北京大学
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

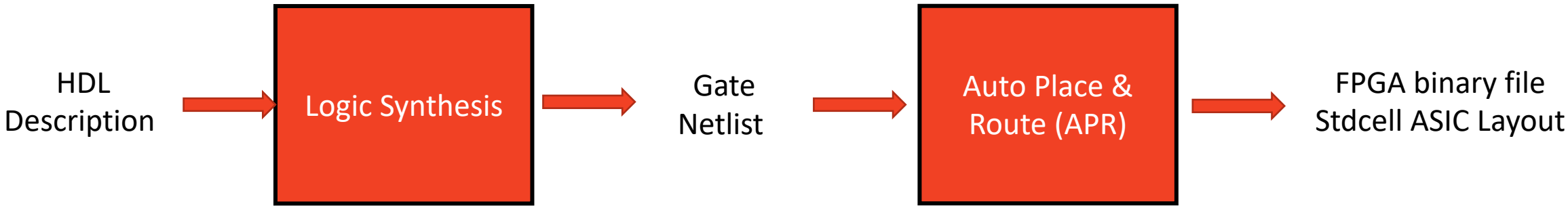
Computer Engineering Basics

- 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

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

```
1   // single-line comments
2   /* multi-line
3   comments
4   */
5   module name(
6       input a,b,
7       input [31:0] c,
8       output z,
9       output reg [3:0] s
10      );
11  // declarations of internal signals, registers
12  // combinational logic: assign
13  // sequential logic: always @ (posedge clock)
14  // module instances
15  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.

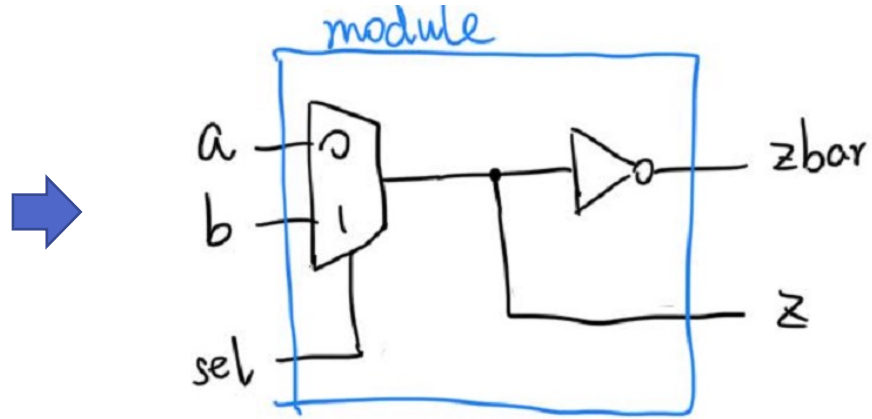
- Format: HDL ignores space “ ”, it only recognize “;”

Wires & Registers

Wires

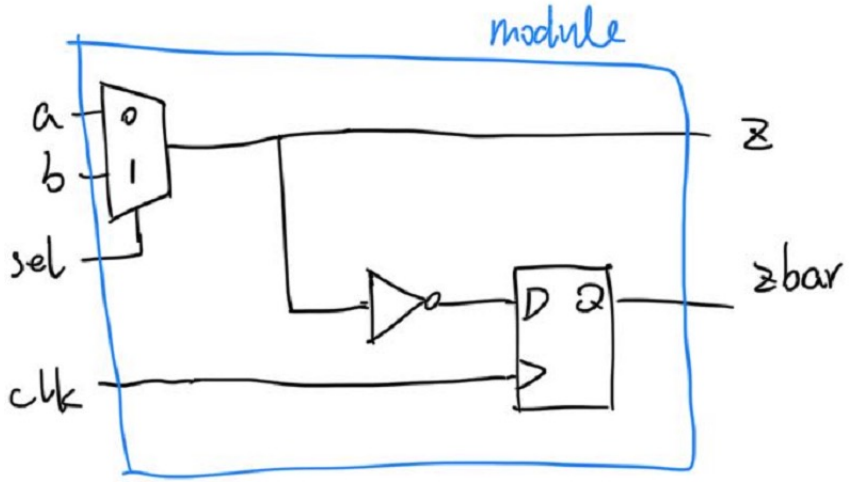
```
1  // 2-to-1 multiplexer with dual-polarity outputs
2  module mux2(
3      input a,b,sel,
4      output z,zbar
5      );
6  // again order doesn't matter (concurrent execution!)
7  // syntax is "assign LHS = RHS" where LHS is a wire/bus
8  // and RHS is an expression
9  assign z = sel ? b : a;
10 assign zbar = ~z;
11 endmodule
```

Directly connect!



Regs

