Hardware Design using Verilog

Prof. Sayandeep Saha and Kalind Karia
Dept. of CSE, IIT Bombay

Overview

Glossary: what terms will we cover in this lecture..

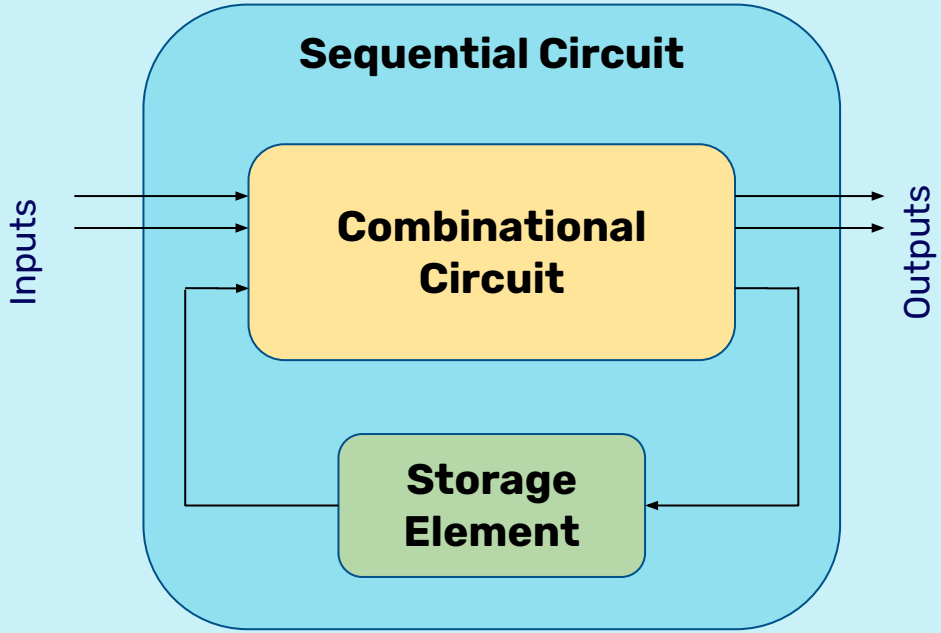
- **Introduction to Verilog**
- **Verilog Data Types**
- **Module Hierarchy**
- **Assignments**
- **Testbench**
- **Simulation**
- **Some digital logic designs**

Combinational and Sequential Circuit

Combinational and Sequential Circuit

- Combinational
  - determine boolean function
  - design using logic gates
  - no memory

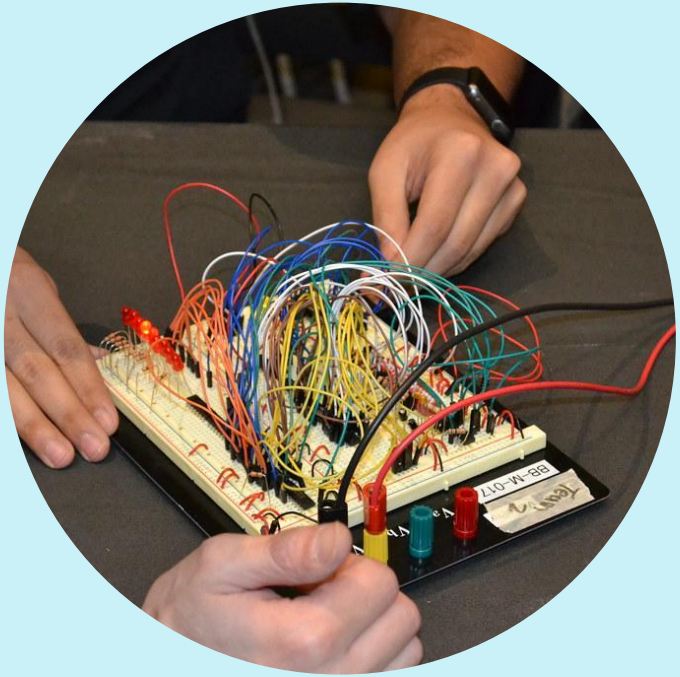
- Sequential
  - have memory
  - circuit that remembers!
  - model the circuit to create a **state**



Intro to FPGA and HDLs

What is Electronic Design?

- Process of developing a circuit by using **known** electronic components in order to meet the given **specifications**.
- Specifications: detailed description of desired behavior of the circuit
- Known devices: devices whose behavior can be modeled by known equations or algorithms, with known values of parameters



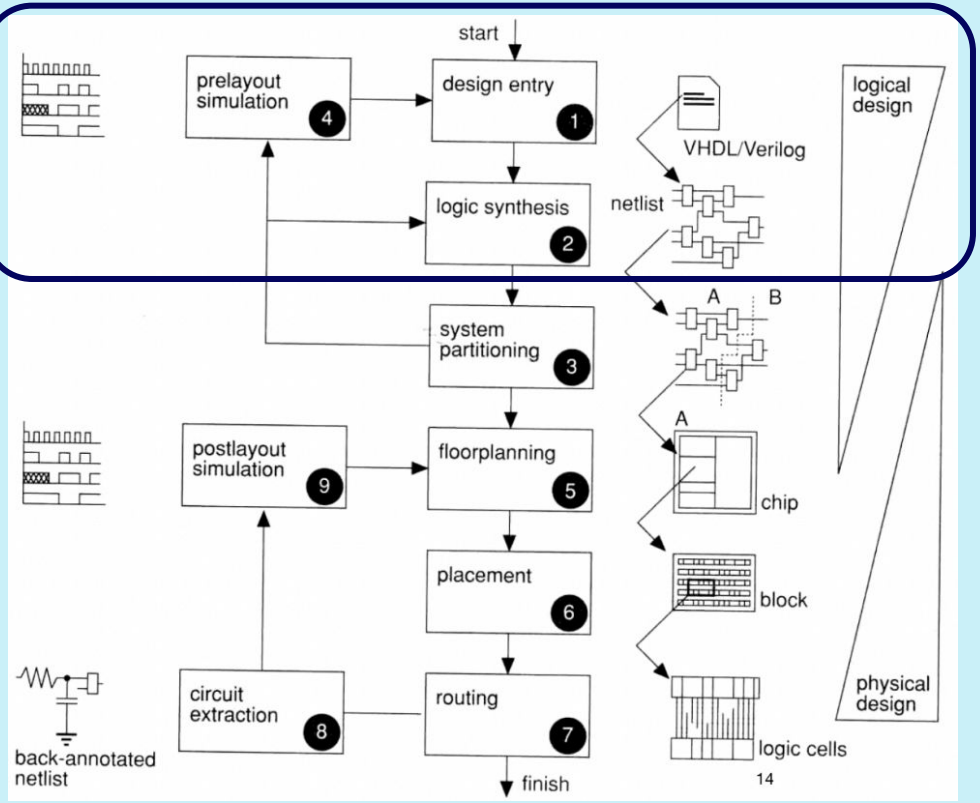
Hardware Description Languages (HDLs)

- A computer language used to describe the structure and behavior of electronic circuits (usually digital circuits)
- HDL: easy to describe hardware circuits following a particular syntax
- Need for HDLs → Circuits were designed on PCBs, testing and designing of large circuits not easy, slow time to market and complex design flow

➔ **Verilog**

➔ VHDL (Very High Speed Integrated Circuit Hardware Description Lang.)

