北京大學
PEKING UNIVERSITY

22530007

人工智能与芯片设计

2-Verilog HDL与数字集成电路基础

燕博南

2023秋

Computer Engineering Basics

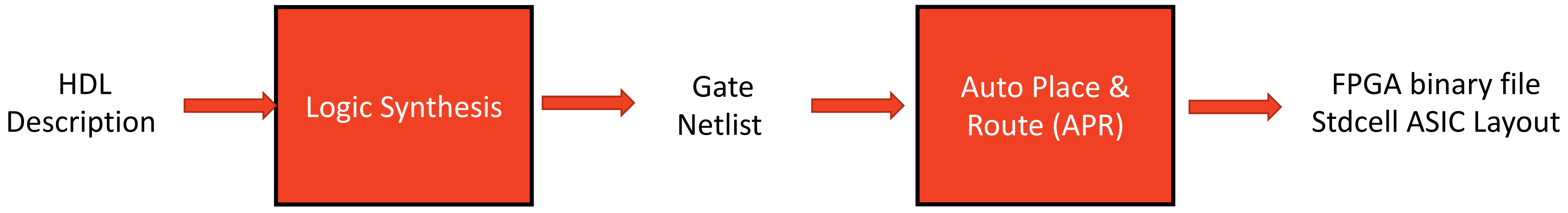
- Why AI chip?
- Logic Gates? What is computer hardware?
- Flip-Flops? Registers?
- How computer executes program?
- What is GPU?
- Difference between CPU, FPGA, GPU
- http://www.asic-world.com/verilog/veritut.html

A good tutorial for open-source simulation:
全平台轻量开源verilog仿真工具iverilog+GTKWave使用教程

Part 1

Verilog HDL grammar, operators, synthesizable design

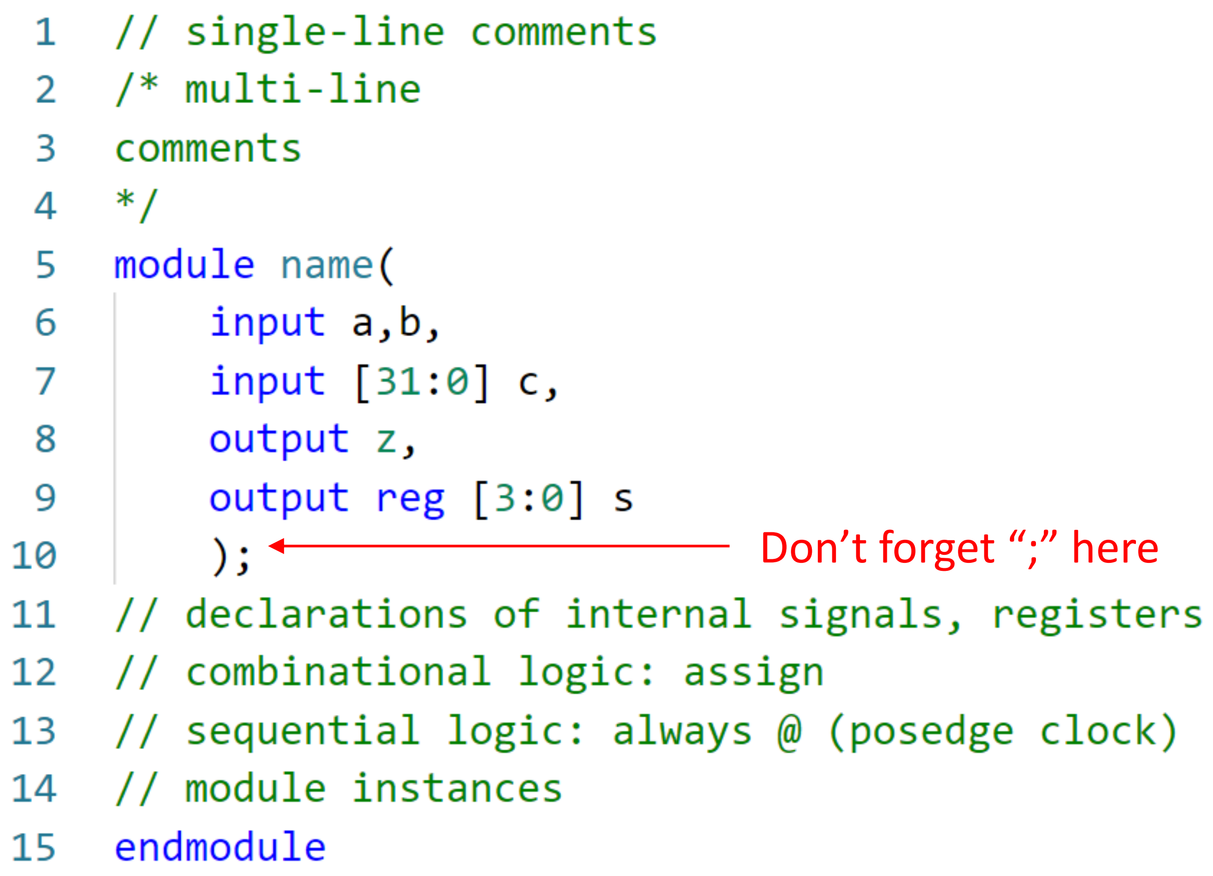
Digital Circuit Design Flow



- HDL -> Logic
- Map to target lib (stdcell/LUTs)
- Optimize speed, area

- Create floorplan blocks
- Place cells in blocks
- Route interconnect
- Optimize iteratively

Basic Building - Module



```verilog
// single-line comments
/* multi-line
comments
*/
module name(
    input a,b,
    input [31:0] c,
    output z,
    output reg [3:0] s
    );
// declarations of internal signals, registers
// combinational logic: assign
// sequential logic: always @ (posedge clock)
// module instances
endmodule
```

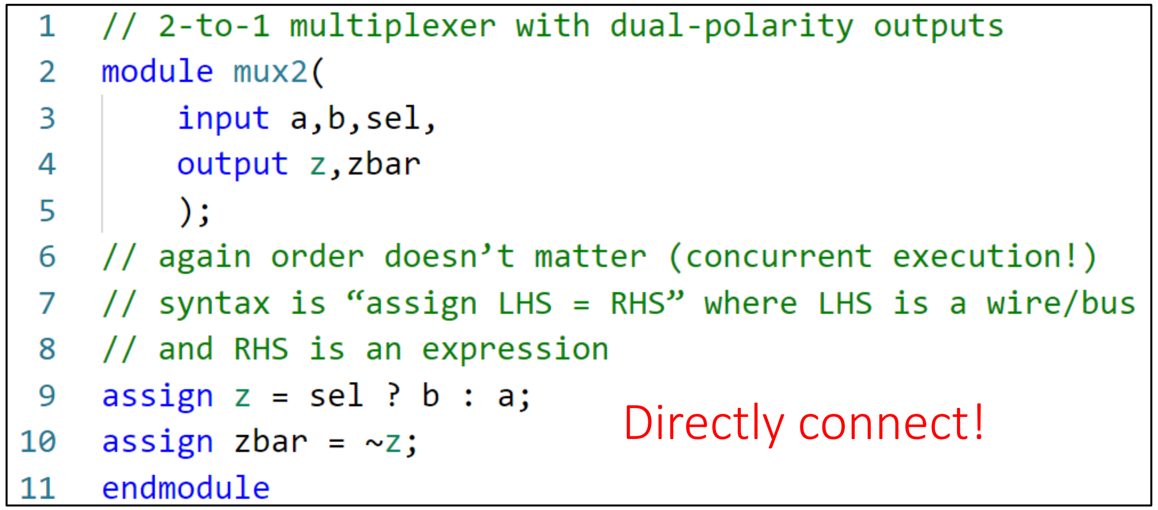
Don’t forget “;” here

In Verilog we design modules, one of which will be identified as our top-level module. Modules usually have named, directional ports (specified as input, output) which are used to communicate with the module.