```
1  // 2-to-1 multiplexer with dual-polarity outputs
2  module weird_mux2(
3      input a,b,sel,
4      output z,
5      output reg zbar
6      );
7  // again order doesn't matter (concurrent execution!)
8  // syntax is "assign LHS = RHS" where LHS is a wire/bus
9  // and RHS is an expression
10 assign z = sel ? b : a;
11
12 always @(posedge clk) begin
13     zbar = ~z;
14 end
15
16 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”

```
10   assign zbar = ~z;
```

```
12   always @(posedge clk) begin
13       zbar = ~z;
14   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 ??

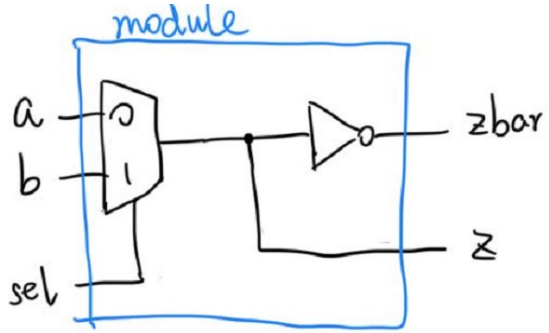
is 1'd1

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).

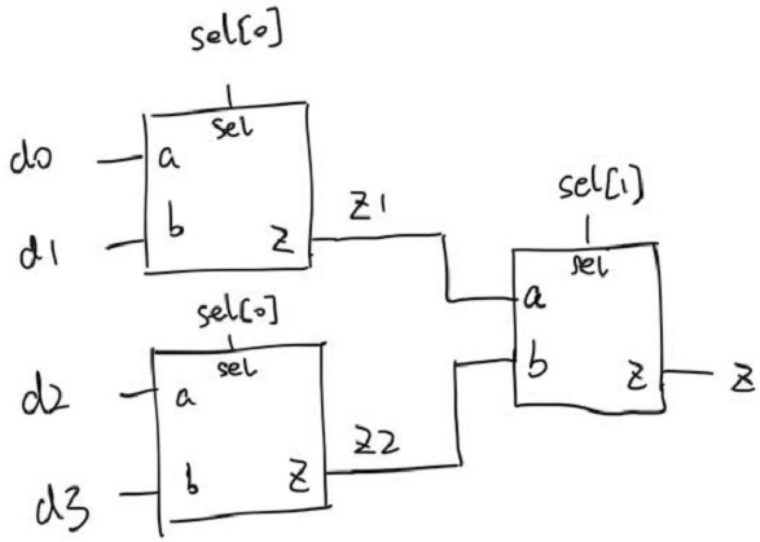
You can specify a signed constant as follows:
8'shFF // 8-bit twos-complement representation of -1
To be absolutely clear in your intent it's usually best to explicitlyspecify the width and radix.

Hierarchy: module instances

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



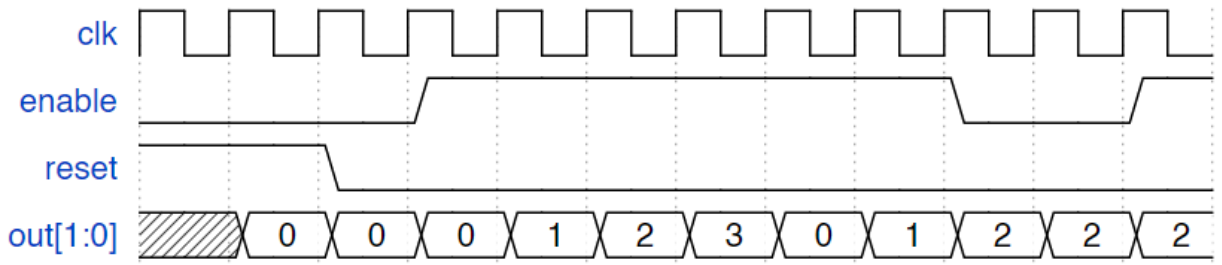
- Write all original names (style requirement)
- Connection are concurrently executed



Example 1: A counter

