Hardware Design using Verilog

Sayandeep Saha

Assistant Professor
Dept. of CSE, IIT Bombay

Overview

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

FPGA

LUT and CLB

Comparison between FPGA, ASIC and Microcontroller

HDLs

FPGA design flow

Introduction to Verilog

Verilog Data Types

Module Hierarchy

Assignments

Parameters

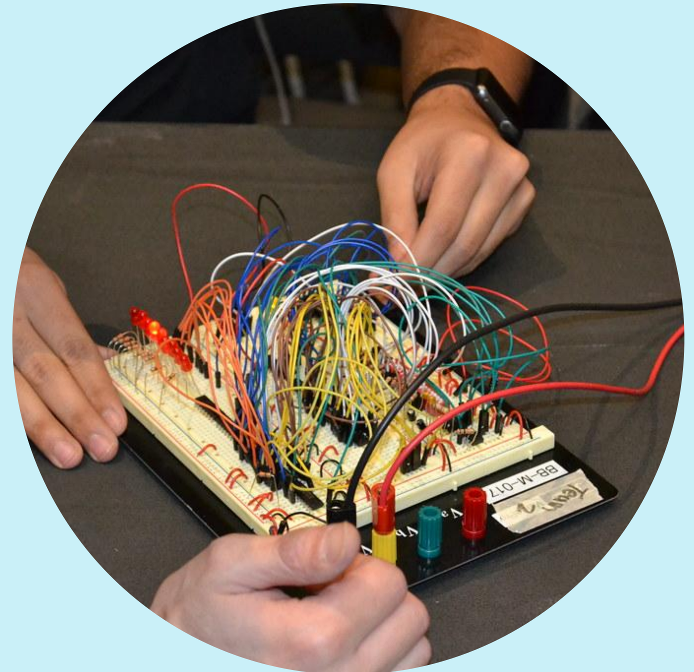
Testbench

Simulation

Introduction to FPGA

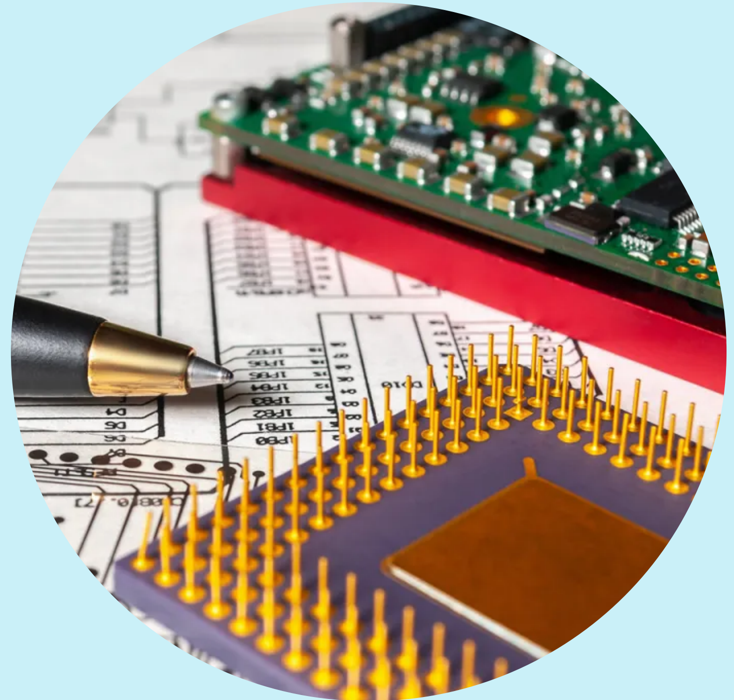
What is Electronic Design?

- 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



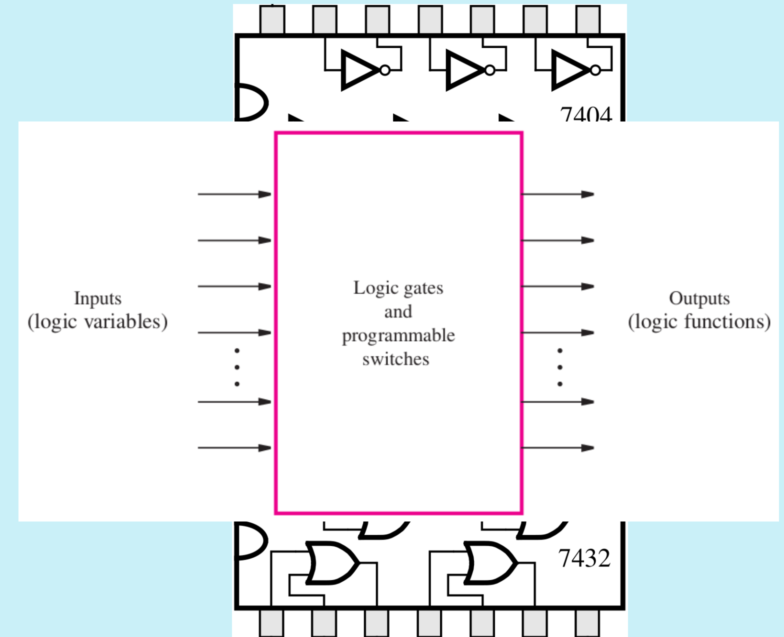
Electronic Design

- Process of converting a behavioral description of a circuit to a structural description.
- After the conversion, comes the **physical design** which involves choosing component sizes, their physical arrangement, defining the connections.
- Physical design is already done in **FPGA** based design.

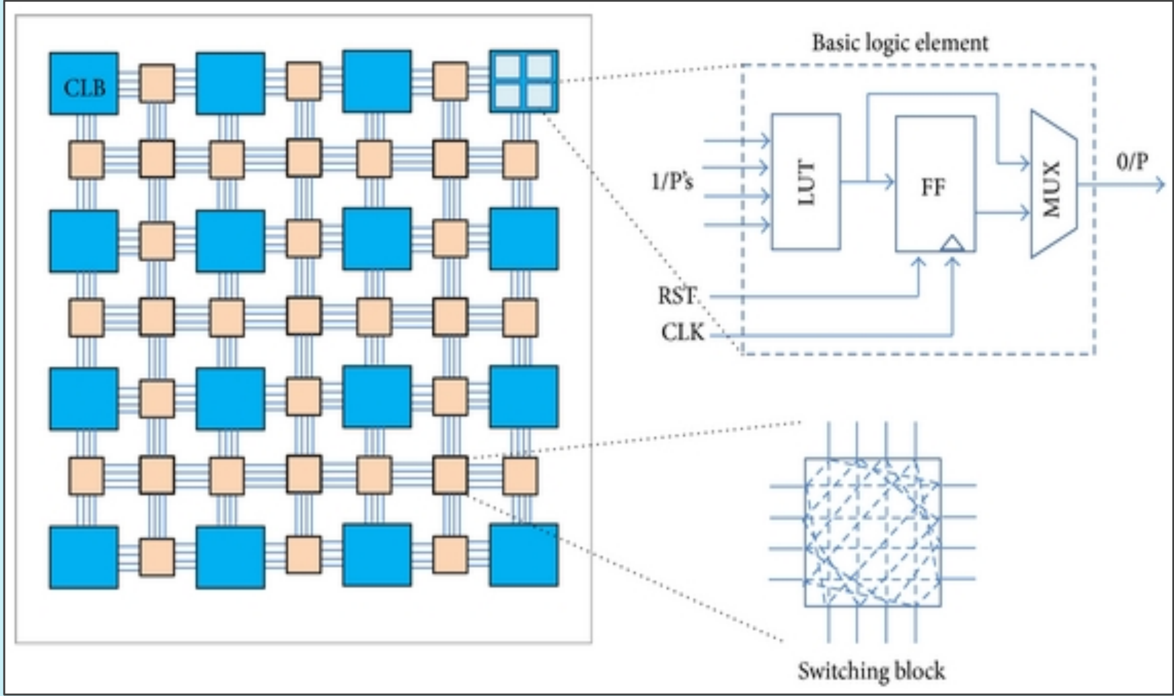


Are standard chips the best choice?

