STUDY PAGE · AREA 01 OF 14

Digital Logic & Number Systems

01

The logic every testbench checks: how numbers live in bits, how gates combine them, and how flip-flops and state machines remember. Ends with your first self-checking design — a 4-bit ALU.





A	B	Y	
0	0	1	1011
0	1	1	0x2D
1	0	1	-19
1	1	0	

12 h of study

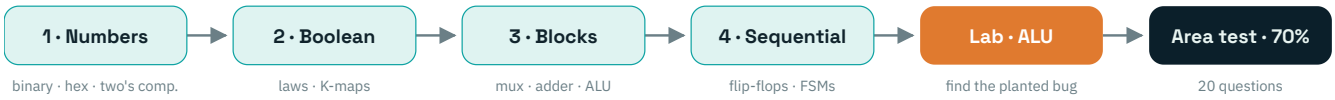
4 modules

8 diagrams · **3** interactive

Free tools · Windows, macOS, Linux

20-question test · 70% to pass

Lab: 4-bit ALU + checker



binary · hex · two's comp. | laws · K-maps | mux · adder · ALU | flip-flops · FSMs | find the planted bug | 20 questions

BEFORE YOU START · A

Who this area is for

The first area of the roadmap. No hardware background needed — just curiosity and a computer.



EE / ECE students

Final-year and master's students preparing for VLSI internships and campus placements in design or verification.

Software engineers

Developers moving into hardware who want the foundations explained from first principles.

QA & test engineers

Test automation people heading into design verification — the same stimulus-and-check mindset applied to silicon.

FPGA & embedded engineers

Engineers who write some HDL already and want the rules and habits that verification teams expect.

RTL designers

Designers who want to move to verification and think like the person checking their code.

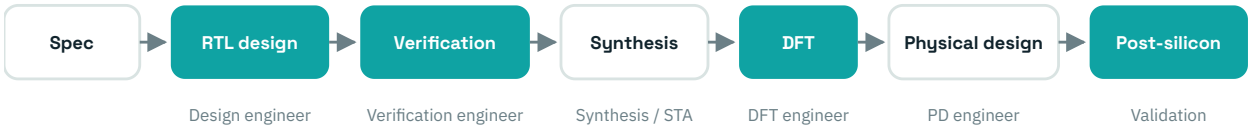
Career returners

Engineers refreshing fundamentals before interviews after time away from hands-on work.

Prerequisites

You need	Level	Where to get it
Basic arithmetic	Comfortable	School maths
Powers of two	Know 1, 2, 4 … 256	Module 1 recaps them
Any programming language	Helpful, not required	—
Command line	Can run a command	Install guide below

Where this area sits in the chip flow — and the roles that use it



Highlighted: stages where engineers reason about bits, gates and flops every day

Every stage of the flow speaks this language. Designers build from these blocks, verifiers predict their outputs, DFT and validation engineers debug them in silicon.

Role	How you use this area on the job	Typical first title
Design verification (DV)	Predict expected values, read waveforms bit by bit, judge overflow and sign bugs	DV Engineer / Verification Engineer I
RTL design	Choose encodings, build datapaths from muxes and adders	ASIC Design Engineer / RTL Engineer
FPGA	Map logic onto LUTs and flip-flops	FPGA Engineer
DFT	Reason about scan flops and stuck-at faults on gates	DFT Engineer
Post-silicon validation	Decode register dumps and hex values from silicon	Validation Engineer

BEFORE YOU START · B

What you will be able to do

Every outcome is practised in the lab and checked by the area test.

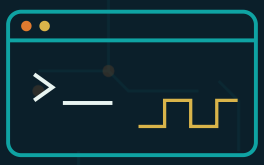


- Convert between binary, hex and decimal and read two's complement values.
- Tell carry from signed overflow and compute both.
- Simplify logic with Boolean laws and Karnaugh maps.
- Describe mux, decoder, adder, comparator and ALU behaviour from their truth tables.
- Explain setup, hold and clock-to-Q, and what violating them does.
- Draw and trace a Moore state machine.
- Write a reference model and compare a design against it.
- Find the planted overflow bug in a 4-bit ALU.

BEFORE YOU START · C

Tools & software

This area needs very little: a simulator for the lab and, optionally, a drawing tool for circuits.



Tool	What it does	Used in this area for	Cost
Icarus Verilog `iverilog` + `vvp`	Event-driven Verilog simulator: compiles, then runs	The ALU lab (`make`) and the first simulation	Free
Logisim-evolution or **Digital**	Draw gates and flip-flops and watch them switch	Trying circuits before writing HDL	Free
GTKWave or **Surfer**	Waveform viewers for `.vcd` / `.fst` dumps	Optional: watching ALU outputs change	Free
VS Code + Verilog extension	Editor with syntax highlighting and lint on save	Writing code	Free
EDA Playground	Browser-based simulators, including commercial ones, after a free sign-up	Zero-install option for the lab	Free
make and **git**	Run builds with one command; track your changes	Lab Makefile; saving your work	Free

Commercial simulators you will meet at work

Simulator	Vendor	Notes
VCS	Synopsys	Widely used for SystemVerilog/UVM regressions
Xcelium	Cadence	Successor of Incisive / NC-Sim
Questa	Siemens EDA	Successor of ModelSim; starter editions ship with FPGA tools

i

Same language, same rules

The ALU you check in the lab is plain Verilog-2005: it runs unchanged on Icarus, Verilator and every commercial simulator.

BEFORE YOU START · D

Install step by step

Pick your computer. About 15 minutes. If anything fails, the browser option works with nothing installed.



Windows

Recommended: WSL 2 (Ubuntu inside Windows). You get the same commands as Linux and as most companies' compute farms.

1

Turn on WSL and install Ubuntu

Open **PowerShell as Administrator**, run the command, then restart.

```
wsl --install
```

After the restart an Ubuntu window opens: choose a user name and password.

2

Install the tools

In the Ubuntu window:

```
sudo apt update
sudo apt install -y iverilog verilator gtkwave make git
```

3

Install VS Code and connect it to WSL

Install VS Code from code.visualstudio.com, add the **WSL** extension, then type `code .` in Ubuntu to open the current folder.

4 **Add Verilog support**

Install the **Verilog-HDL/SystemVerilog/Bluespec SystemVerilog** extension (publisher mshr-h).

5 **Check it works**

```
iverilog -V | head -1
verilator --version
```

> i **No WSL allowed on your laptop?**
> Install the native Icarus Verilog for Windows build (installer from bleyer.org/icarus, which also bundles GTKWave) and tick "Add executable folder to the user PATH". Use EDA Playground for Verilator.

macOS

1 **Install Apple's command-line tools**

```
xcode-select --install
```

2 **Install Homebrew**

Follow the one-line command on brew.sh, then reopen Terminal.

3 **Install the simulators**

```
brew install icarus-verilog verilator make git
```

4 **Install a waveform viewer**

Download **Surfer** from surfer-project.org (native on Apple silicon), or try `brew install --cask gtkwave`.

5 **Editor**

Install VS Code and the **Verilog-HDL/SystemVerilog/Bluespec SystemVerilog** extension.

6 **Check it works**

```
iverilog -V | head -1
verilator --version
```

Linux

1 **Ubuntu / Debian**

```
sudo apt update
sudo apt install -y iverilog verilator gtkwave make git
```

2 **Fedora**

```
sudo dnf install -y iverilog verilator gtkwave make git
```

3 **Arch**

```
sudo pacman -S iverilog verilator gtkwave make git
```

4 **Check the versions**

```
iverilog -V | head -1      # 11 or newer
verilator --version        # 5.x recommended
```

Browser only

1 **Open EDA Playground**

Go to edaplayground.com and create a free account (simulation needs a login).

2 **Paste the code**

Design on the right, testbench on the left.

3 **Choose a simulator**

Under *Tools & Simulators* pick **Icarus Verilog**; tick *Open EPWave after run* for waveforms.

4

Run

Press **Run**; the log shows your `$display` output.

BEFORE YOU START · E

Your first simulation in 5 minutes

Simulate a full adder and print its whole truth table.



1

Make a folder and a file called `full_adder.v`

```
module full_adder (input a, b, cin, output sum, cout);
  assign sum  = a ^ b ^ cin;
  assign cout = (a & b) | (cin & (a ^ b));
endmodule

module tb;
  reg a, b, cin;
  wire sum, cout;
  full_adder dut (.a(a), .b(b), .cin(cin), .sum(sum), .cout(cout));
  integer i;
  initial begin
    $display(" a b cin | cout sum");
    for (i = 0; i < 8; i = i + 1) begin
      {a, b, cin} = i[2:0];
      #1 $display(" %b %b  %b  |   %b   %b", a, b, cin, cout, sum);
    end
  end
endmodule
```

2

Compile it

```
iverilog -g2012 -o fa.vvp full_adder.v
```

3

Run it

```
vvp fa.vvp
```

Expected output:

```
 a b cin | cout sum
 0 0  0  |   0   0
 0 0  1  |   0   1
 0 1  0  |   0   1
 0 1  1  |   1   0
 1 0  0  |   0   1
 1 0  1  |   1   0
 1 1  0  |   1   0
 1 1  1  |   1   1
```

4

What to notice

The loop drives all 8 input combinations — an **exhaustive** test. `#1` lets the outputs settle before printing. Compare each row with the full adder table in Module 3.

