To execute a program we need:

- **Communication**: getting data from one place to another
- **Computation**: perform arithmetic or logical operations
- **Memory**: store the program, variables, results

Everything is expressed in terms of bits (0s and 1s).

- Communication: Low or high voltage on a wire
- Computation: Compute boolean functions
- Storage: Store bits
Use voltage thresholds to extract discrete values from a continuous signal.

Simplest version: 1-bit signal
- Either high range (1) or low range (0)
- With a guard range between them.

Not strongly affected by noise or low-quality elements; circuits are simple, small and fast.
**Truth Tables**

**And:** $A \& B = 1$ when both $A = 1$ and $B = 1$.

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$A &amp; B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</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>1</td>
</tr>
</tbody>
</table>

**Not:** $\neg A = 1$ when $A = 0$.

<table>
<thead>
<tr>
<th>$A$</th>
<th>$\neg A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Or:** $A \mid B = 1$ when either $A = 1$ or $B = 1$.

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$A \mid B$</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>1</td>
</tr>
</tbody>
</table>

**Xor:** $A \hat{\lor} B = 1$ when either $A = 1$ or $B = 1$, but not both.

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$A \hat{\lor} B$</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>
Computing with Logic Gates

How are these logic functions actually computed in hardware?

- Logic gates are constructed from transistors.
- The output is a boolean function of inputs.
- The gate responds continuously to changes in input with a small delay.

How many of these do you really need?
Aside: Multiple-Input Gates

Some gates allow multiple inputs. For example, a 3-input AND is essentially just a cascade of two 2-input ANDs.

For which gates does it make sense to have extra inputs? For which doesn’t it make sense?
A small circle on either the input or output of a gate means that that signal is inverted. That is, it’s as if there were an inverter (not) gate there.
A Complex Function

Simple boolean functions are implemented by logic gates; more complex functions, by combinations of gates.

\[
\begin{array}{cccc}
    A & B & C & Z \\
    0 & 0 & 0 & 1 \\
    0 & 0 & 1 & 1 \\
    0 & 1 & 0 & 1 \\
    0 & 1 & 1 & 1 \\
    1 & 0 & 0 & 0 \\
    1 & 0 & 1 & 0 \\
    1 & 1 & 0 & 0 \\
    1 & 1 & 1 & 1 \\
\end{array}
\]

\[Z = \neg A \lor (B \land C)\]
Can you see what this circuit does?
Can you see what this circuit does? This is called a *majority circuit*. What function does it compute?
It’s pretty easy to see that any boolean function can be implemented with AND, OR and NOT. Why? We call that a functionally complete set of gates.

You can get by with fewer gates. How would you show each of the following?

- AND and NOT is complete.
- OR and NOT is complete.
- NAND is complete.
- NOR is complete.
- AND alone is not complete.
- OR alone is not complete.

Often circuit designers will restrict themselves to a small subset of gates (e.g., just NAND gates). Why would they do that?
Suppose you wanted to do addition with logic. How might you go about that?
Suppose you wanted to do addition with logic. How might you go about that?

Define a circuit (full adder) that does one step in an addition:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Cin</th>
<th>Cout</th>
<th>S</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>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<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>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
The following circuit is a 1-bit (full) adder:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Cin</th>
<th>Cout</th>
<th>S</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>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</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>1</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>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Adding a Pair of 4-bit Ints

How do you subtract? How do you multiply?
Combinational Circuits

The box contains an acyclic network of logic gates.

- Continuously responds to changes in inputs.
- Outputs become (after a short delay) boolean functions of the inputs.
The following circuit generates a 1 iff a and b are equal.

\[
\text{int eq = (a && b) || (!a && !b);}
\]

Can you design a simpler circuit to do this?

**Hardware description languages (Verilog, VHDL)**

- Describe control, data movement, ...
- “Compile” (synthesize) a hardware description into a circuit.
One of the more widely used HDL’s is Verilog:

```verilog
module simp_circuit (A, B, C, x, y);
    input A, B, C;
    output x, y;
    wire e;
    and g1 (e, A, B);
    not g2 (y, C);
    or g3 (x, e, y);
endmodule
```
Hardware Control Language (HCL)

- Very simple hardware description language.
- Boolean operations have syntax similar to C logical operations.
- We’ll use it to describe control logic for processors.

Data types

- `bool`: Boolean (a, b, c, ...)
- `int`: words (A, B, C, ...)
- Does not specify word size

Statements

- `bool a = bool-expr;`
- `int A = int-expr;`
Boolean expressions

- Logic operations: `a && b`, `a || b`, `!a`
- Set membership: `A in {B, C, D}`

Word expressions

- Case expressions: `[a: A; b: B; c: C]`
- Evaluate Boolean expressions `a`, `b`, `c` in sequence
- Return corresponding word expression for first successful Boolean evaluation.
Word Equality

Word-level representation:

HCL Representation:

Eq = (A == B)

Assume 32-bit word size.