ASIC design flow



Introduction to Verilog

Verilog language

- One of the two most commonly-used languages in digital hardware design (other is VHDL).
- Virtually every chip (FPGA, ASIC, etc.) is designed in part using one of these two languages.

Logic Simulation and Synthesis

- Logic Simulation
  - Runs your circuit in computer before you map it to silicon.
  - Essential for ensuring correctness, testing, etc.
- Logic Synthesis
  - Converts your logic described in high-level to a gate-level description => equivalent to a compiler.
  - Register-transfer level (RTL) to gates conversion
  - Such compilation also involves logic minimization

Concurrency

- Verilog or any HDL has the power to model concurrency which is natural to a piece of hardware.
- There may be two hardware circuits running parallelly.
- Verilog provides the following constructs for concurrency:
  - always
  - assign
  - module instantiation
  - non-blocking assignments inside a sequential block

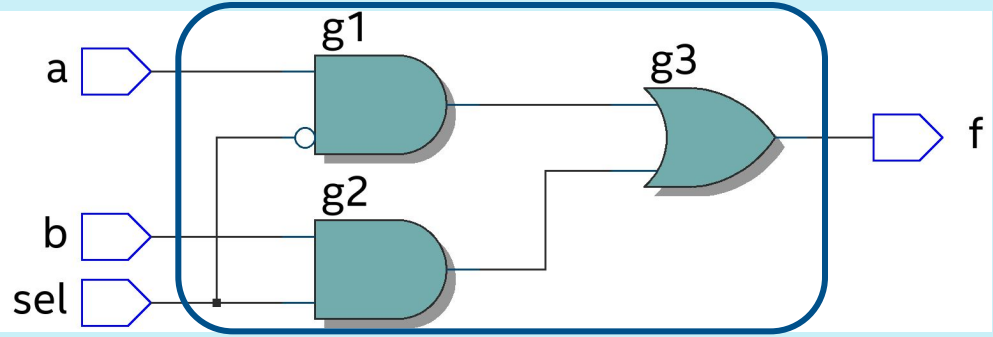
Multiplexer built with primitives (structural modelling)

Verilog programs built from modules

Each module has an interface

Identifiers not explicitly defined default to wires

Module may contain structure: instances of primitives and other modules




```
module mux (f, a, b, sel);
input a, b, sel;
output f;

wire f1, f2;

and g1 (f1,   a,  nsel),
    g2 (f2,   b,  sel);
or  g3 (f,    f1, f2);
not g4 (nsel, sel);

endmodule
```

Verilog logic values

- Predefined logic value system or value set to:
  - ‘0’, ‘1’, ‘x’ or ‘z’
- ‘x’ means uninitialized or unknown logic value
- ‘z’ means high impedance value

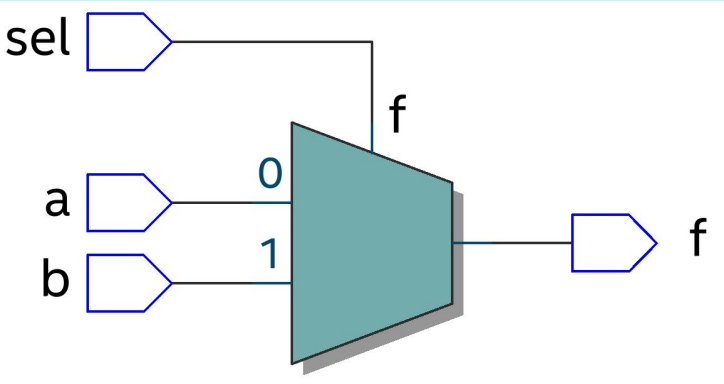
Verilog data types

- Nets: wire
  - Analogous to a wire in any circuit
  - Cannot store or hold a value
  - To model connectivity, any value driven by a device must be driven continuously onto that wire, in parallel with the other driving values.
- Integer: used for the index variables of say for loops. No hardware implication.

Multiplexer built with always (behavioral modelling)

Modules may contain one or more *always* blocks

Sensitivity list contains the signals whose change triggers the execution of the block




```
module mux (f, a, b, sel);
input a, b, sel;
output f;

reg f;

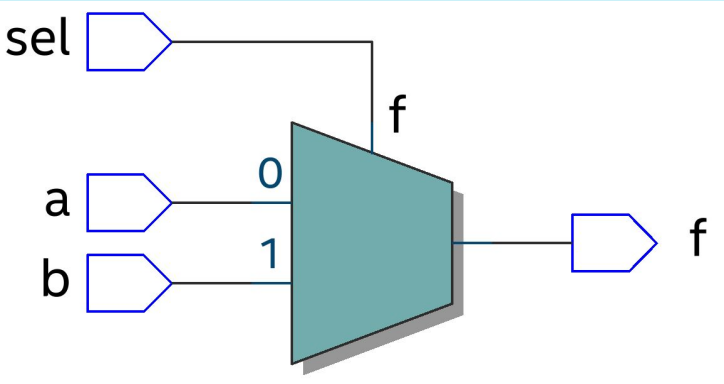
always @(a or b or sel) begin
    if (sel) f = b;
    else f = a;
end

endmodule
```

Multiplexer built with always (behavioral modelling)

A *reg* behaves like memory: holds its value until imperatively assigned otherwise

Body of an *always* block contains traditional imperative code




```
module mux (f, a, b, sel);
input a, b, sel;
output f;

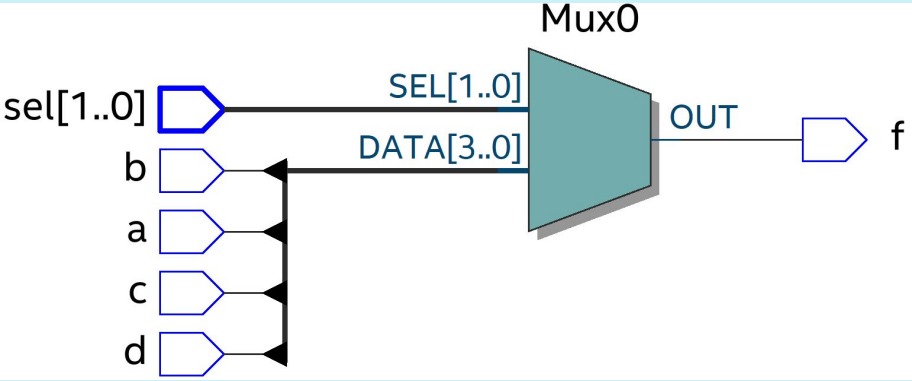
reg f;

always @(a or b or sel) begin
    if (sel) f = b;
    else f = a;
end

endmodule
```