- Functions provided by these chips are fixed and inefficient for building large circuits.
- Need for a chip suitable for large logic circuitry where structure is not fixed.
- **PLD** (Programmable Logic Device): general-purpose chip with a pool of logic gates and programmable switches.

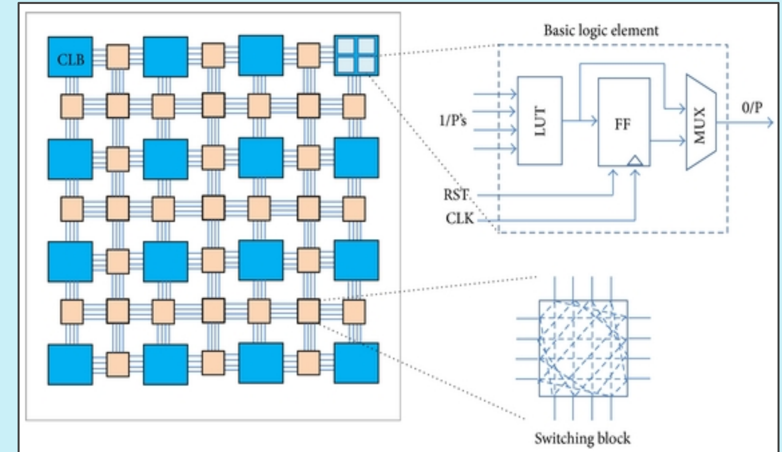


PLD example: FPGA



PLD example: FPGA

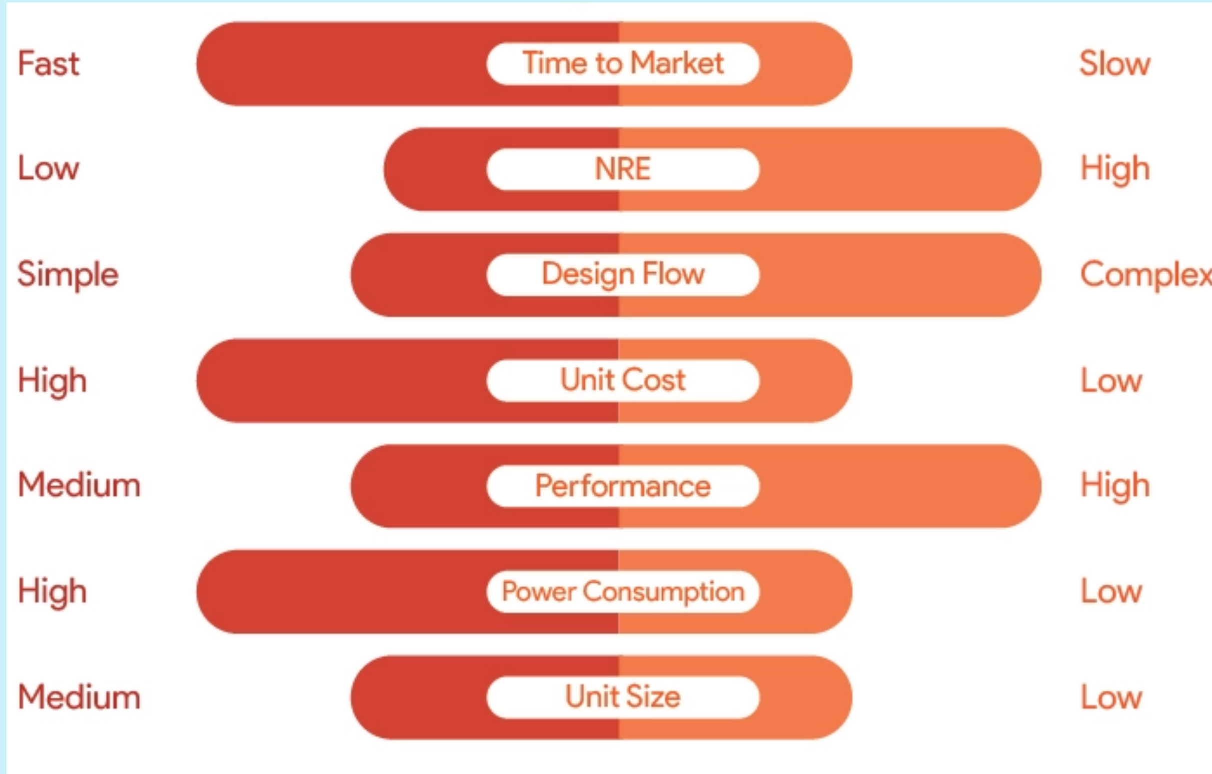
- FPGA: an array of pre-placed, prefabricated configurable logic blocks/elements (CLBs)
- These blocks are also connected by a prefabricated signal routing conductor path segments
    - Appropriate paths are stitched together by configuring switches
    - Switches: pass-transistors



Comparison between FPGA, ASIC and Microcontroller & Applications

FPGA vs ASIC

FPGA



ASIC

Source : https://hardwarebee.com/fpga-vs-asic-difference/

FPGA vs Microcontroller

FPGA	Microcontroller
• All parts of circuit operate independently • Max. clock speed depends on design • Many IOs can be accessed simultaneously • Power usage depends on design but typically more than microcontroller • Relatively costly	• Executes one instruction at a time • Fixed max. clock speed • Limited IOs (typically 8) can be accessed at a time • Often very power efficient with advanced sleep modes • Relatively cheap

Applications of FPGA

- Prototyping
  - Ensemble of gate arrays used to emulate a circuit before manufacturing
  - Faster and more accurate simulation
- Less cost production
  - Very less time to market
- Reconfigurable Computing
  - Same hardware used to configure on the fly!
- Special Purpose Computing Platform
  - Dedicated to solve one problem
  - Can be attached to general purpose computers

HDLs and FPGA Design Flow

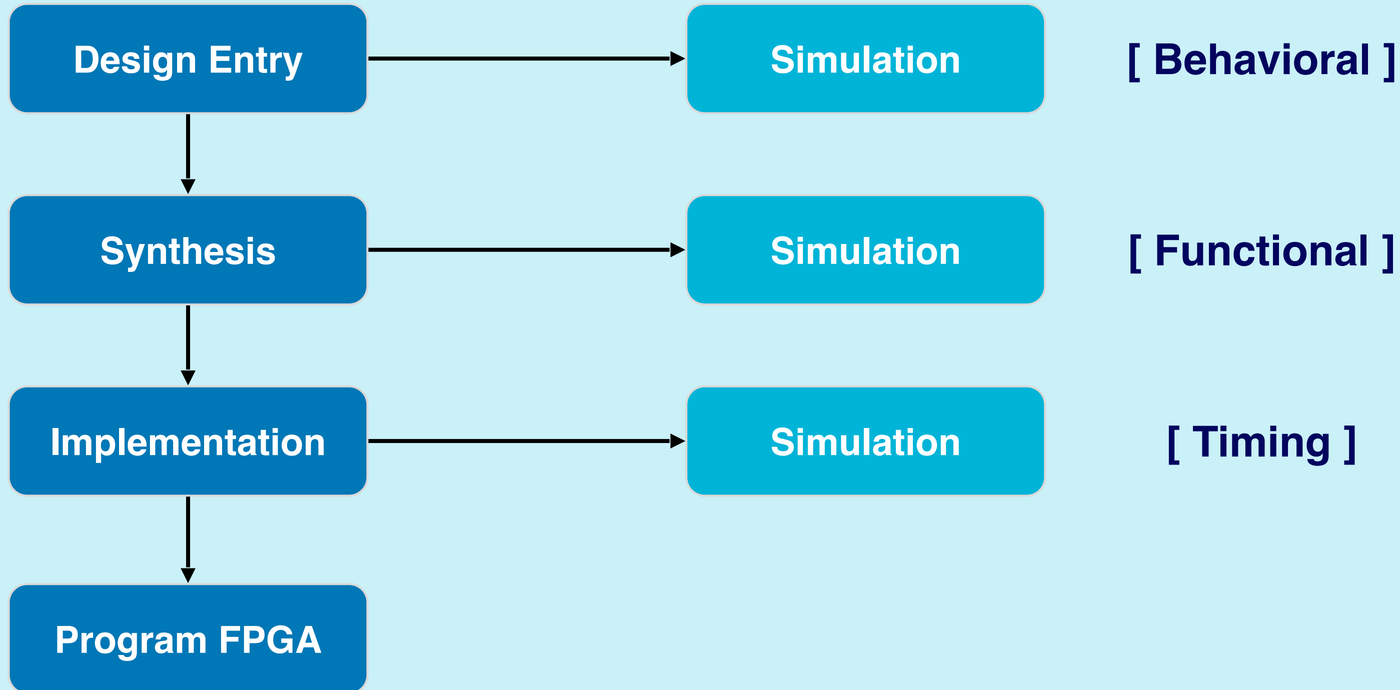
Hardware Description Languages (HDLs)

