Quiz 1: next week
Covers L1-L8
Oct 11, 7:30-9:30PM
Walker memorial
50-340
Using Combinational and Sequential Logic, we can build **special-purpose hardware**: a finite state machine that solves a particular problem (e.g., Modulo 4 Counter, GCD)

Soon we will be able to build a **general-purpose computer**: a machine that can solve any solvable problem, given enough time and memory.
The von Neumann Model

- Many ways to build a general-purpose computer
- Almost all modern computers are based on the von Neumann model (John von Neumann, 1945)
- Components:
  - Main memory holds programs and their data
  - Central processing unit accesses and processes memory values
  - Input/output devices to communicate with the outside world
Main Memory = Random-Access Memory

- Registers and a Register File can only be used to store a small number of data elements. To support all of our storage needs, we use main memory.

- Array of bits, organized in $W$ words of $N$ bits each
  - Typically, $W$ is a power of two: $W = 2^k$
  - Example: $W=8$ (k=3 address bits), $N=32$ bits per word

- Can read from and write to individual words

- Many possible implementations (later in the course)

<table>
<thead>
<tr>
<th>Address</th>
<th>11101000 10111010 01011010 10010101</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>10111010 00000000 11110101 00000000</td>
</tr>
<tr>
<td>001</td>
<td>10000000 00000000 11110101 00000000</td>
</tr>
<tr>
<td>010</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>00000000 00000000 11110101 11011000</td>
</tr>
</tbody>
</table>
Storage Conventions: Registers vs Memory

- Typically variables live in memory
- Registers hold temporary values or values that we need to use repeatedly
- ALU operations on performed on registers
- To operate with memory variables
  - Load them into a register
  - Compute on them
  - Store the results back to memory

```
inj x, y;
y = x - 37;
```

```
0x1000: | n |
0x1004: | r |
0x1008: | x |
0x100C: | y |
0x1010: |
```

- `R1 ← Mem[0x1008]`
- `R1 ← R1 - 37`
- `Mem[0x100C] ← R1`
Von Neumann Computer: Stored-Program Computer

- Express program as a sequence of **coded instructions**
- Memory holds both data and instructions
- CPU fetches, interprets, and executes successive instructions of the program

**Central Processing Unit**

**Main Memory**

- instruction
- instruction
- instruction
- data
- data
- data

**Example Instruction:**

```
0xba5eba11
```

**How does CPU distinguish between instructions and data?**

- **rs2 rs1 rd op**
- **rd ← op(rs1, rs2)**
Anatomy of a von Neumann Computer

- **Instructions** coded as binary data
- **Program Counter** or PC: Address of the instruction to be executed
- Logic to translate instructions into control signals for datapath
Instructions

- Instructions are the fundamental unit of work
- Each instruction specifies:
  - An operation or opcode to be performed
  - Source and destination operands

- A von Neumann machine executes instructions sequentially
  - CPU logically implements this loop:
    - By default, the next PC is current PC + size of current instruction unless the instruction says otherwise
Instruction Set Architecture (ISA)

- ISA: The contract between software and hardware
  - Functional definition of operations and storage locations
  - Precise description of how software can invoke and access them

- The ISA is a new layer of abstraction:
  - ISA specifies what hardware provides, not how it’s implemented
  - Enables fast innovation in hardware (no need to change software!)
    - 8086 (1978): 29 thousand transistors, 5 MHz, 0.33 MIPS
    - Pentium 4 (2003): 44 million transistors, 4 GHz, ~5000 MIPS
    - Skylake (2015): 1.75 billion transistors, 4 GHz, ~30k MIPS
    - All implement the x86 ISA

- Down side: Commercially successful ISAs last for decades
  - Today’s x86 CPUs carry baggage of design decisions from 70’s
RISC-V ISA

In this course we will use the RISC-V ISA

- A new, open, free ISA from Berkeley

- Several variants
  - RV32, RV64, RV128: Different data widths
  - ‘I’: Base Integer instructions
  - ‘M’: Multiply and Divide
  - ‘F’ and ‘D’: Single- and Double-precision floating point
  - And many other modular extensions

- We will design an RV32I processor, which is the base integer 32-bit variant
RISC-V ISA: Storage

CPU State

PC

General-Purpose Registers

x0  000000...0
x1
x2
...

x31

x0 hardwired to 0

Main Memory

Address

0x00
0x04
0x08
0x0C
0x10
0x12

Up to $2^{32}$ bytes (4GB) of memory, organized as $2^{30}$ 4-byte words

Each memory word is 32-bits wide, but we use byte memory addresses. Since each word contains 4 bytes, addresses of consecutive words differ by 4.
RISC-V ISA: Instructions

Three types of instructions:

- Computational: Perform operations on general registers
- Loads and stores: Move data between general registers and main memory
- Control: Change the program counter

All instructions have a fixed 32-bit length (4 bytes)

Why fixed instead of variable-length instructions?

