Reading Quiz
Today

• C Programming Wrap Up
  – Arrays, Strings
  – Structs
  – Functions

• Hardware basics
  – Machine memory models
  – Digital signals
  – Logic gates

• Manipulating/Representing values in hardware
  – Adders
  – Storage & memory (latches)
Hardware Models (1940’s)

• Harvard Architecture:

![Program Memory](image)
Hardware Models (1940’s)

- Harvard Architecture:
  - Program Memory
  - CPU (Control and Arithmetic)
  - Data Memory
  - Input/Output
Von Neumann

John von Neumann
“The father of modern machines”

Stored Program Concept

EDVAC 1945
Stored Program Concept

• **Fixed Machines**
  – Early machines had “fixed” programs
  – Can’t be used for other purpose
  – Change required re-design & re-wiring!

• **General Purpose Machine**
  – Need more versatility
  – Programs stored in a “memory”
  – Machine can be re-programmed!

Instructions encode functionality – programs!
Von Neumann Architecture

- Von Neumann Architecture:

  - CPU (Control and Arithmetic)
  - Program and Data Memory
  - Input/Output
Von Neumann Architecture Model

Based on Alan Turing’s Universal Turing Machine

Computer is a generic computing machine:

• **Stored program model**: computer stores program rather than encoding it (feed in data and instructions)

**No distinction between data and instructions memory**
Von Neumann Architecture Model

5 parts connected by buses (wires):
Control, Memory, Processing, Input, Output
Control: The CPU

Processing Unit: executes instructions selected by control unit

**ALU** (arithmetic logic unit): simple functional units: ADD, SUB...

Registers: temporary storage directly accessible by instructions
Memory

Memory: data and instructions are stored.

Memory is addressable: addr 0, 1, 2, ....

“Register”
Small, very vast storage space.
Fixed size (e.g., 32 bits).

Stores what is currently being worked on.

The CPU

1. Processing Unit
   ALU registers

2. Control Unit
   PC, IR

3. Memory Unit

4. Input Units

5. Output Units

addr bus
cntrl bus
data bus
Input/Output: keyboard (can trigger actions), terminal, disk
Digital Computers

• All input is discrete (driven by periodic clock)

• All signals are binary (0: no voltage, 1: voltage) data, instructions, control signals, arithmetic, clock

• To run program, need different types of circuits

Circuits to execute program instructions that act on program data

CPU
ALU, Cntrl, Storage

RAM
Cntrl & Storage

Circuits to store program data and instructions and support reading and writing addressable storage locations

bus
Goal: Build a CPU (model)

Three main classifications of HW circuits:

1. **ALU**: implement arithmetic & logic functionality
   
   (ex) adder to add two values together

2. **Storage**: to store binary values
   
   (ex) Register File: set of CPU registers, Also: main memory (RAM)

3. **Control**: support/coordinate instruction execution
   
   (ex) fetch the next instruction to execute
Abstraction

User / Programmer
Wants low complexity

Applications
Specific functionality

Software library
Reusable functionality

Operating system
Manage resources

Complex devices
Compute & I/O
Abstraction

Complex devices
Compute & I/O

Hardware Circuits
Logic Gates
Transistors

Here be dragons.
(Electrical Engineering)

... 
(Physics)
Logic Gates

Input: Boolean value(s) (high and low voltages for 1 and 0)
Output: - Boolean value result of boolean function
  - Always present, but may change when input changes

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A &amp; B</th>
<th>A</th>
<th>B</th>
<th>~A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
More Logic Gates

NAND

\[ \text{out} = \neg(a \& b) \]

Note the circle on the output. This means “not.” (flip bits)

NOR

\[ \text{out} = \neg(a \mid b) \]

A  B  |  A  NAND  B  |  A  NOR  B
---|---|---
0  0  | 1  | 1  
0  1  | 1  | 0  
1  0  | 1  | 0  
1  1  | 0  | 0  

Slide 27
Combinational Logic Circuits

• Build up higher level processor functionality from basic gates

Outputs are Boolean functions of inputs
Outputs continuously respond to changes to inputs
What does this circuit output?

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Out_A</th>
<th>Out_B</th>
<th>Out_C</th>
<th>Out_D</th>
<th>Out_E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Clicker Choices
What does this circuit output?

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Out_A</th>
<th>Out_B</th>
<th>Out_C</th>
<th>Out_D</th>
<th>Out_E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
What does this circuit output?