- A computer language used to describe the structure and behavior of electronic circuits (usually digital circuits).
- Need for HDLs
  - Circuits were designed on PCBs, test and design of large circuits not easy
  - HDL: easy to describe hardware circuits in terms of software algorithm with a particular syntax
  - CAD tools now generate the complex hardware circuit from the HDL algorithm

➔ **Verilog**

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

FPGA Design Flow



Any Questions?

Introduction to Verilog

Verilog language

- Originally a modeling language for a very efficient event-driven digital *logic simulator*.
- Later pushed into use as a specification language for *logic synthesis*.
- 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 that are independent of each other.
- 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 programas built from modules

Each module has an interface

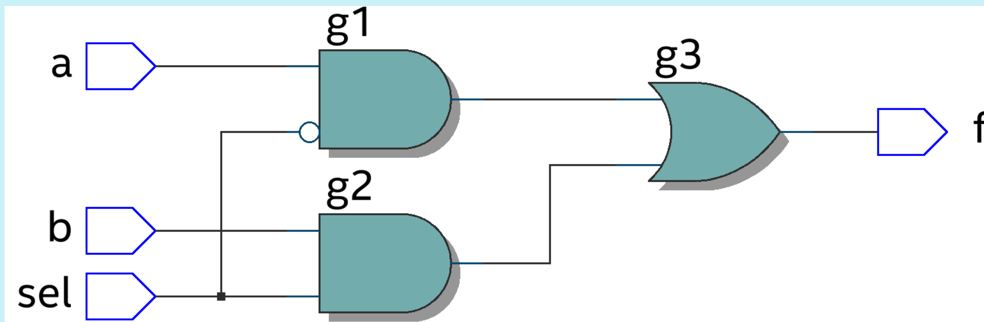
Module may contain structure: instances of primitives and other modules

```
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

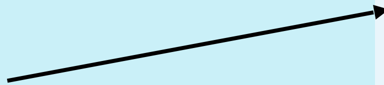
- Predefined logic value system or value set to:
  - ‘0’, ‘1’, ‘x’ or ‘z’
- ‘x’ means uninitialized or unknown logic value
- ‘z’ means high impedance value

Verilog data types

- Nets: wire
  - Analogous to a wire in an ASIC
  - Cannot <u>store</u> or <u>hold</u> 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 primitives (structural modelling)

Identifiers not explicitly defined default to wires

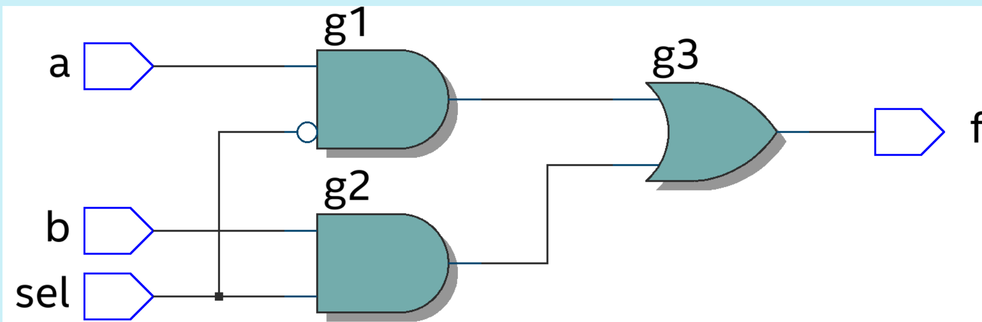


```
module mux (f, a, b, sel);
input a, b, sel;
output f;

wire f1, f2;

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

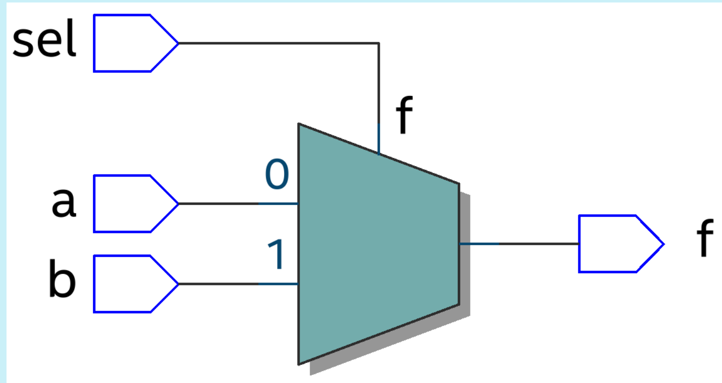
endmodule
```



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;

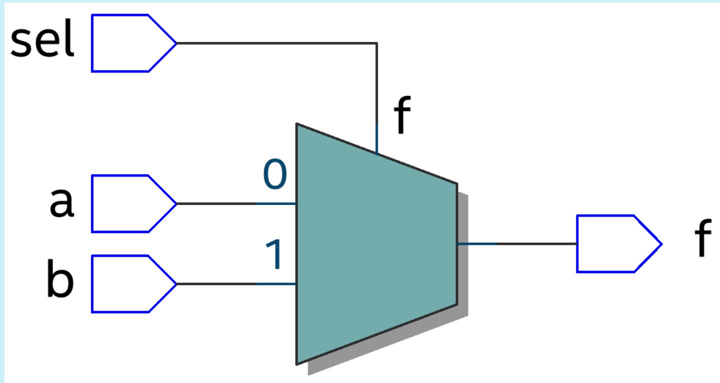
always @(a or b or sel) begin
            if (sel) f = b;
            else f = a;
end

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
```