- Format: HDL ignores space “ ”, it only recognize “;”

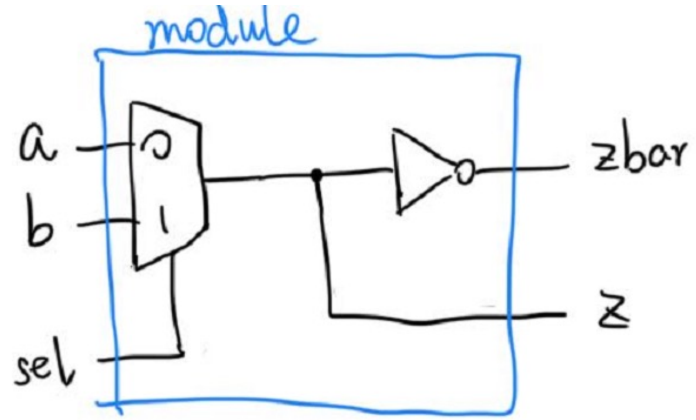
Wires & Registers

Wires

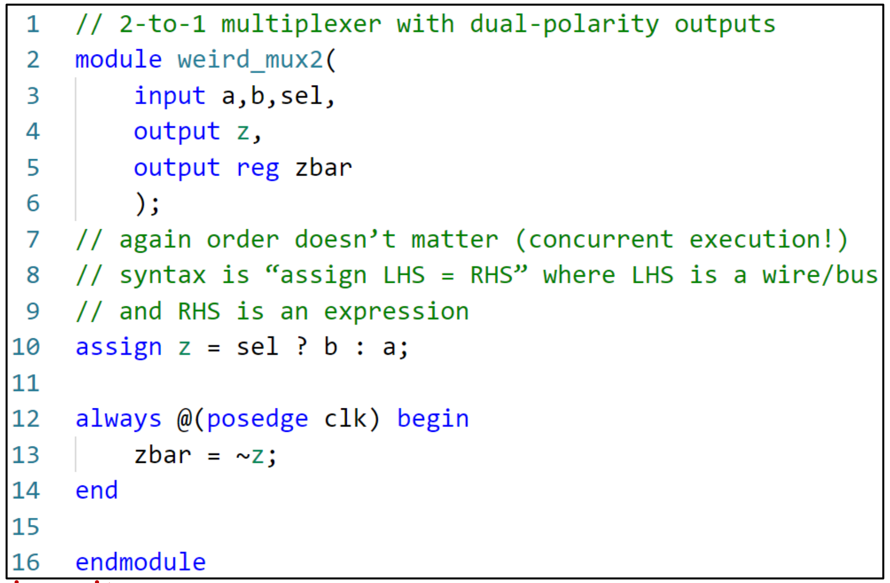


```
// 2-to-1 multiplexer with dual-polarity outputs
module mux2(
    input a,b,sel,
    output z,zbar
    );
// again order doesn’t matter (concurrent execution!)
// syntax is “assign LHS = RHS” where LHS is a wire/bus
// and RHS is an expression
assign z = sel ? b : a;
assign zbar = ~z;
endmodule
```

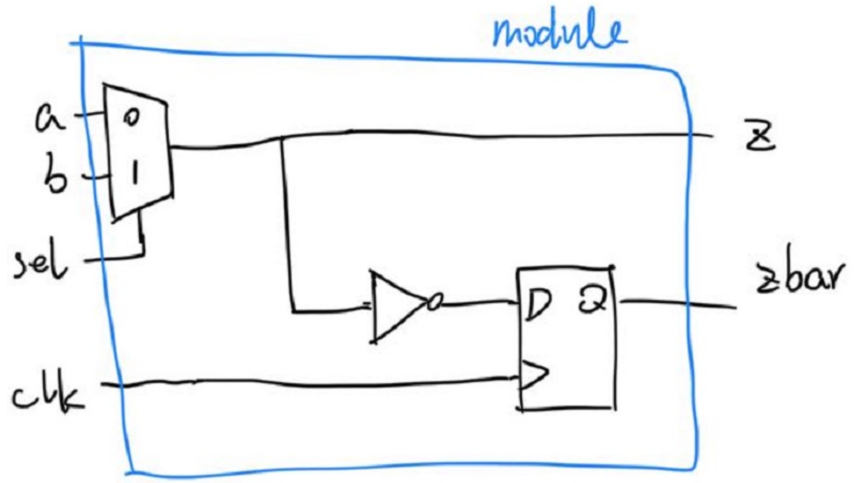
Directly connect!



Regs



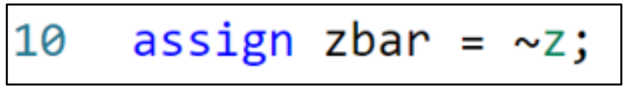
```
// 2-to-1 multiplexer with dual-polarity outputs
module weird_mux2(
    input a,b,sel,
    output z,
    output reg zbar
    );
// again order doesn’t matter (concurrent execution!)
// syntax is “assign LHS = RHS” where LHS is a wire/bus
// and RHS is an expression
assign z = sel ? b : a;

always @(posedge clk) begin
    zbar = ~z;
end

endmodule
```



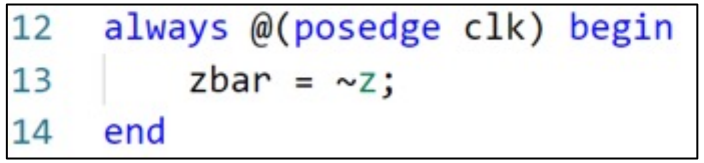
Secrets of Wires & Regs

- Without "reg" declaration, variables are always wires
- Reg can only be output and inside signals

- Assignment:
  - Wires can only be changed using "assign" outside "always"
  - Regs can only be changed inside "always"



```verilog
assign zbar = ~z;
```



```verilog
always @(posedge clk) begin
    zbar = ~z;
end
```

Operators

Arithmetic	*	Multiply
	/	Division
	+	Add
	-	Subtract
	%	Modulus
	+	Unary plus
	-	Unary minus
Logical	!	Logical negation
	&&	Logical and
	||	Logical or
Relational	>	Greater than
	<	Less than
	>=	Greater than or equal
	<=	Less than or equal
Equality	==	Equality
	!=	inequality
Shift	>>	Right shift
	<<	Left shift
	<<<, >>>	Arithmetic shift

Reduction	~	Bitwise negation
	~&	nand
	|	or
	~|	nor
	^	xor
	^~	xnor
	~^	xnor

Concatenation	{ }	Concatenation
Conditional	?	conditional

What are the difference between (logic) shift and arithmetic shift?

Numeric Constants

Constant values can be specified with a specific width and radix:

```
123 // default: decimal radix, 32-bit width
'd123 // 'd = decimal radix
'h7B // 'h = hex radix
'o173 // 'o = octal radix
'b111_1011 // 'b = binary radix, "_" are ignored
'hxx // can include X, Z or ? in non-decimal constants
16'd5 // 16-bit constant 'b0000_0000_0000_0101
11'h1X? // 11-bit constant 'b001_XXXX_ZZZZ
```

1'd3 ??

is 1'd1

By default constants are unsigned and will be extended with 0's on left if need be (if high-order bit is X or Z, the extended bits will be X or Z too).

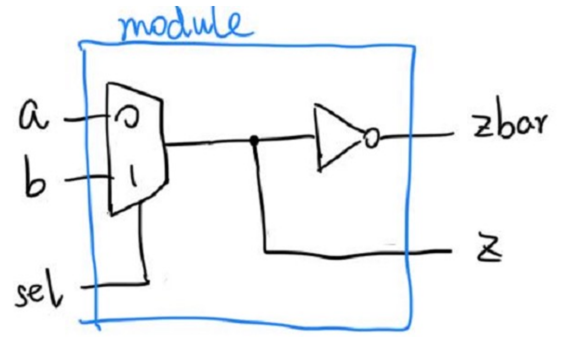
You can specify a signed constant as follows:

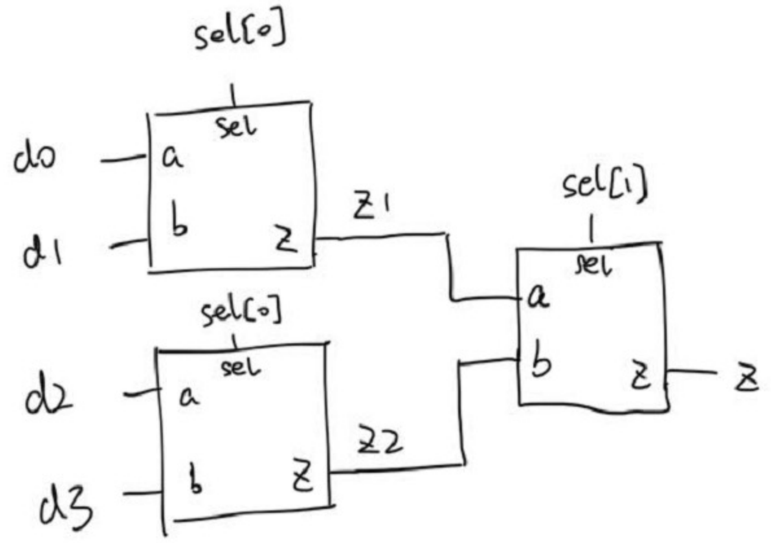
```
8'shFF // 8-bit twos-complement representation of -1
```

To be absolutely clear in your intent it's usually best to explicitlyspecify the width and radix.