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Out_A</th>
<th>Out_B</th>
<th>Out_C</th>
<th>Out_D</th>
<th>Out_E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
What can we do with these?

• Build-up XOR from basic gates (AND, OR, NOT)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A ^ B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Q: When is $A^{\wedge}B == 1$?
Which of these is an XOR circuit?

And

Or

Not

Draw an XOR circuit using AND, OR, and NOT gates.

I’ll show you the clicker options after you’ve had some time.
Build a Circuit from Basic Gates

• Build XOR using (AND, OR, NOT)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A ^ B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**How:**

(1) Construct Logic Table
(2) When is $$A \oplus B = 1$$?

(a) express condition of each row of 1 result, in terms of input values $$A$$, $$B$$ combined with $$\&$$, $$\sim$$

(b) combine each row expression with $$|$$

$$A \oplus B = (\sim A \& B) | (A \& \sim B)$$

(3) : Translate expression to circuit
Build XOR circuit from basic gates (AND, OR, NOT)

And  Or  Not

Use $A \oplus B == 1 : (\neg A \& B) \mid (A \& \neg B)$

Test out circuit

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>$A \oplus B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Which of these is an XOR circuit?

A: 

B: 

C: 

D: 

E: None of these are XOR.
Which of these is an XOR circuit?

A: 

B: 

C: 

D: 

E: None of these are XOR.
XOR Circuit

\[ A \oplus B \equiv (\neg A \land B) \lor (A \land \neg B) \]

A: 0  B: 0  A^B:  
A: 1  B: 0  A^B:  
A: 0  B: 1  A^B:  
A: 1  B: 1  A^B:  

out = A^B
**XOR Circuit: Abstraction!**

\[
A \oplus B \equiv (\sim A \land B) \lor (A \land \sim B)
\]

---

Treat XOR Circuit as a building block for other circuits.
Three main classifications of HW circuits:

1. **ALU**: implement arithmetic & logic functionality
   - (ex) adder to add two values together
2. **Storage**: to store binary values
   - (ex) Register File: set of CPU registers
3. **Control**: support/coordinate instruction execution
   - (ex) fetch the next instruction to execute
Three main classifications of HW circuits:

1. ALU: implement arithmetic & logic functionality
   (ex) adder to add two values together

Start with ALU components (e.g., adder)
Combine into ALU!

<table>
<thead>
<tr>
<th>HW Circuits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Gates</td>
</tr>
<tr>
<td>Transistor</td>
</tr>
</tbody>
</table>
Arithmetic Circuits

• 1 bit adder: $A + B$

• Two outputs:
  1. Obvious one: the sum
  2. Other one: ??

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Sum (A + B)</th>
<th>$C_{out}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Which of these circuits is a one-bit adder?

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Sum (A + B)</th>
<th>C_{out}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

A:

B:

C:

D:
Which of these circuits is a one-bit adder?

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Sum (A + B)</th>
<th>C_{out}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

A:  
B:  
C:  
D:  

Slide 44
More than one bit?

- When adding, sometimes have *carry in* too

```
1111
0011010
+ 0001111
```

```
0101001
```
One-bit (full) adder

Need to include:

Carry-in & Carry-out

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Cin</th>
<th>Sum</th>
<th>Cout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

• When is Sum 1?

• When is $C_{out}$ 1?
One-bit (full) adder

Need to include:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Cin</th>
<th>Sum</th>
<th>Cout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- When is Sum 1?

\[ \sim C_{in} \land (A \land B) \lor C_{in} \land \sim(A \land B) = (C_{in} \land (A \lor B)) \]

- When is \( C_{out} \) 1?
One-bit (full) adder

Need to include:
- Carry-in & Carry-out

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Cin</th>
<th>Sum</th>
<th>Cout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- When is Sum 1?

\[ \sim C_{\text{in}} \& (A^\wedge B) \mid C_{\text{in}} \& \sim (A^\wedge B) = (C_{\text{in}} \wedge (A^\wedge B)) \]

- When is $C_{\text{out}}$ 1?

\[ (A \& B) \mid ((A^\wedge B) \& C_{\text{in}}) \]
One-bit (full) adder

Need to include:
Carry-in & Carry-out

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C_in</th>
<th>Sum</th>
<th>C_out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Multi-bit Adder (Ripple-carry Adder)

