Intro to Verilog

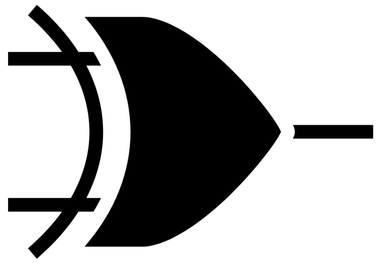
Kosi Nwabueze <kosinw@mit.edu>

Spring 2026

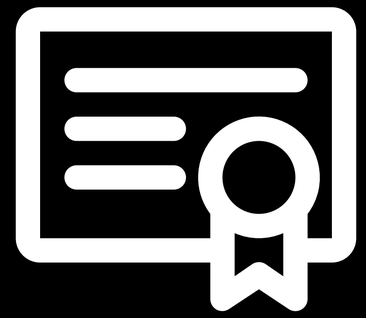
What is Verilog?

Verilog is a *hardware description language* — it describes digital circuits, not software programs.

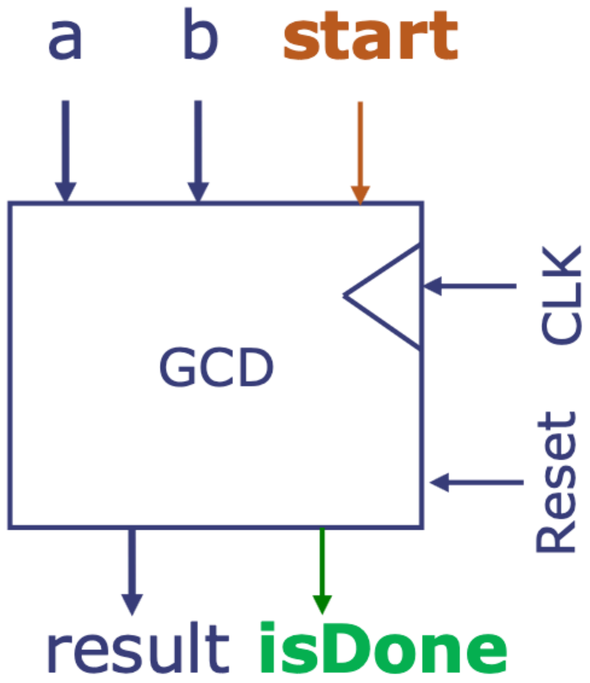
When you write **Verilog**, you are specifying digital logic behavior: logic gates, flip-flops, and latches.



Verilog (synthesizable)

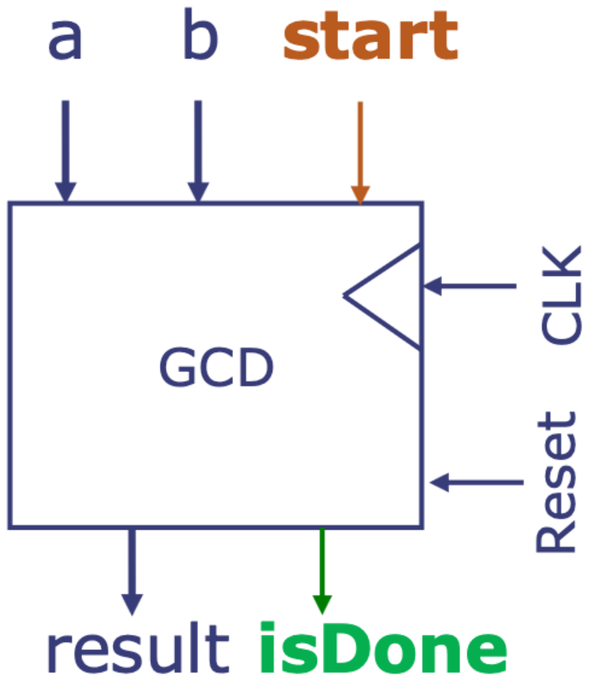


Verilog (simulatable)




```
1  module GCD (
2      input  logic          clk, rst, start,
3      input  logic  [31:0] a, b,
4      output logic  [31:0] result,
5      output logic          isDone
6  );
7      // TODO: implement me
8  endmodule
```

Modules are the unit of ***abstraction*** in a circuit.

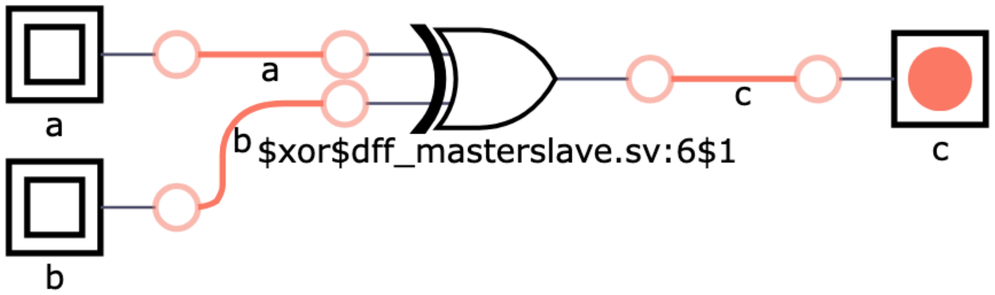



```
1    module GCD (
2        input  logic          clk, rst, start,
3        input  logic  [31:0] a, b,
4        output logic  [31:0] result,
5        output logic          isDone
6    );
7        // TODO: implement me
8    endmodule
```