BEFORE YOU START · F

Study plan

12 hours at about 6 hours a week.

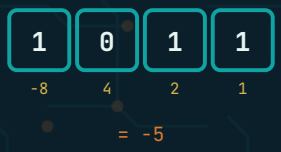


Session	Do this	Time
Day 1	Install tools · first simulation · Module 1 (number systems, two's complement, carry vs overflow)	3 h
Day 2	Module 2 (gates, laws, K-maps); try the gate lab	2.5 h
Day 3	Module 3 (mux, decoder, adders, ALU)	2 h
Day 4	Module 4 (flip-flops, timing, FSMs); feed bits to the sequence detector	2 h
Day 5	Lab: complete the reference model, find the bug, reach PASS · area test	2.5 h

MODULE 1 OF 4

Number systems & two's complement

How numbers live in wires — and the two ways an addition can fall off the end.



- Binary, hex and decimal
- Two's complement and sign extension
- Carry vs signed overflow
- Overflow equations for add and subtract

⏱ 3 hours · Area 01 · Digital logic

Start from zero

New to this? Work through these steps first, in order. Already comfortable? Skim them and jump to the reference below.

STEP 1

What "digital" means

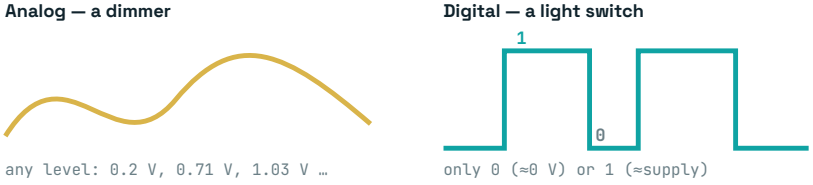
In plain words

A wire inside a chip is either at a **low** voltage (close to 0 V) or a **high** voltage (close to the supply). We call low **0** and high **1**. Nothing in between counts.

Think of it like

A dimmer switch can sit at any brightness — that is **analog**. An ordinary light switch is only off or on — that is **digital**.

Worked example



Same wire, two ways to use it. Chips use the right-hand way.

Aa The word for it

That single 0-or-1 value is a **bit** — short for *binary digit*.

! Why it matters

Two levels are easy to tell apart even when the signal is noisy: a slightly wobbly high is still clearly high. That is why every value you will ever check in a testbench is built from bits.

2 STEP 2 — A transistor is a switch

≈ In plain words

Something has to flip those 0s and 1s. That is a **transistor**: a switch controlled by another electrical signal instead of a finger.

✳ Think of it like

A tap controlled by a lever: the lever (the **gate**) decides whether water (current) flows.

✎ Worked example



Gate 0: no current, lamp dark. Gate 1: the switch closes, current flows, the lamp lights.

Aa The word for it

The control input is the **gate**. A modern chip has billions of these switches.

! Why it matters

Every gate, adder and memory cell in this course is just transistor switches wired together — nothing more mysterious than that.

3 STEP 3 — Counting with bits

≈ In plain words

Put four bits side by side and they can count. Each bit is one lamp: off is 0, on is 1.

Think of it like

A car odometer: the right wheel goes 0…9, rolls over to 0 and pushes the next wheel up. Binary is the same, but each wheel has only two values, so it rolls over after 1, not 9.

Worked example

Decimal	Bits	Decimal	Bits	Decimal	Bits	Decimal
0	`0000`	4	`0100`	8	`1000`	12
1	`0001`	5	`0101`	9	`1001`	13
2	`0010`	6	`0110`	10	`1010`	14
3	`0011`	7	`0111`	11	`1011`	15

Watch 1 → 2: the right bit rolls over to 0 and the next bit turns on (`0010`).

The word for it

Four bits give **16** different patterns: 0 to 15.

Why it matters

Counting in binary is how you will read counters, addresses and state numbers in waveforms.

STEP 4

Place value: every position has a weight

In plain words

In decimal, 345 means 3 hundreds + 4 tens + 5 ones. Each position is worth **10×** the one to its right. In binary each position is worth **2×** the one to its right.

Think of it like

Coins of 8, 4, 2 and 1: to pay any amount up to 15 you either use each coin once or not at all.

Worked example

1. Write the weights over the bits: `8 4 2 1`.
2. Read `1011`: one 8, no 4, one 2, one 1.
3. Add the weights where the bit is 1: 8 + 2 + 1 = **11**.
4. A byte continues the weights: 128 64 32 16 8 4 2 1. So `1011 0110` = 128 + 32 + 16 + 4 + 2 = **182**.

The word for it

The weights are powers of two. The leftmost bit is the **MSB** (most significant), the rightmost the **LSB**.

Why it matters

To check any value in a waveform you add weights in your head. It becomes automatic within a week.

STEP 5

5 Decimal to binary: keep dividing by two

In plain words

To turn a decimal number into binary, divide by 2 again and again and write down each remainder.

Think of it like

Sharing sweets between two people: each time, note whether one is left over.

Worked example

Step	Divide	Result	Remainder
1	45 ÷ 2	22	**1**
2	22 ÷ 2	11	**0**
3	11 ÷ 2	5	**1**
4	5 ÷ 2	2	**1**
5	2 ÷ 2	1	**0**
6	1 ÷ 2	0	**1**

Read the remainders from the **bottom up**: `101101`. Check with weights: 32 + 8 + 4 + 1 = 45 ✓

Aa The word for it — Repeated division by the **radix** (base) 2. The first remainder is the LSB.

! Why it matters — You will often turn a number from a spec (a length of 45, an address of 200) into the bit pattern you expect on a bus.

6 STEP 6 — Hexadecimal: four bits in one character

≈ In plain words — Long strings of 0s and 1s are hard to read. Hex is a shorthand with 16 digits: 0–9, then A=10, B=11, C=12, D=13, E=14, F=15.

☀ Think of it like — Writing “1 dozen” instead of drawing 12 dots.

✎ Worked example

1. Split the bits into groups of four from the right: `1011 0110`.
2. `1011` = 11 = **B**; `0110` = 6 = **6**.
3. So 182 = **B6** in hex. In Verilog: `8'hB6` — 8 bits, h for hex, value B6.

Aa The word for it — One **hex digit** = exactly four bits (a *nibble*).

! Why it matters — Simulators, logs and register maps print values in hex. You must read `32'hDEAD_BEEF` as fast as a phone number.

PAUSE & TRY **What is `8'h3C` in binary and in decimal?** Answer

3 = `0011`, C = `1100` → `0011 1100` = 32 + 16 + 8 + 4 = **60**.

7 STEP 7 — Fixed width and range

≈ In plain words — In hardware a number always has a fixed number of wires: 4, 8, 32 bits. With *n* bits you get 2^n patterns.

Think of it like

An odometer at 999 rolls to 000 — there is no fourth wheel for the 1.

Worked example

Width	Patterns	Unsigned range
4 bits	16	0 … 15
8 bits	256	0 … 255
16 bits	65,536	0 … 65,535
32 bits	≈ 4.3 billion	0 … 4,294,967,295

8 bits: 255 + 1 → every bit rolls over → `0000_0000`. The 1 that fell off the top is the carry.

The word for it

The bit that falls off the top of an unsigned addition is the **carry**.

Why it matters

“Wrap-around at the maximum” is one of the most common corner cases you will test.

8 STEP 8 Negative numbers: two’s complement

In plain words

There is no minus sign on a wire. Hardware stores negatives by giving the **leftmost bit a negative weight**.

Think of it like

A clock face: going back 1 from 0 lands on 15 — the patterns wrap around a wheel.

Worked example

1. 4-bit weights become `-8 4 2 1`.
2. `1000` = −8. `1111` = −8 + 4 + 2 + 1 = **−1**.
3. `0xxx` patterns are 0…7; `1xxx` patterns are −8…−1.

The word for it

Two's complement. The leftmost bit is the **sign bit**: 0 positive, 1 negative. n bits hold -2^{n-1} … $2^{n-1}-1$.

Why it matters — The same bits mean different numbers depending on whether the design treats them as signed. Many real bugs are exactly that mix-up.