```
1  module counter (
2  out     ,  // Output of the counter
3  enable  ,  // enable for counter
4  clk     ,  // clock Input
5  reset      // reset Input
6  );
7
8  output [1:0] out; //Output Ports
9  input enable, clk, reset; //Input Ports
10
11 reg [1:0] out; //Internal Variables
12
13 always @(posedge clk)
14 if (reset) begin
15   out <= 2'b0 ;
16 end else if (enable) begin
17   out <= out + 1;
18 end
19
20 endmodule
```

What is the timing diagram?



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

Verification: Simulation & Testbench

Design:

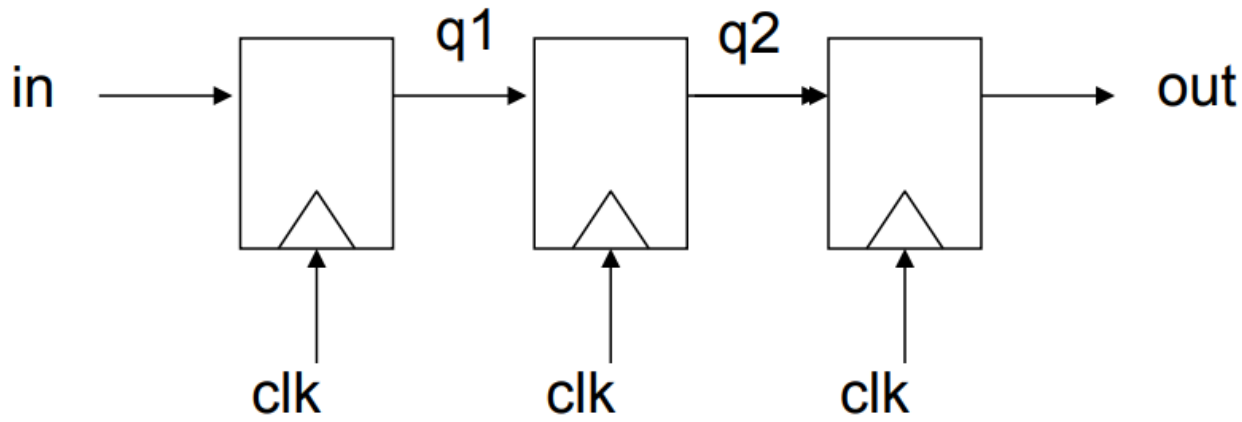
```
1  module counter (
2  out     ,  // Output of the counter
3  enable  ,  // enable for counter
4  clk     ,  // clock Input
5  reset      // reset Input
6  );
7
8  output [1:0] out; //Output Ports
9  input enable, clk, reset; //Input Ports
10
11 reg [1:0] out; //Internal Variables
12
13 always @(posedge clk)
14 if (reset) begin
15   out <= 2'b0 ;
16 end else if (enable) begin
17   out <= out + 1;
18 end
19
20 endmodule
```

Testbench:

```
1  `timescale 1ns/1ps
2  module Testbench;
3
4  wire [1:0] OUT;
5  reg EN, CLK, RST;
6
7  initial CLK = 0;
8  always #2 CLK=~CLK;
9
10 initial begin
11     #1
12     EN = 0;
13     RST = 1;
14     #4
15     EN = 1;
16     RST = 0;
17     #(4*7)
18     EN = 0;
19 end
20
21 counter u1(
22 .out (OUT)    ,  // Output of the counter
23 .enable (EN) ,  // enable for counter
24 .clk  (CLK)   ,  // clock Input
25 .reset  (RST)    // reset Input
26 );
27
28 endmodule
```

Testbench has no ports

Blocking and Non-blocking




```
1   // shift register
2   reg q1,q2,out;
3   always @(posedge clk) begin
4     q1 = in;
5     q2 = q1;
6     out = q2;
7   end
```

Wrong!