*Simpler to decode & to compute next PC (but larger code)*
Computational Instructions
Computational Instructions (R-type)

- Register-to-register instructions (R-type)

<table>
<thead>
<tr>
<th>7</th>
<th>5</th>
<th>5</th>
<th>3</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>funct7</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
</tr>
</tbody>
</table>

- Performs \( R[rd] \leftarrow R[rs1] \text{ op } R[rs2] \)
- Operation \( \text{ op } \) specified by \( (\text{funct7}, \text{funct3}) \)
- Example: ADD instruction

\[
\begin{align*}
\text{op } &= \text{add} \\
\text{rs2} &= 4 \\
\text{rs1} &= 3 \\
\text{rd} &= 1
\end{align*}
\]


- We prefer a symbolic representation: \text{add } x1, x3, x4
- Similar instructions for other operations:

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Comparisons</th>
<th>Logical</th>
<th>Shifts</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD, SUB</td>
<td>SLT, SLTU</td>
<td>AND, OR, XOR</td>
<td>SLL, SRL, SRA</td>
</tr>
</tbody>
</table>

October 4, 2018
Many programs use small constants frequently (e.g., comparisons, loop indices, etc.)

Using registers to hold these constants is wasteful!

Solution: **Register-immediate instructions (I-type)**
- Performs $R[rd] \leftarrow R[rs1] \text{ op } \text{imm}$
- **Immediate** operand is encoded into instruction (12 bits (instr_imm))
- To use it: $\text{imm} = \text{signExtend}($instr_imm$[11:0])$
- Example: $\text{addi } x5, x2, -3$
Computational Instructions (I-type)

- Similar set of operations as R-type instructions:
  - Shift instructions use lower 5 bits of immediate:
    - \( R[rd] \leftarrow R[rs1] \ op \ instr\_imm[4:0] \)

Why is the \( imm \) 5 bits for shift instructions?

Data is 32 bits wide,
5 bits can specify shift between 0-31

Why is there no SUBI?

Equivalent to ADDI with a negative immediate
Example 1

- Execute \( a = ((b+3) \gg c) - 1; \)

  1. Break up complex expression into basic computations.

  2. Assume \( a, b, c, t0, \) and \( t1 \) are in registers. \( x1: a, x2: b, x3: c, x4: t0, x5: t1 \)

  \[
  \begin{align*}
  t0 &= b + 3; \\
  t1 &= t0 \gg c; \\
  a &= t1 - 1;
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{addi} & \quad x4, x2, 3 \\
  \text{srl} & \quad x5, x4, x3 \\
  \text{addi} & \quad x1, x5, -1;
  \end{align*}
  \]
Handling Large Constants

- Sometimes we need to use full 32-bit constants
- Solution: Add a format with a 20-bit immediate

\[
\text{imm} = \{\text{long_imm}[19:0], 12\text{'b}0\}\]

- Load upper immediate (LUI): \text{lui rd, long_imm}
- \(R[\text{rd}] \leftarrow \text{imm}\)

- **Example:** Write code to load constant 0xCAFE0123 into x3

\[
\begin{align*}
\text{lui x3, 0xCAFE0} & \quad // \ x3 = 0xCAFE00000 \\
\text{addi x3, x3, 0x123} & \quad // \ x3 = 0xCAFE0123
\end{align*}
\]
Load and Store Instructions
Load Instruction

- Loads move data from main memory into a register
- Load word: \( \text{lw} \) \( \text{rd} \), \( \text{inst\textunderscore imm}(\text{rs1}) \)
- Example: \( \text{lw} \) \( \text{x2, 4(x3)} \)
- Performs \( R[\text{rd}] \leftarrow \text{Mem}[R[\text{rs1}] + \text{imm}] \)

- Encode 12 bit immediate offset in instruction
  - \( \text{imm} = \text{signExtend}(\text{inst\textunderscore imm}[11:0]) \)

- RISC-V has a few other load instructions (load half-word, load byte, etc.) that we will not use in this course
Store Instruction