Hierarchy: module instances

```
// 4-to-1 multiplexer
module mux4(input d0,d1,d2,d3, input [1:0] sel, output z);
 wire z1,z2;
 // instances must have unique names within current module.
 // connections are made using .portname(expression) syntax.
 // once again order doesn’t matter…
 mux2 m1(.sel(sel[0]),.a(d0),.b(d1),.z(z1)); // not using zbar
 mux2 m2(.sel(sel[0]),.a(d2),.b(d3),.z(z2));
 mux2 m3(.sel(sel[1]),.a(z1),.b(z2),.z(z));
 // could also write “mux2 m3(z1,z2,sel[1],z,)” NOT A GOOD IDEA!
endmodule
```



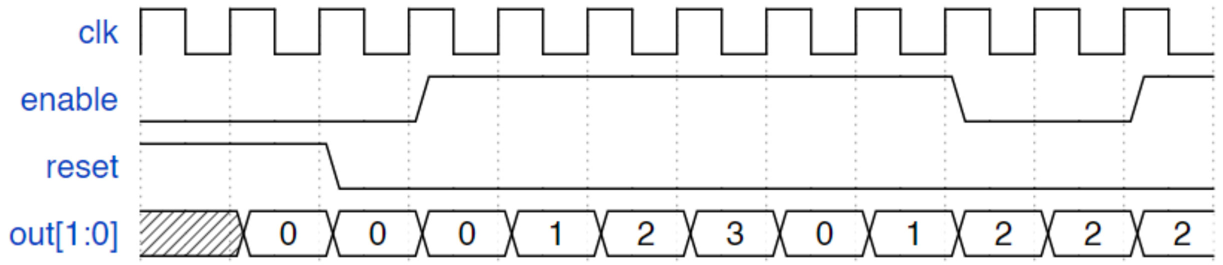


- Write all original names (style requirement)
- Connection are concurrently executed

Example 1: A counter

```verilog
module counter (
out      ,  // Output of the counter
enable   ,  // enable for counter
clk      ,  // clock Input
reset       // reset Input
);

output [1:0] out; //Output Ports
input enable, clk, reset; //Input Ports

reg [1:0] out; //Internal Variables

always @(posedge clk)
if (reset) begin
  out <= 2'b0 ;
end else if (enable) begin
  out <= out + 1;
end

endmodule
```

What is the timing diagram?



- Beware of “before” & “after” clock edge

Verification: Simulation & Testbench

Design:

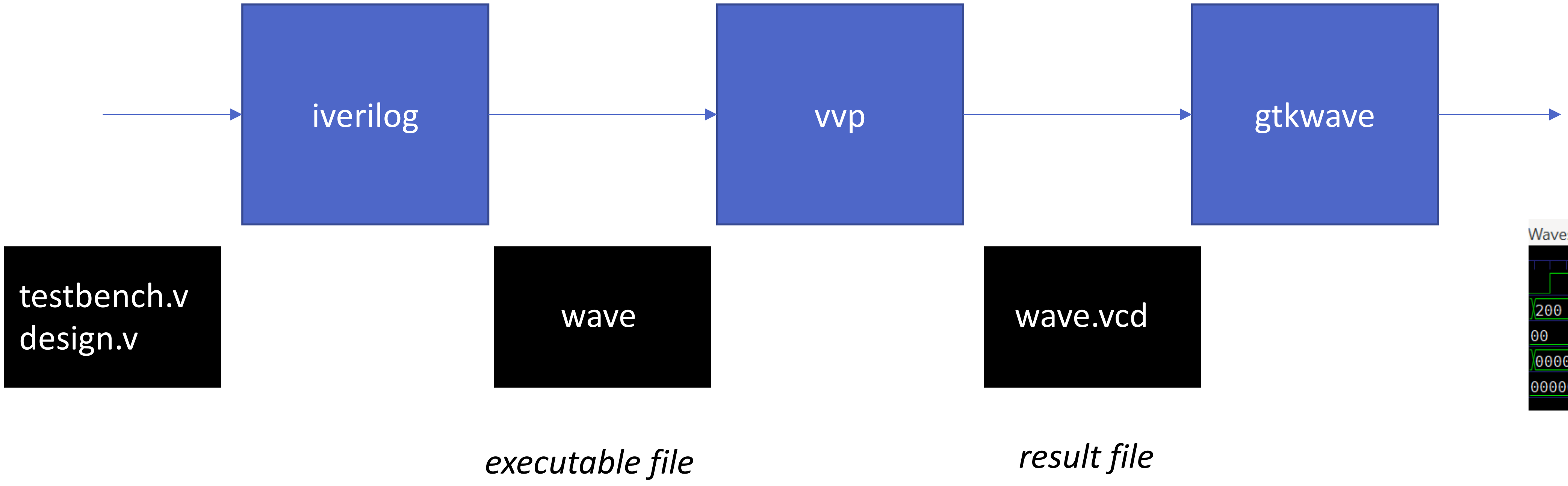
```
module counter (
out      ,  // Output of the counter
enable   ,  // enable for counter
clk      ,  // clock Input
reset       // reset Input
);

output [1:0] out; //Output Ports
input enable, clk, reset; //Input Ports

reg [1:0] out; //Internal Variables

always @(posedge clk)
if (reset) begin
  out <= 2'b0 ;
end else if (enable) begin
  out <= out + 1;
end

endmodule
```

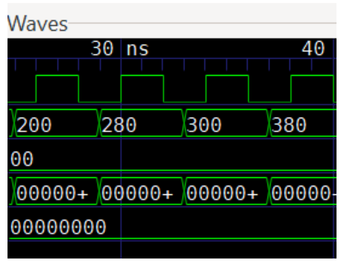
Testbench:

```
`timescale 1ns/1ps
module Testbench;

wire [1:0] OUT;
reg EN, CLK, RST;

initial CLK = 0;
always #2 CLK=~CLK;

initial begin
    #1
    EN = 0;
    RST = 1;
    #4
    EN = 1;
    RST = 0;
    #(4*7)
    EN = 0;
end

counter u1(
.out (OUT)     ,  // Output of the counter
.enable (EN) ,  // enable for counter
.clk  (CLK)    ,  // clock Input
.reset  (RST)     // reset Input
);

endmodule
```