Multiplexer built with always (behavioral modelling)

Triggers the execution of the always block on the change of all signals being read in the block




```
module mux (f, a, b, c, d, sel);
input a,b,c,d; input [1:0] sel;
output f;

reg f;

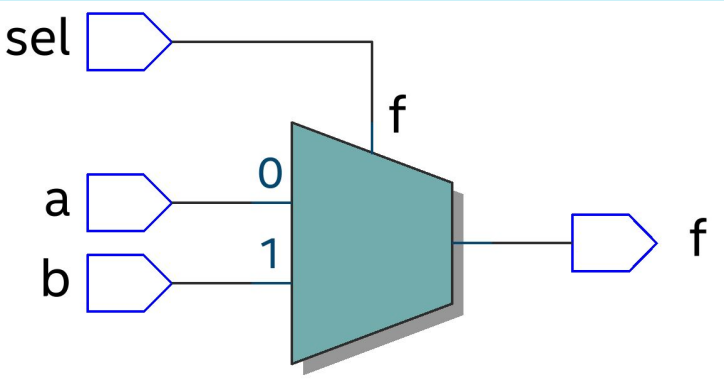
always @(*) begin
     case (sel)
          2'b00: f = a;
          2'b01: f = b;
          2'b10: f = c;
          2'b11: f = d;
          default: f = 2'd0;
     endcase
end
endmodule
```

The *reg* data type

- Register data type: similar to a variable in programming language
- Default initial value: ‘x’
- ```
  module reg_ex1;
  reg q; wire d;
  always @(posedge clk) q = d;
  ```
- A reg is not always equivalent to a hardware register, flip-flop or latch
- ```
  module reg_ex2; // purely combinational
  reg c;
  always @(a or b) c = a|b;
  ```

Multiplexer with assign (dataflow modelling)

LHS is always set to the value on the RHS
Any change on the RHS causes re-evaluation




```
module mux (f, a, b, sel);
input a, b, sel;
output f;

assign f = sel ? b : a;

endmodule
```

Any Questions?

Module Hierarchy and Instantiation

- Module interface provides the means to interconnect two Verilog modules.
- Note that a ***reg*** **cannot** be an **input** or **inout** port.
- A module may instantiate other modules.
- Instances of `module mymod (y, a, b);` can be of two ways:
  - Connect-by-position: `mymod mm1 (y1, a1, b1);`
  - Connect-by-name: `mymod mm2 (.a(a2), .b(b2), .y(y2));`

Sequential blocks and Procedures

- Sequential block is a group of statements between a `begin` and an `end`.
- A sequential block, in an `always` statement executes repeatedly.
- Inside an `initial` statement, it executes only once.
- A procedure is an `always` or `initial` statement or a function.
- Procedural statements within a sequential block executes concurrently with other procedures.

Assignments

- Let’s see the various module assignments
- ```
  module xyz ();
  // continuous assignments
  always // beginning of a procedure
      begin // beginning of a sequential block
      // …… procedural assignments
      end
  endmodule
  ```
- A continuous assignment assigns a value to a wire like a real gate driving a wire.
  ```

Assignments example

```
module holiday_1 (sat,sun,weekend);
input sat, sun;
output weekend;


// continuous assignment
assign weekend = sat | sun;

endmodule
```

```
module holiday_2 (sat,sun,weekend);
input sat, sun;
output weekend;

reg weekend;

always @(sat or sun)
    // procedural assignment
    weekend = sat | sun;

endmodule
```

Blocking and Non-blocking assignments

- Blocking procedural assignments must be executed **before** the procedural flow can pass to the subsequent statement.
- A non-blocking procedural assignment is **scheduled** to occur **without** blocking the procedural flow to subsequent statements.

```
a = 1; b = a; c = b;
```

Blocking assignment:

```
a = b = c = 1
```

```
a <= 1; b <= a; c <= b;
```

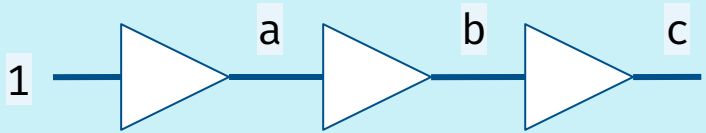
Non-blocking assignment:

```
a = 1
b = old value of a
c = old value of b
```

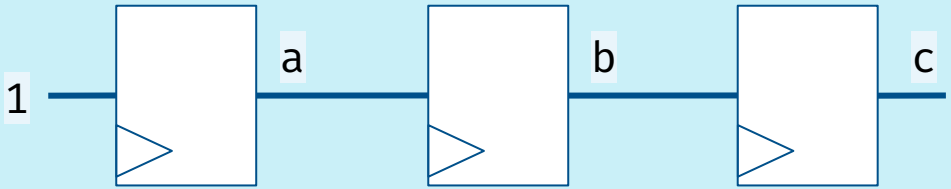
Non-blocking looks like latches!

- RHS of non-blocking taken from latches
- RHS of blocking taken from wires

```
a = 1; b = a; c = b;
```




```
a <= 1; b <= a; c <= b;
```



Examples

```
// blocking assignment
always @(a1 or b1 or c1 or m1) begin
    m1 = #3 (a1 & b1);
    y1 = #1 (m1 | c1);
end
```

Statement executed at time t causing m1 to be assigned at t+3

Statement executed at time t+3 causing y1 to be assigned at time t+4

```
// non-blocking assignment
always @(a2 or b2 or c2 or m2) begin
    m2 <= #3 (a2 & b2);
    y2 <= #1 (m2 | c2);
end
```

Statement executed at time t causing m2 to be assigned at t+3

Statement executed at time t causing y2 to be assigned at time t+1. Uses old values.

Numbers

- Format of integer constants:

  ```
  width’ radix value
  ```

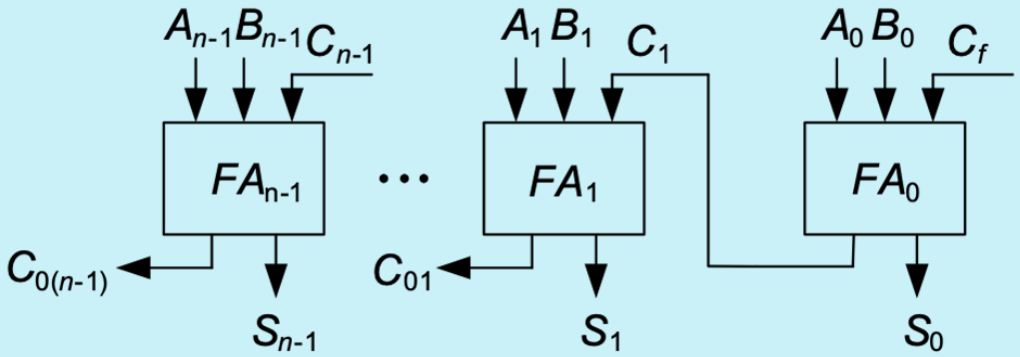
- Example: `2’b00` or `2’d0`

Any Questions?

Ripple Carry Adder

Ripple Carry Adder

- Consists of N cascaded stages of full adder.



C_f : forced carry
$C_{0(n-1)}$: overflow carry

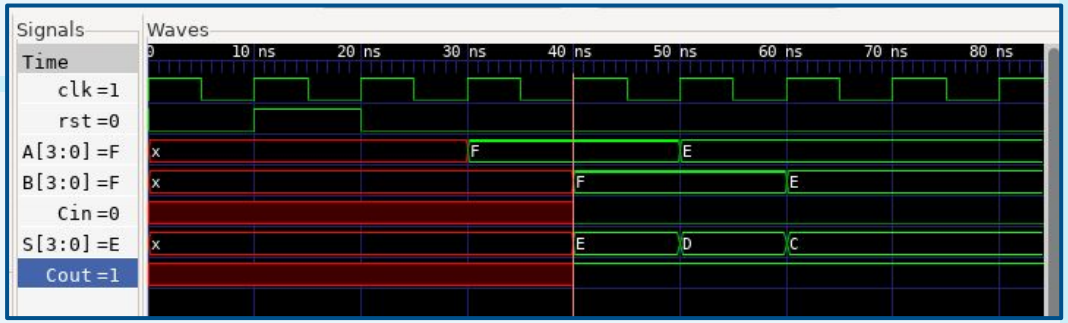
$$S_i = A_i \oplus B_i \oplus C_i$$
$$C_{0i} = A_iB_i + B_iC_{0(i-1)} + C_{0(i-1)}A_i$$

Code for 4-bit Ripple Carry Adder

```
module full_adder (a, b, cin, s, cout);
input a, b, cin;
output s, cout;

assign s = a ^ b ^ cin;
assign cout = (a & b) | (cin & (a ^ b));

endmodule
```




```
module adder_4bit (a, b, cin, s, cout);
input [3:0] a, b; input cin;
output [3:0] s; output cout;

wire c1, c2, c3;

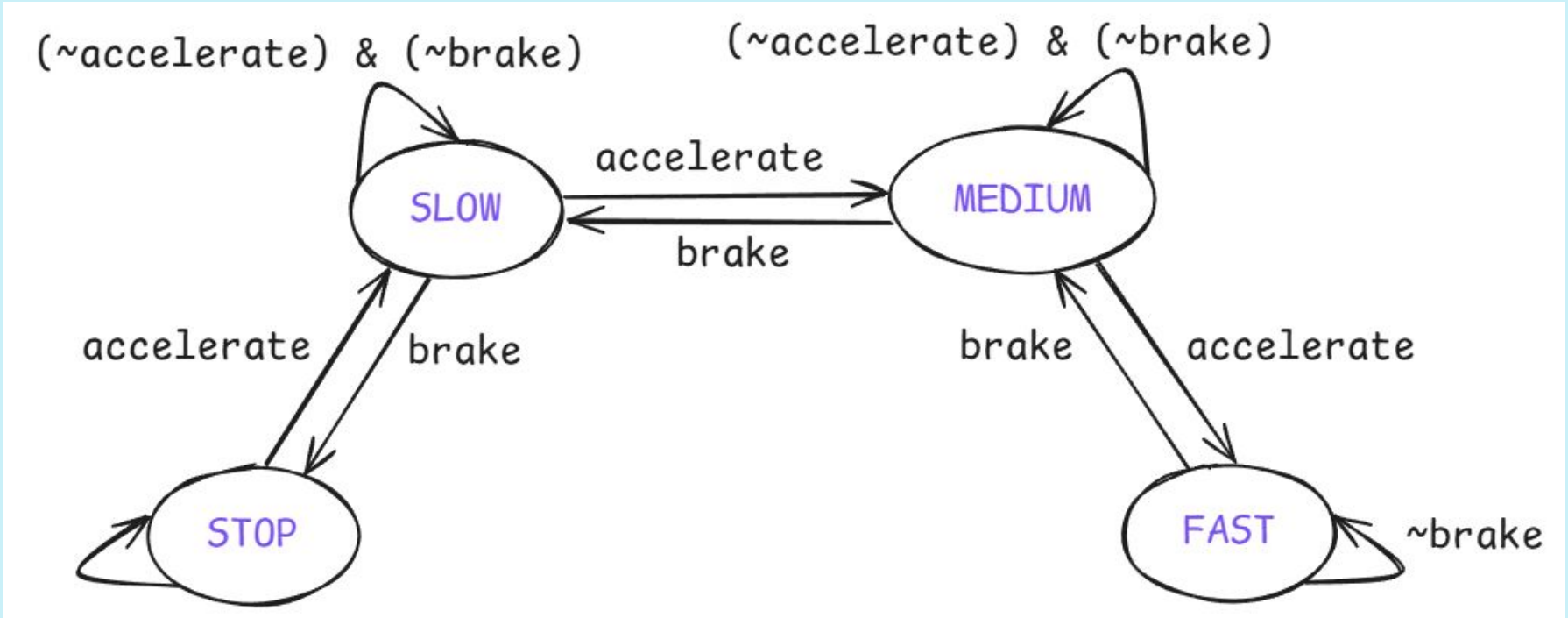
full_adder fa1 (.a(a[0]), .b(b[0]), .cin(cin), .s(s[0]), .cout(c1));
full_adder fa2 (.a(a[1]), .b(b[1]), .cin(c1), .s(s[1]), .cout(c2));
full_adder fa3 (.a(a[2]), .b(b[2]), .cin(c2), .s(s[2]), .cout(c3));
full_adder fa4 (.a(a[3]), .b(b[3]), .cin(c3), .s(s[3]), .cout(cout));