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;
  ```

The *reg* data type

- Programming languages provide variables that can contain arbitrary values of a particular type.
- They are implemented as simple memory locations.
- Assigning a value to such variables is like storing the value at the memory location.
- Verilog *reg* operates in the same way.
- Previous assignments have no effect on the final result.

Example of *reg*

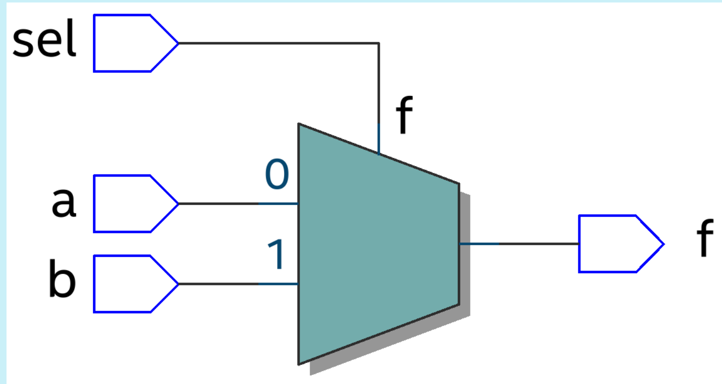
```
module assignments;
reg r;

initial r <= #20 3;
initial begin
        r = 5; r <= #35 2;
end
initial begin
        r <= #100 1;
        #15 r = 4;
        #220;
        r = 0;
end
endmodule
```

- The variable r is shared by all the concurrent blocks.
- r takes the value that was last assigned.
- This is *like* a hardware register which also stores the value that was last loaded into it.
- But a *reg* is **not necessarily** a hardware register.

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;

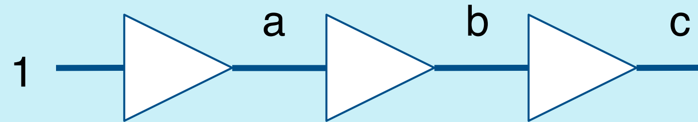
Non-blocking assignment:

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

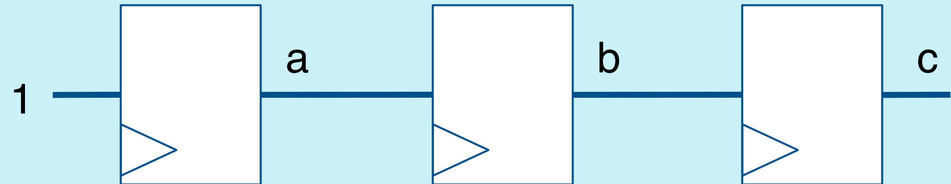
Non-blocking looks like latches!

- 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;

- Verilog keeps track of the sign if it is assigned to an integer or assigned to a parameter.
- Once Verilog loses the sign the designer has to be careful.

Parameterized design

```
module vector_and (z, a, b);
parameter cardinality = 1;
input [cardinality-1:0] a, b;
output [cardinality-1:0] z;
        wire [cardinality-1:0] z = a & b;
endmodule
```

```
module four_and_gates (outbus, inbus_a, inbus_b);
input [3:0] inbus_a, inbus_b; output [3:0] outbus;

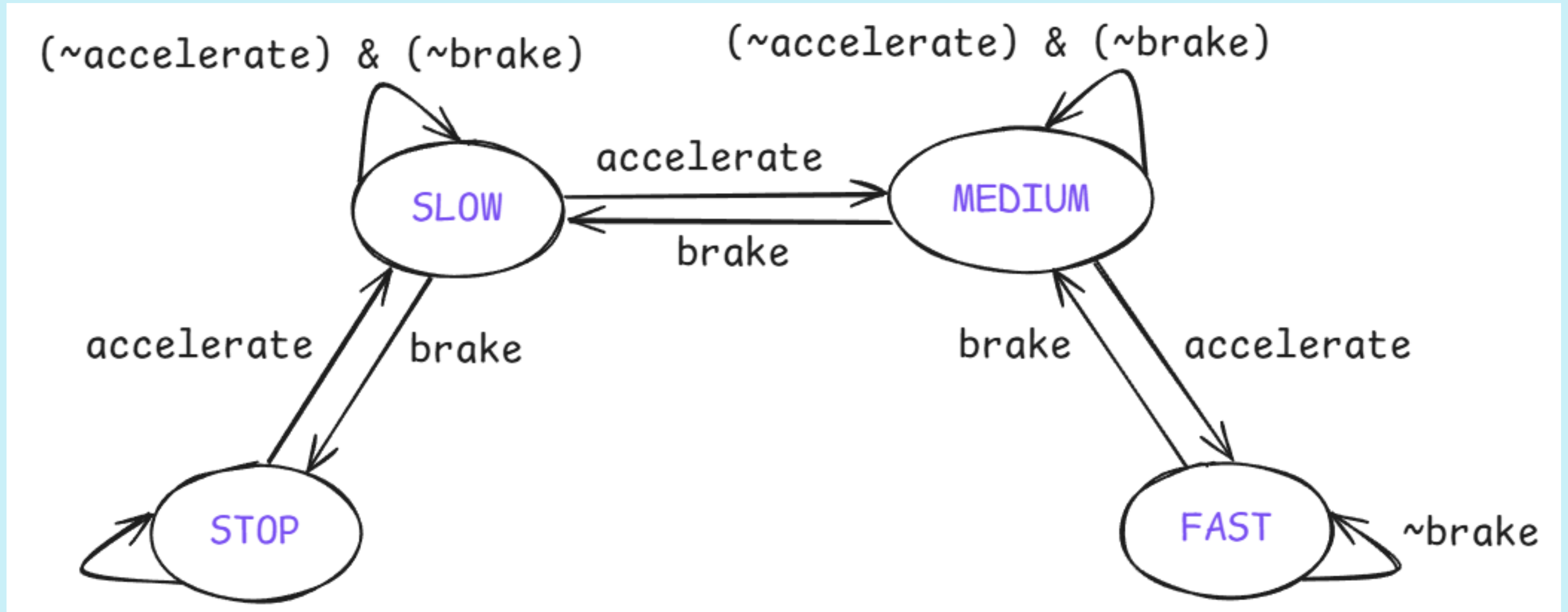
vector_and #(4) my_and(outbus, inbus_a, inbus_b);
endmodule
```

Any Questions?

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



Car Speed Controller Code (I)

```
module fsm_car_speed1 (clk,keys,accelerate,brake,speed);
input clk, keys, accelerate, brake;
output [1:0] speed;

reg [1:0] speed;

// states of speed
parameter        stop  = 2'b00,
                 slow  = 2'b01,
                 mdium = 2'b10,
                 fast  = 2'b11;
```

Car Speed Controller Code (II)

```
always @(posedge clk or
            negedge keys)
        begin
        if (!keys)
                speed = stop;
        else if (accelerate)
                case (speed)
                        stop: speed =
slow;
                        slow: speed =
mdium;
                        mdium: speed =
fast;
                        fast: speed =
fast;
```

```
                else if (brake)
                case (speed)
                        stop: speed =
stop;
                        slow: speed =
stop;
                        mdium: speed =
slow;
                        fast: speed =
mdium;
                endcase
        else
                speed = stop;
        end
```

Car Speed Controller: a better way!

- We keep a separate control part where the next state is computed.
- The other part generates the output from the next state.
- We follow this architecture in the implementation of any finite state machines, like ALU, CPU, etc.

Car Speed Controller [better] Code

```
module fsm_car_speed2 (clk,keys,accelerate,brake,speed);
input clk, keys, accelerate, brake;
output [1:0] speed;

reg [1:0] speed; reg [1:0] newspeed;

// states of speed
parameter          stop  = 2'b00,
                   slow  = 2'b01,
                   mdium = 2'b10,
                   fast  = 2'b11;
```

```
// datapath
always @(keys or accelerate or
                brake or speed)
        begin
        case (speed)
                        stop:
                if (accelerate)
                        newspeed = slow;
                else
                        newspeed = stop;
                        slow:
                if (brake)
                        newspeed = stop;
                else if (accelerate)
                        newspeed = mdium;
                else
                        newspeed = slow;
            mdium:
                if (brake)
                        newspeed = slow;
                else if (accelerate)
                        newspeed = fast;
                else
                        newspeed = mdium;
                        fast:
                if (brake)
                        newspeed = mdium;
                else
                        newspeed = fast;
            default:
                newspeed = stop;
        endcase
    end