Testbench has no ports

Verilog Simulator Workflow

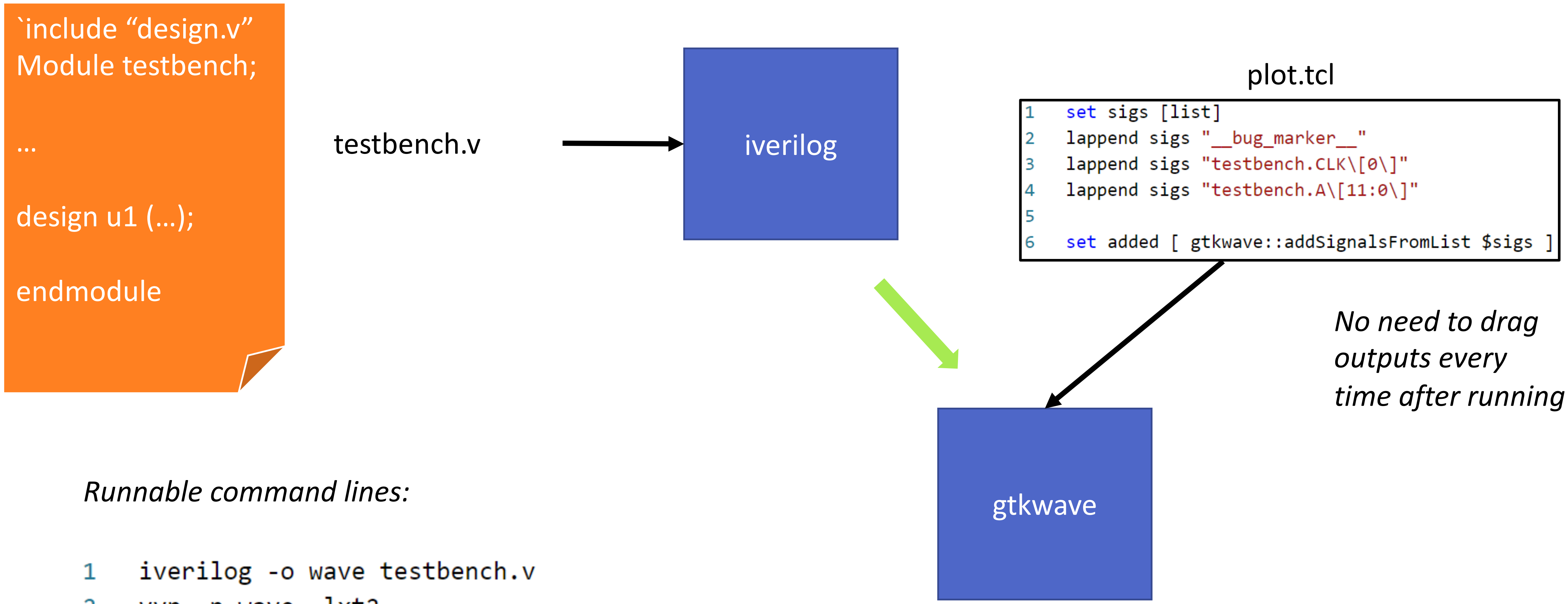




Runnable commend lines:

```
iverilog -o wave testbench.v
vvp -n wave -lxt2
gtkwave wave.vcd -S plot.tcl
```

Verilog Simulator Workflow



```
`include "design.v"
Module testbench;

…

design u1 (…);

endmodule
```

```
set sigs [list]
lappend sigs "__bug_marker__"
lappend sigs "testbench.CLK\[0\]"
lappend sigs "testbench.A\[11:0\]"

set added [ gtkwave::addSignalsFromList $sigs ]
```

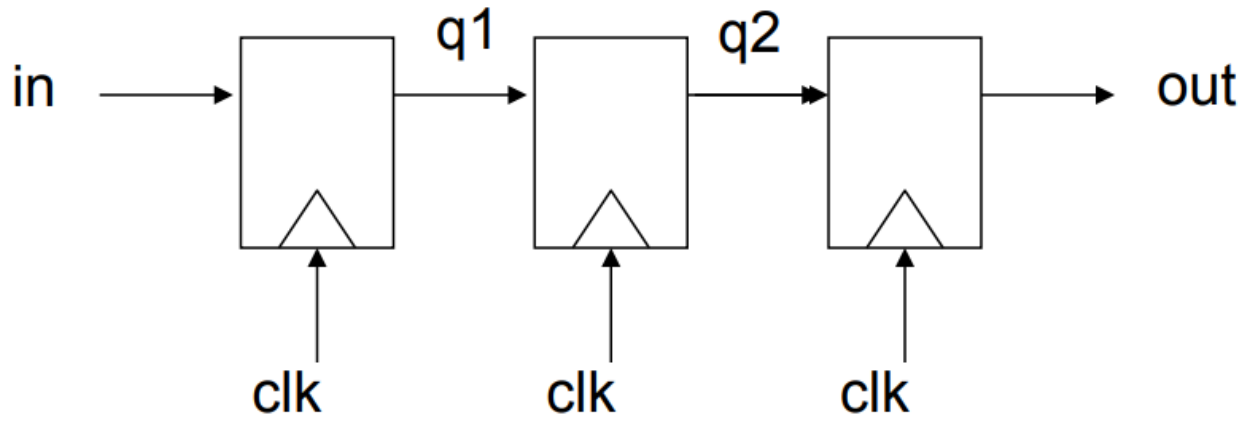
No need to drag outputs every time after running

Runnable command lines:

```
iverilog -o wave testbench.v
vvp -n wave -lxt2
gtkwave wave.vcd -S plot.tcl
```

A good tutorial:
全平台轻量开源verilog仿真工具iverilog+GTKWave使用教程

Blocking and Non-blocking



```
// shift register
reg q1,q2,out;
always @(posedge clk) begin
  q1 = in;
  q2 = q1;
  out = q2;
end
```

Wrong!

```
// shift register
reg q1,q2,out;
always @(posedge clk) q1 <= in;
always @(posedge clk) q2 <= q1;
always @(posedge clk) out <= q2;
```

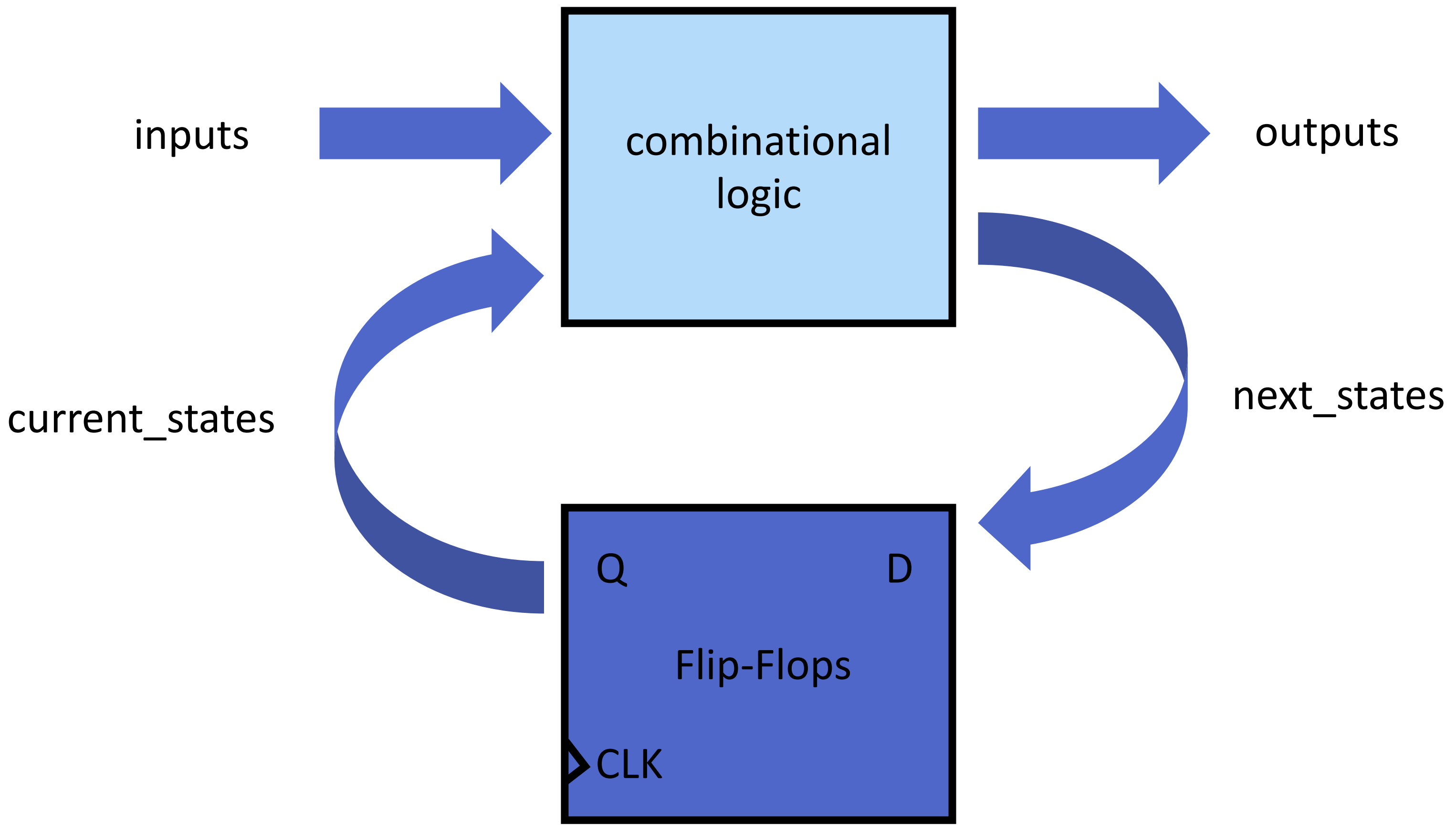
Correct!

