CS 31: Intro to Systems C Programming
L05-L06: Digital Logic

Vasanta Chaganti & Kevin Webb
Swarthmore College
September 19 - 21, 2023
Announcements

• Clickers will count for credit from this week
Reading Quiz

• Note the red border!

• 1 minute per question

• No talking, no laptops, phones during the quiz

Check your frequency:
• Iclicker2: frequency AA
• Iclicker+: green light next to selection

For new devices this should be okay, For used you may need to reset frequency

Reset:
1. hold down power button until blue light flashes (2secs)
2. Press the frequency code: AA vote status light will indicate success
Agenda

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

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

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

• Harvard Architecture:
  
  ![Harvard Architecture Diagram]

• Von Neumann Architecture:
  
  ![Von Neumann Architecture Diagram]
Von Neumann

John von Neumann
“The father of modern machines”

Stored Program Concept

EDVAC 1945
Von Neumann Architecture Model

• Computer is a generic computing machine:
  • Based on Alan Turing’s Universal Turing Machine
  • Stored program model: computer stores program rather than encoding it (feed in data and instructions)
  • No distinction between data and instructions memory

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

1. Processing Unit: Execute instructions to produce a result
   – ALU (arithmetic logic unit): set of circuits for arithmetic (ADD, SUB, etc.)
   – Registers: temporary storage for instructions (scratch space)

2. Control Unit: Keep track of which instruction to execute next and what that instruction says to do.
Memory

3. Data and instruction storage in “main memory” (RAM)
   – Each byte in memory has a unique address
Memory

4. Input: Data coming into the CPU from outside sources
   – keyboard, mouse, network, hard drive

5. Output: Data leaving the CPU to the outside world
   – video display, audio, network, hard drive, printer
Goal: Build a CPU (model)

Three main classifications of hardware circuits:

1. ALU: implement arithmetic & logic functionality
   – Example: adder circuit to add two values together

2. Storage: to store binary values
   – Example: set of CPU registers (“register file”) to store temporary values

3. Control: support/coordinate instruction execution
   – Example: circuitry to fetch the next instruction from memory and decode it
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>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
More Logic Gates

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A NAND B</th>
<th>A NOR B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</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>0</td>
</tr>
</tbody>
</table>

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

Clicker Choices

<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 does this circuit output?

Clicker Choices

<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>
Building more interesting circuits…

• 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^B ==1?
Which of these is an XOR circuit?

General strategy:
1. Determine truth table (given inputs)
   - express these in terms of input values A, B combined with AND, NOT
   - then, combine each row expression with OR
2. Find rows with output = 1
3. Translate expression to a circuit

<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>
Which of these is an XOR circuit?

![Diagram of AND, OR, and NOT gates]

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

I’ll show you the clicker options after you’ve had some time.

<table>
<thead>
<tr>
<th></th>
<th></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>
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: Abstraction

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

<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>( \sim A \land B )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>( A \land \sim B )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>( A \land \sim B )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>( A \land B )</td>
</tr>
</tbody>
</table>

\[ \text{out} = A \oplus B \]
Recall Goal: Build a CPU (model)

Three main classifications of hardware circuits:

1. ALU: implement arithmetic & logic functionality
   – Example: adder circuit to add two values together

2. Storage: to store binary values
   – Example: set of CPU registers (“register file”) to store temporary values

3. Control: support/coordinate instruction execution
   – Example: circuitry to fetch the next instruction from memory and decode it
Recall Goal: Build a CPU (model)

Three main classifications of hardware circuits:

1. **ALU**: implement arithmetic & logic functionality
   - Example: adder circuit to add two values together

Start with ALU components (e.g., adder circuit, bitwise operator circuits)

Combine component circuits into ALU!
Arithmetic Circuits

- 1 bit adder: \( A + B \)
- Two outputs:
  1. Obvious one: the sum
  2. Other one: ??

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Sum (A + B)</th>
<th>C_{\text{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>
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>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>
Which of these circuits is a one-bit adder?