- $A_0$, $B_0$ -> 1-bit adder -> Sum$_0$, Cout
- $A_1$, $B_1$ -> 1-bit adder -> Sum$_1$, Cout
- $A_2$, $B_2$ -> 1-bit adder -> Sum$_2$, Cout
- ... (continued with $A_{N-1}$, $B_{N-1}$)
- $A_3$, $B_3$ -> 1-bit adder -> Sum$_3$, Cout
- $A_{N-1}$, $B_{N-1}$ -> 1-bit adder -> Sum$_{N-1}$, Cout
Three-bit Adder (Ripple-carry Adder)

\[ \begin{align*}
010 & \quad + \quad 011 \\
\Rightarrow & \quad 101
\end{align*} \]
Arithmetic Logic Unit (ALU)

• One component that knows how to manipulate bits in multiple ways
  – Addition
  – Subtraction
  – Multiplication / Division
  – Bitwise AND, OR, NOT, etc.

• Built by combining components
  – Take advantage of sharing HW when possible (e.g., subtraction using adder)
Simple 3-bit ALU: Add and bitwise OR

At any given time, we only want the output from ONE of these!
Simple 3-bit ALU: Add and bitwise OR

3-bit inputs
A and B:
A_0  A_1  A_2
B_0  B_1  B_2

Extra input: control signal to select Sum vs. OR

Circuit that takes in Sum_{0-2} / Or_{0-2} and only outputs one of them, based on control signal.
Which of these circuits lets us select between two inputs?

A:
- Input 1
- Control Signal
- Input 2

B:
- Input 1
- Control Signal
- Input 2

C:
- Input 1
- Control Signal
- Input 2
Which of these circuits lets us select between two inputs?

A: 
Input 1
Control Signal
Input 2

B: 
Input 1
Control Signal
Input 2

C: 
Input 1
Control Signal
Input 2
Multiplexor: Chooses an input value

**Inputs:** $2^N$ data inputs, N signal bits

**Output:** is one of the $2^N$ input values

- Control signal $c$, chooses the input for output
  - When $c$ is 1: choose $a$, when $c$ is 0: choose $b$

\[
\text{out} = (c \& a) \lor (\neg c \& b)
\]
N-Way Multiplexor

Choose one of N inputs, need \( \log_2 N \) select bits

<table>
<thead>
<tr>
<th>( c_1 )</th>
<th>( c_2 )</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>D0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>D1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>D2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>D3</td>
</tr>
</tbody>
</table>
Example, 1-bit 4-way MUX

When select input is 2 (0b10): C chosen as output

<table>
<thead>
<tr>
<th>S</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
</tr>
</tbody>
</table>
Simple 3-bit ALU: Add and bitwise OR

3-bit inputs
A and B:

A₀
A₁
A₂

B₀
B₁
B₂

Sum₀
Sum₁
Sum₂

Extra input: control signal to select Sum vs. OR

Circuit that takes in Sum₀-2 / Or₀-2 and only outputs one of them, based on control signal.
Simple 3-bit ALU: Add and bitwise OR

3-bit inputs
A and B:

A₀ A₁ A₂

B₀ B₁ B₂

Extra input: control signal to select Sum vs. OR

3-bit adder

Sum₀ Sum₁ Sum₂

Or₀ Or₁ Or₂

Multiplexer!
ALU: Arithmetic Logic Unit

- Arithmetic and logic circuits: ADD, SUB, NOT, ...
- Control circuits: use op bits to select output
- Circuits around ALU:
  - Select input values X and Y from instruction or register
  - Select op bits from instruction to feed into ALU
  - Feed output somewhere

CPU Instruction:

```
ADD 2 3
```

X \( \circ \) Y

Output flags: set as a side effect of \( \circ \)
(e.g., overflow detected)
Three main classifications of HW circuits:

1. **ALU**: implement arithmetic & logic functionality
   - (ex) adder to add two values together
2. **Storage**: to store binary values
   - (ex) Register File: set of CPU registers
3. **Control**: support/coordinate instruction execution
   - (ex) fetch the next instruction to execute

Circuits are built from Logic Gates which are built from transistors
Three main classifications of HW circuits:

2. Storage: to store binary values
   (ex) Register File: set of CPU registers

Give the CPU a “scratch space” to perform calculations and keep track of the state its in.

<table>
<thead>
<tr>
<th>HW Circuits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Gates</td>
</tr>
<tr>
<td>Transistor</td>
</tr>
</tbody>
</table>
CPU so far…

- We can perform arithmetic!