HCL representation
- Equality operation
- Generates Boolean value
HCL Expression:

```
int out = (s && a) || (!s && b);
```

- Control signal \( s \) selects between two inputs \( a \) and \( b \).
- Output is \( a \) when \( s == 1 \), and \( b \) otherwise.
Word Multiplexor

Word-level representation:

HCL Representation:

\[
\text{int Out} = [ \\
   \quad \text{s : A;} \\
   \quad 1 : \text{B;} \\
]\]

Select input word A or B depending on control signal S.
Minimum of 3 words

```
int Min3 = [
    A <= B && A <= C : A;
    B <= A && B <= C : B;
    1 : C;
];
```

4-way Multiplexor

```
int Out4 = [
    !s1 && !s0 : D0;
    !s1 : D1;
    !s0 : D2;
    1 : D3;
];
```

What do these do?
An ALU is an Arithmetic Logic Unit

- Multiple functions: add, subtract, and, xor, others
- Combinational logic to perform functions.
- Control signals select function to be performed.
- Modular: multiple instances of 1-bit ALU
A 4-bit ALU

- Combinational logic: continuously responding to inputs.
- Control signal selects function computed; Y86 ALU has only 4 arithmetic/logical operations.
- Also computes values of condition codes. Note these are not the same as the three Y86 flags:
  - OF: overflow flag
  - ZF: zero flag
  - SF: sign flag
int Out = [
    !s1 && !s0: X+Y;
    !s1 && s0 : X-Y;
    s1 && !s0: X&Y;
    1 : X^Y;
];
How would you design a circuit that records a bit? What does that even mean?
Sequential Logic

How would you design a circuit that records a bit? What does that even mean?

Ideally, you’d like a device (latch) as follows:

```
/0/0/0/0/0/0
/0/0/0/0/0/0
/0/0/0/0/0/0
/0/0/0/0/0/0
/0/0/0/0/0/0
/0/0/0/0/0/0
/0/0/0/0/0/0
/0/0/0/0/0/0
/0/0/0/0/0/0
/0/0/0/0/0/0
/0/0/0/0/0/0
/0/0/0/0/0/0
/0/0/0/0/0/0
/1/1/1/1/1/1
/1/1/1/1/1/1
/1/1/1/1/1/1
/1/1/1/1/1/1
/1/1/1/1/1/1
/1/1/1/1/1/1
/1/1/1/1/1/1
/1/1/1/1/1/1
/1/1/1/1/1/1
/1/1/1/1/1/1
/1/1/1/1/1/1
/1/1/1/1/1/1
/1/1/1/1/1/1
```

Enable

Q

Data

The value on line $Q$ is the current stored value.

To store a new value:

1. Line Enable should be low (0).
2. Place the bit to store on line Data.
3. Raise Enable to high (1).
4. The value on line Data is stored in the device.
5. Lower Enable to low (0).
6. Reading Q returns the stored bit until next store.

Such “state-holding” devices are called sequential logic as opposed to combinational logic.
An SR flip flop is a step in the direction of a latch.

Pulse the R (reset) input to record a 0.

Pulse the S (set) input to record a 1.

<table>
<thead>
<tr>
<th>S</th>
<th>R</th>
<th>Qnext</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Q</td>
<td>hold state</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>reset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>set</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
<td>not allowed</td>
</tr>
</tbody>
</table>

Characteristic table
Gated D Latch: Store and Access One Bit

Higher level representation
D Latch Truth table

<table>
<thead>
<tr>
<th>E/CP</th>
<th>D</th>
<th>Q</th>
<th>Q̅</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>Q</td>
<td>Q̅</td>
<td>No change</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Reset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Set</td>
</tr>
</tbody>
</table>

E (enable) and CP (clock pulse) are just two names for the same input.
A 4-bit Register

4 D latches:
- All share the E (aka WE or Write Enable) input
- D0–D3 are the data input
- Q0–Q3 are the output
Register file provides the CPU with temporary, fast storage.
- N registers.
- Each of K bits.
- L output ports.

Suppose we want eight 4-bit registers and one output port.
Write Enable (WE) must be held at “1” long enough to allow:

- Data to be read;
- Operation (e.g., addition) to be performed;
- Result to be stored in target register.
An edge-triggered flip-flop changes states either at the positive edge (rising edge) or at the negative edge (falling edge) of the clock pulse on the control input.

- A register is made up of several flip flops, each providing storage and access for an individual bit.
- A register file is made up of several registers and control logic.
Clocking

The clock acts to enforce timing control on the chip.

- An integral part of every synchronous system.
- Can be global

Clock Frequency = $1 / \text{clock period}$

- Measured in cycles per second (Hertz)
- $1 \text{ KHz} = 1000 \text{ cycles / second}$
- $1\text{ns} \left(10^{-9} \text{ seconds}\right) = 1\text{GHz} \left(10^9\right) \text{ clock frequency}$
- Higher frequency means faster machine speed.
Random Access Memory (RAM)

Stores many words
- Conceptually, a large array where each row is uniquely addressable.
- In reality, much more complex to increase throughput.
- Multiple chips and banks, interleaved, with multi-word operations.

Many implementations
- Dynamic (DRAM) is large, inexpensive, but relatively slow.
  - 1 transistor and 1 capacitor per bit.
  - Reads are destructive.
  - Requires periodic refresh.
  - Access time takes hundreds of CPU cycles.
- Static (SRAM) is fast but expensive.
  - 6 transistors per bit.
  - Streaming orientation.
**Summary**

**Computation**
- Performed by combinational logic.
- Implements boolean functions.
- Continuously reacts to inputs.

**Storage**
- Registers: part of the CPU.
  - Each holds a single word.
  - Used for temporary results of computation.
  - Loaded on rising clock.
- Memory is much larger.
- Variety of implementation techniques.