9 STEP 9 — Negating: invert, then add one

In plain words — To make a number negative: flip every bit, then add 1.

Think of it like — A two-step recipe you can do on paper in five seconds.

Worked example

1. 6 = `0110`.
2. Invert every bit → `1001`.
3. Add 1 → `1010` = −8 + 2 = **−6** ✓
4. Add 6 + (−6) with an ordinary adder: `0110 + 1010 = 1 0000`. Keep 4 bits → `0000`. The same adder handles both signs.

The word for it — This is the **two's complement negation**.

Why it matters — It is why chips need only one adder for signed and unsigned maths — and why you check subtraction as “add the negation”.

10 STEP 10 — Carry versus overflow

In plain words — There are two different ways a result can “fall off the end”. **Carry** is about unsigned numbers; **overflow** is about signed numbers.

Think of it like — Carry: the odometer ran out of wheels. Overflow: the answer ran off the edge of the signed wheel, from +7 straight to −8.

Worked example

1. Unsigned 8-bit: 255 + 1 needs 9 bits → **carry = 1**.
2. Signed 4-bit: 7 + 1 should be 8, but `1000` reads as −8 → **overflow = 1**, carry = 0.

Aa The word for it	**Carry** (unsigned out of range) and **overflow** (signed out of range) are separate flags.
! Why it matters	You check both flags separately — exactly what you will do in this area's ALU lab, where the planted bug is in a flag.

PAUSE & TRY **In 4 bits, what is 5 + 4 read as unsigned, and read as signed? Is there carry? Overflow?** Answer

The bits are `1001`. Unsigned: 9, fits → no carry. Signed: −7, but the true answer is 9 → **overflow, no carry**.

Going further: the reference

The same ideas at the next level: precise definitions, tables, rules and interview questions.

Key definitions

Bit
One binary digit, 0 or 1 — one wire in hardware.

Nibble / byte / word
4 bits / 8 bits / the natural width of a datapath (often 32 or 64).

Radix (base)
How many digit values a number system has: 2 (binary), 10 (decimal), 16 (hexadecimal).

MSB / LSB
Most / least significant bit: the leftmost bit has the largest weight.

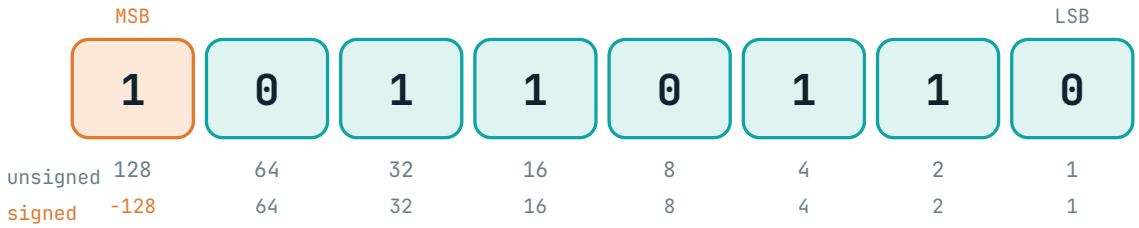
Two's complement
The way hardware stores signed integers: the MSB has a negative weight, -2^{n-1}.

Sign extension
Widening a signed value by copying its MSB into the new upper bits.

Carry
The extra bit produced when an unsigned result needs one more bit.

Overflow
When a signed result falls outside the range the width can represent.

Positional notation: every bit has a weight




```
unsigned 128+32+16+4+2 = 182 · signed -128+32+16+4+2 = -74 · hex 8'hB6
```

Same bits, two meanings. The hardware stores only the pattern; whether it means 182 or −74 is decided by how the design interprets it.

Base	Name	Digits	Verilog literal	Example (decimal 45)
2	Binary	0–1	`8'b0010_1101`	0010 1101
16	Hexadecimal	0–9, A–F	`8'h2D`	2D
10	Decimal	0–9	`8'd45`	45
8	Octal	0–7	`8'o55`	55 (rare in hardware)

Try it: number explorer

Value (0–255): 182

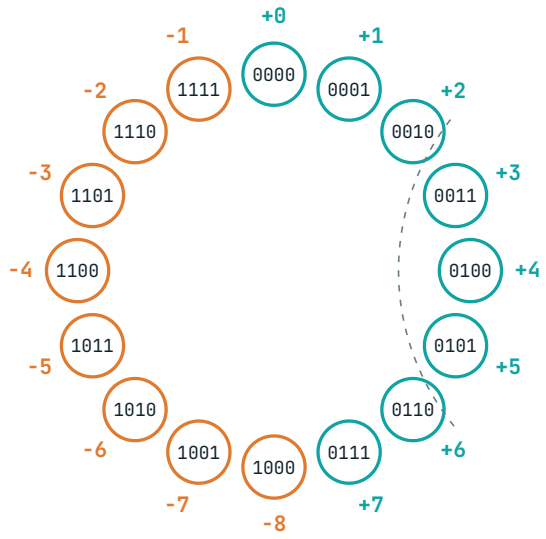


```
binary  8'b1011_0110
hex     8'hB6
unsigned 182
signed   -74   (MSB = 1 → negative)
16-bit sign-extended: 16'hFFB6
```

Converting by hand

Conversion	Method	Worked example
Binary → decimal	Add the weights of the 1 bits	`1011_0110` = 128+32+16+4+2 = **182**
Decimal → binary	Subtract the largest power of two that fits, repeat	45 = 32+8+4+1 → `0010_1101`
Binary ↔ hex	Group four bits from the right	`1011_0110` → B 6 → `8'hB6`
Negate (two's complement)	Invert every bit, add 1	−10: `0000_1010` → `1111_0101` + 1 = `1111_0110`
Sign-extend 4 → 8 bits	Copy the MSB into the new bits	`1010` (−6) → `1111_1010` (still −6)

The two's complement wheel



Read the wheel

- Adding 1 moves one step clockwise.
- +7 + 1 lands on −8: that jump is **signed overflow**.
- 1111 + 1 lands on 0000: that wrap is the **unsigned carry** (15 + 1 = 16 needs 5 bits).
- Negate = reflect: invert all bits, add 1.
- Range of n bits: -2^{n-1} … $2^{n-1} - 1$.

Why one adder handles signed and unsigned. The wheel is the same; only the labels you read change. Carry and overflow are two different ways of falling off it.

Carry versus overflow

Operation (8-bit)	Unsigned view	Signed view	carry	overflow
200 + 100	300 doesn't fit → 44	−56 + 100 = 44 ✓	1	0
100 + 50	150 ✓	+150 doesn't fit → −106	0	1
−1 + −1 (`FF + FF`)	510 → 254	−2 ✓	1	0
−100 + −100	312 → 56	−200 doesn't fit → +56	1	1

```
// Overflow rules (n-bit, result in sum / diff)
add_overflow = (a[n-1] == b[n-1]) && (sum[n-1]  != a[n-1]);  // same signs in, d
sub_overflow = (a[n-1] != b[n-1]) && (diff[n-1] != a[n-1]);  // different signs
```

★ **Verification habit: compute one bit wider**

A checker should calculate the expected result in a wider integer (or plain 32-bit `int`) so it can see the carry and the true signed value, then compare flags explicitly.

× **Common mistakes**

Mixing signed and unsigned operands in one Verilog expression (the whole expression becomes unsigned) · zero-extending a negative number · treating overflow and carry as the same flag · forgetting that hex digits map to exactly four bits.

INTERVIEW CORNER

Q. What is −1 in 8-bit two's complement? **A.** `8'hFF` (all ones).

Q. How do you detect signed overflow in an adder? **A.** Both operands have the same sign and the result's sign is different; equivalently, carry into the MSB ≠ carry out of the MSB.

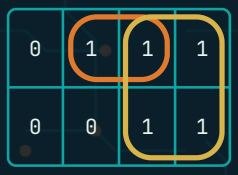
Key takeaways

1. Hardware stores patterns; signed or unsigned is an interpretation.
2. Two's complement: MSB weighs -2^{n-1}; negate by invert-and-add-one.
3. Carry is about unsigned results, overflow about signed ones.
4. Sign-extend signed values; zero-extend unsigned ones.

MODULE 2 OF 4

Boolean algebra & Karnaugh maps

The algebra of 0 and 1, and the fastest way to simplify it by hand.



- Gates and truth tables
- Laws: De Morgan, absorption, consensus
- Sum of products and minterms
- K-map grouping and don't-cares

⏱ 2.5 hours | Area 01 · Digital logic

Start from zero

New to this? Work through these steps first, in order. Already comfortable? Skim them and jump to the reference below.