Input / output ports

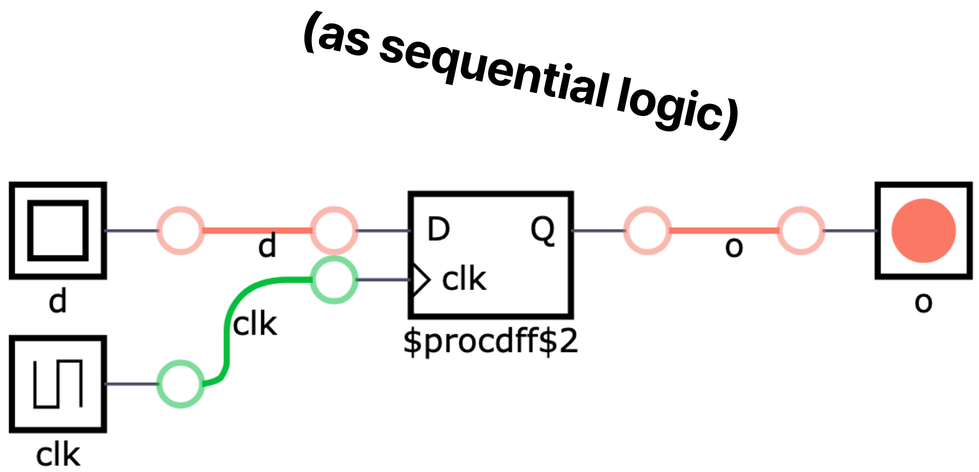
Inputs ports drive the circuit; **output ports** carry results.

(as combinational logic)




```
1  logic a;
2  logic b;
3
4  assign c = a + b;
```

Signals are declared using `logic`.
The **usage** determines its synthesis **behavior**.




```
1  logic d;
2  logic clk;
3  logic o;
4
5  always_ff @(posedge clk) begin
6    o <= d;
7  end
```

Signals are declared using `logic`.
The **usage** determines its synthesis **behavior**.

```
1   logic a;                 // create one bit variable
2   logic [3:0] b;          // create four bit variable
3   logic [11:0] c,d,e;  // create several 12 bit variables
```

Signals can be declared with flexible size in hardware.
Sizing is specified from leftmost bit to rightmost.

```
1  logic a;
2  wire b;
3  reg c;
```

Signals may also be declared with `wire` and `reg`.
For most cases, prefer using `logic`.

```
1  logic [7:0] in1, in2;
2  logic [7:0] a, b;
3
4  assign a = in1 ^ in2;
5
6  always_comb begin
7    b = in1 + in2;
8  end
```

Signals assigned to within combinational logic use ***blocking assignment*** (=) w/ `assign` or `always_comb`.

```
1  logic [7:0] a;
2
3  always_ff @(posedge clk) begin
4    a <= b + c;
5  end
```

Signals assigned to within sequential logic use ***non-blocking assignment*** (<=) in an `always_ff`.

```
1    logic [7:0] a;
2
3    always_ff @(posedge clk) begin
4      a <= b + c;
5    end
```

sensitivity list

Sensitivity lists describes when the procedural block fires. The block fires when at least one of the signals changes (posedge/negedge).

(below are bad examples)

```
always_ff @(posedge clk) begin
  a = b;
  c = a;
end

always_ff @(posedge clk) begin
  addition_result = input + 2'd2;
  xor_result <= input & addition_result;
  output = xor_result;
end
```

In general, separate **blocking** and **non-blocking** assignments into `always_comb` and `always_ff` blocks.

```
always_comb begin
    out = 32'h0;
    if (enable) begin
        case (mode)
            2'b00:   out = a + b;
            2'b01:   out = a - b;
            2'b10:   out = a ^ b;
            default: out = a;
        endcase
    end
end

```

Conditionals are used for *data-dependent* behavior in combinational logic. Assignment within a conditional will be synthesized as a **multiplexer**.

```
// --- sized literals ---
logic [31:0] word = 32'hdeadbeef;
logic [3:0]  nibble = 4'b1010;
logic [7:0]  byte = 8'd255;
```

Sized literals are of the form size (in bits), base (hex, decimal, or binary), and literal.

```
1     // --- bit slice ---
2     logic [31:0] instr = 32'h00c58533;
3     logic [11:0] imm_i = instr[31:20]; // 12'h00c
```

Bit-slicing enables accessing a subset of the bits from a signal (from MSB to LSB).

```
1    // --- concatenation ---
2    logic [15:0] hi = 16'h1234;
3    logic [15:0] lo = 16'h5678;
4    logic [31:0] combined = {hi, lo}; // 32'h12345678
```

Concatenation allows combining multiple signals together by sequentially combining their bits.

```
1    // --- bitwise ops (operate bit-by-bit on vectors) ---
2    logic [7:0] a = 8'h0f;
3    logic [7:0] b = 8'h33;
4    logic [7:0] bw_and = a & b;
5    logic [7:0] bw_or  = a | b;
6    logic [7:0] bw_xor = a ^ b;
7    logic [7:0] bw_not = ~a;
```

Bitwise operations apply Boolean functions bit-by-bit on given bitvectors.

```
1    // --- logical ops (treat vectors as "true if nonzero") ---
2    logic enable = 1'b1;
3    logic ready  = 1'b0;
4    logic go     = enable && !ready;
```

Logical operations allow Boolean operations on bitvectors where they are true only if nonzero.