Combinational Building Blocks

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 2: multiplexers, decoders, comparators — and `always_comb`.

> In A01 we described single gates with `assign`. Today we compose those gates into standard building blocks and introduce `always_comb` — the construct for describing more complex combinational logic in a readable way.

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

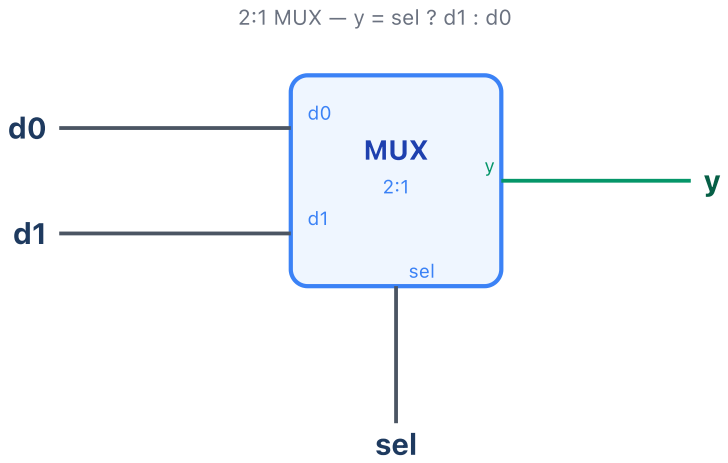
- Implement multiplexers, decoders, and comparators in SystemVerilog using `always_comb` and `case`.
- Explain the latch inference trap and write latch-free combinational blocks with complete case coverage.
- Compose standard combinational building blocks using both structural and behavioral SystemVerilog.
- Use parameterized modules to build reusable, width-generic components.

The multiplexer (MUX)

A **2:1 MUX** selects one of two data inputs based on a select signal:

$$Y = \bar{S} \cdot D_0 + S \cdot D_1$$

```
module mux2 #(parameter WIDTH = 1) (
    input  logic [WIDTH-1:0] d0, d1,
    input  logic             sel,
    output logic [WIDTH-1:0] y
);
    assign y = sel ? d1 : d0;
endmodule
```

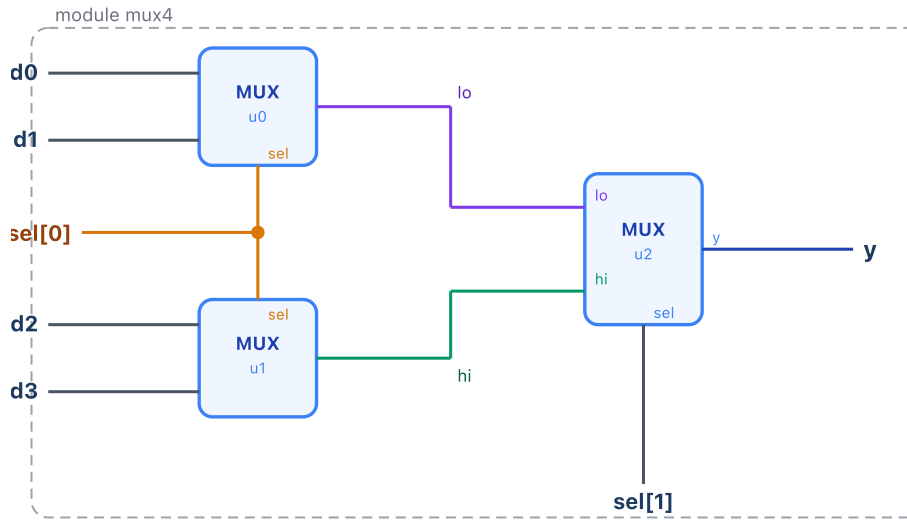


The ternary `? :` operator is shorthand for a 2:1 MUX — you will see it constantly in datapath descriptions.

