Combinational Logic & SystemVerilog — First Contact

Rodolfo Azevedo

Institute of Computing, University of Campinas (UNICAMP), Brazil

rodolfo.azevedo@unicamp.br

http://www.ic.unicamp.br/~rodolfo/mo801

Goal of this class

Module 1, Class 1: from logic gates to SystemVerilog — and a first circuit running on real hardware.

> This class assumes you have seen logic gates before. The goal is not to re-teach them — it is to connect what you already know to the language (SystemVerilog) and tools (OSS CAD Suite) we will use for the rest of the course.

At the end of this class, you should be able to:

- Describe the behavior of logic gates (AND, OR, NOT, XOR, NAND, NOR) using truth tables and Boolean equations.
- Write synthesizable SystemVerilog modules with `assign`, port declarations, and module instantiation.
- Apply Boolean algebra and De Morgan's theorem to simplify combinational logic.
- Use vectors, concatenation, and width casting in SystemVerilog to handle multi-bit signals.
- Synthesize a combinational circuit and load it onto the Tang Nano 9K FPGA.

Why start here?

The end goal of this course is to run a keyword-spotting AI on a processor *you built from scratch*. That processor will be tens of thousands of gates. Every one of them obeys the same rules as the AND gate on this slide.

Understanding hardware at the gate level is what separates someone who *uses* an FPGA from someone who *builds* what runs on it.