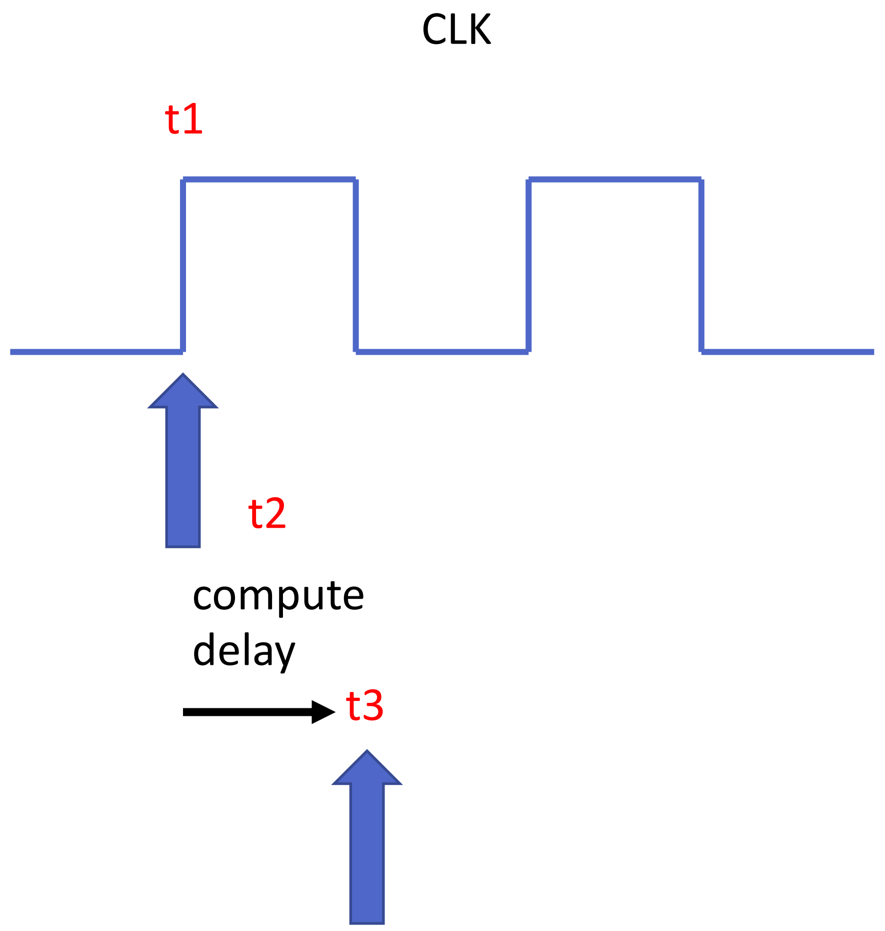
Part 2

Finite State Machine

Finite State Machine

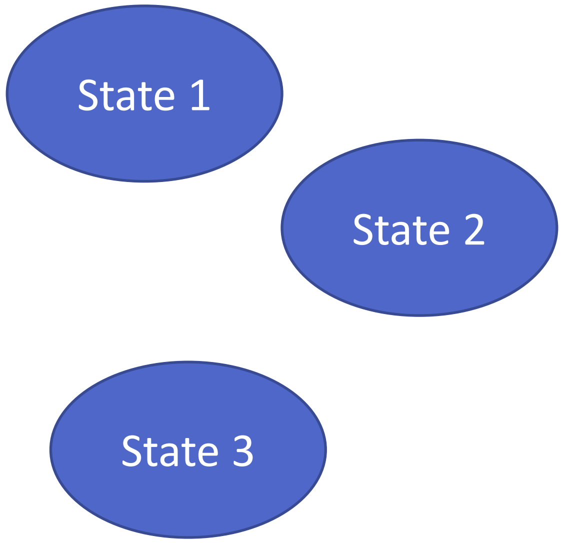
- **Hardware/Circuits**



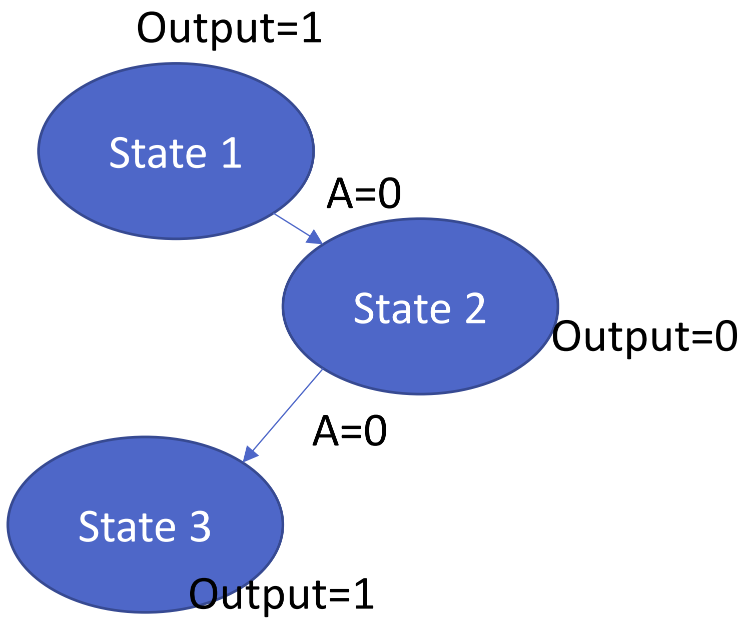


Time point: t1, t2, t3

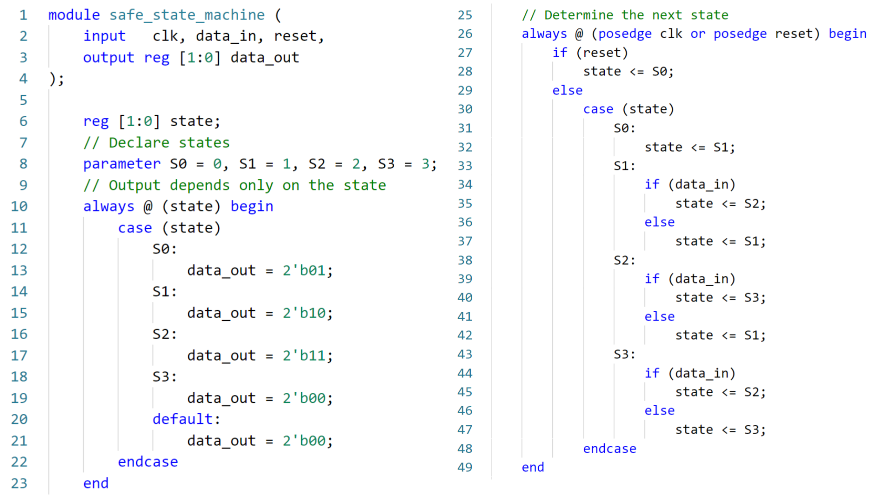
Design Methodologies & Templates



Step 1: define states



Step 2: draw state-transfer diagram



```verilog
module safe_state_machine (
    input   clk, data_in, reset,
    output reg [1:0] data_out
);

    reg [1:0] state;
    // Declare states
    parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3;
    // Output depends only on the state
    always @ (state) begin
        case (state)
            S0:
                data_out = 2'b01;
            S1:
                data_out = 2'b10;
            S2:
                data_out = 2'b11;
            S3:
                data_out = 2'b00;
            default:
                data_out = 2'b00;
        endcase
    end
```