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Sum (A + B)</th>
<th>C\text{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>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Sum (A + B)</th>
<th>C\text{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>
Which of these circuits is a one-bit adder?

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Sum (A + B)</th>
<th>C\textsubscript{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:
More than one bit?

• When adding, sometimes have *carry in* too

\[
\begin{align*}
0011010 \\
+ & 0001111
\end{align*}
\]
More than one bit?

• When adding, sometimes have carry in too

\[
\begin{align*}
1111 \\
0011010 \\
+ \quad 0001111
\end{align*}
\]
Write Boolean expressions for $\text{Sum} = 1$ and $\text{C}_{\text{out}} = 1$

When is $\text{Sum} 1$?

When is $\text{C}_{\text{out}} 1$?

<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>
Write Boolean expressions for \( \text{Sum} = 1 \)

When is \( \text{Sum} = 1 \)?

\[ \sim C_{in} \; \& \; (A \land B) \]
Write Boolean expressions for \( \text{Sum} = 1 \)

When is \( \text{Sum} = 1 \)?

\[
\begin{align*}
\sim C_{\text{in}} \ & \ (A \lor B) \\
C_{\text{in}} \ & \ \sim (A \lor B) \\
= \ & (C_{\text{in}} \lor (A \lor B))
\end{align*}
\]
Write Boolean expressions for $\text{Sum} = 1$ and $\text{C}_{\text{out}} = 1$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>$C_{\text{in}}$</th>
<th>Sum</th>
<th>$C_{\text{out}}$</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>

When is $\text{Sum}$ 1?

$\sim C_{\text{in}} \ &(A^B) | C_{\text{in}} \ & \sim(A^B) \ == \ (C_{\text{in}} \ ^ (A^B))$

When is $\text{C}_{\text{out}}$ 1?
Write Boolean expressions for \( \text{Sum} = 1 \) and \( C_{\text{out}} = 1 \)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C_{\text{in}}</th>
<th>Sum</th>
<th>C_{\text{out}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<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>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<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>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>
<td>1</td>
</tr>
</tbody>
</table>

When is \( \text{Sum} = 1? \)
\[ \neg C_{\text{in}} \land (A \lor B) \lor C_{\text{in}} \land \neg (A \lor B) = (C_{\text{in}} \land (A \lor B)) \]

When is \( C_{\text{out}} = 1? \)
\[ (A \land B) \lor ((A \lor B) \land C_{\text{in}}) \]
Write Boolean expressions for $\text{Sum} = 1$ and $\text{C}_{\text{out}} = 1$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>$\text{C}_{\text{in}}$</th>
<th>$\text{Sum}$</th>
<th>$\text{C}_{\text{out}}$</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>

When is $\text{Sum} = 1$?

$\neg \text{C}_{\text{in}} \& (\text{A} \& \text{B}) \mid \text{C}_{\text{in}} \& \neg (\text{A} \& \text{B})$ == $(\text{C}_{\text{in}} \land (\text{A} \& \text{B}))$

When is $\text{C}_{\text{out}} = 1$?

$(\text{A} \& \text{B}) \mid (\text{A} \& \text{B}) \land \text{C}_{\text{in}}$
One-bit (full) adder

- Need to include: carry-in and 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)
Three-bit Adder (Ripple-carry Adder)

\[ 010 + 011 = 100 \]
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)
3-bit inputs
A and B:

A0, A1, A2
B0, B1, B2

At any given time, we only want the output from ONE of these!

Out0, Out1, Out2
Simple 3-bit ALU: Add and bitwise OR

3-bit inputs A and B:

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

Sum

Out

Extra input: control signal to select Sum vs. OR

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

A:  

B:  

C:  

A:  

B:  

C:
Which of these circuits lets us select between two inputs?

