CS 31: Intro to Systems
Digital Logic

Kevin Webb
Swarthmore College
September 13, 2022
Reading Quiz
Today

- Hardware basics
  - Machine memory models
  - Digital signals
  - Logic gates

- Manipulating/Representing values in hardware
  - Adders
  - Storage & memory (latches)

Circuits: Borrow some paper if you need to!
Hardware Models (1940’s)

- Harvard Architecture:
- Von Neumann Architecture:
Hardware Models (1940’s)

• Harvard Architecture:

  ![Harvard Architecture Diagram]

• Von Neumann Architecture:

  ![Von Neumann Architecture Diagram]
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
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></th>
<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>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</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>0</td>
<td>0</td>
</tr>
</tbody>
</table>
More Logic Gates

NAND

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

NOR

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

Note the circle on the output. This means “negate it.”

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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A NOR B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Combinational Logic Circuits

• Build up higher level processor functionality from basic gates

Acyclic Network of Gates

Inputs

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 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^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.
Which of these is an XOR circuit?

A:  

B:  

C:  

D:  

E: None of these are XOR.
XOR Circuit: Abstraction

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

A: 0  B: 0  A^B:
A: 0  B: 1  A^B:
A: 1  B: 0  A^B:
A: 1  B: 1  A^B:
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
Digital Circuits - Building a CPU

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!
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>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>
More than one bit?

• When adding, sometimes have *carry in* too

\[
\begin{align*}
0011010 \\
+ & 0001111 \\
\end{align*}
\]
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)
Three-bit Adder (Ripple-carry Adder)

$010 \, (2) \ + \ 011 \, (3) \ = \ 101 \, (2)$
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

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:

A₀ A₁ A₂
B₀ B₁ B₂

3-bit adder
Sum₀ Sum₁ Sum₂

Or₀ Or₁ Or₂

Out₀ Out₁ Out₂

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

3-bit adder

Circuit that takes in \text{Sum}_{0-2} / \text{Or}_{0-2} and only outputs one of them, based on control signal.

Out_0
Out_1
Out_2
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>

![Diagram of 4-Way Multiplexor and truth table]
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

Multiplexor!

Or0
Or1
Or2
**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: | 2 | 3
|---|---|---|
ADD |
```

*op bits: selects which op to output*

```
ALU
```

*Output flags: set as a side effect of *op* (e.g., overflow detected)*
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
Digital Circuits - Building a CPU

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.
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
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:
Registers

• Fixed-size storage (8-bit, 32-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.
  • Grouping N-bit registers gives us register file.

• Register file’s simple interface:
  • Read $R_x$’s value, use for calculation
  • Write $R_y$’s value to store result
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
Digital Circuits - Building a CPU

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.
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 our 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.

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

Mem Addresses (buckets)

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

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

MUX

MUX

ALU

...
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

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

Interpret the instruction bits: What operation? Which arguments?

**Program Counter (PC):**

**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.
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)
Storing results.

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

Program Counter (PC):

Instruction Register (IR):

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

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

32-bit Register #0

MUX

32-bit Register #1

MUX

32-bit Register #2

MUX

32-bit Register #3

... 

Result location specifies where to store ALU output.
Questions so far?

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
WE
Data in
WE
Data in
WE
Data in
WE

32-bit Register #0

MUX

32-bit Register #1

MUX

32-bit Register #2

MUX

32-bit Register #3

...

Register File

Result location specifies where to store ALU output.
Why do we need a program counter? Can’t we just start executing instruction at address 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
MUX
32-bit Register #1
MUX
32-bit Register #2
32-bit Register #3

MUX

ALU

Result might be: Memory Register PC

Register File
Recap CPU Model

Four stages: fetch instruction, decode instruction, execute, store result

Program Counter (PC): Memory address of next instr

Instruction Register (IR): Instruction contents (bits)

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

MUX
MUX

ALU

Register File
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
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

…

MUX
MUX

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

OP Code tells ALU which operation to perform.

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)
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.
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)
Storing results.

Interpret the instruction bits: Store result in register, memory, PC.

Program Counter (PC): Address 0

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

Data in

32-bit Register #0

MUX

32-bit Register #1

MUX

32-bit Register #2

MUX

32-bit Register #3

Register File

ALU

Result might be: Memory

Register

PC
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 a CPU 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
Clock and Circuits

Clock Edges Triggers events

• Circuits have continuous values
• Rising Edge: trigger new input values
• Falling Edge: consistent output ready to read
• Between rising and falling edge can have inconsistent state as new input values flow through circuit

![Clock Diagram]

Clock: ^ new input ^ output ready ^ new input
Cycle Time: Laundry Analogy

- Discrete stages: fetch, decode, execute, store

- Analogy (laundry): washer, dryer, folding, dresser

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\textsuperscript{st} hour:
\begin{tikzpicture}
  \node at (0,0) {W};
\end{tikzpicture}

2\textsuperscript{nd} hour:
\begin{tikzpicture}
  \node at (0,0) {W};
  \node at (1,0) {Dy};
\end{tikzpicture}

3\textsuperscript{rd} hour:
\begin{tikzpicture}
  \node at (0,0) {W};
  \node at (1,0) {Dy};
  \node at (2,0) {F};
\end{tikzpicture}

4\textsuperscript{th} hour:
\begin{tikzpicture}
  \node at (0,0) {W};
  \node at (1,0) {Dy};
  \node at (2,0) {F};
  \node at (3,0) {Dr};
\end{tikzpicture}

5\textsuperscript{th} hour:
\begin{tikzpicture}
  \node at (0,0) {W};
  \node at (1,0) {Dy};
  \node at (2,0) {F};
  \node at (3,0) {Dr};
\end{tikzpicture}

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: \( F D E S \)

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