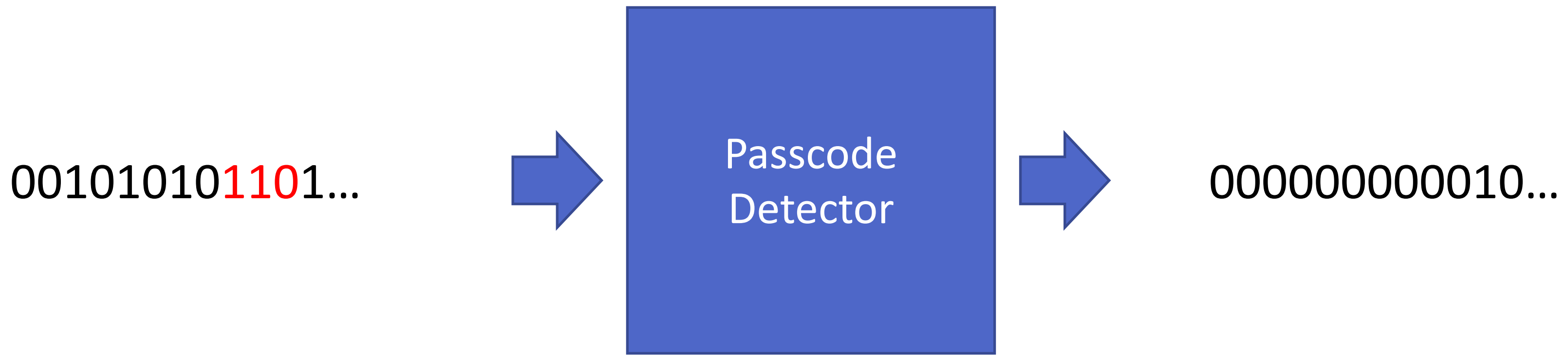
```verilog
    // Determine the next state
    always @ (posedge clk or posedge reset) begin
        if (reset)
            state <= S0;
        else
            case (state)
                S0:
                    state <= S1;
                S1:
                    if (data_in)
                        state <= S2;
                    else
                        state <= S1;
                S2:
                    if (data_in)
                        state <= S3;
                    else
                        state <= S1;
                S3:
                    if (data_in)
                        state <= S2;
                    else
                        state <= S3;
            endcase
    end
```

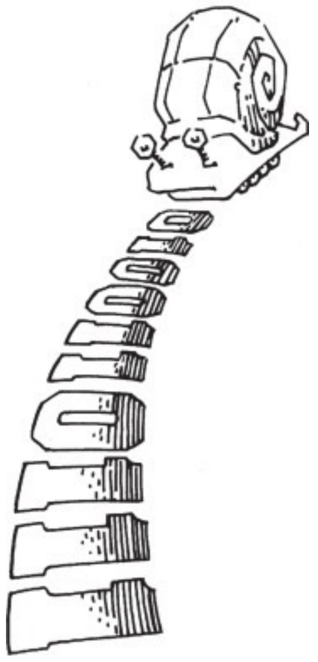
Step 3: fill in the template

Verilog HDL Templates for State Machines (intel.com)

FSM Example 1 Passcode Detector

Question: build circuits that outputs 1 pulse, whenever receives “110”

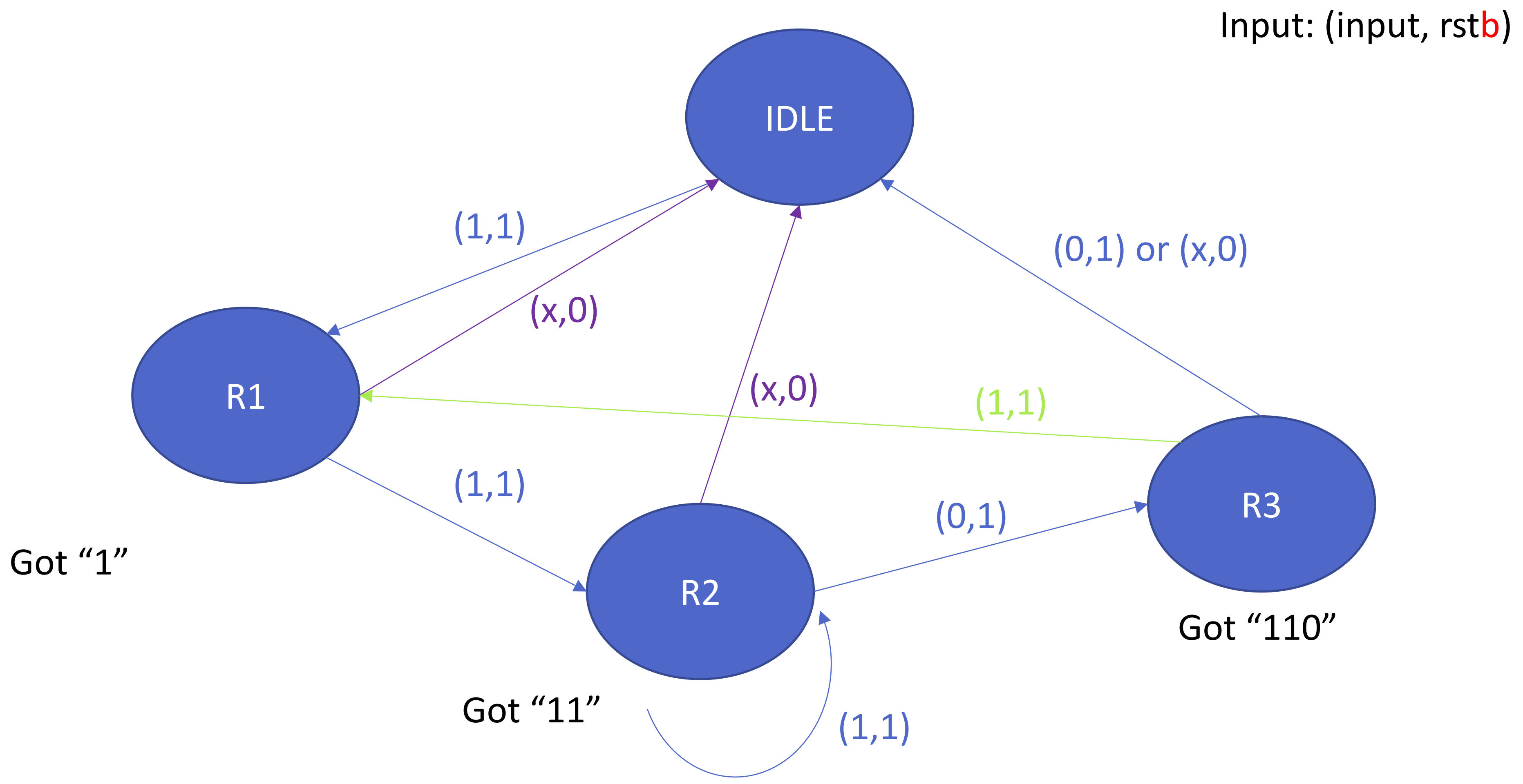




Sarah L. Harris, David Harris, in Digital Design and Computer Architecture, 2022

FSM Example 1 – Step 1 & 2

Question: build circuits that outputs 1 pulse, whenever receives "110"