- Storage questions:
  - Where do the ALU input values come from?
  - Where do we store the result?
  - What does this “register” thing mean?
Memory Circuit Goals: Starting Small

• Store a 0 or 1

• Retrieve the 0 or 1 value on demand (read)

• Set the 0 or 1 value on demand (write)
R-S (Reset-Set) Latch: Stores Value $Q$

When $R$ and $S$ are both 1: Maintain a value

$R$ and $S$ are never both simultaneously 0

• To write a new value:
  • Set $S$ to 0 momentarily ($R$ stays at 1): to write a 1
  • Set $R$ to 0 momentarily ($S$ stays at 1): to write a 0
**R-S (Reset-Set) Latch: Stores Value Q**

When R and S are both 1: Maintain a value
R and S are never both simultaneously 0

To write a new value:
- Set S to 0 momentarily (R stays at 1): to write a 1
- Set R to 0 momentarily (S stays at 1): to write a 0

\[ Q = 1 \text{ (value stored)} \]
\[ \sim Q = 0 \]
R-S (Reset-Set) Latch: Stores Value Q

To write 0 to an RS latch momentarily set R to 0

R and S are never both simultaneously 0
Gated D Latch

Controls RS latch writing, ensures S & R never both 0

D: into top NAND, \(\sim D\) into bottom NAND
WE: write-enabled, when set, latch is set to value of D

Latches used in registers (up next) and SRAM (caches, later)
Fast, not very dense, expensive

DRAM: capacitor-based:
An N-bit Register

- Fixed-size storage (8-bit, 32-bit, etc.)
- Gated D latch stores one bit
  - Connect N of them to the same write-enable wire!

Abstraction!
“Register file”

• A set of registers for the CPU to store temporary values.

• This is (finally) something you will interact with!

• Instructions of form:
  – “add R1 + R2, store result in R3”
Memory Circuit Summary

• Lots of abstraction going on here!
  – Gates hide the details of transistors.
  – Build R-S Latches out of gates to store one bit.
  – Combining multiple latches gives us N-bit register.

• Register file’s simple interface:
  – Read $R_x$’s value, use for calculation
  – Write $R_y$’s value to store result
CPU so far…

We know how to store data (in register file).
We know how to perform arithmetic on it, by feeding it to ALU.
Remaining questions:
Which register(s) do we use as input to ALU?
Which operation should the ALU perform?
To which register should we store the result?

All this info comes from the program: a series of instructions.
Recall: Von Neumann Model

We’re building this.

Our program (instructions) live here. We’ll assume for now that we can access it like an array.

Mem Addresses (buckets)

0:
1:
2:
3:
4:
...
N-1:
Digital Circuits - Building a CPU

Three main classifications of HW circuits:

1. **ALU**: implement arithmetic & logic functionality
   (ex) adder to add two values together
2. **Storage**: to store binary values
   (ex) Register File: set of CPU registers
3. **Control**: support/coordinate instruction execution
   (ex) fetch the next instruction to execute

Circuits are built from Logic Gates which are built from transistors
Three main classifications of HW circuits:

3. Control: support/coordinate instruction execution
   (ex) fetch the next instruction to execute

Keep track of where we are in the program.
Execute instruction, move to next.
Control Unit

Which register(s) do we use as input to ALU?
Which operation should the ALU perform?
To which register should we store the result?

All this info comes from our program: a series of instructions.
CPU Game Plan

- **Fetch** instruction from memory

- **Decode** what the instruction is telling us to do
  - Tell the ALU what it should be doing
  - Find the correct operands

- **Execute** the instruction (arithmetic, etc.)

- **Store** the result
Let’s add **two more special registers** (not in register file) to keep track of program.

Program Counter (PC): Memory address of next instr

Instruction Register (IR): Instruction contents (bits)

![Diagram of register file and ALU](image)
Fetching instructions.

Load IR with the contents of memory at the address stored in the PC.

Program Counter (PC): Address 0

Instruction Register (IR): Instruction at Address 0

Register File
Decoding instructions.

Interpret the instruction bits: What operation? Which arguments?

Program Counter (PC): | Address 0
---|---
Instruction Register (IR): | OP Code | Reg A | Reg B | Result

32-bit Register #0
32-bit Register #1
32-bit Register #2
32-bit Register #3

Register File

Slide 82
Decoding instructions.

Interpret the instruction bits: What operation? Which arguments?