4:1 MUX from 2:1 MUXes

A 4:1 MUX selects among four inputs using two select bits. Build it from three 2:1 MUXes:

```
module mux4 #(parameter WIDTH = 8) (
    input  logic [WIDTH-1:0] d0, d1, d2, d3,
    input  logic [1:0]       sel,
    output logic [WIDTH-1:0] y
);
    logic [WIDTH-1:0] lo, hi;
    mux2 #(WIDTH) u0 (.d0(d0), .d1(d1), .sel(sel[0]), .y(lo));
    mux2 #(WIDTH) u1 (.d0(d2), .d1(d3), .sel(sel[0]), .y(hi));
    mux2 #(WIDTH) u2 (.d0(lo), .d1(hi), .sel(sel[1]), .y(y));
endmodule
```



MUXes as universal logic

Any Boolean function of n variables can be implemented with a 2^n:1 MUX whose data inputs are the truth-table entries. This is exactly how LUTs (Look-Up Tables) on FPGAs work — a 6-input LUT is a 64:1 MUX.

`always_comb` — describing combinational logic procedurally

`assign` is fine for simple expressions. For more complex logic (if/else, case), use `always_comb`:

```
module mux4_comb #(parameter WIDTH = 8) (
    input  logic [WIDTH-1:0] d0, d1, d2, d3,
    input  logic [1:0]       sel,
    output logic [WIDTH-1:0] y
);
    always_comb begin
        case (sel)
            2'b00: y = d0;
            2'b01: y = d1;
            2'b10: y = d2;
            2'b11: y = d3;
            default: y = d0;
        endcase
    end
endmodule
```

- `always_comb` re-evaluates whenever **any** signal it reads changes — exactly the semantics of combinational hardware.
- Tools verify that every output is assigned on every path. A missing `default` that leaves `y` undriven on some input combination would infer a latch.

> **Add `default` to every `case`.**

Arithmetic vs. bitwise operators

Two categories of operators; confusing them is a frequent source of bugs:

Category	Operators	Operand(s)	Result
Bitwise	`&` `|` `^` `~`	Two vectors	Vector, same width — each bit independently
Reduction	`&` `|` `^` (prefix)	One vector	1-bit result
Arithmetic	`+` `-` `*`	Two vectors	Sum/diff (wraps on overflow)
Logical	`&&` `||` `!`	Any	1-bit boolean
Comparison	`==` `!=` `<` `>`	Two values	1-bit boolean

```
logic [7:0] a = 8'hFF, b = 8'h01;
assign and_bits = a & b;     // 8'h01  — bitwise AND per bit
assign sum      = a + b;     // 8'h00  — arithmetic (255+1 wraps to 0)
assign both_nz  = a && b;    // 1      — logical: "are both nonzero?"
assign any_bit  = |a;        // 1      — reduction: "is any bit of a set?"
assign eq       = (a == b);  // 0      — comparison: 1-bit result
```

Rule: use `&&` / `||` inside `if` conditions and assertions; use `&` / `|` for bit manipulation of vectors. Mixing them compiles but produces wrong hardware.

The accidental latch — The most common beginner bug in SV

```
// WRONG — infers a latch because y is not assigned when sel==2'b10 or 2'b11
always_comb begin
    case (sel)
        2'b00: y = d0;
        2'b01: y = d1;
        // sel==2'b10 and 2'b11: y is undriven → latch inferred
    endcase
end
```

Fix 1 — add `default`:

```
        default: y = d0;   // or any defined value
```

Fix 2 — assign a default before the `case` (idiomatic for complex logic):

```
always_comb begin
    y = d0;           // safe default; case overrides specific values
    case (sel)
        2'b01: y = d1;
        2'b10: y = d2;
        2'b11: y = d3;
    endcase
end
```

The second pattern is especially useful when most cases share the same output — you only enumerate the exceptions.