STEP 1

1 From bits to decisions: the logic gate

≈ In plain words	A logic gate is a tiny circuit of a few transistors. It takes one or two bits in and gives one bit out, following a fixed rule.
✳ Think of it like	A vending-machine rule: “if the coin is in AND the button is pressed, drop the snack”.
Aa The word for it	A **logic gate**. Every digital circuit — a calculator, a processor — is thousands or millions of gates wired together.
! Why it matters	When you read RTL you are reading a description of gates. Knowing the few basic rules lets you predict any output.

STEP 2

2 Truth tables: the full specification

≈ **In plain words**

A truth table lists **every** combination of inputs and the output for each one. Nothing is left out.

☀ **Think of it like**

A complete price list: every item, every price — no “ask at the counter”.

✎ **Worked example**

1. 1 input → 2 rows (0, 1).
2. 2 inputs → 4 rows (00, 01, 10, 11).
3. 3 inputs → 8 rows. Each new input doubles the rows: 2^n.

Aa **The word for it**

A **truth table** with 2^n rows for n inputs.

! **Why it matters**

For a verification engineer the truth table **is** the specification: if every row behaves, the block is correct.

PAUSE & TRY **How many rows does a truth table with four inputs need?** Answer

2, 4, 8, **16** — two to the power of four.

3 STEP 3

NOT: the opposite

≈ **In plain words**

NOT has one input, and its output is the opposite: 0 → 1, 1 → 0.

☀ **Think of it like**

A “do not disturb” sign that flips: when you're in, the sign says out.

✎ **Worked example**

a	y = ~a
0	1
1	0

Aa **The word for it**

An **inverter**. Symbol: a triangle with a small circle at the tip — the circle always means *invert*. Verilog: `y = ~a;`

! **Why it matters**

Active-low signals (like `rst_n`) are everywhere; you invert in your head all day.

4 STEP 4 — AND: all inputs must be 1

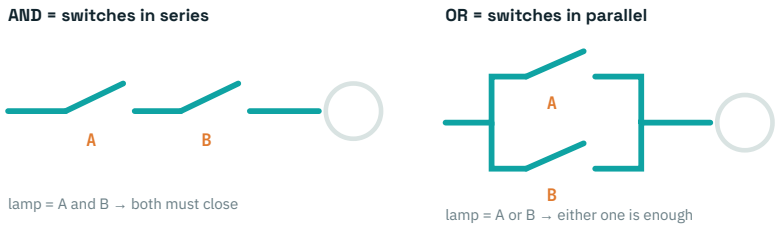
≈ **In plain words**

AND outputs 1 only when **every** input is 1.

☀ **Think of it like**

A bank vault that opens only when two people turn their keys at the same time. As switches: two switches in **series**.

✎ **Worked example**



Series switches behave like AND; parallel switches behave like OR.

a b	a & b
0 0	0
0 1	0
1 0	0
1 1	**1**

AND as a mask: `1011_0110 & 0000_1111 = 0000_0110` — the top four bits are forced to 0, the bottom four pass through.

Aa **The word for it**

Symbol: flat back, round front. Verilog: `a & b`.

! **Why it matters** Masks are how designs pick fields out of registers, and how you write checks on just the bits you care about.

5 STEP 5

OR: any input is enough

≈ **In plain words** OR outputs 1 when **at least one** input is 1.

☀ **Think of it like** A car's interior light comes on if the driver's door OR the passenger's door is open. As switches: two switches in **parallel**.

✎ **Worked example**

a b	a | b
0 0	0
0 1	1
1 0	1
1 1	1

OR forces bits on: `0011_0110 | 1000_0000 = 1011_0110` — the top bit becomes 1 whatever it was.

Aa **The word for it** Verilog: `a | b`.

! **Why it matters** Setting a flag bit in a register is an OR. Collecting “any error happened” is an OR of all error bits.

6 STEP 6

XOR: are the inputs different?

≈ **In plain words** XOR outputs 1 when exactly one input is 1 — when the inputs **differ**.

Think of it like — A staircase light with a switch at the bottom and the top: flipping either one changes the light.

Worked example

a b	a ^ b
0 0	0
0 1	1
1 0	1
1 1	0

Bit by bit: `1010 ^ 1100 = 0110` — 1 wherever the columns differ.

The word for it — Exclusive OR. Verilog: `a ^ b`.

Why it matters — XOR is the sum bit of an adder (1 + 1 = 0, carry 1), the heart of parity and CRC checks, and the quickest way to compare two values bit by bit.

PAUSE & TRY **XOR bit by bit: `1010` and `1100` ?** Answer

1⊕1 = 0, 0⊕1 = 1, 1⊕0 = 1, 0⊕0 = 0 → **`0110`**.

STEP 7

7 NAND and NOR: the universal gates

In plain words — NAND is AND followed by NOT; NOR is OR followed by NOT.

Think of it like — LEGO bricks: with enough copies of one brick you can build any shape.

Worked example

a b	NAND	NOR
0 0	1	1
0 1	1	0
1 0	1	0
1 1	0	0

Tie both NAND inputs together → NOT. NAND then NOT → AND. Add De Morgan → OR.

The word for it

NAND and NOR are **universal**: either one alone can build any circuit.

Why it matters

Real silicon is built mostly from NAND/NOR cells; you will see them in gate-level netlists.

STEP 8

8 Combining gates

In plain words

Gates connect: the output of one becomes the input of the next.

Think of it like

Plumbing: water from one pipe feeds the next junction.

Worked example

1. Function: `y = (a & b) | c`.
2. Inputs a = 1, b = 0, c = 1.
3. AND first: 1 & 0 = **0**.
4. Then OR: 0 | 1 = **1**. So y = 1.

The word for it

A **combinational circuit**: the output depends only on the inputs right now.

Why it matters

This is exactly how you compute the expected value of any RTL expression: evaluate from the inside out.

PAUSE & TRY **Same circuit, `y = (a & b) | c`, with a = 1, b = 1, c = 0. What is y?** Answer

AND: 1 & 1 = 1. OR: 1 | 0 = **1**.

9 STEP 9

Simplifying logic

≈ **In plain words**

Many different gate circuits give the same truth table. Simple rules let you remove the extra gates.

☀ **Think of it like**

Tidying a sentence: “the very very big” → “the big” — same meaning, fewer words.

✎ **Worked example**

1. Rules: `a & 1 = a`, `a & 0 = 0`, `a | 0 = a`, `a | 1 = 1`.
2. A Karnaugh map groups neighbouring 1s: f = 1 for m1, m3, m5, m6, m7.
3. A group of four where c = 1 gives `c`; a pair where a = b = 1 gives `a & b`.
4. Result: `f = c | (a & b)`.

Aa **The word for it**

Boolean algebra and the **Karnaugh map** (the full method is in the reference below).

! **Why it matters**

Synthesis tools simplify for you, but you must be able to read a truth table and predict the output.

Going further: the reference

The same ideas at the next level: precise definitions, tables, rules and interview questions.

Key definitions

Boolean algebra
Algebra on values 0 and 1 with AND (·), OR (+) and NOT (¯).

Truth table
Every input combination listed with its output: 2^n rows for n inputs.

Minterm
A product term that is 1 for exactly one input combination (m0, m1, ...).

Sum of products (SOP)
An OR of AND terms, e.g. `a·b + c` — maps directly to two levels of gates.

Karnaugh map
A truth table drawn in Gray order so that adjacent 1s can be grouped to simplify.

Don’t care (X)
An input combination that never happens, so its output can be chosen freely.

Universal gate
NAND or NOR alone can build every other function.

Hazard (glitch)
A short wrong pulse on an output while inputs change through different path delays.