// controller
always @(posedge clk or negedge keys)
        begin
        if (!keys)
            speed = stop;
        else
            speed = newspeed;
        end

endmodule
```

Conclusion

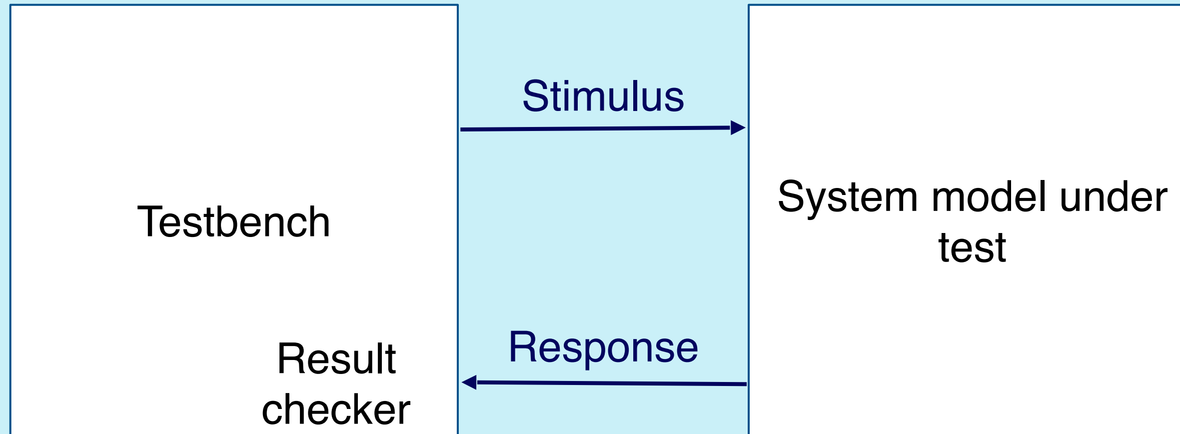
- Write codes which can be translated into hardware!
- Following cannot be translated into hardware (non-synthesizable)
  - initial blocks
    - Used to set up initial state or describe finite testbench stimuli
    - Don’t have obvious hardware component
  - Delays
    - Maybe in the Verilog source,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

Any Questions?

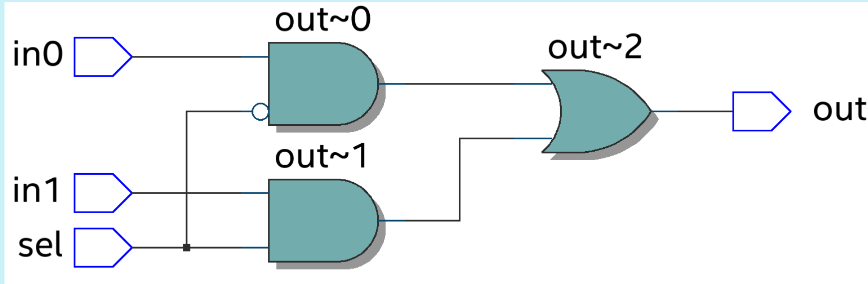
Behavioral Simulation

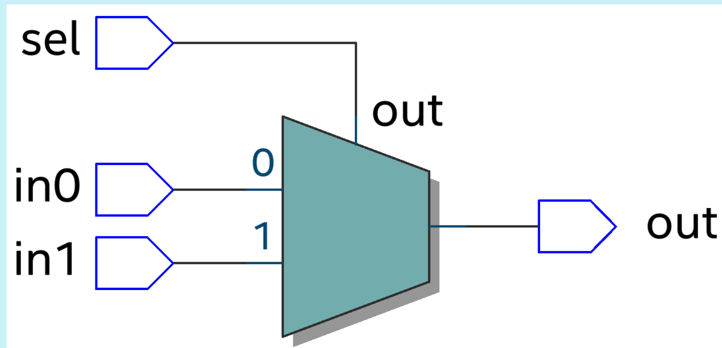
How do we test the behavior of a design?

- 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);

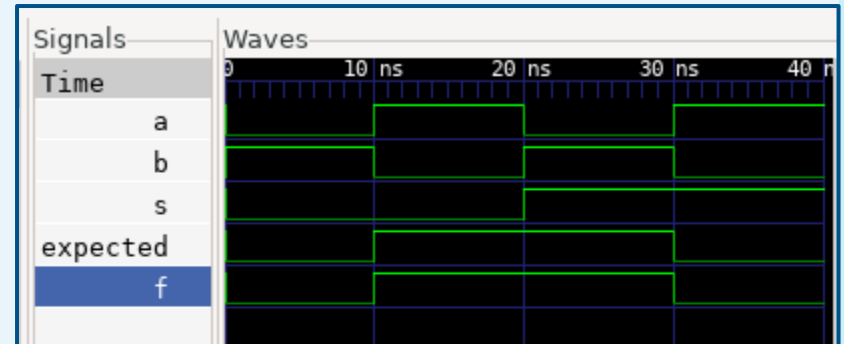
// alternative
// assign out = sel ? in1 : in0;

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
```



Any Questions?

Some more design examples

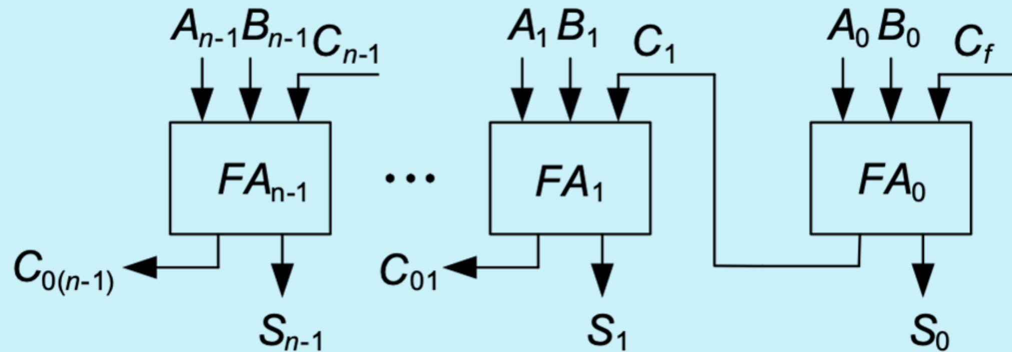
Recap…

- 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.

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));

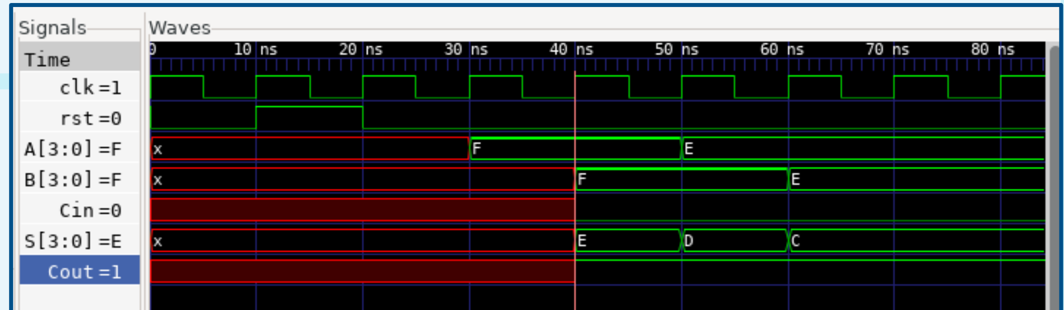
endmodule
```

```
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
```



Code for N-bit Ripple Carry Adder