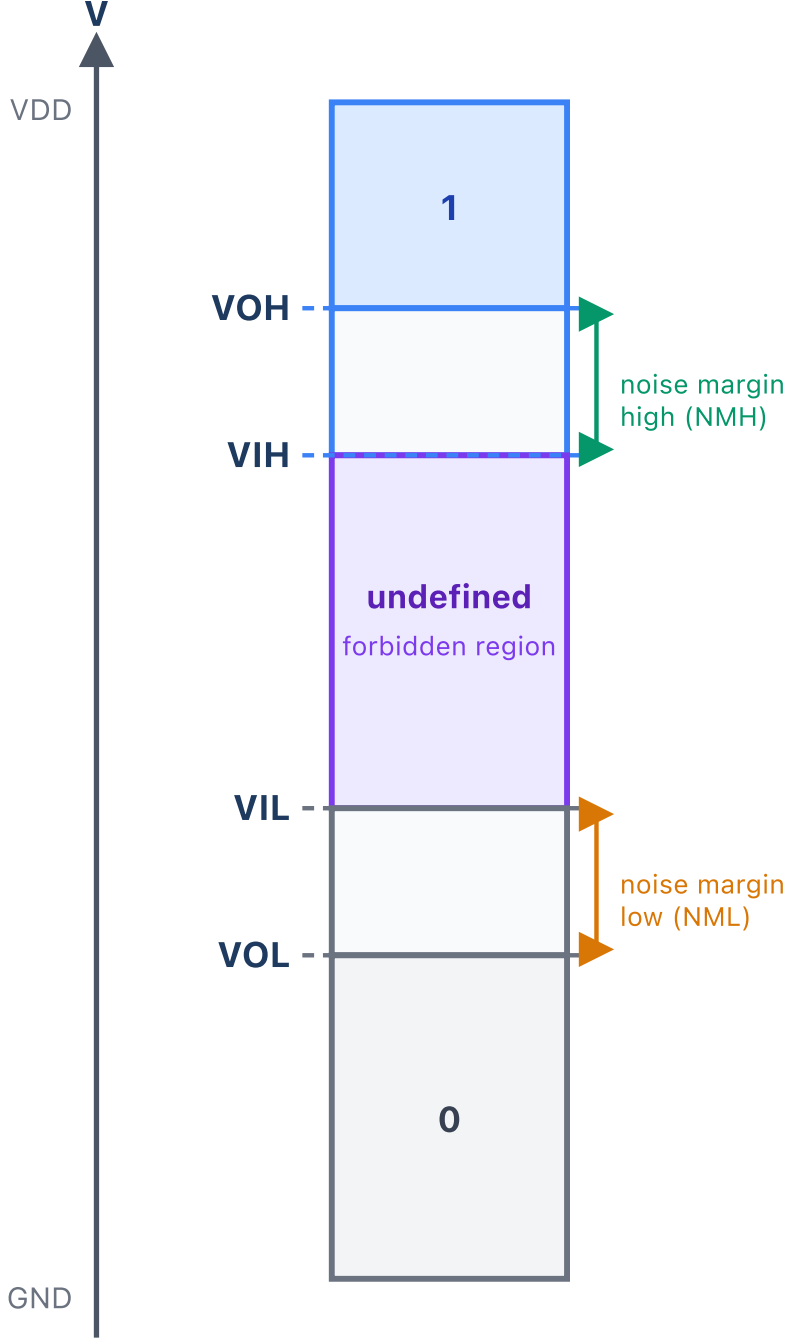
The digital abstraction

Real circuits deal in voltages. We simplify them to two values:

- **0** → low voltage (GND, typically 0 V)
- **1** → high voltage (V_{DD}, typically 1.8–3.3 V in modern devices)

This works because logic families define **noise margins**: a range of voltages around each level that are still reliably interpreted as 0 or 1. As long as noise stays within the margins, the digital abstraction holds — we never have to think about voltages again.

The Gowin GW1NR-9C on the Tang Nano 9K is a 3.3 V device; its I/O banks operate at 3.3 V or 1.8 V depending on pin configuration.