`begin` / `end` — block delimiters

In an `always_comb` (or any procedural block), a `case` arm or `if` branch executes **one statement** unless you group with `begin` / `end`:

```
always_comb begin
    if (sel)
        y = d1;         // single statement: no begin/end needed
    else begin
        y = d0;         // multiple statements require begin/end
        carry = 1'b0;
    end
end
```

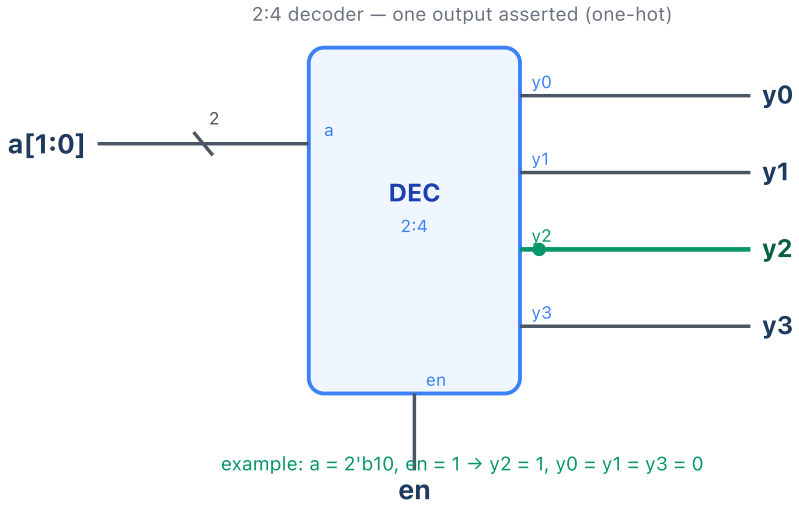
Rules of thumb:

- The `always_*` block itself always uses `begin` / `end`.
- Inside `case` arms: add `begin` / `end` whenever the arm has more than one statement.
- When in doubt, always use `begin` / `end` — it costs nothing and avoids the classic dangling-else bug.

Decoder

An n-**to-**2^n **decoder** asserts exactly one output for each input combination:

```
module decoder2to4 (
    input  logic [1:0] a,
    input  logic       en,
    output logic [3:0] y
);
    always_comb begin
        y = 4'b0000;        // default: all outputs low
        if (en)
            y = 4'b0001 << a;   // shift a one-hot '1' into position a
    end
endmodule
```



Priority encoder

A **priority encoder** outputs the binary index of the highest-priority asserted input. Convention, lower index = higher priority:

```
module prio_enc4 (
    input  logic [3:0] req,     // request lines; req[0] = highest priority
    output logic [1:0] grant,   // binary index of highest-priority request
    output logic       valid    // 1 if at least one request is active
);
    always_comb begin
        valid = |req;
        casez (req)
            4'b???1: grant = 2'd0;
            4'b??10: grant = 2'd1;
            4'b?100: grant = 2'd2;
            4'b1000: grant = 2'd3;
            default: grant = 2'd0;
        endcase
    end
endmodule
```

- `|req` is a reduction OR — 1 if any bit is set.
- `casez` evaluates top-to-bottom; the first matching arm wins, encoding the priority order.
- A priority encoder is the inverse of a decoder: decoder (index→one-hot), encoder (one-hot→index).

Where this appears in the course: the bus arbiter in Project 2 uses priority encoding to resolve simultaneous requests.

Comparator

Comparing two n-bit numbers is a standard building block for branches (`BEQ`, `BLT` in RISC-V):

```
module comparator #(parameter WIDTH = 32) (
    input  logic signed [WIDTH-1:0] a, b,
    output logic                    eq, lt, gt
);
    assign eq = (a == b);
    assign lt = (a <  b);    // signed comparison because of 'signed' declaration
    assign gt = (a >  b);
endmodule
```

- The `signed` keyword tells the tool to treat the vector as two's complement.
- Without `signed`, `8'hFF` is *greater* than `8'h01` (255 > 1 unsigned), but `8'hFF` is *less than* `8'h01` (–1 < 1 signed).
- `==` produces a single-bit `logic` — exactly what you need for a branch condition.

Instruction decode: a plain `case`

RISC-V opcode bits `instr[6:0]` select the instruction format — each value here is fully specified, so a plain `case` is all you need:

```
always_comb begin
    opcode_type = UNKNOWN;
    case (instr[6:0])
        7'b0110011: opcode_type = R_TYPE;
        7'b0010011: opcode_type = I_TYPE;
        7'b0000011: opcode_type = LOAD;
        7'b0100011: opcode_type = STORE;
        default:    opcode_type = UNKNOWN;
    endcase
end
```

- No don't-care bits here — every opcode is matched exactly.
- Always add a `default`.