```
// n-bit ripple carry adder
module nbit_rca #(parameter N = 32) (a, b, cin, s, cout);
input [N-1:0] a, b; input cin;
output [N-1:0] s; output cout;

wire [N:0] cr;
assign cr[0] = cin;

generate
        genvar i;
        for (i = 0; i < N; i = i + 1) begin : full_adder_instance
                full_adder fa (.a(a[i]), .b(b[i]), .cin(cr[i]), .s(s[i]), .cout(cr[i+1]));
        end
endgenerate

assign cout = cr[N];

endmodule
```

Testbench for N-bit Ripple Carry Adder

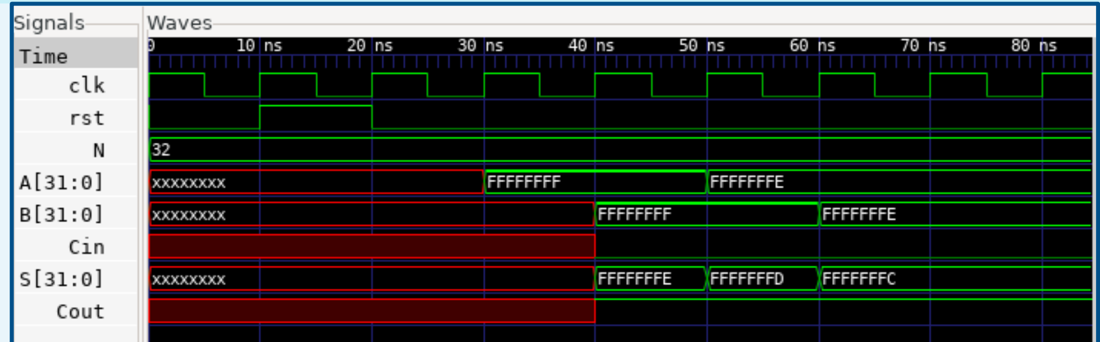
```
`timescale 1ns/1ps
module tb_nbit_rca;
parameter N = 32;

reg clk; reg rst;
reg [N-1:0] A; reg [N-1:0] B; reg Cin;
wire [N-1:0] S; wire Cout;

initial begin
	$display ("time\t, clk\t, rst\t, A\t, B\t, Cin\t, S\t, Cout");
	$monitor ("%g\t %b\t %b\t %d\t %d\t %b\t %d\t %b\t", $time, clk, rst, A, B, Cin, S, Cout);

	clk = 1; rst = 0;
	#10 rst = 1; #10 rst = 0;
	#10 A = 4294967295; #10 B = 4294967295; Cin = 0;
	#10 A = 4294967294; #10 B = 4294967294;
	#20; #5 $finish;
end

always begin
	#5 clk = ~clk;
end
```




```
nbit_rca #(N) dut (.a(A), .b(B), .cin(Cin), .s(S), .cout(Cout));

initial begin
	$dumpfile ("nbit_rca.vcd");
	$dumpvars (1, tb_nbit_rca);
end
endmodule
```

Counter

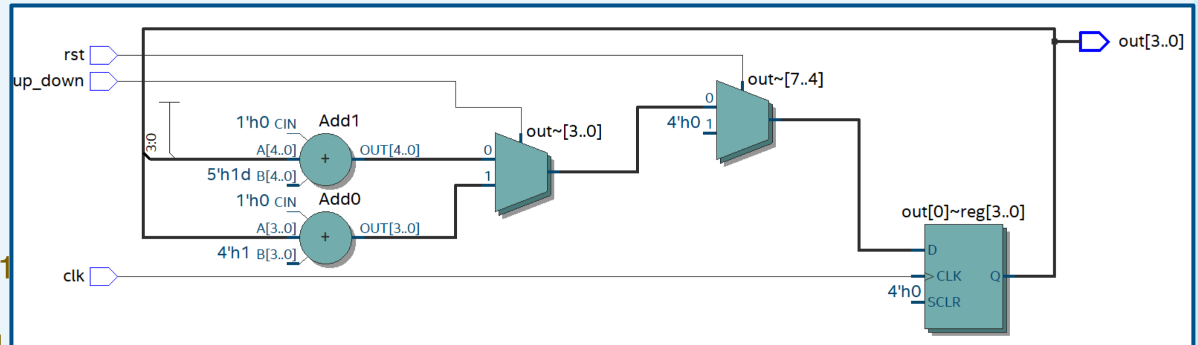
Code for 4-bit Up Down Counter

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

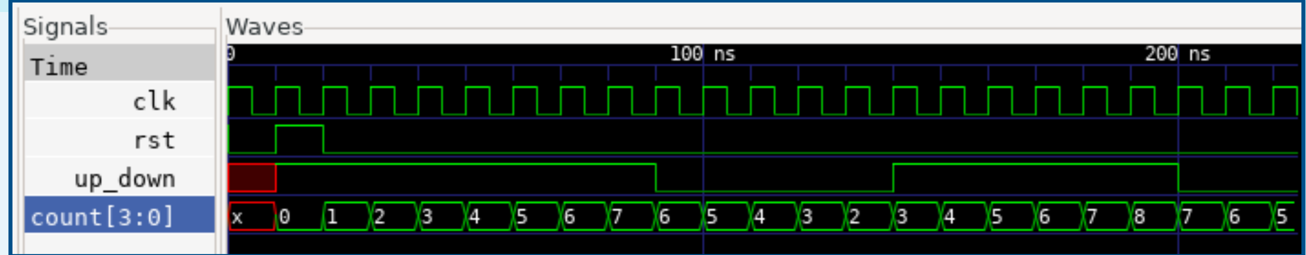
reg [3:0] out;

always @(posedge clk) begin
        if (rst)
                out <=  4’b0;
                else if (up_down)
                        out <=  out + 1
                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
```

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,

$$\mathbf{x = x_1B^m + x_0} \text{ and } \mathbf{y = y_1B^m + y_0}$$

where $\mathbf{x_0}$ and $\mathbf{y_0}$ are less than $\mathbf{B^m}$.

- The product is then,

$$\mathbf{xy = (x_1B^m + x_0)(y_1B^m + y_0)}$$

$$\mathbf{= x_1y_1B^{2m} + (x_1y_0 + x_0y_1)B^m + x_0y_0}$$

$$\mathbf{= z_2B^{2m} + z_1B^m + z_0}$$

Karatsuba algorithm

- The product is then,

$$xy = z_2B^{2m} + z_1B^m + z_0$$

where $z_2 = x_1y_1$, $z_1 = x_1y_0 + x_0y_1$, $z_0 = x_0y_0$

- Karatsuba's contribution:

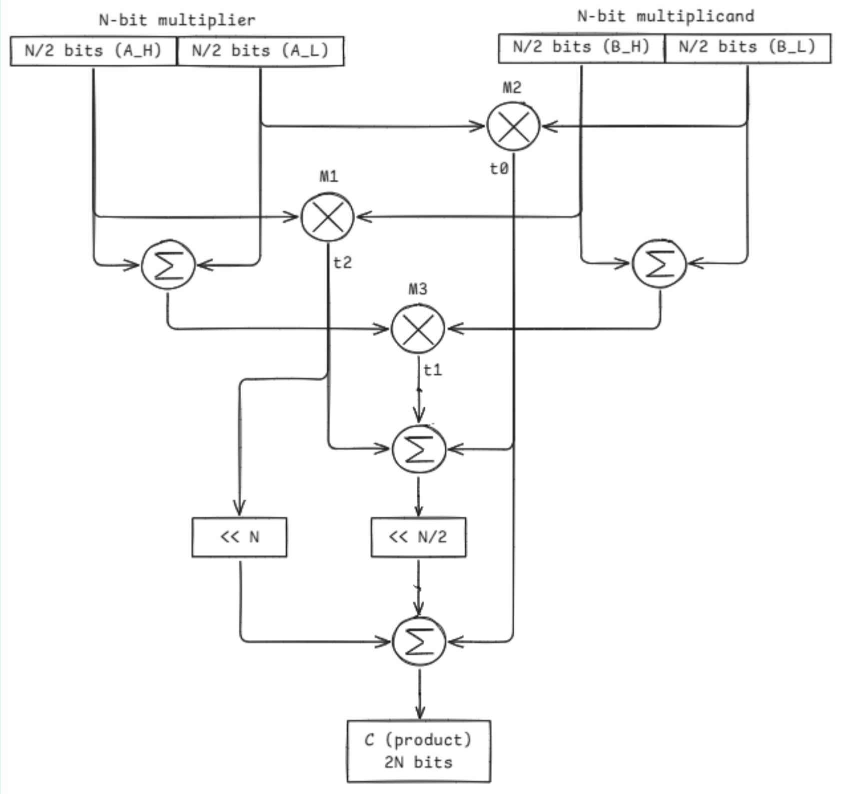
with z_2 and z_0 as before and $z_3 = (x_1 + x_0)(y_1 + y_0)$,