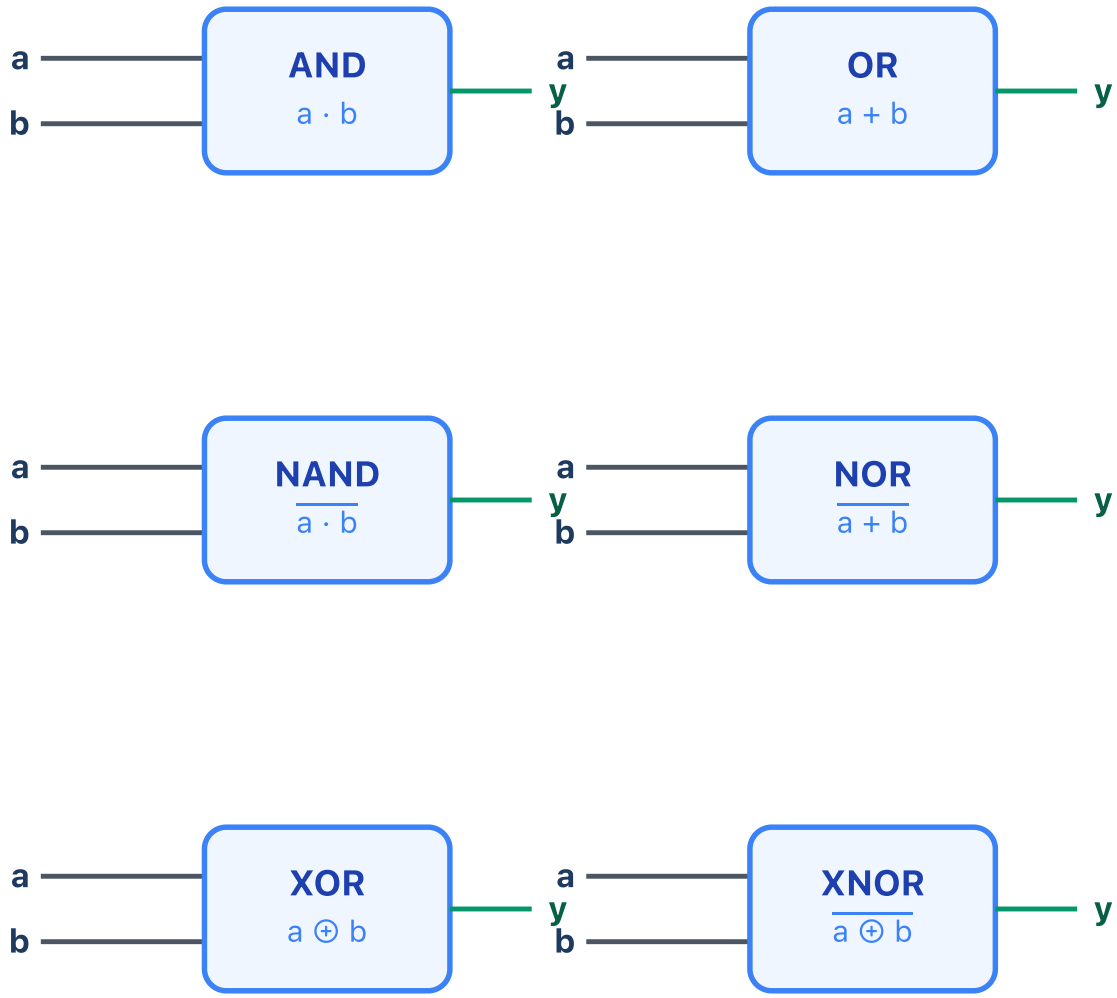


Noise margins: valid-1 / undefined / valid-0 voltage zones

Logic gates — the vocabulary

Gate	Symbol (text)	Boolean	Behaviour
NOT	`~a`	$\bar{a}$	Inverts the input
AND	`a & b`	$a \cdot b$	1 only when **both** inputs are 1
OR	`a | b`	$a + b$	1 when **at least one** input is 1
NAND	`~(a & b)`	$\overline{a \cdot b}$	Inverted AND
NOR	`~(a | b)`	$\overline{a + b}$	Inverted OR
XOR	`a ^ b`	$a \oplus b$	1 when inputs **differ**
XNOR	`~(a ^ b)`	$\overline{a \oplus b}$	1 when inputs are **equal**

These operators appear in SystemVerilog exactly as shown in the **Symbol (text)** column — so the language and the circuit description are already the same notation.



2-input gates

Truth tables and Boolean equations — example

Seat-belt alert: a car sounds an alarm (S) if the ignition is on (K) and either the driver (D) or passenger (P) is not wearing a belt.

$$S = K \cdot (\bar{D} + \bar{P})$$

K	D	P	S
0	×	×	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

This is a combinational circuit: the output depends **only** on the current inputs, with no memory of the past.

From equation to SystemVerilog — immediately

```
module seatbelt_alert (
    input  logic k,   // ignition
    input  logic d,   // driver belt
    input  logic p,   // passenger belt
    output logic s    // alarm
);
    assign s = k & (~d | ~p);
endmodule
```

- `module` / `endmodule` — the building block of a SystemVerilog design
- `input` / `output` — the ports of the module, visible from the outside
- `assign s = ...` — **continuous assignment**
  - `s` is permanently wired to this expression
  - Change any input, `s` updates instantly (in the model) or within nanoseconds (in real hardware)
- The SV operators (`&` , `|` , `~`) are exactly the Boolean operators from the previous slide.
- `logic` — the type for almost everything in this course
  - A 4-state type: `0` , `1` , `x` (unknown), `z` (undriven)
  - For synthesizable logic, you will see only `0` and `1`

Boolean algebra — the rules you already know

These identities let you simplify circuits (fewer gates = smaller, faster hardware):

Identity	AND form	OR form
Identity	$a \cdot 1 = a$	$a + 0 = a$
Null	$a \cdot 0 = 0$	$a + 1 = 1$
Idempotent	$a \cdot a = a$	$a + a = a$
Complement	$a \cdot \bar{a} = 0$	$a + \bar{a} = 1$
De Morgan	$\overline{a \cdot b} = \bar{a} + \bar{b}$	$\overline{a + b} = \bar{a} \cdot \bar{b}$
Distributive	$a(b + c) = ab + ac$	$a + bc = (a + b)(a + c)$

De Morgan is the most important: it explains why NAND and NOR are universal gates (any function can be built from NAND alone), and why we write `~(a & b)` instead of `~a | ~b` — they are the same circuit.