endmodule
```

Modelling Sequential Circuits

- `always @(posedge clk) begin <procedural_statements> end`
- "`posedge clk`" means that the value in the flip-flops change at the positive edge of the `clk`
- "`negedge clk`" can also be used
- `@(sensitivity_list)` triggers the always block when one of the signals in the list changes

Car Speed Controller

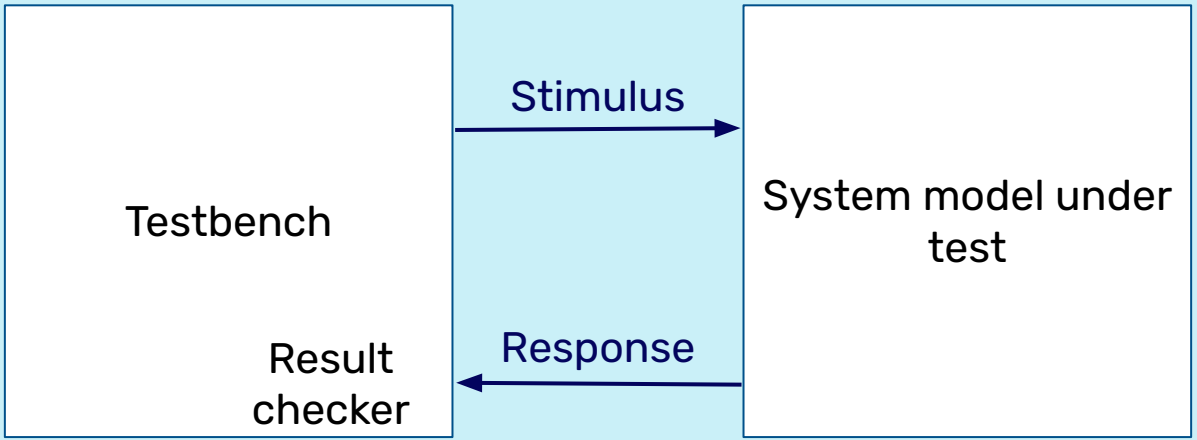


Any Questions?

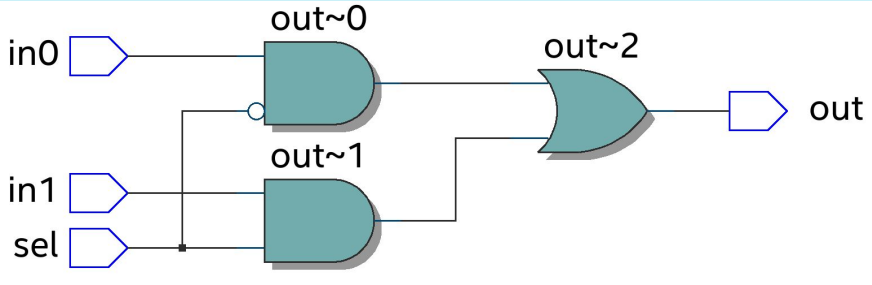
Behavioral Simulation

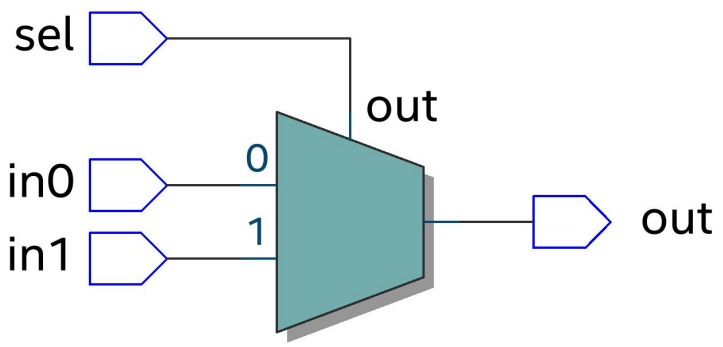
How do we test the behavior of a design?

- Testbench generates stimulus and checks response
- Coupled to model of the system under test
- Testbench and system under test are run simultaneously



Looking back at the multiplexer design






```
// dataflow modelling
module mux2 (in0,in1,sel,out);
input in0, in1, sel;
output out;

assign out = (~sel & in0)
           | (sel & in1);

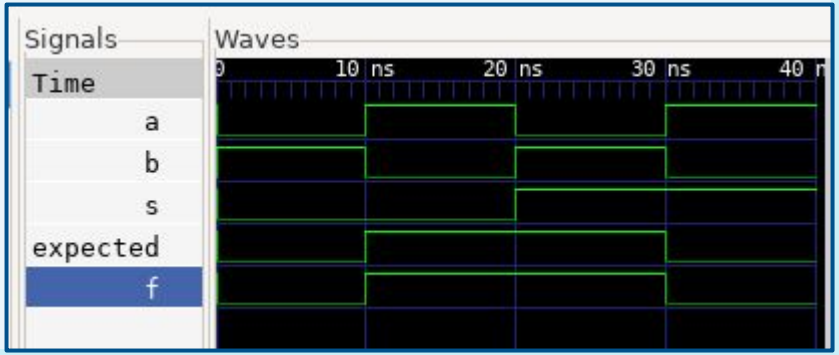
// alternative
// assign out = sel ? in1 : in0;

endmodule
```

Testbench for the multiplexer

```
`timescale 1ns/1ps
module testmux;
reg a, b, s;
wire f;
reg expected;

mux2 dut (.sel(s), .in0(a), .in1(b), .out(f));
initial begin
$monitor ("sel=%b in0=%b in1=%b out=%b,expected out=%b time=%d",s,a,b,f,expected,$time);
     s = 0; a = 0; b = 1; expected = 0;
     #10 a = 1; b = 0; expected = 1;
     #10 s = 1; a = 0; b = 1; expected = 1;
     #10 a = 1; b = 0; expected = 0; #10;
end
initial begin
     $dumpfile ("dump.vcd");
     $dumpvars (1, testmux);
end
endmodule
```



Caution!

- Write codes which can be translated into hardware!
- Following cannot be translated into hardware (non-synthesizable)
  - `initial` blocks
    - Used to set up initial state or describe finite testbench stimuli
    - Don’t have obvious hardware component
  - Delays
    - Maybe in the Verilog source, but are simply ignored
- Finally, remember that you are a better designer than the tool.

Any Questions?

Some more design examples

Counter

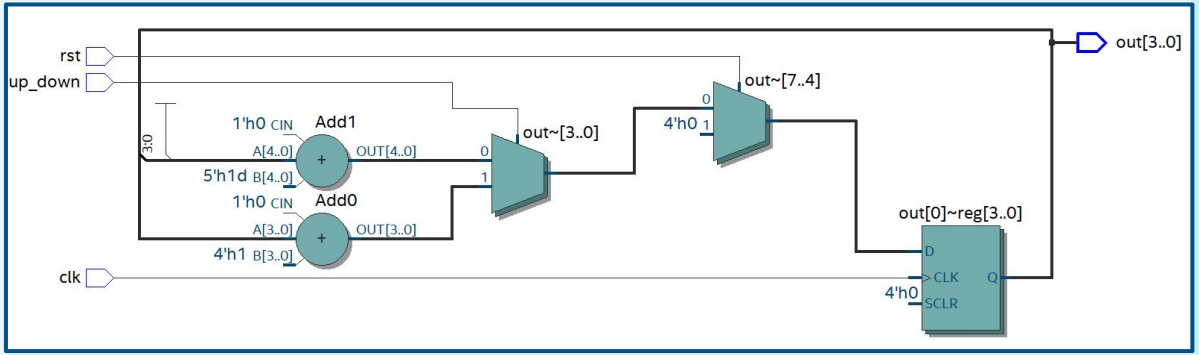
Code for 4-bit Up Down Counter

```
module up_down_counter (clk, rst, up_down, out);
input clk, rst, up_down;
output [3:0] out;