$$z_1 = x_1y_0 + x_0y_1$$

$$= (x_1 + x_0)(y_1 + y_0) - x_1y_1 - x_0y_0$$

$$= z_3 - z_2 - z_0$$

Datapath of Karatsuba multiplier



Composite Function

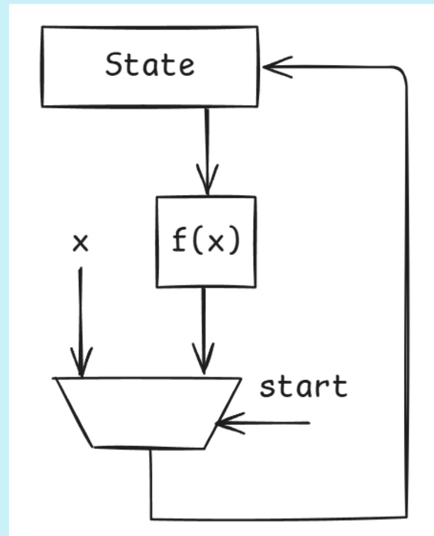
Problem statement

- 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++;
```

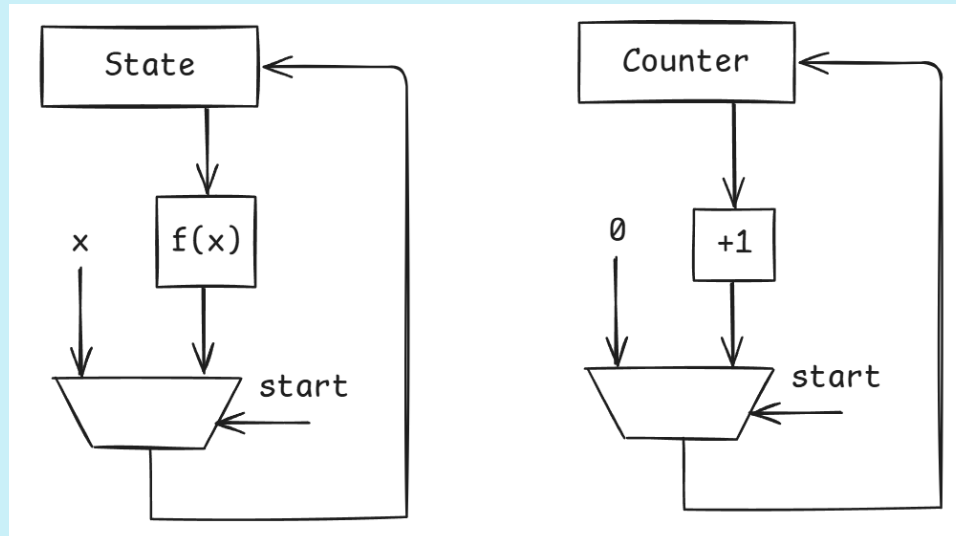
How to design the datapath?

- Let’s draw the design in steps



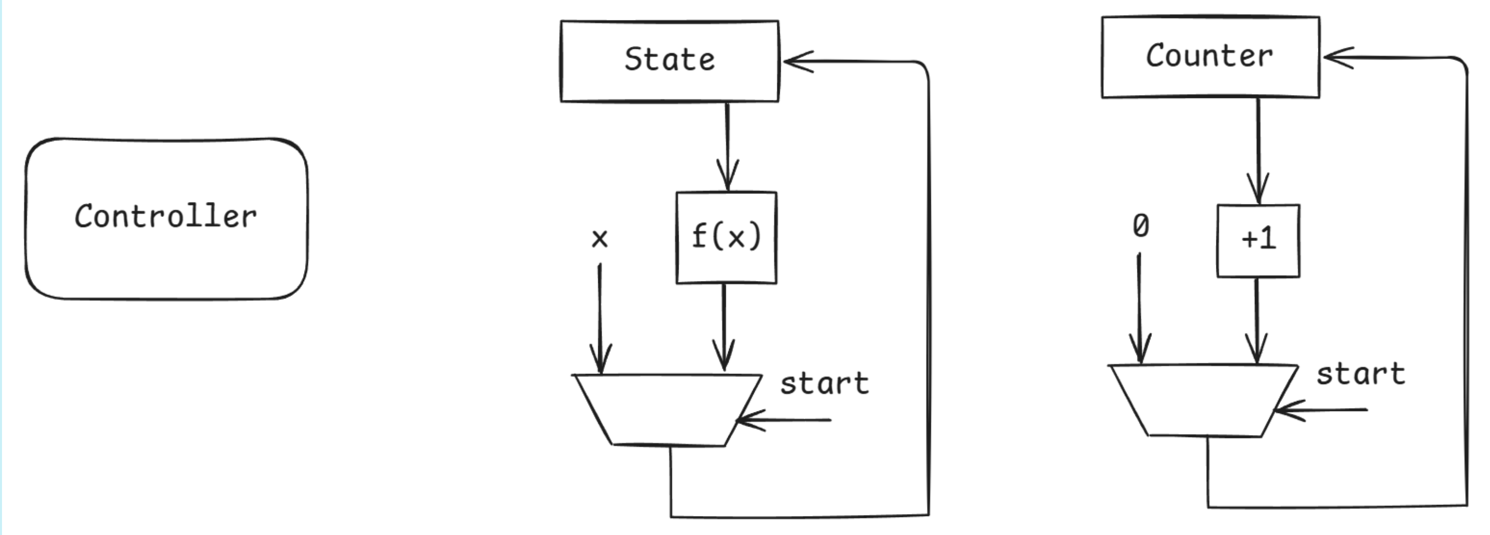
How to design the datapath?