Vectors: multi-bit signals

A single `logic` is one wire. A **vector** is a bundle of wires:

```
logic [3:0] a;    // 4-bit vector: a[3] (MSB) ... a[0] (LSB)
logic [7:0] b;    // 8-bit vector
logic [0:3] r;    // r[0] is MSB — less common, avoid mixing conventions
```

Operators apply **bitwise** across vectors:

```
logic [3:0] x, y, z;
assign z = x & y;    // z[3]=x[3]&y[3], z[2]=x[2]&y[2], ...
assign z = x ^ y;    // bitwise XOR: 1 wherever x and y differ
assign z = ~x;       // bitwise NOT: inverts every bit
```

Reduction operators collapse a vector to one bit:

```
assign all_ones = &x;   // 1 only if every bit of x is 1
assign any_one  = |x;   // 1 if at least one bit of x is 1
assign parity   = ^x;   // XOR of all bits: 1 if an odd number of 1s
```

Number literals

```
4'b1010         // 4-bit binary = decimal 10
8'hFF           // 8-bit hex = 255
12'd2217        // 12-bit decimal
32'hDEAD_BEEF   // underscores for readability, ignored by the tool
```

- Default base is decimal if no prefix is given: `8'd255` = `255` = `8'b11111111`
  - `b` = binary, `h` = hex, `d` = decimal, `o` = octal
- Width is optional: `8'hFF` = `hFF` = `255` = `8'b11111111`
- `'0` and `'1` are width-agnostic: `'0` = all-zeros, `'1` = all-ones, whatever the context width is.
  - Notice the difference between `0` (decimal 0) and `'0` (all-zeros, width determined by context), `1` (decimal 1) and `'1` (all-ones, width determined by context)
- `x` and `z` are also width-agnostic: `'x` = all unknown, `'z` = all undriven

Concatenation

`{a, b}` joins two vectors into one — essential for building wider signals:

```
logic [3:0] hi, lo;
logic [7:0] word;
assign word = {hi, lo};              // hi is bits [7:4], lo is bits [3:0]
assign word = {4'b0000, lo};         // zero-extend lo to 8 bits
assign word = {{4{lo[3]}}, lo};      // sign-extend lo (replicate MSB 4 times)
```

The `{N{expr}}` replication syntax is used constantly for sign extension — you will write it dozens of times in Project 1's immediate-reconstruction logic.

> **Sign extension** is the process of increasing the width of a binary number while preserving its value, typically by replicating the most significant bit.

Width casting: N'(expr)

When expressions mix different bit-widths, SystemVerilog silently zero-extends or truncates the shorter operand. This is a frequent source of subtle bugs.

The fix: use `N'(expr)` to make the intended width explicit:

```
logic [7:0] a;
logic [3:0] b;
logic [7:0] sum;
assign sum = a + 8'(b);       // zero-extend b to 8 bits explicitly

localparam int HALF = 127;
logic [7:0] cnt;
// Without casting, (HALF - 1) is a 32-bit constant; comparison works but
// generates a width-mismatch warning. Cast makes intent clear:
if (cnt == 8'(HALF - 1)) ...
```

Key rules:

- Use `N'(expr)` whenever the right-hand side has a different bit-width from the target — especially with `localparam` arithmetic.
- `'0` extends to all-zeros matching the context width; `'1` extends to all-ones.
- Explicit casts make both code and synthesis reports easier to read, and prevent Verilator `-Wall` warnings that hide real bugs.