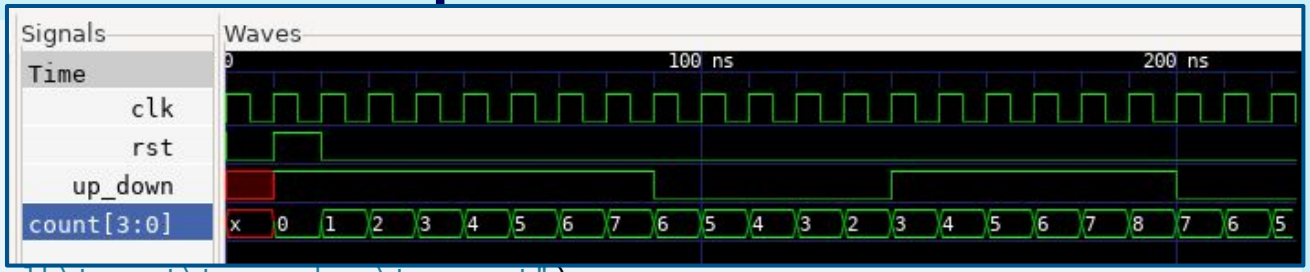
reg [3:0] out;

always @(posedge clk) begin
     if (rst)
          out <= 4'b0;
     else if (up_down)
          out <= out + 1'b1;
     else
          out <= out - 1'b1;
end

endmodule
```



Testbench for 4-bit Up Down Counter




```
`timescale 1ns/1ps
module tb_up_down_counter;
reg clk, rst, up_down;
wire [3:0] count;

initial begin
        $display ("time\t, clk\t, rst\t, up_down\t, count");
        $monitor ("%g\t %b\t %b\t %b\t %d", $time, clk, rst, up_down, count);

        clk = 1; rst = 0;
        #10 rst = 1; up_down = 1; #10 rst = 0;
        #70 up_down = 0; #50 up_down = 1; #60 up_down = 0;
        #20; #5 $finish;
end
always begin
        #5 clk = ~clk;
end

up_down_counter dut (.clk(clk), .rst(rst), .up_down(up_down), .out(count));
initial begin
        $dumpfile ("up_down_counter.vcd"); $dumpvars (1, tb_up_down_counter);
end
endmodule
```

Composite Function

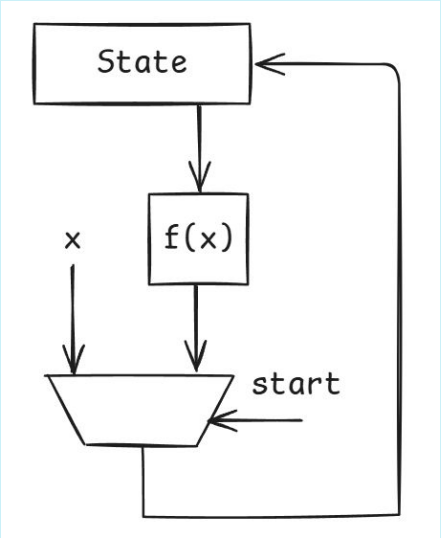
Problem statement

- Goal: Let's say we have a function **`y = f (x) = (A & x) ^ B`** where **A** and **B** are constants
- You have to implement the below functionality