```
1   // shift register
2   reg q1,q2,out;
3   always @(posedge clk) q1 <= in;
4   always @(posedge clk) q2 <= q1;
5   always @(posedge clk) out <= q2;
```

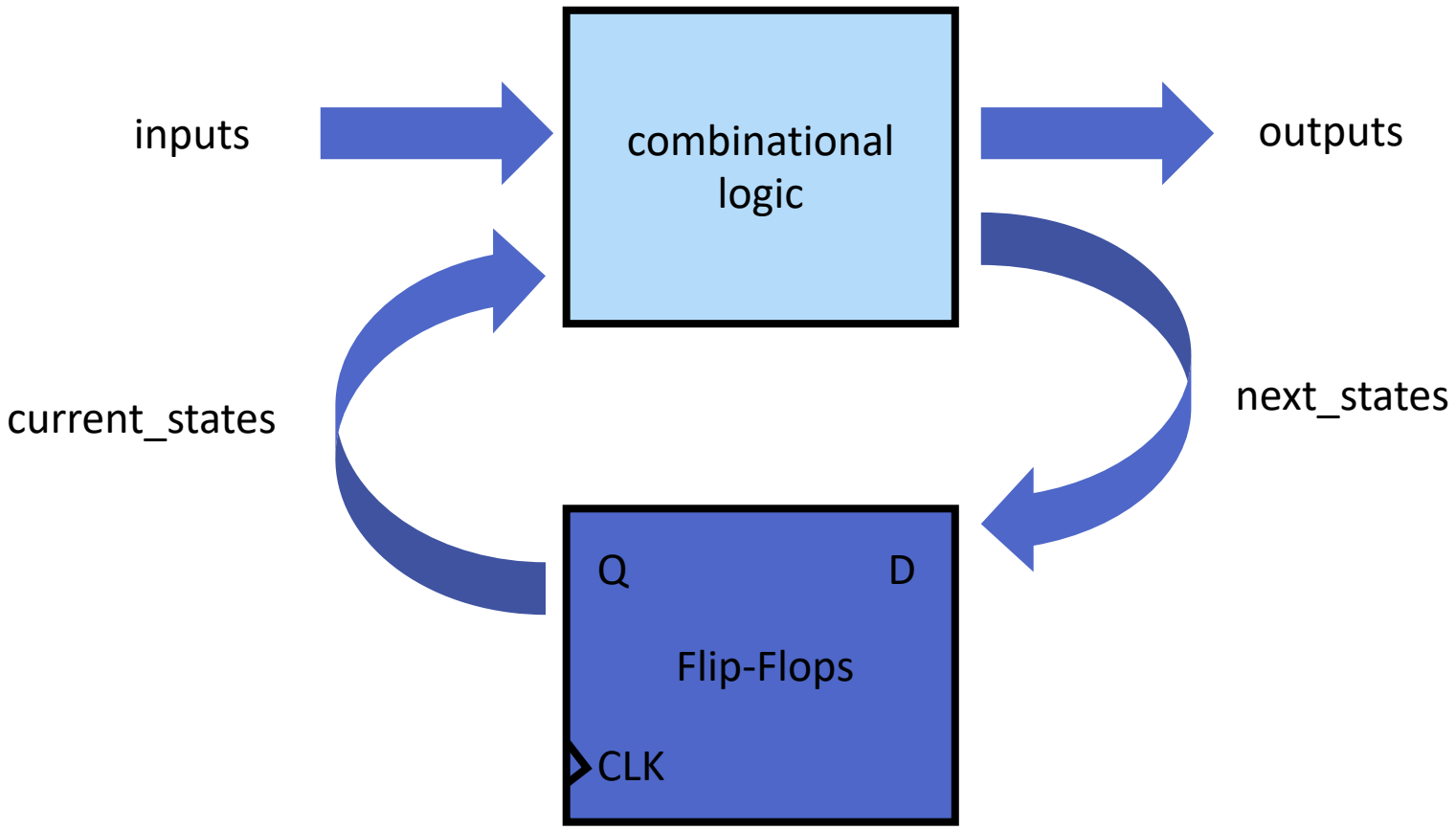
Correct!

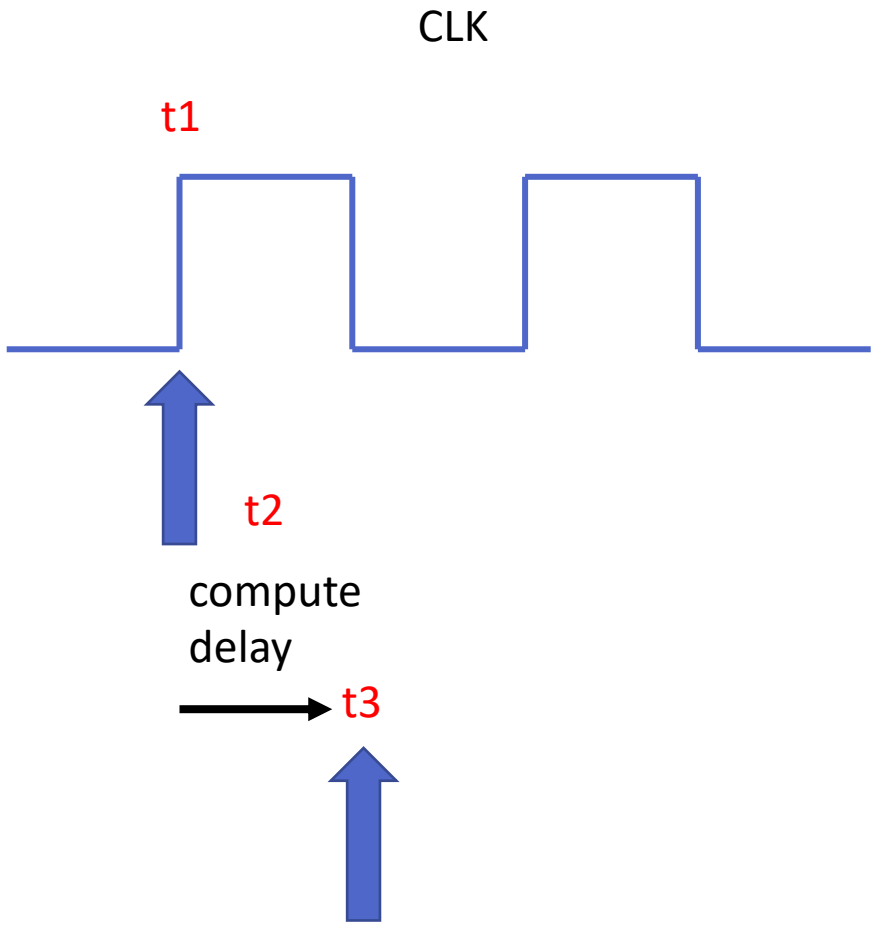
Part 2