Modules and ports — the building block

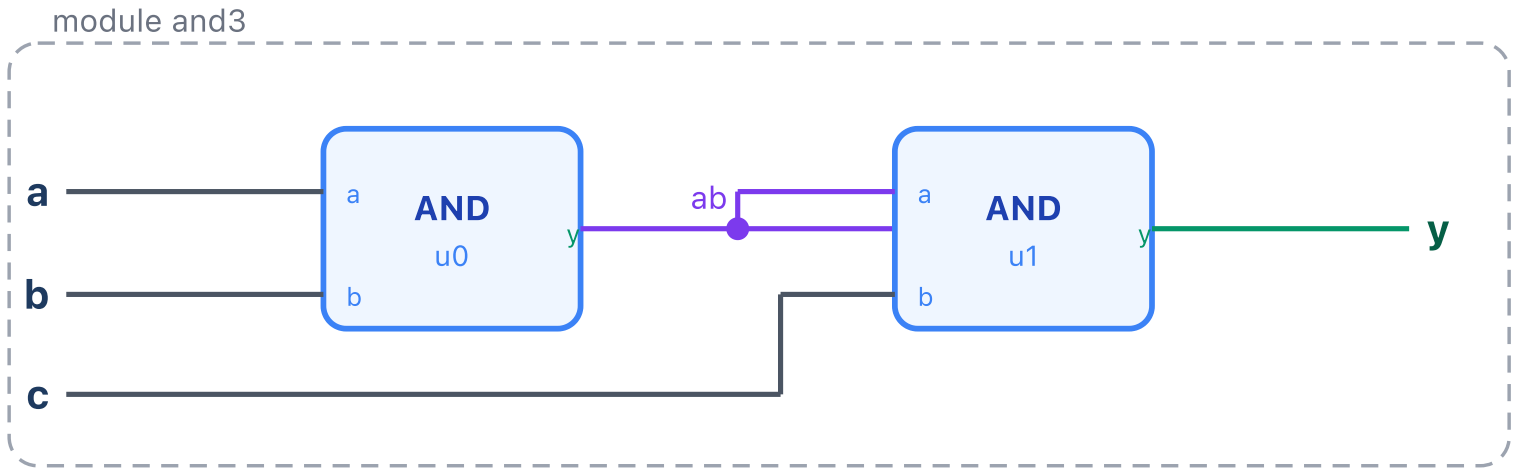
```
module and_gate (
    input  logic a,
    input  logic b,
    output logic y
);
    assign y = a & b;
endmodule
```

A module is a **black box**: visible from the outside only through its ports. Inside can be any logic; the rest of the design does not care.

Instantiation — using a module inside another

```
module and3 (input logic a, b, c, output logic y);
    logic ab;
    and_gate u0 (.a(a), .b(b), .y(ab));   // named port connections
    and_gate u1 (.a(ab), .b(c), .y(y));
endmodule
```

- `.port(signal)` — always prefer named connections over positional; self-documenting and survives port-order changes.
- `logic ab` — an internal signal connecting the two instances, visible only inside `and3`.



A more interesting example: majority function

Output is 1 when at least 2 of 3 inputs are 1. The Boolean equation (from SOP):

$$M = ab + ac + bc$$

```
module majority (
    input  logic a, b, c,
    output logic m
);
    assign m = (a & b) | (a & c) | (b & c);
endmodule
```

Equivalently with De Morgan (using only NAND):

```
    assign m = ~(~(a & b) & ~(a & c) & ~(b & c));
```

Both descriptions produce the same circuit — the synthesis tool picks the implementation.

Toolchain express — before touching the board

You need four tools from the **OSS CAD Suite** (one installer, all platforms):

```
# Install OSS CAD Suite (Linux/Mac — adjust path for your OS)
# Download from https://github.com/YosysHQ/oss-cad-suite-build/releases
tar -xf oss-cad-suite-*.tgz
source oss-cad-suite/environment   # add tools to PATH

# Verify
yosys --version          # synthesis
nextpnr-himbaechel --version  # place & route
openFPGALoader --version      # bitstream upload
verilator --version    # simulation (later classes)
```

Minimal project layout:

```
project/
  src/top.sv      ← your SystemVerilog
  pins.cst        ← pin constraints for the Tang Nano 9K
  Makefile        ← build rules
```

To synthesize and load: `make load` (full Makefile in M01A06; for now, use the Lab 2 starter template).