Try it: gate lab

Inputs:

AND	OR	XOR	NAND
0	0	0	1

NOR	XNOR
1	1

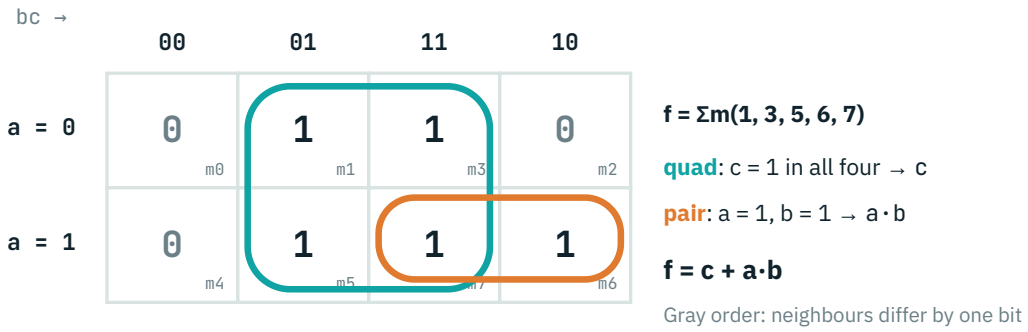
Gates and their truth tables

A B	AND a&b	OR a | b	XOR a^b	NAND	NOR	XNOR
0 0	0	0	0	1	1	1
0 1	0	1	1	1	0	0
1 0	0	1	1	1	0	0
1 1	1	1	0	0	0	1

The laws you will actually use

Law	Form	Use it to
Identity / null	a·1 = a, a+0 = a, a·0 = 0, a+1 = 1	Remove constants
Complement	a·ā = 0, a+ā = 1	Kill impossible terms
Absorption	a + a·b = a, a·(a+b) = a	Delete redundant terms
De Morgan	$(a \cdot b)^{-} = \bar{a} + \bar{b}$, $(a+b)^{-} = \bar{a} \cdot \bar{b}$	Push inversions; build with NAND/NOR
Distributive	a·(b+c) = a·b + a·c, a + b·c = (a+b)(a+c)	Factor or expand
Consensus	a·b + ā·c + b·c = a·b + ā·c	Spot redundant (hazard-covering) terms

Karnaugh maps: simplify by grouping



Group 1s in powers of two (1, 2, 4, 8), as large as possible, wrapping around edges. Each group becomes one product term; variables that change inside the group drop out.

```
// The simplified function in Verilog — synthesis would find this too
assign f = c | (a & b);
```

Why a verifier cares

You rarely minimise logic by hand at work — synthesis does it. But reading a truth table, spotting a missing don't-care, or deriving the expected value of a decoder from its spec is daily verification work.

Common mistakes

Grouping in non-power-of-two sizes · forgetting edges wrap around · drawing the map in binary order 00 01 10 11 instead of Gray order · treating a don't-care as 1 when it doesn't help a group.

INTERVIEW CORNER

Q. Implement an inverter using only a NAND gate. **A.** Tie both inputs together: $\overline{(a \cdot a)} = \bar{a}$.

Q. What is a static-1 hazard? **A.** A momentary 0 on an output that should stay 1 when one input changes; covered by adding the consensus term.

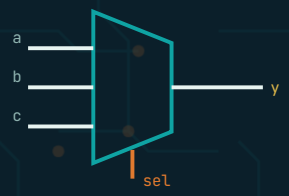
Key takeaways

1. A truth table fully specifies combinational logic.
2. K-maps group adjacent 1s in Gray order to drop variables.
3. NAND and NOR are universal; De Morgan converts between forms.

MODULE 3 OF 4

Combinational building blocks

Mux, decoder, adder, comparator — and the ALU that combines them.



- Multiplexers and decoders
- Half, full and ripple-carry adders
- Comparators
- The 4-bit ALU's operations and flags

⏱ 2 hours · Area 01 · Digital logic

Start from zero

New to this? Work through these steps first, in order. Already comfortable? Skim them and jump to the reference below.

STEP 1

The multiplexer: a selector

In plain words

A mux passes **one** of several inputs to its output. A separate **select** input decides which.

Think of it like

A railway switch: the lever decides which track the train goes down.

Worked example

1. 2:1 mux: select = 0 → out = input 0; select = 1 → out = input 1.
2. 4:1 mux: four inputs need **two** select bits, because 2 bits name 4 choices.
3. select = `10` (2) → out = input 2.

The word for it

Multiplexer (mux). In Verilog every `? :` and every `case` that picks a value becomes one.

Why it matters — Whenever a design chooses between values, there is a mux — and a select signal you must drive through every value.

STEP 2

2 The decoder: one code, one output

In plain words — A decoder does the opposite: it takes a code and turns on **exactly one** of its outputs.

Think of it like — A hotel key card: the code opens exactly one door.

Worked example

code	out 3	out 2	out 1	out 0
00	0	0	0	1
01	0	0	1	0
10	0	**1**	0	0
11	1	0	0	0

The word for it — **Decoder**; its outputs are **one-hot** (exactly one is 1).

Why it matters — That is how a chip picks one register or one memory block out of many: address decoding. A check: “exactly one output is high”.

STEP 3

3 Adding two bits: the half adder

In plain words — Adding bits is just counting: 0+0 = 0, 0+1 = 1, 1+0 = 1, and 1+1 = 2, which is `10` in binary — a sum of 0 with a carry of 1.

Think of it like — Column addition at school: 5 + 7 = 12 → write 2, carry 1.

✎ **Worked example**

a b	carry	sum
0 0	0	0
0 1	0	1
1 0	0	1
1 1	**1**	**0**

Look at the columns: **sum** is exactly XOR, **carry** is exactly AND.

Aa **The word for it**

A **half adder**: `sum = a ^ b`, `carry = a & b`.

! **Why it matters**

You just built arithmetic from two gates. Every adder you will verify grows from this.

4 STEP 4 Full adders and the ripple carry

≈ **In plain words**

A **full adder** also adds a carry coming in from the right. Chain one per bit and the carry passes from bit to bit.

☀ **Think of it like**

A line of people passing a bucket: each waits for the one before.

✎ **Worked example**

1. 0111 (7) + 0001 (1), starting at the right.
2. Bit 0: 1 + 1 = 0, carry 1.
3. Bit 1: 1 + 0 + carry 1 = 0, carry 1.
4. Bit 2: 1 + 0 + carry 1 = 0, carry 1.
5. Bit 3: 0 + 0 + carry 1 = 1. Result `1000` = 8 ✓

Aa **The word for it**

A **ripple-carry adder**: the carry ripples right to left.

! **Why it matters**

The longest carry chain (like `0111 + 0001` or `1111 + 0001`) is a classic corner case to include in tests.

5

STEP 5

The ALU: many operations, one block

In plain words

An ALU (Arithmetic Logic Unit) computes several operations — ADD, AND, OR, XOR … — and an **opcode** chooses which result comes out. Inside, a mux picks one of the results.

Think of it like

A Swiss-army knife: many tools, you flip out one at a time.

Worked example

opcode	operation	a = 6 (0110), b = 3 (0011)	y
000	ADD	6 + 3	`1001` = 9
010	AND	0110 & 0011	`0010` = 2
011	OR	0110 | 0011	`0111` = 7
100	XOR	0110 ^ 0011	`0101` = 5

The ALU also reports **flags**: carry, overflow and zero. For 6 + 3 in 4 bits: unsigned 9 fits (carry 0), but signed 9 is more than +7 → **overflow = 1**.

The word for it

ALU, **opcode**, **flags**.

Why it matters

This 4-bit ALU is exactly what you verify in Lab 01 — with an exhaustive test of all 2,048 input combinations.

PAUSE & TRY

4-bit ALU, a = 5 (0101), b = 3 (0011). What do ADD, AND and XOR give?

Answer

ADD: `1000` = 8 (unsigned fits, signed overflows). AND: `0001` = 1. XOR: `0110` = 6.

Going further: the reference

The same ideas at the next level: precise definitions, tables, rules and interview questions.

Key definitions

Multiplexer (mux)
Selects one of 2^n inputs onto one output using an n-bit select.

Decoder
Turns an n-bit code into 2^n one-hot outputs; used for address decoding.

Encoder / priority encoder
Reverse of a decoder; a priority encoder reports the highest active input.

Half / full adder
Adds two bits (half) or two bits plus carry-in (full).

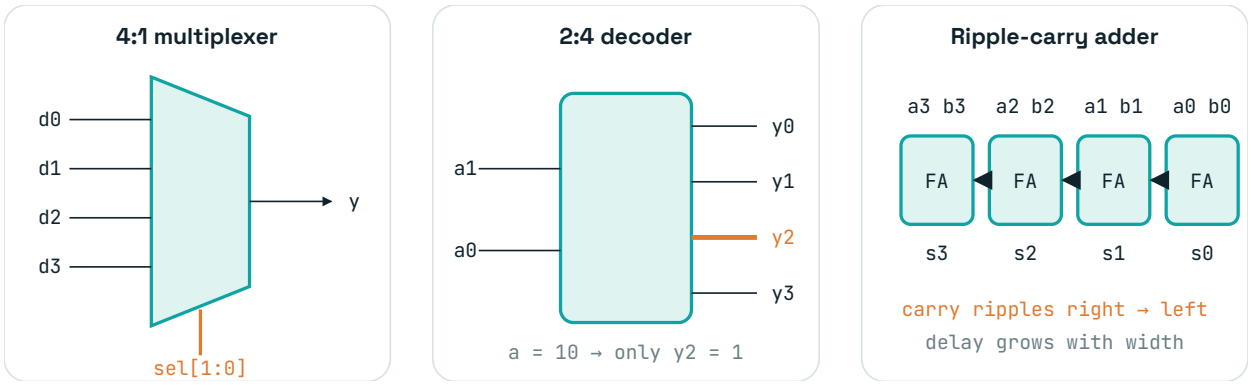
Ripple-carry adder
Full adders chained so each carry feeds the next; simple but slow for wide words.