A:

B:

C:
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 $b$

$$\text{out} = (c \land a) \lor (\neg c \land b)$$
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
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

3-bit adder

Or_0, Or_1, Or_2

Extra input: control signal to select Sum vs. OR

Sum_0, Sum_1, Sum_2

Multiplexor!
• 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

Output flags: set as a side effect of \( op \) (e.g., overflow detected)
Goal: Build a CPU (model)

Three main classifications of hardware circuits:

1. **ALU**: implement arithmetic & logic functionality
   - Example: adder circuit to add two values together

2. **Storage**: to store binary values
   - Example: set of CPU registers (“register file”) to store temporary values

3. **Control**: support/coordinate instruction execution
   - Example: circuitry to fetch the next instruction from memory and decode it
Goal: Build a CPU (model)

Three main classifications of hardware circuits:

2. Storage: to store binary values
   – Example: set of CPU registers (“register file”) to store temporary values

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

• We can perform arithmetic!

• Storage questions:
  – Where to 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 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 Latch: Stores Value Q

Assume that the RS Latch currently stores 1.

To write 0 into the latch, set R’s value to 0.

A. Set R to 0 to store 0
Gated D Latch

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

- **D**: into top NAND, \(~D\) 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, 64-bit, etc.)

• Gated D latch lets us store one bit
  – Connect N of them to the same write-enable wire!
“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
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
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.

CPU (Control and Arithmetic) ➔ Program and Data Memory ➔ Input/Output

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](attachment:image.png)
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

Diagram:
- Data in
- WE
- Data in
- WE
- Data in
- WE

- MUX
- 64-bit Register #0
- 64-bit Register #1
- 64-bit Register #2
- 64-bit Register #3

- Register File
- ALU
- (Memory)
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

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

MUX

ALU

Data in

WE

Data in

WE

Data in

WE

Data in

WE

MUX

Register File
Decoding instructions.

Interpret the instruction bits: What operation? Which arguments?

Program Counter (PC): Address 0

Instruction Register (IR):

<table>
<thead>
<tr>
<th>OP Code</th>
<th>Reg A</th>
<th>Reg B</th>
<th>Result</th>
</tr>
</thead>
</table>

OP Code tells ALU which operation to perform.

ALU

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

MUX

Data in
Data in
Data in
Data in

WE
WE
WE
WE

Register File

(Memory)

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

N-1:
Decoding instructions.

Interpret the instruction bits: What operation? Which arguments?

Program Counter (PC): Address 0

Instruction Register (IR):

<table>
<thead>
<tr>
<th>OP Code</th>
<th>Reg A</th>
<th>Reg B</th>
<th>Result</th>
</tr>
</thead>
</table>

Register ID #'s specify input arguments.

Data in

WE

Data in

WE

Data in

WE

Data in

WE

64-bit Register #0

64-bit Register #1

64-bit Register #2

64-bit Register #3

MUX

MUX

ALU

Register File

(Memory)

0:

1:

2:

3:

4:

...:

N-1:
Executing instructions.

Interpret the instruction bits: What operation? Which arguments?

Program Counter (PC): Address 0

Instruction Register (IR):

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

Data in
64-bit Register #0
Data in
64-bit Register #1
Data in
64-bit Register #2
Data in
64-bit Register #3

MUX
MUX

Register File

(e.g., Add)
Storing results.

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

Program Counter (PC): Address 0

Instruction Register (IR):

<table>
<thead>
<tr>
<th>OP Code</th>
<th>Reg A</th>
<th>Reg B</th>
<th>Result</th>
</tr>
</thead>
</table>

Result location specifies where to store ALU output.
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

Result might be:
- Memory
- Register
- PC

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

Register File

MUX
MUX

ALU

(result)

Data in
64-bit Register #0

Data in
64-bit Register #1

Data in
64-bit Register #2

Data in
64-bit Register #3

Data in

(Data)

WE

WE

WE

WE

0:
1:
2:
3:
4:
...:
N-1:

(Memory)
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 → 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....
4-hour cycle time.

Finishes a laundry load every cycle.

(6 laundry loads per day)
Pipelining (Laundry)

1 Hour

1st hour:

2nd hour:

3rd hour:

4th hour:

5th hour:

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

CPU Stages: fetch, decode, execute, store results

1 Nanosecond

1st nanosecond:

2nd nanosecond:

3rd nanosecond:

4th nanosecond:

5th nanosecond:

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.)