```
while (counter <= 31):
    x = f(x);
    counter++;
```

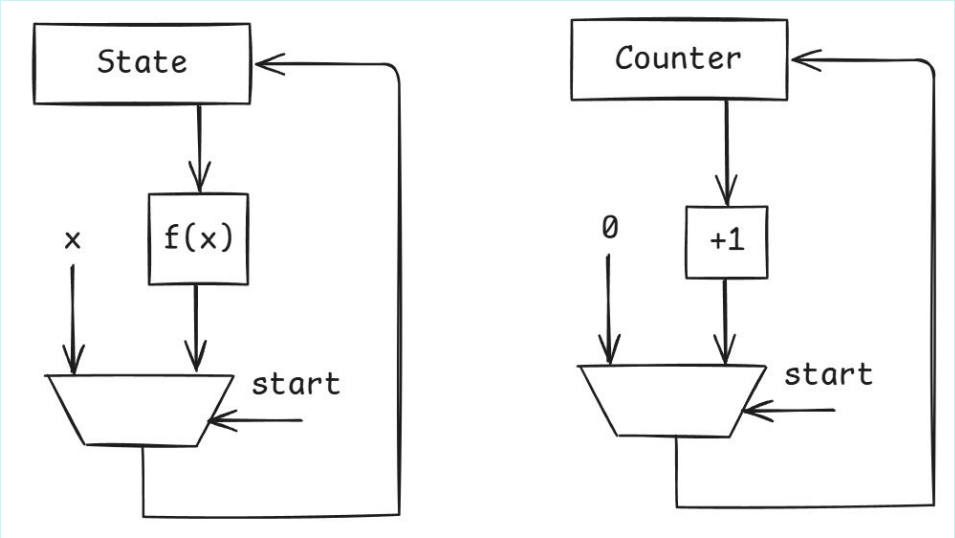
How to design the datapath?

- Let’s draw the design in steps



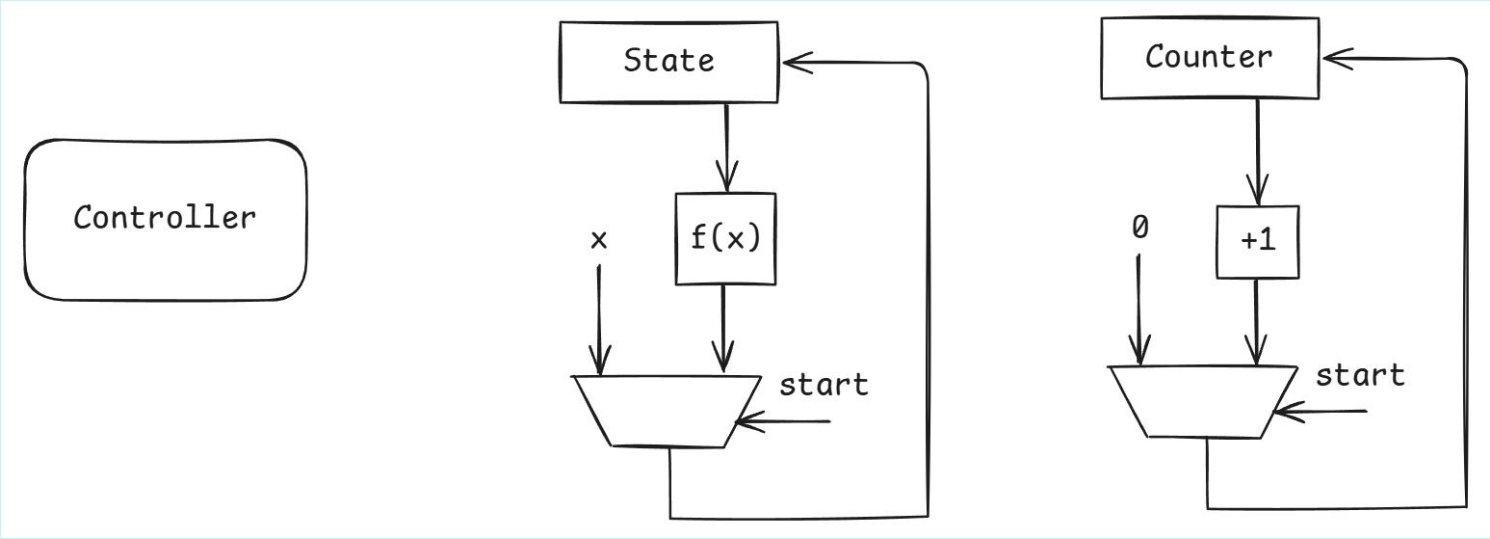
How to design the datapath?

- We also need a counter
- The counter counts and the state updates for each clock while start = 1



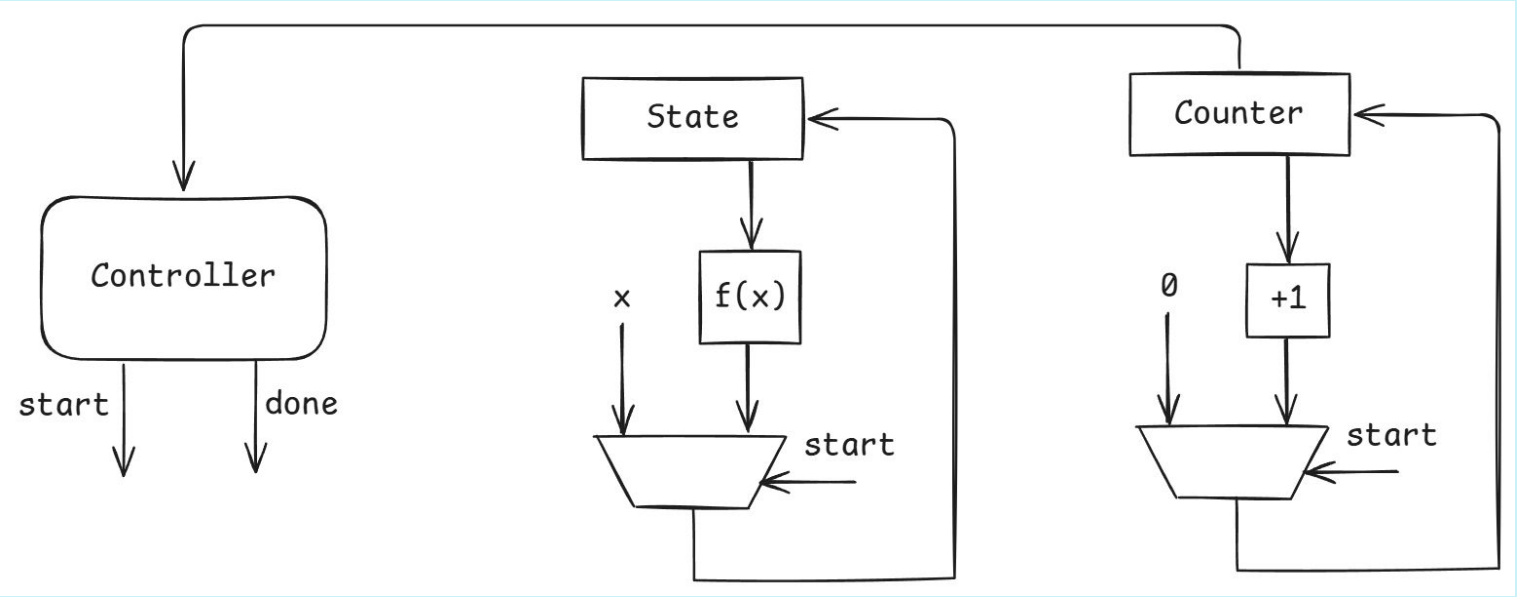
How to design the datapath?

- Now let’s bring in the controller



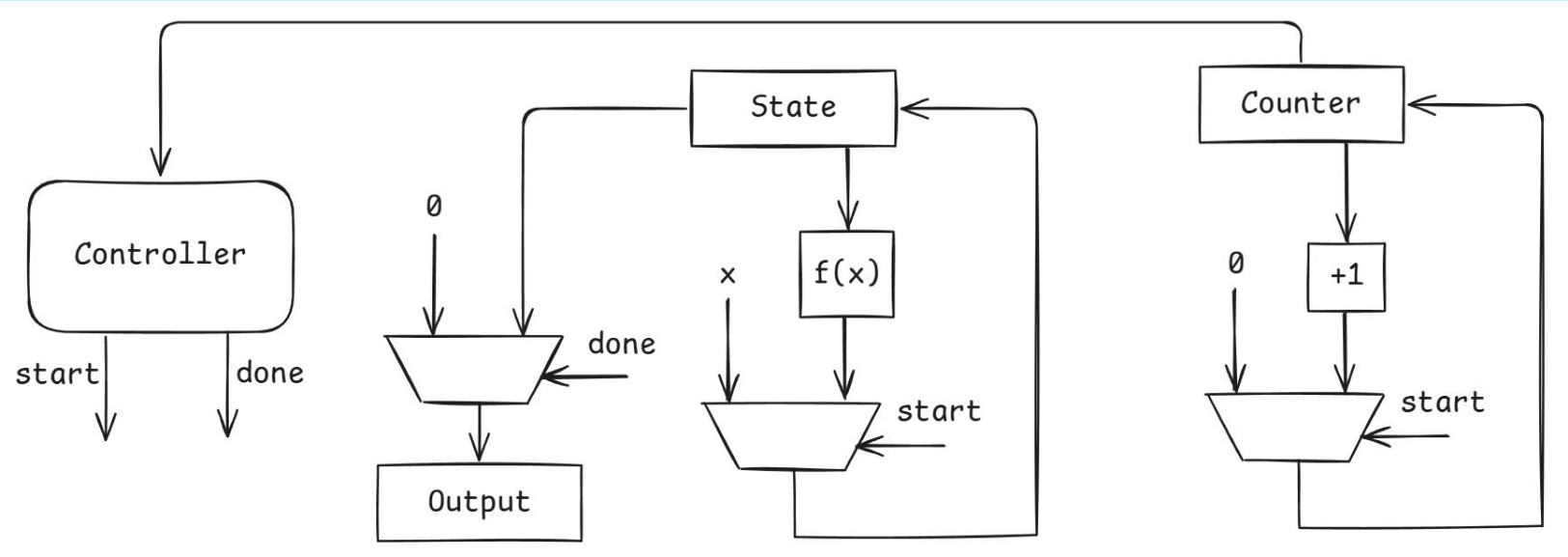
How to design the datapath?

- The controller stops the circuit when counter == 31



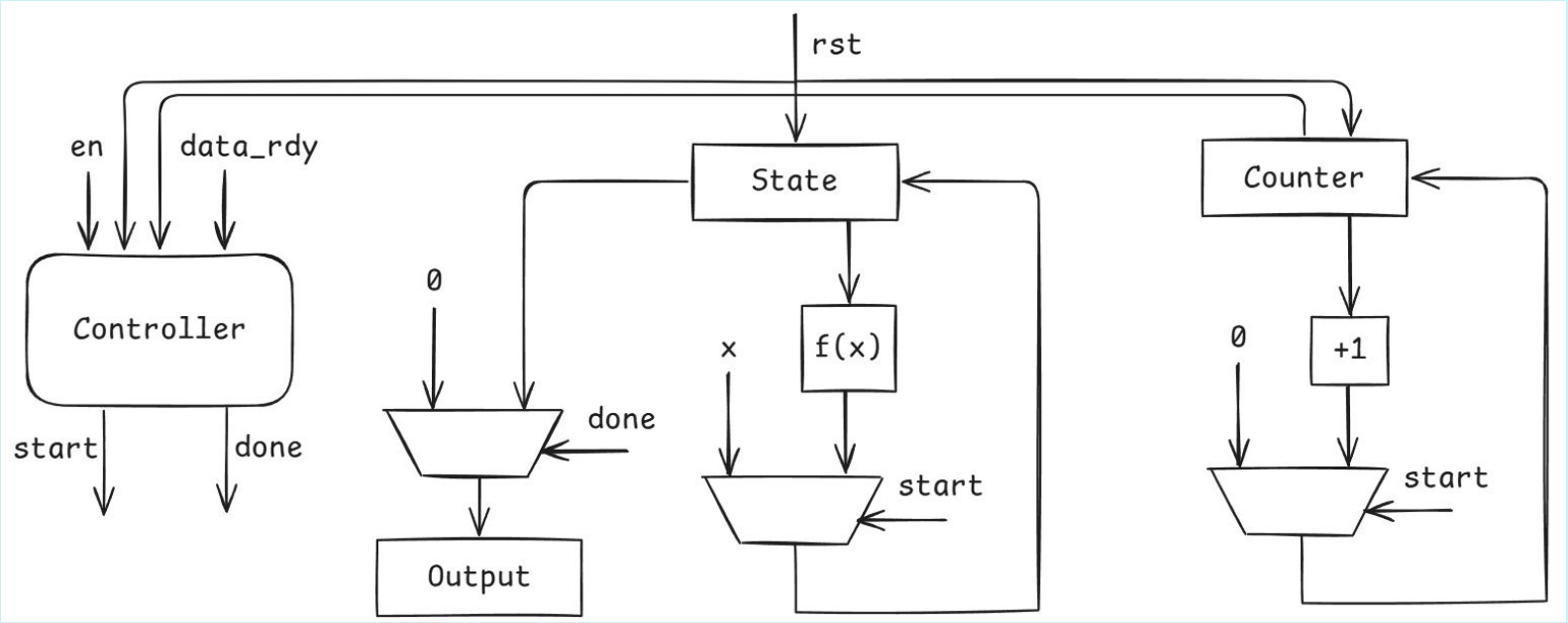
How to design the datapath?

- The controller stops the circuit when counter == 31
- It also generates output at that point, controlled by “done” signal



How to design the datapath?

- Some more essential input signals



Karatsuba Multiplier

Karatsuba algorithm

- Basic principle: divide-and-conquer
- Computes product of two numbers **x** and **y** using three multiplications of smaller numbers, each having half the digits as **x** or **y**, and some additions and logical shifts.

Karatsuba algorithm

- Let x and y be n-digit strings in base B. For any positive integer m less than n, we can write,

  $$x = x_1 B^m + x_0 \text{ and } y = y_1 B^m + y_0$$