Program Counter (PC): Address 0

Instruction Register (IR): OP Code | Reg A | Reg B | Result

Data in
WE
Data in
WE
Data in
WE
Data in
WE

32-bit Register #0
32-bit Register #1
32-bit Register #2
32-bit Register #3

... Register File

OP Code tells ALU which operation to perform.
Decoding instructions.

Interpret the instruction bits: What operation? Which arguments?

**Program Counter (PC):** Address 0

**Instruction Register (IR):**

- **OP Code**
- **Reg A**
- **Reg B**
- **Result**

Register ID #'s specify input arguments.

Data in
WE
Data in
WE
Data in
WE
Data in
WE
Data in
WE

32-bit Register #0
32-bit Register #1
32-bit Register #2
32-bit Register #3

Register File
Executing instructions.

Interpret the instruction bits: What operation? Which arguments?

Program Counter (PC): Address 0

Instruction Register (IR): OP Code | Reg A | Reg B | Result

Let the ALU do its thing. (e.g., Add)

Data in
WE
Data in
WE
Data in
WE
Data in
WE

32-bit Register #0

32-bit Register #1

32-bit Register #2

32-bit Register #3

... 

Register File

(Memory)
Storing results.

We’ve just computed something. Where do we put it?

**Program Counter (PC):**

```
| Address 0 |
```

**Instruction Register (IR):**

```
| OP Code | Reg A | Reg B | Result |
```

Data in

32-bit Register #0

32-bit Register #1

32-bit Register #2

32-bit Register #3


... 

Result location specifies where to store ALU output.

(Memory)
Why do we need a program counter? Can’t we just start at 0 and count up one at a time from there?

A. We don’t, it’s there for convenience.
B. Some instructions might skip the PC forward by more than one.
C. Some instructions might adjust the PC backwards.
D. We need the PC for some other reason(s).
Why do we need a program counter? Can’t we just start at 0 and count up one at a time from there?

A. We don’t, it’s there for convenience.
B. Some instructions might skip the PC forward by more than one.
C. Some instructions might adjust the PC backwards.
D. We need the PC for some other reason(s).
Storing results.

Interpret the instruction bits: What operation? Which arguments?

Program Counter (PC):

Address 0

Instruction Register (IR):

OP Code | Reg A | Reg B | Result

Data in
WE
Data in
WE
Data in
WE
Data in
WE

32-bit Register #0
32-bit Register #1
32-bit Register #2
32-bit Register #3

... 

Register File

Result might be:
- Memory
- Register
- PC

(Memory)

0: 1: 2: 3: 4: ...

N-1:
Clocking

• Need to periodically transition from one instruction to the next.

• It takes time to fetch from memory, for signal to propagate through wires, etc.
  – Too fast: don’t fully compute result
  – Too slow: waste time
Clock Driven System

- Everything in is driven by a discrete clock
  - clock: an oscillator circuit, generates hi low pulse
  - clock cycle: one hi-low pair

- Clock determines how fast system runs
  - Processor can only do one thing per clock cycle
    - Usually just one part of executing an instruction
  - 1GHz processor:
    - 1 billion cycles/second $\rightarrow$ 1 cycle every nanosecond
Cycle Time: Laundry Analogy

- Discrete stages: fetch, decode, execute, store
- Analogy (laundry): washer, dryer, folding, dresser

4 Hours (each stage takes 1 hour)

You have big problems if you have millions of loads of laundry to do....
Laundry

4-hour cycle time.

Finishes a laundry load every cycle.

(6 laundry loads per day)
Pipelining (Laundry)

1 Hour

1<sup>st</sup> hour: W

2<sup>nd</sup> hour: W Dy

3<sup>rd</sup> hour: W Dy F

4<sup>th</sup> hour: W Dy F Dr

5<sup>th</sup> hour: W Dy F Dr

Steady state: One load finishes every hour! (Not every four hours like before.)
Pipelining (CPU)

1 Nanosecond

1\textsuperscript{st} nanosecond: F

2\textsuperscript{nd} nanosecond: F D

3\textsuperscript{rd} nanosecond: F D E

4\textsuperscript{th} nanosecond: F D E S

5\textsuperscript{th} nanosecond: F D E S

CPU Stages: fetch, decode, execute, store results

Steady state: One instruction finishes every nanosecond! (Clock rate can be faster.)
Pipelining

(For more details about this and the other things we talked about here, take architecture.)
Up next

• Talking to the CPU: Assembly language