Comparator
Tells whether a = b, a < b or a > b; equality is an AND of XNORs.

ALU
Arithmetic Logic Unit: one block that performs the operation chosen by an opcode.

Propagation delay
Time from an input change to the output settling; sets how fast a circuit can run.

Three blocks you will meet in every design



Mux, decoder, adder. Most datapaths are these three, wired together and registered.

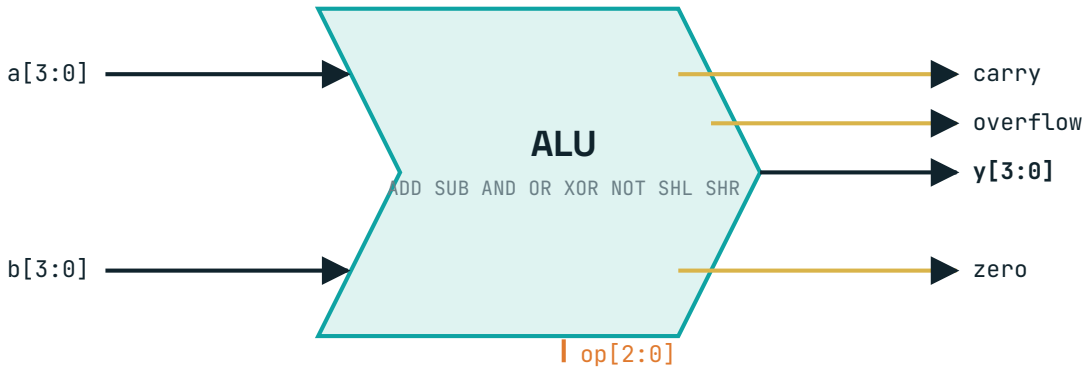
```
assign y   = sel[1] ? (sel[0] ? d3 : d2) : (sel[0] ? d1 : d0);   // 4:1 mux
assign dec = 4'b0001 << a;                                        // 2:4 decoder
assign {cout, sum} = a + b + cin;                                 // adder with carry
```

Full adder truth table

a b cin	cout sum	a b cin	cout sum
0 0 0	0 0	1 0 0	0 1
0 0 1	0 1	1 0 1	1 0
0 1 0	0 1	1 1 0	1 0
0 1 1	1 0	1 1 1	1 1

`sum = a ⊕ b ⊕ cin` and `cout = a·b + cin·(a ⊕ b)`. Chaining n full adders gives an n-bit adder; faster designs (carry-lookahead, carry-select) compute carries in parallel.

The ALU you will verify in the lab



One block, eight operations. The opcode chooses the operation; the flags describe the result.

op	Name	y	carry	overflow
000	ADD	a + b	carry out	signed add overflow
001	SUB	a − b	borrow (a < b unsigned)	signed sub overflow
010	AND	a & b	0	0
011	OR	a | b	0	0
100	XOR	a ^ b	0	0
101	NOT	~a	0	0
110	SHL	a << 1	a[3]	0
111	SHR	a >> 1	a[0]	0

★ **Exhaustive when you can**

A 4-bit ALU has 4 + 4 + 3 = 11 input bits: 2,048 combinations. Testing all of them takes milliseconds, so the lab's testbench is exhaustive — no luck involved. Beyond about 20 input bits you switch to constrained-random stimulus (Area 05).

Common mistakes

Letting the adder result truncate silently (declare it one bit wider) · forgetting that SUB's carry means borrow · shift amount and direction mixed up · comparing signed values with an unsigned comparator.

INTERVIEW CORNER

Q. How many 2:1 muxes make a 4:1 mux? **A.** Three: two on the first level, one on the second.

Q. Why is a ripple-carry adder slow? **A.** The carry may travel through every stage, so delay grows linearly with width.

Key takeaways

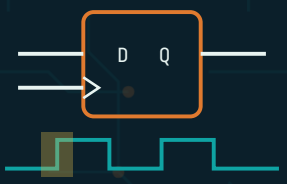
1. Datapaths are built from mux, decoder, adder and comparator.
2. An adder's result needs n+1 bits; the extra bit is the carry.
3. Small blocks can be verified exhaustively.

MODULE 4 OF 4

Flip-flops & state machines

How hardware remembers, how fast it can run, and how it follows a sequence.

- Latches vs flip-flops
- Setup, hold, clock-to-Q, metastability
- Moore and Mealy FSMs
- Resets



2 hours | Area 01 · Digital logic

Start from zero

New to this? Work through these steps first, in order. Already comfortable? Skim them and jump to the reference below.

STEP 1

Circuits that remember

In plain words

Everything so far has no memory: change the inputs and the output changes at once. An adder can't remember the last sum; a gate can't count.

Think of it like

A calculator without a memory key versus one with it.

Worked example

Kind	Output depends on	Examples
Combinational	the inputs **now**	gates, mux, adder, ALU
Sequential	the inputs **and the past**	flip-flops, registers, counters, state machines

The word for it

Combinational versus **sequential** logic.

Why it matters

Real chips must remember: a counter value, a received byte, which step of a protocol they are on. Most bugs live in that memory.

STEP 2

2 The clock: the chip's heartbeat

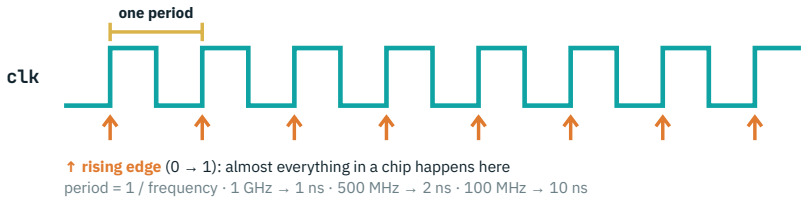
In plain words

A clock is a signal that flips 0, 1, 0, 1 forever at a steady rate, so that every memory element updates together.

Think of it like

A drummer keeping time for a whole band.

Worked example



The moment the clock goes from 0 to 1 is the rising edge.

The word for it

The 0→1 moment is the **rising edge**. Time between rising edges is the **period**; period = 1 / frequency.

Why it matters

Almost everything in a chip happens on the rising edge. Testbenches drive and sample relative to it.

PAUSE & TRY **If the clock is 500 MHz, how long is one period?** Answer

Half the frequency of 1 GHz means double the time: **2 ns**.

STEP 3

3 The D flip-flop: a camera for one bit

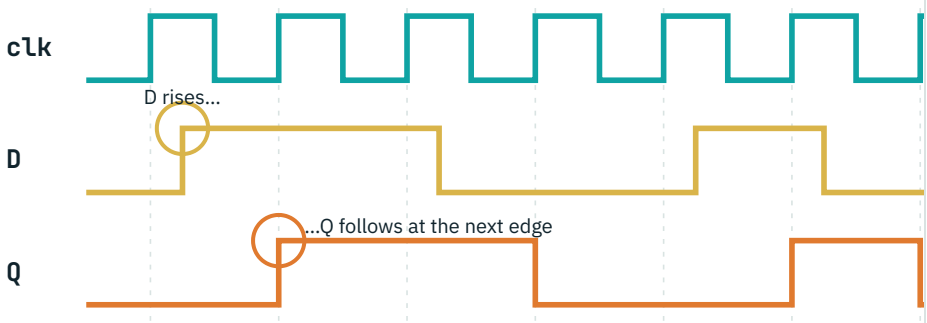
In plain words

A D flip-flop has a data input **D**, a clock input and an output **Q**. On each rising edge, Q copies D. Between edges Q holds that value, whatever D does.

Think of it like

A camera: D is the scene, the rising edge is the shutter click, Q is the photo. The scene can change between clicks — the photo doesn't.

Worked example



D rises between edges; nothing happens until the next rising edge, then Q c
it.

```
always @(posedge clk)  q <= d;   // <= : non-blocking, for
```

The word for it

A **D flip-flop**: it **samples** D on the edge and **holds** it until the next edge.

Why it matters

Seeing Q change one cycle after D is the most important waveform-reading skill in verification.

STEP 4

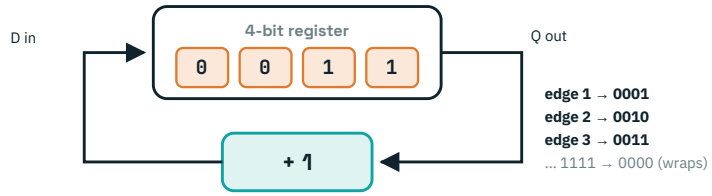
4 Registers and counters

In plain words

One flip-flop stores one bit. Four side by side, sharing one clock, store a 4-bit number: a **register**. Feed its output through a +1 adder back into its input and it counts.