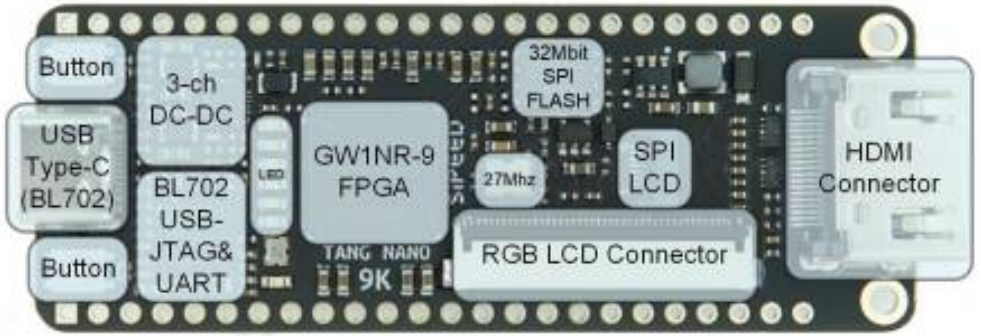
> **Lab 2** walks through the full setup step by step — complete it before the next class so you can do Lab 3 on the board.

The Tang Nano 9K: your hardware target

- **FPGA**: Gowin GW1NR-9C — 8,640 LUTs, 6,480 FFs, 468 Kbit BRAM, 64 Mbit external PSRAM.

> - **LUT** = lookup table, implements any combinational function of N inputs
> - **FF** = D flip-flop, registered state
> - **BRAM** = block RAM, on-chip memory, inferred from `logic` arrays
>
> See M01A06 for the full primitives table.

- **On board**: 27 MHz oscillator, 6 user LEDs (active-low), 2 push buttons, UART via USB.



Your job today

- Synthesize a combinational circuit whose inputs are the 2 buttons and whose outputs are some of the 6 LEDs.

```
// pins.cst (excerpt — Tang Nano 9K)
IO_LOC "btn[0]"  3;    // push button 0
IO_LOC "btn[1]"  4;    // push button 1
IO_LOC "led[0]" 10;    // LED 0 (active-low: 0 = ON)
IO_LOC "led[1]" 11;
IO_LOC "led[2]" 13;
```

Board exercise: logic functions on buttons and LEDs

```
module btn_logic (
    input  logic [1:0] btn,   // active-low: 0 when pressed
    output logic [5:0] led    // active-low: 0 = ON
);
    logic b0, b1;
    assign b0 = ~btn[0];   // invert: b0=1 when button 0 pressed
    assign b1 = ~btn[1];

    assign led[0] = ~(b0 & b1);    // LED 0 on when BOTH pressed
    assign led[1] = ~(b0 | b1);    // LED 1 on when EITHER pressed
    assign led[2] = ~(b0 ^ b1);    // LED 2 on when they DIFFER
    assign led[3] = ~b0;           // LED 3 mirrors button 0
    assign led[4] = ~b1;           // LED 4 mirrors button 1
    assign led[5] = ~(b0 | b1);    // LED 5 = OR again
endmodule
```

- Notice that the LED logic maps directly to the Boolean expressions from the first slides.
- Synthesize with `make load` (Yosys → nextpnr → openFPGALoader) and press the buttons. You are observing real combinational hardware — no CPU, no code.

Next class

Combinational Building Blocks: multiplexers, decoders, comparators — and `always_comb` and `case` in SystemVerilog, the tools for describing more complex combinational logic cleanly.