Finite State Machine

Finite State Machine

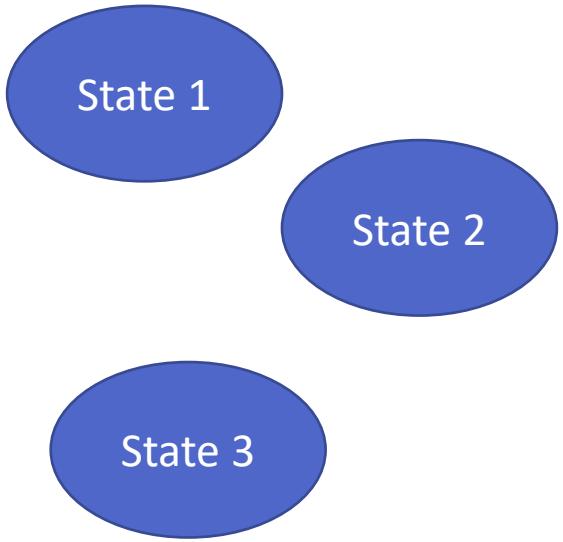
- **Hardware/Circuits**



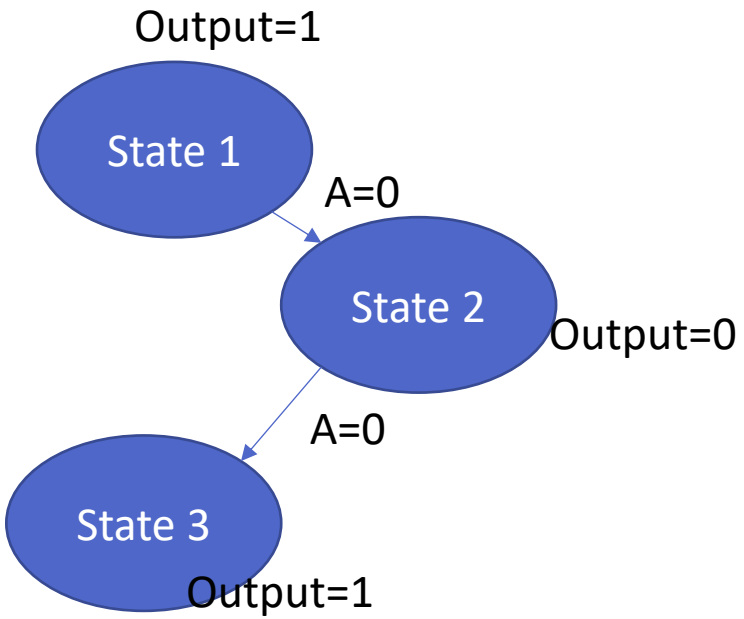


Time point: t1, t2, t3

Design Methodologies & Templates



Step 1: define states



Step 2: draw state-transfer diagram