Think of it like A turnstile counter that clicks up by one each time someone passes.

Worked example



Every rising edge the register captures its old value plus one: 0, 1, 2, 3 …

The word for it A **register** and a **counter**.

Why it matters Counters are everywhere (timers, addresses, FIFO pointers), and their wrap-around is a favourite place for bugs.

PAUSE & TRY **A 4-bit counter is at 15. What does it show after the next edge?**

Answer

15 = `1111` ; + 1 = `1 0000` ; four bits keep `0000` → it **wraps to 0**.

STEP 5

5 Reset: start from a known value

In plain words At power-on, flip-flops wake up holding random values. A **reset** signal forces them to a known value, usually 0.

Think of it like Zeroing a kitchen scale before you weigh anything.

✎ **Worked example**

```
always @(posedge clk)
  if (rst) q <= 0;        // reset check comes first
  else     q <= q + 1;
```

Forget the reset and simulation shows **X** (unknown) — and X spreads through every gate it touches.

Aa **The word for it**

Reset; the unknown value **X**.

! **Why it matters**

“Every register has its reset value after reset” is the first check in almost every testbench.

6 STEP 6 Setup and hold: keep still while the shutter clicks

≈ **In plain words**

D must be stable for a short time **before** the edge and a short time **after** it.

☀ **Think of it like**

If someone moves just as the shutter clicks, the photo is blurred.

✎ **Worked example**

1. Setup time: D stable before the edge.
2. Hold time: D stable after the edge.
3. If D changes inside that window, the flip-flop may capture the wrong value.

Aa **The word for it**

Setup time, **hold time**, and a **setup/hold violation**.

! **Why it matters**

Static timing analysis tools catch these in the design. In testbenches, you avoid them by driving inputs away from the clock edge.

7 STEP 7 Metastability and the two-flop synchronizer

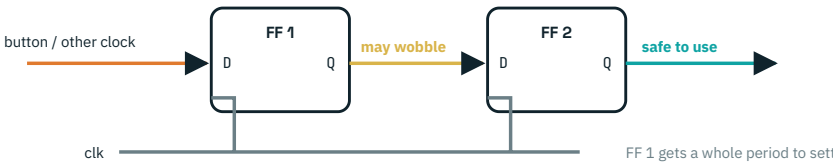
In plain words

In the worst case a violation leaves the output stuck halfway between 0 and 1 for a while. This is most likely when a signal comes from a button or another clock, because it can change at any moment.

Think of it like

A ball balanced on the top of a hill: it will roll to one side, but you don't know which or when.

Worked example



The first flip-flop may wobble, but it has a whole clock period to settle before the second one samples it.

The word for it

Metastability; the fix is a **two flip-flop synchronizer**.

Why it matters

Every signal that crosses from one clock to another must be synchronized — reviewers and checkers look for it.

STEP 8

8 State machines

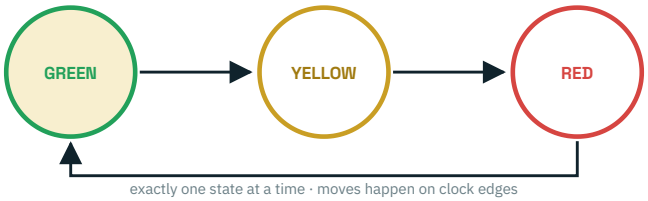
In plain words

A finite state machine (FSM) is always in exactly one **state**, and on a clock edge it moves to the next state according to its inputs.

Think of it like

A traffic light: green → yellow → red → green.

✎ **Worked example**



States are circles; arrows are transitions.

Aa **The word for it**

FSM: a **state register** (flip-flops that remember the current state) plus **next-state logic** (gates that decide where to go).

! **Why it matters**

Protocol controllers, arbiters and handshakes are all FSMs. Verifying them means covering every state and every transition.

9 STEP 9 Tracing an FSM step by step

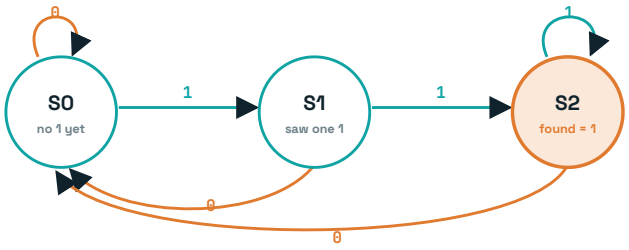
≈ **In plain words**

This FSM watches a stream of bits and raises **found** when it sees two 1s in a row.

☀ **Think of it like**

Counting how many times in a row someone says “yes”.

✎ **Worked example**



A 1 moves forward (S0 → S1 → S2); a 0 sends it back to S0 from anywhere.

input	—	0	1	1	0
state after	S0	S0	S1	**S2**	S0
found	0	0	0	**1**	0

Aa **The word for it**

A **Moore** FSM: the output depends only on the state (found = 1 in S2).

! **Why it matters**

Tracing by hand like this is how you write the expected sequence a checker compares against.

PAUSE & TRY **From S0, the inputs are 1, 1, 1. Which state are we in at the end?** Answer

1 → S1, 1 → S2, 1 → stays in **S2**; found stays on.

10 STEP 10

An FSM in Verilog: two blocks

≈ **In plain words**

Write an FSM as two blocks that match its two parts: one for the state register, one for the next-state logic.

Think of it like — A form with two sections: “where am I now” and “where do I go next”.

Worked example

```
always @(posedge clk)                    // 1 · state registe
  if (rst) state <= S0;
  else     state <= next;

always @(*)                               // 2 · next-state l
  case (state)
    S0: next = in ? S1 : S0;
    S1: next = in ? S2 : S0;
    S2: next = in ? S2 : S0;
  endcase

assign found = (state == S2);
```

The word for it — The **two-process FSM** pattern.

Why it matters — It is what reviewers expect, and it keeps the flip-flops and the logic clearly apart — which makes bugs easier to find.

11 STEP 11 — What verification checks in sequential logic

In plain words — Three things, every time: reset, every transition, and the corners.

Think of it like — A building inspector: does the door lock (reset), does every room connect correctly (transitions), what happens in a fire (corners)?

Worked example

1. **Reset:** after reset, is every register at its known value?
2. **Every transition:** for each state and each input, does it go to the right next state? (3 states × 2 inputs = 6 transitions above)
3. **Corners:** the counter wrapping at 15, two events on the same edge, a reset in the middle of a sequence.

The word for it — **Transition coverage** and **corner cases**.

Why it matters Bugs hide in transitions and corners, not on the happy path. That is where your tests should spend their time.

Going further: the reference

The same ideas at the next level: precise definitions, tables, rules and interview questions.

Key definitions

Latch
Level-sensitive: transparent while enable is high, holds while low.

Flip-flop
Edge-triggered: samples D only at the clock edge, holds Q in between.

Setup time (t_su)
How long D must be stable before the clock edge.

Hold time (t_h)
How long D must stay stable after the edge.

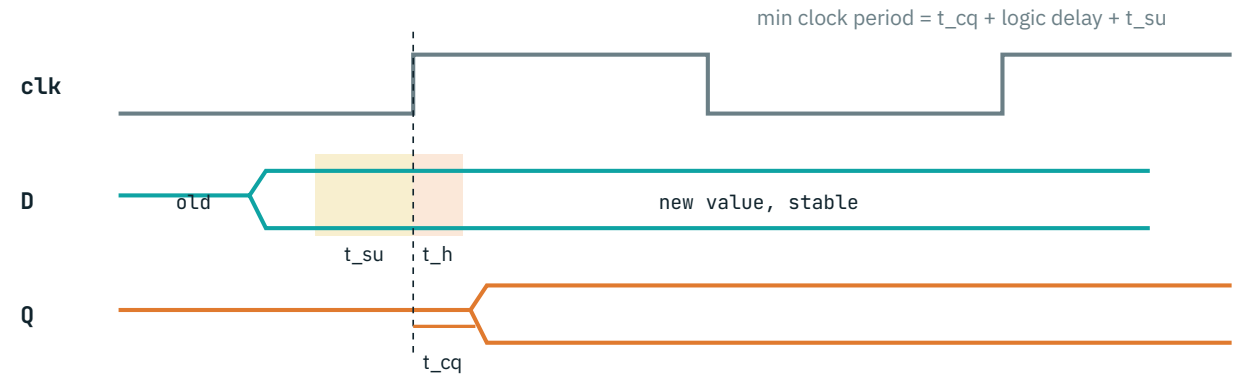
Clock-to-Q (t_cq)
Delay from the clock edge to Q changing.

Metastability
An unresolved, in-between state when setup or hold is violated; fixed with a 2-flop synchronizer.

Moore machine
Outputs depend only on the current state.

Mealy machine
Outputs depend on state and current inputs.