`casez` and `casex` — wildcard matching

Now suppose the decoder only needs to know **"is this an ALU op?"** — it doesn't need to distinguish R-type (register operand) from I-type (immediate operand), and those two opcodes differ only in bit 5:

```
always_comb begin
    opcode_type = UNKNOWN;
    casez (instr[6:0])
        7'b0?10011: opcode_type = ALU_OP;  // R-type or I-type — bit 5 don't-care
        7'b0000011: opcode_type = LOAD;
        7'b0100011: opcode_type = STORE;
        default:    opcode_type = UNKNOWN;
    endcase
end
```

- `casez`: `?` and `z` bits in the case items are don't-cares.
- `casex`: `x` bits are also don't-cares — avoid in synthesizable code, as `x` has simulation-only semantics and can mask real bugs.
- **Prefer `casez` over `casex`** when wildcard bits are needed — don't let `x` hide real bugs. Always add a `default`.

One-hot select: a minimal bus

A **one-hot select** picks one of N sources based on a one-hot enable vector — the combinational core of a simple bus:

```
module bus_mux #(parameter N = 4, WIDTH = 32) (
    input  logic [WIDTH-1:0] data  [0:N-1],  // N data sources
    input  logic [N-1:0]     sel,             // one-hot: exactly one bit set
    output logic [WIDTH-1:0] y
);
    integer i;
    always_comb begin
        y = '0;
        for (i = 0; i < N; i++)
            if (sel[i]) y = data[i];
    end
endmodule
```

- One-hot `sel` means at most one `if` branch fires — the `for` loop collapses to a chain of `?:`.
- Synthesis generates a tree of 2:1 MUXes, one per source.
- In Project 2's bus, the address decoder produces a one-hot `sel` vector; this MUX routes the response back to the CPU.

The `for` loop inside `always_comb` is *unrolled at elaboration time* — it generates N parallel `if` statements, not a sequential loop in hardware.

In-class exercise — combinational blocks

Exercise 1 (5 min): write a 3-to-8 decoder with enable, using `always_comb` and the shift pattern (`8'b1 << a`). Test mentally: if `en=1` and `a=3'd5`, which output bit should be 1?

Exercise 2 (10 min): the `casez` below has a subtle error. Find it and fix it.

```
module prio_wrong (
    input  logic [3:0] req,
    output logic [1:0] grant
);
    always_comb
        casez (req)
            4'b1???: grant = 2'd3;   // req[3] highest
            4'b?1??: grant = 2'd2;
            4'b??1?: grant = 2'd1;
            4'b???1: grant = 2'd0;
        endcase
endmodule
```

> **Hint**: what happens when `req = 4'b0000` ? What does the synthesizer infer?

Exercise 3 (discussion): rewrite the priority encoder using the two-process pattern (default assignment before `casez`) to eliminate the latch. Which style do you prefer for readability?

Lab 3 out — decoder on the Tang Nano 9K

Goal: implement a 2-to-4 decoder where inputs are the 2 push buttons and outputs drive 4 of the 6 LEDs (active-low).

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

    always_comb begin
        led = 6'b111111;       // all LEDs off (active-low: 1 = off)
        case (sel)
            2'b01: led[0] = 0; // btn0 only: LED 0 on
            2'b10: led[1] = 0; // btn1 only: LED 1 on
            2'b11: led[2] = 0; // both: LED 2 on
            default: ;         // 2'b00: nothing pressed, all off
        endcase
    end
endmodule
```

Extend for full credit: add a free-running 2-bit counter so the decoder cycles through all four outputs automatically when no button is pressed. Synthesize, load, and verify each LED lights exclusively.

Next class

Sequential Logic & SystemVerilog: flip-flops, registers, counters — and `always_ff` — the building blocks that add memory to circuits.