```
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
```

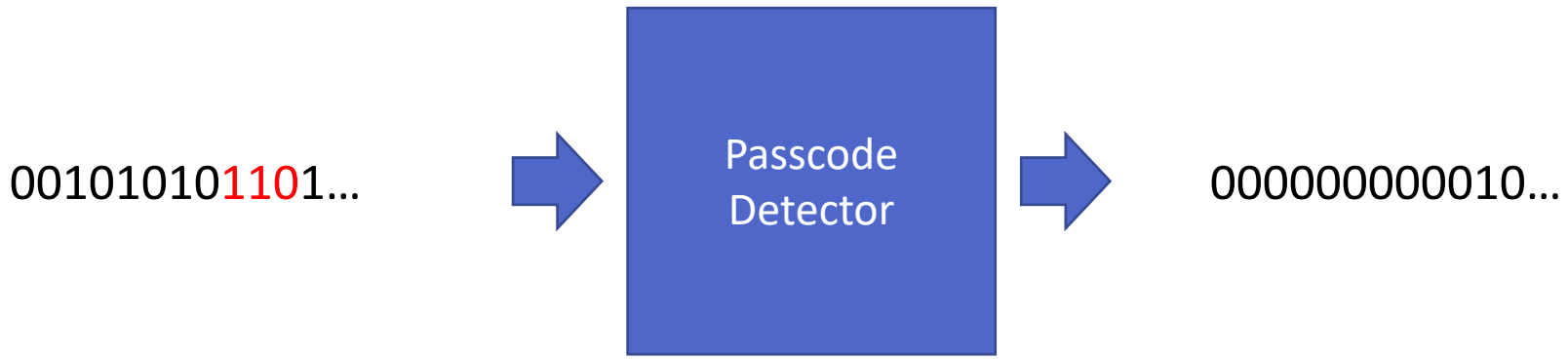
```
    // Determine the next state
    always @ (posedge clk or posedge reset) begin
        if (reset)
            state <= S0;
        else
            case (state)
                S0:
                    state <= S1;
                S1:
                    if (data_in)
                        state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