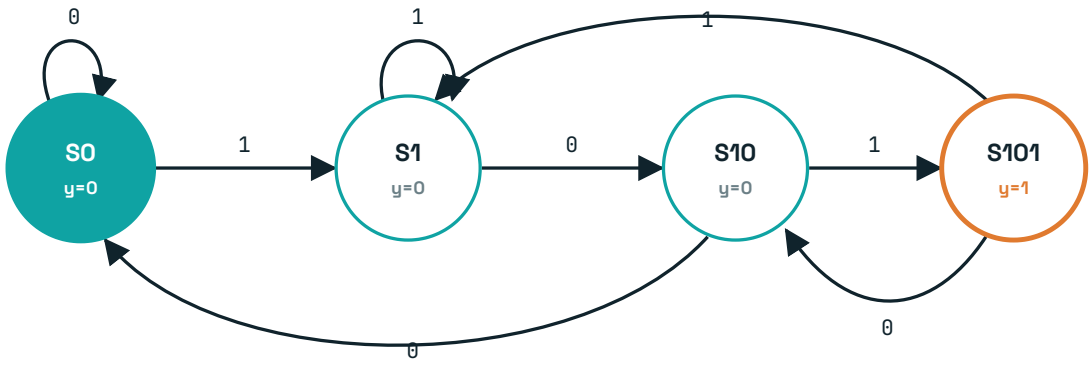
Timing words you must know



The window around the edge. D must not change inside setup + hold; Q follows t_cq later. The fastest clock is limited by t_cq + logic delay + t_su.

Term	Meaning	If violated
Setup	D stable before the edge	Wrong value or metastability — fix by slowing the clock or shortening logic
Hold	D stable after the edge	Wrong value at any clock speed — fix by adding delay on the short path
Clock skew	Clock arrives at different flops at different times	Eats into setup or hold margin
Reset recovery/removal	Setup/hold for releasing asynchronous reset	Flops leave reset in different cycles

Finite state machines



Detect 1 0 1 in a bit stream (overlapping). Each circle is a state; arrows are labelled with the input bit that takes them. The output is 1 only in S101 (Moore).

Feed bits:

```
state = S0 · y = 0 · input history: (none) · matches so far: 0
```

State	in = 0	in = 1	Output y
S0	S0	S1	0
S1	S10	S1	0
S10	S0	S101	0
S101	S10	S1	1

Resets

Style	Code	Notes
Asynchronous	`always @(posedge clk or negedge rst_n)`	Acts immediately; release must be synchronised
Synchronous	`always @(posedge clk) if (!rst_n) …`	Acts on the clock edge; needs a running clock

★ **From logic to verification**

Every checker in this course follows the same recipe: write a reference model in plain integer arithmetic, drive inputs, compare every output with `!==` (so X is a failure), and print the first few mismatches then `PASS n/n` or `FAIL`.

× **Common mistakes**

Confusing latches with flip-flops · fixing a hold violation by slowing the clock (it doesn't help) · FSMs with unreachable or missing transitions · forgetting to reset the state register.

INTERVIEW CORNER

Q. What is metastability and how do you handle it? **A.** A flop sampling during its setup/hold window can hover between 0 and 1; pass asynchronous signals through two flops to give it a cycle to resolve.

Q. Moore or Mealy — which reacts faster? **A.** Mealy: its output can change in the same cycle as the input.

Key takeaways

1. Flip-flops sample on an edge; latches are transparent on a level.
2. Setup limits clock speed; hold is independent of it.
3. An FSM is a state register plus next-state logic; draw it before you code it.

PRACTICE

Check yourself

Answer in your head, then open each card. The area test has 20 more.

What is `8'hB6` as an unsigned and as a signed number?

Unsigned 182; signed −74 (−128 + 54).

Write −6 in 4-bit two's complement, then sign-extend it to 8 bits.

`1010` → `1111_1010`.

200 + 100 in 8 bits: carry, overflow, result?

Result 44, carry 1. As signed it is −56 + 100 = 44, so no overflow.

Simplify `a + a·b`.

a (absorption).

Why does a K-map use Gray order?

So that physically adjacent cells differ in exactly one variable, which is what lets a group drop that variable.

How many input combinations does the 4-bit ALU have?

4 + 4 + 3 = 11 bits → 2,048.

Which timing violation cannot be fixed by slowing the clock?

Hold.

In the 101 detector, what state follows S101 on input 0?

S10 — the trailing 1 0 can start a new match (overlap).

REFERENCE

Interview questions

Asked in DV and RTL interviews. Try an answer out loud before opening.



Explain two's complement and why hardware uses it.

MSB has weight -2^{n-1}; one adder works for signed and unsigned; zero has one representation; negate by invert and add one.

Difference between carry and overflow?

Carry: unsigned result needs another bit. Overflow: signed result is out of range. They are independent — all four combinations occur.

Build an XOR from NAND gates.

Four NANDs: $n1 = \overline{(a \cdot b)}$, then $\overline{(\overline{(a \cdot n1)} \cdot \overline{(b \cdot n1)})}$.

What is a glitch and when does it matter?

A brief wrong output while inputs change through unequal paths; it matters on asynchronous signals like clocks and resets, not on data sampled by flops.

Latch vs flip-flop?

Latch is level-sensitive (transparent when enabled); flip-flop is edge-triggered.

Why do we need synchronizers?

To reduce the chance that metastability from an asynchronous input propagates into the design.

Design a counter that counts 0 → 9 and wraps.

`always @(posedge clk) cnt <= (cnt == 9) ? 0 : cnt + 1;` with a 4-bit register and reset.

How would you test a 32-bit adder?

Not exhaustively (2^{64} cases): corner values (0, max, min, −1), carry chains, random values, and a reference model computed one bit wider.

REFERENCE

Glossary

Every term used on this page, A to Z.

A a

ALU
Block performing the operation selected by an opcode.

Bit / byte / nibble
1 / 8 / 4 binary digits.

Carry
Extra bit from an unsigned addition.

Clock-to-Q
Edge-to-output delay of a flip-flop.

Comparator
Tells equal / less / greater.

De Morgan
Inverting an AND gives an OR of inverses, and vice versa.

Decoder
n-bit code → one-hot outputs.

Don’t care
Unused input combination whose output may be chosen freely.

Flip-flop
Edge-triggered storage element.

Full adder
Adds a, b and carry-in.

Gray code
Sequence where neighbours differ by one bit.

Hold time
Stable time required after the clock edge.

Karnaugh map
Grid for simplifying Boolean functions.

Latch
Level-sensitive storage element.

Metastability
Unresolved state after a timing violation.

Minterm
Product term true for exactly one input combination.

Moore / Mealy
FSM outputs from state only / from state and inputs.

MSB / LSB
Most / least significant bit.

Multiplexer
Selects one of several inputs.

Overflow
Signed result out of range.

Setup time
Stable time required before the clock edge.

Sign extension
Copying the MSB when widening a signed value.

Truth table
All inputs with their outputs.

Two’s complement
Signed representation with MSB weight -2^{n-1}.

REFERENCE

FAQ & troubleshooting

The problems learners hit most in this area, and the fix for each.



You see	Why	Fix
Overflow flag looks random in the lab	Checking signed overflow with the add rule for SUB	Use: different signs in, and the result's sign differs from a
Mismatch only for op = SUB	Borrow vs carry confusion	For SUB, carry means borrow: 1 when a < b (unsigned)
`iverilog: command not found`	Not installed or not on PATH	Re-run the install step; reopen the terminal
Simulation prints X	Inputs never assigned before the first print	Drive every input before `#1 $display`
Testbench reports PASS with 0 checks	Loop bounds or counters wrong	Print the number of checks; require it to equal 2,048

REFERENCE

Further reading

Go deeper when you are ready.



Resource	Why read it
Harris & Harris, *Digital Design and Computer Architecture*	Number systems, Boolean algebra and sequential logic with HDL examples
M. Morris Mano, *Digital Design*	The classic reference for K-maps and combinational design
Charles Petzold, *Code*	A readable story of how bits become computers — good for newcomers
Icarus Verilog documentation	Command-line options for the lab

HANDS-ON LAB · ABOUT 2 HOURS

Lab 01 · Build and self-check a 4-bit ALU

You get `alu4.v` (eight operations, one planted bug), `tb_alu4.v` with an exhaustive stimulus loop and a reference model with TODOs, and a Makefile.

1. Run `make`: with the TODOs empty you see mismatches for the operations the model doesn't know yet.
2. Fill in TODO 1–3 so the model covers all eight operations and their carry and overflow flags.
3. Run again: the remaining mismatches are the design's bug — they share one operation and one flag.
4. Fix `alu4.v` and re-run.

Done when you see `PASS: 2048/2048`.