  where x_0 and y_0 are less than B^m.

- The product is then,

  $$\begin{aligned} xy &= (x_1 B^m + x_0)(y_1 B^m + y_0) \\ &= x_1 y_1 B^{2m} + (x_1 y_0 + x_0 y_1) B^m + x_0 y_0 \\ &= t_2 B^{2m} + t_1 B^m + t_0 \end{aligned}$$

Karatsuba algorithm

- The product is then,

  $$xy = t_2 B^{2m} + t_1 B^m + t_0$$

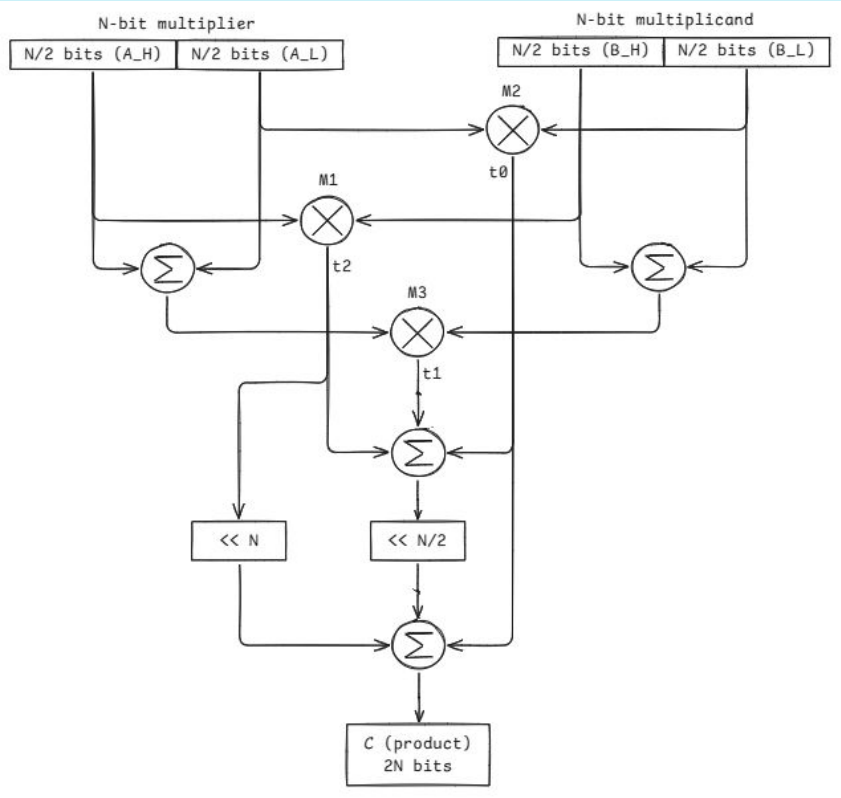
  where $t_2 = x_1 y_1$, $t_1 = x_1 y_0 + x_0 y_1$, $t_0 = x_0 y_0$

- Karatsuba's contribution:

  with t_2 and t_0 as before and $t_3 = (x_1 + x_0)(y_1 + y_0)$,

  $$\begin{aligned} t_1 &= x_1 y_0 + x_0 y_1 \\ &= (x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0 \\ &= t_3 - t_2 - t_0 \end{aligned}$$

Datapath of Karatsuba multiplier



Recap…

- Break the design into data path and control path.
- Datapath is mostly combinational.
- Controller is the sequential logic which sends control signal to the data path.
- Always start by drawing the data path.
- Then identify the control signals.
- Make a module for data path components, this may contain many submodules
- Make a separate module for the controller (optional but recommended).
- Instantiate and connect everything in a top module.
- Top module will contain multiple combinational and sequential blocks.

Thank You!