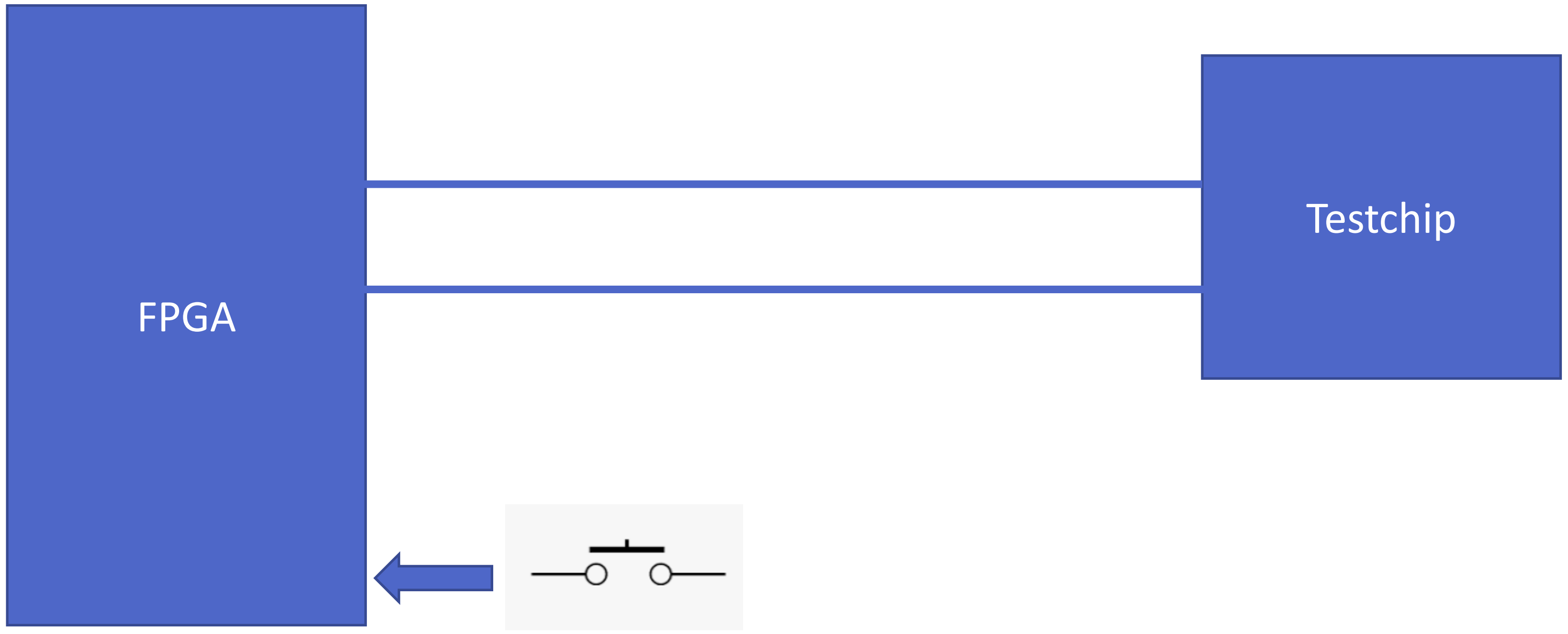
FSM Example 1 – Step 3

Question: build circuits that outputs 1 pulse, whenever receives “110”

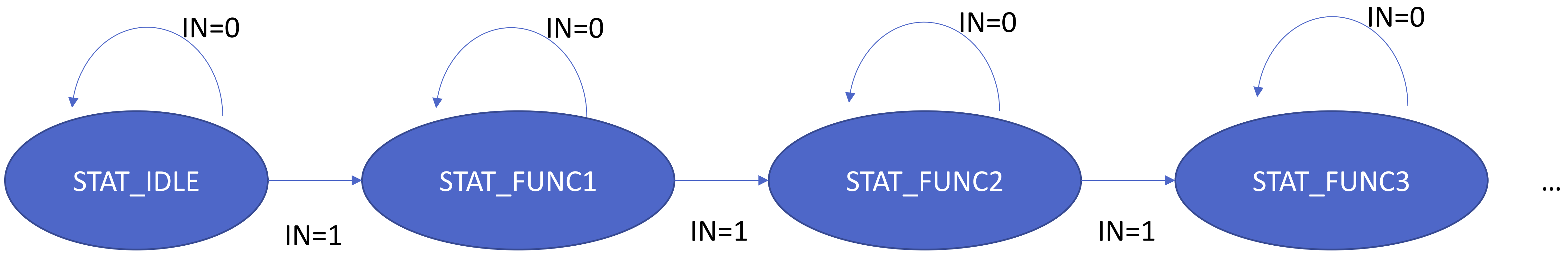
```
module PasscodeDetector (
    input   clk, data_in, rstb,
    output reg data_out
);

    reg [1:0] state;
    // Declare states
    parameter STAT_IDLE = 0,
              STAT_R1 = 1,
              STAT_R2 = 2,
              STAT_R3 = 3;
    // Output depends only on the state
    always @ (state) begin
        if(state == STAT_R3) begin
            data_out <= 1; // alarming!
        end
        else begin
            data_out <= 0;
        end
    end
    // Determine the next state
    always @ (posedge clk) begin
        if (~rstb)
            state <= STAT_IDLE;
        else
            case (state)
                STAT_IDLE: begin
                    if(data_in==1) begin
                        state <= STAT_R1;
                    end
                end
                STAT_R1: begin
                    if (data_in==1)
                        state <= STAT_R2;
                    else
                        state <= STAT_IDLE;
                end
                STAT_R2: begin
                    if (data_in==0)
                        state <= STAT_R3;
                    else
                        state <= STAT_R2;
                end
                STAT_R3: begin
                    if(data_in==1) begin
                        state <= STAT_R1;
                    end
                    else begin
                        state <= STAT_IDLE;
                    end
                end
            endcase
    end
endmodule
```

FSM Example 2 Auto Chip Testing Environment

Push button to test Function 1, Function 2, Function 3, … in series



FSM Example 2 Step 1&2



Step 3 is so easy… that you can fill it yourself.

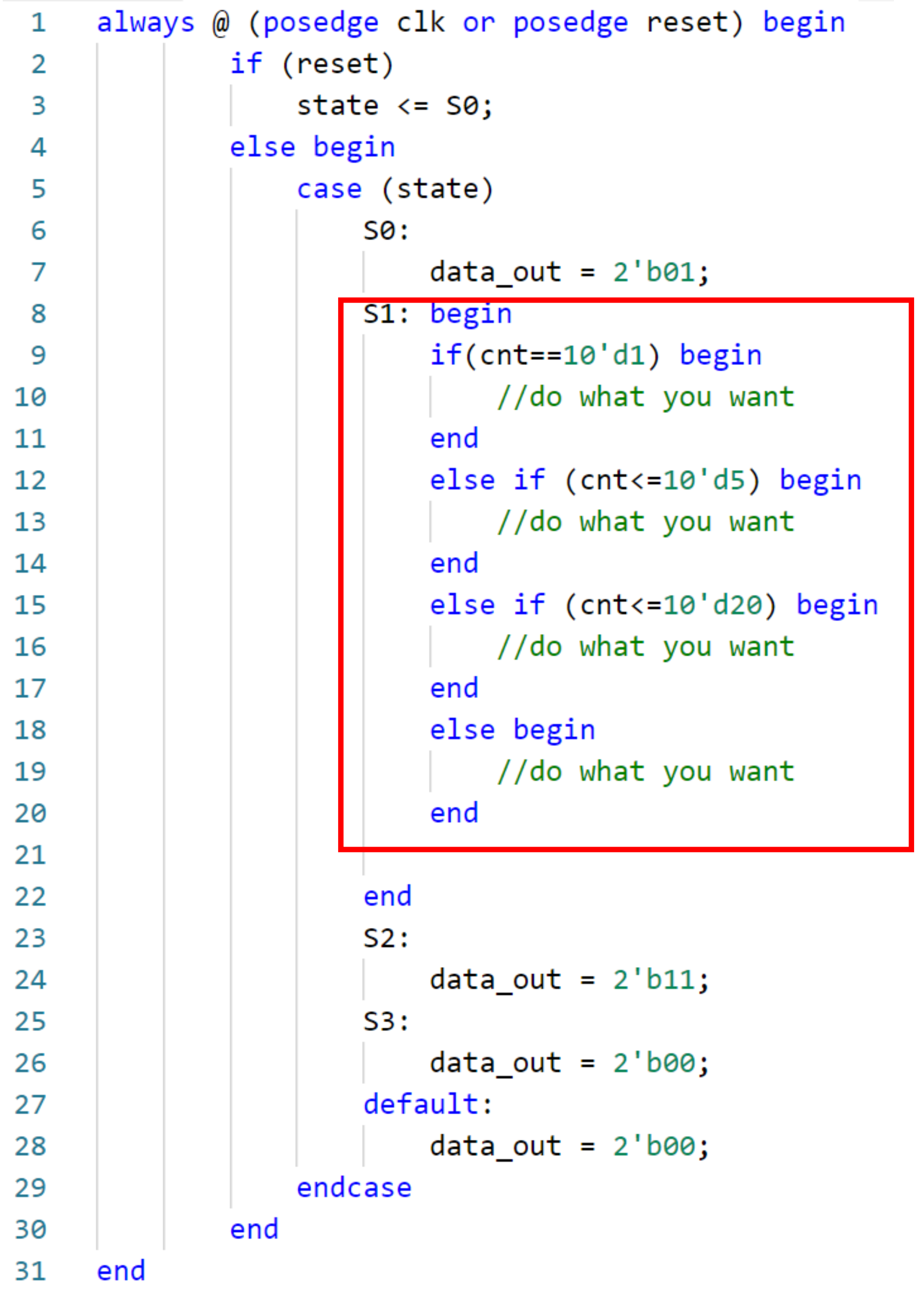
Another Question

Philosophy behind:
Use “state variable” to label timing

Another Q:
How to generate custom waveform after entering some state?