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



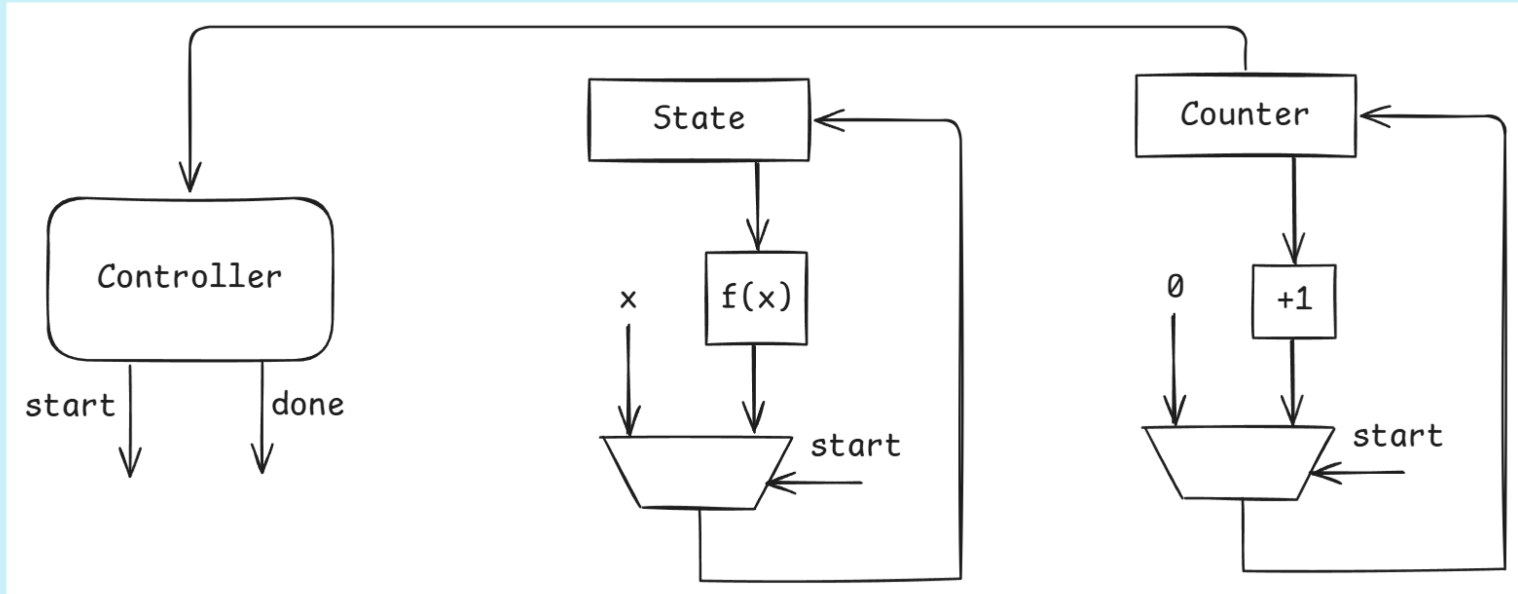
How to design the datapath?

- Now let's bring in the controller



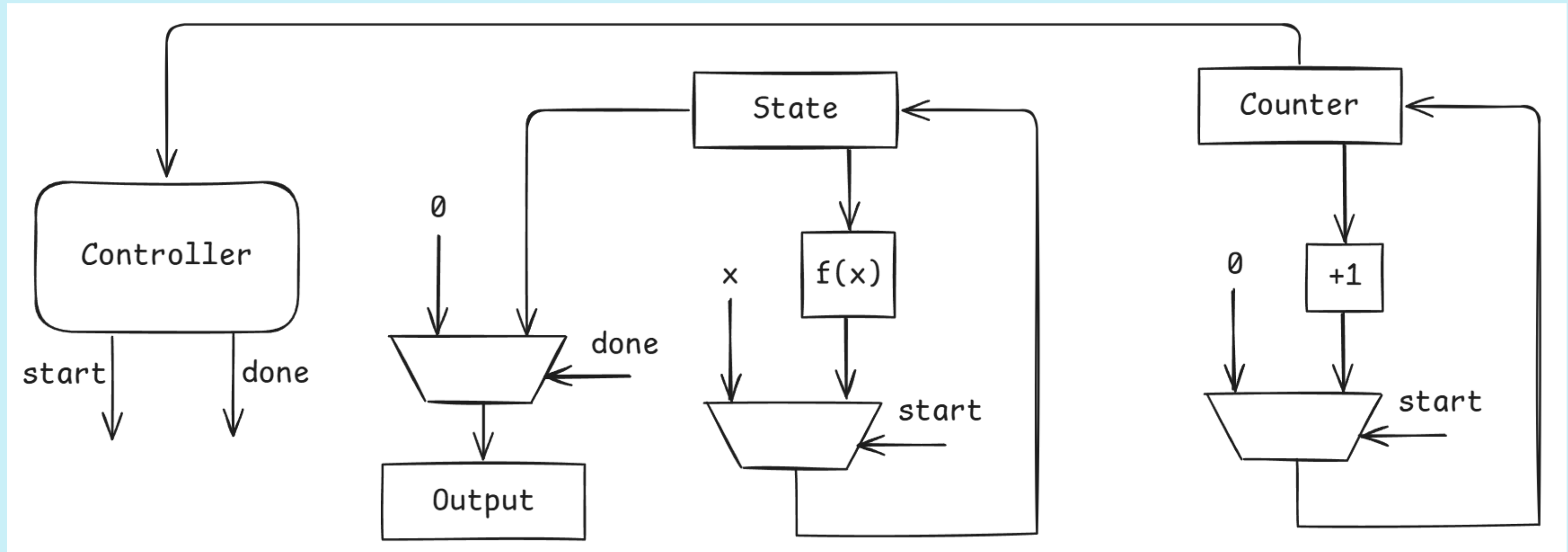
How to design the datapath?

- The controller stops the circuit while counter == 31



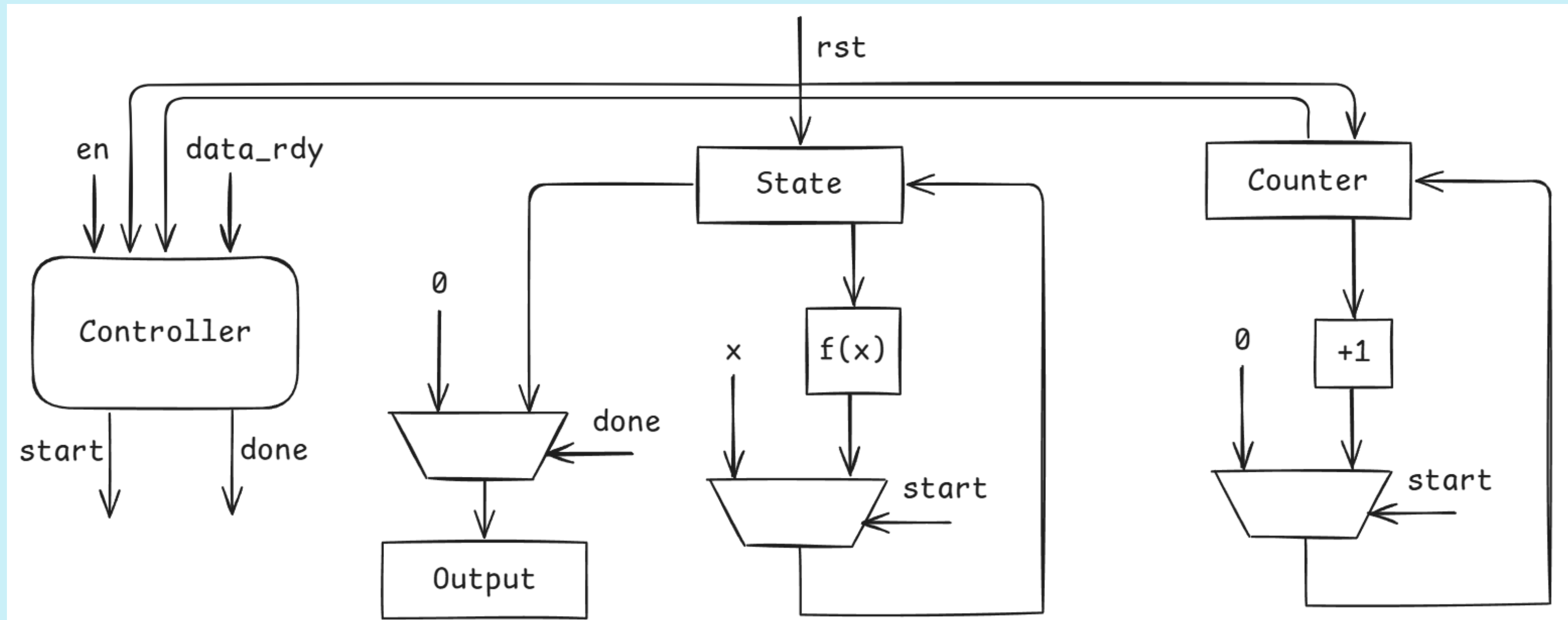
How to design the datapath?

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



How to design the datapath?

- Some more essential input signals



Thank You!