- Stores move data from registers into main memory
- Store word: \texttt{sw rs2, inst\_imm(rs1)}
- Example: \texttt{sw x2, 4(x3)}
- Performs \( \text{Mem}[\text{R}[\text{rs1}] + \text{imm}] \leftarrow \text{R}[\text{rs2}] \)

- Encode 12 bit immediate offset in instruction
  - \( \text{imm} = \text{signExtend(}\text{inst\_imm}[11:0]\text{)} \)

- RISC-V has a few other store instructions (store half-word, store byte) that we will not use in this course
Example 2

```
int x, y;
y = x - 37;
```

- Load x into x2 register
- Get value 0x1008 into x1
- Load Mem[x1] into x2
- Subtract 37 from x2
- Store x2 into y

```
lui x1, 1    // x1 = 0x1000
addi x1, x1, 8 // x1 = 0x1008
lw x2, 0(x1)  // x2 = Mem[0x1008]
addi x2, x2, -37 // x2 = x2 - 37
sw x2, 4(x1)  // Mem[0x100C] = x2
```

```
R1 ← Mem[0x1008]
R1 ← R1 - 37
Mem[0x100C] ← R1
```

```
0x1000:    n
0x1004:    r
0x1008:    x
0x100C:    y
0x1010:    
```

```
Control Instructions
Control Instructions: Branches

- **Conditional branches:** Example: `blt x1, x2, label`
  - First performs comparison to determine if branch is taken or not: `R[rs1] comp R[rs2]`
  - If comparison returns True, then branch is taken:
    - Performs `pc ← pc + imm`
  - Else:
    - Performs `pc ← pc + 4`
  - Offset to label is encoded in 12 bit immediate in instruction
    - `imm = signExtend(inst_imm[11:0],1'b0)`
      - Enforces half word alignment
      - Can change PC only within a ±4KB range
  - Supported comparison operators

<table>
<thead>
<tr>
<th>Instruction</th>
<th>BEQ</th>
<th>BNE</th>
<th>BLT</th>
<th>BGE</th>
<th>BLTU</th>
<th>BGEU</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>comp</code></td>
<td>==</td>
<td>!=</td>
<td>&lt;</td>
<td>≥</td>
<td>&lt;</td>
<td>≥</td>
</tr>
</tbody>
</table>

October 4, 2018
Unconditional Control Instructions: Jumps

- **JAL**: Unconditional jump and link
  - Example: `jal rd, label`
  - Performs `R[rd] ← pc + 4; pc ← pc + imm`
  - Encodes 20 bit immediate in instruction
    - `imm = signExtend(inst_imm[19:0], 1'b0)`
    - Can jump within a ±1MB range of current PC

- **JALR**: Unconditional jump via register and link
  - Example: `jalr rd, 0(rs1)`
  - Performs `R[rd] ← pc + 4; pc ← (R[rs1] + imm) & ~0x01`
    - `~0x01` forces LSB to 0 (half word alignment)
  - Encodes 12 bit immediate in instruction
    - `imm = signExtend(inst_imm[11:0])`
  - Can jump to any 32 bit address – supports long jumps
Assembly Code vs. Binary

- It's too tedious to write programs directly in binary

- To simplify writing programs, assemblers provide:
  - Mnemonics for instructions
    - add x1, x2, x3
  - Symbols for program locations and data
    - bneq x1, x2, loop_begin
    - lw x1, flag
  - Pseudoinstructions
    - mv x1, x2 // short for addi x1, x2, 0
    - j label // short for jal x0, label
    - beqz x1, dest // short for beq x1, x0, dest

- Assemblers translate programs into machine code for the processor to execute
Example 3

GCD Computation:

```c
int a, b;
while (a != 0) {
    if (a >= b) {
        a = a - b;
    } else {
        // Swap a and b
        int t = a;
        a = b;
        b = t;
    }
}
```

Assume x1: a, x2: b, x3: t

```assembly
while:
    beqz x1, done  // if(a==0)
    // goto done
    blt x1, x2, b_bigger  // if(a < b)
    // goto b_bigger
    sub x1, x1, x2  // a ← a - b
    j while  // goto while
b_bigger:
    mv x3, x1  // t ← a
    mv x1, x2  // a ← b
    mv x2, x3  // b ← t
    j while  // goto while
done:
    // now x2: b
    // holds the gcd
```
Take-home problem: Write RISC-V assembly code

- You are told that an array `A` of 10 32-bit values are stored in memory beginning at address 0x100.
  1. `A[0]` is stored at address 0x100. At what address is `A[3]` stored?
Thank you!

Next lecture: Procedures and stacks