Solution: setup an counter variable
Do everything at its pace
[! DO NOT ABUSE. This may cause large comparing logic.]

Possibility of Nested State Machine!



```verilog
always @ (posedge clk or posedge reset) begin
    if (reset)
      state <= S0;
    else begin
      case (state)
        S0:
          data_out = 2'b01;
        S1: begin
          if(cnt==10'd1) begin
            //do what you want
          end
          else if (cnt<=10'd5) begin
            //do what you want
          end
          else if (cnt<=10'd20) begin
            //do what you want
          end
          else begin
            //do what you want
          end

        end
        S2:
          data_out = 2'b11;
        S3:
          data_out = 2'b00;
        default:
          data_out = 2'b00;
      endcase
    end
end
```

A practical application – Vending Machine

All selections are ￥0.30

The machine make changes

Inputs:

- ￥0.25
- ￥0.10
- ￥0.05

Outputs

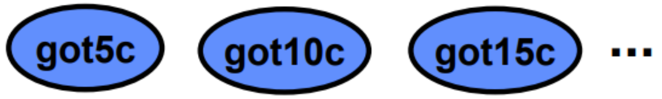
- Dispense can
- Dispense ￥0.10
- Dispense ￥0.05

Example from MIT

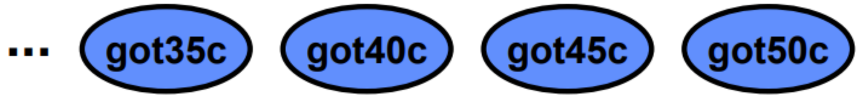
A practical application – Vending Machine

- **A starting (idle) state:**

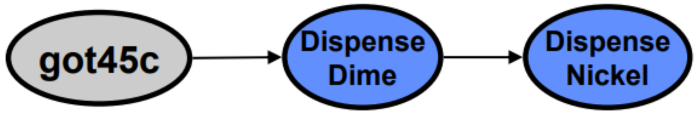
- **A state for each possible amount of money captured:**



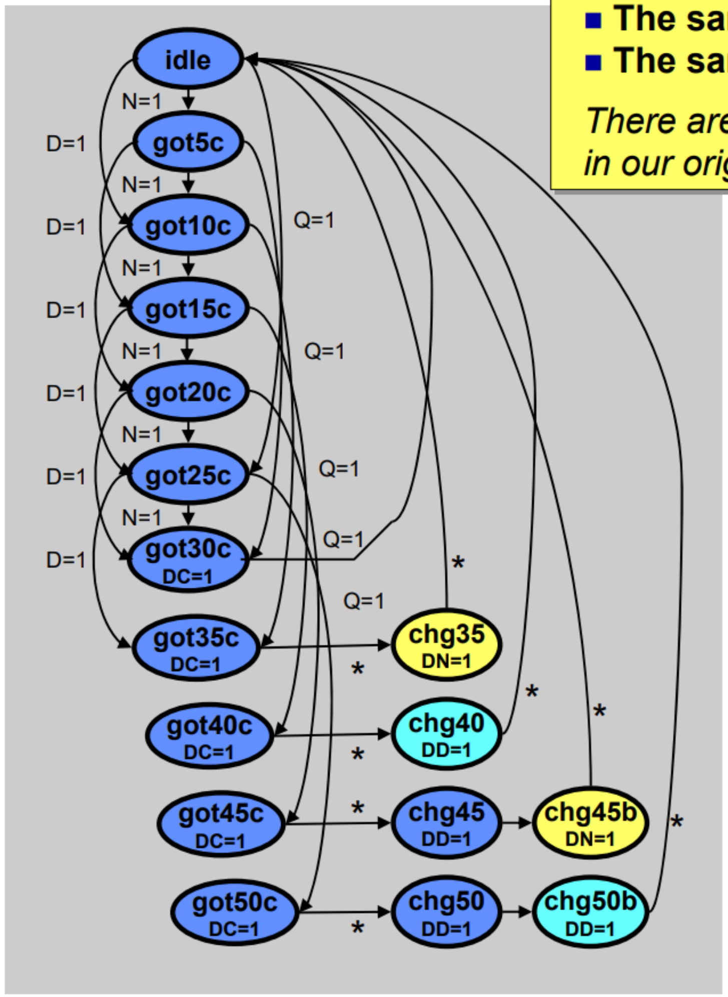
- **What's the maximum amount of money captured before purchase?**
  25 cents (just shy of a purchase) + one quarter (largest coin)



- **States to dispense change (one per coin dispensed):**



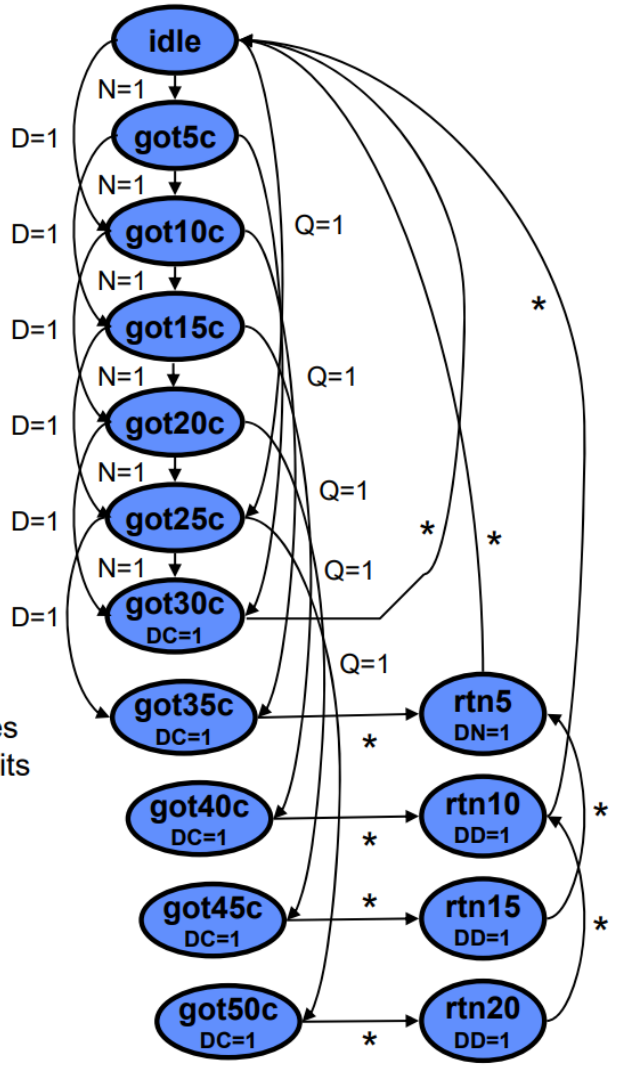
A practical application – Vending Machine



Duplicate states have:

- The same outputs, and
- The same transitions

There are two duplicates in our original diagram.



17 states
5 state bits

15 states
4 state bits

Discussion

- State Machine vs. AI
    - State machines exhaustively cover all cases, which is impossible in AI agent design.
- State machine is appropriate for small scale designs.
- Nested state machines:
    - normally, do not nest inside FSM in >2 folds.
    - Otherwise, too complicated timing path dependency.