- last 7 bits are for the opcode

Immediate I type

- first 12 bits are instead used for the immediate

Immediate S/B type

- The first 7 bits, as well as the return operand are used for the immediate.
- For only S/B types the immediate is a single value but split between two bit fields.

Immediate U/J type

- The first 20 bits are used for the immediate

When decoding the first step is to parse the opcode. The opcode will specify how to split the rest of the bits.

The opcode can only specify 128 instructions directly, there are and additional 10 bits which can be used for the R type instructions.