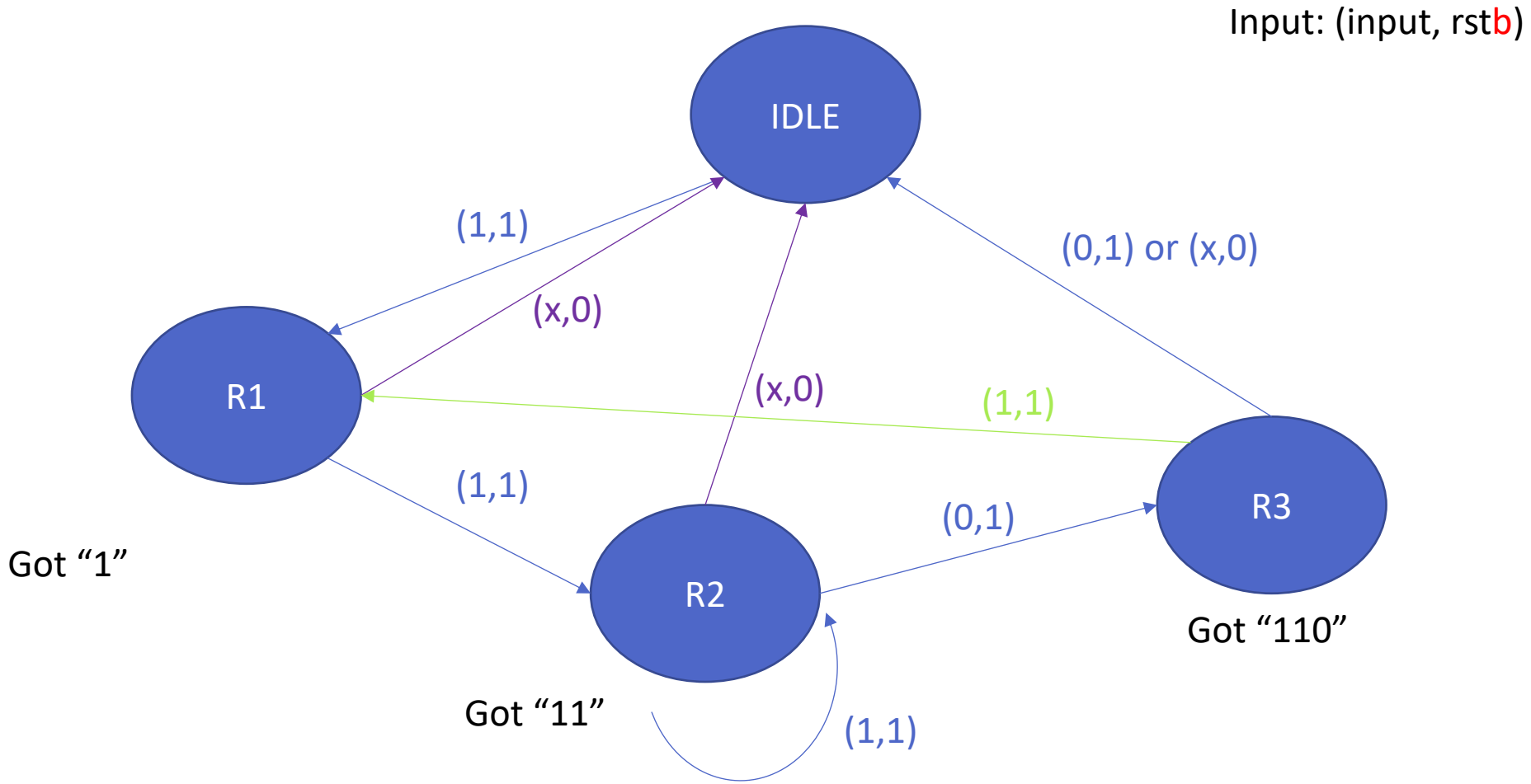
FSM Example 1 Passcode Detector

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



FSM Example 1 – Step 1 & 2

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



FSM Example 1 – Step 3

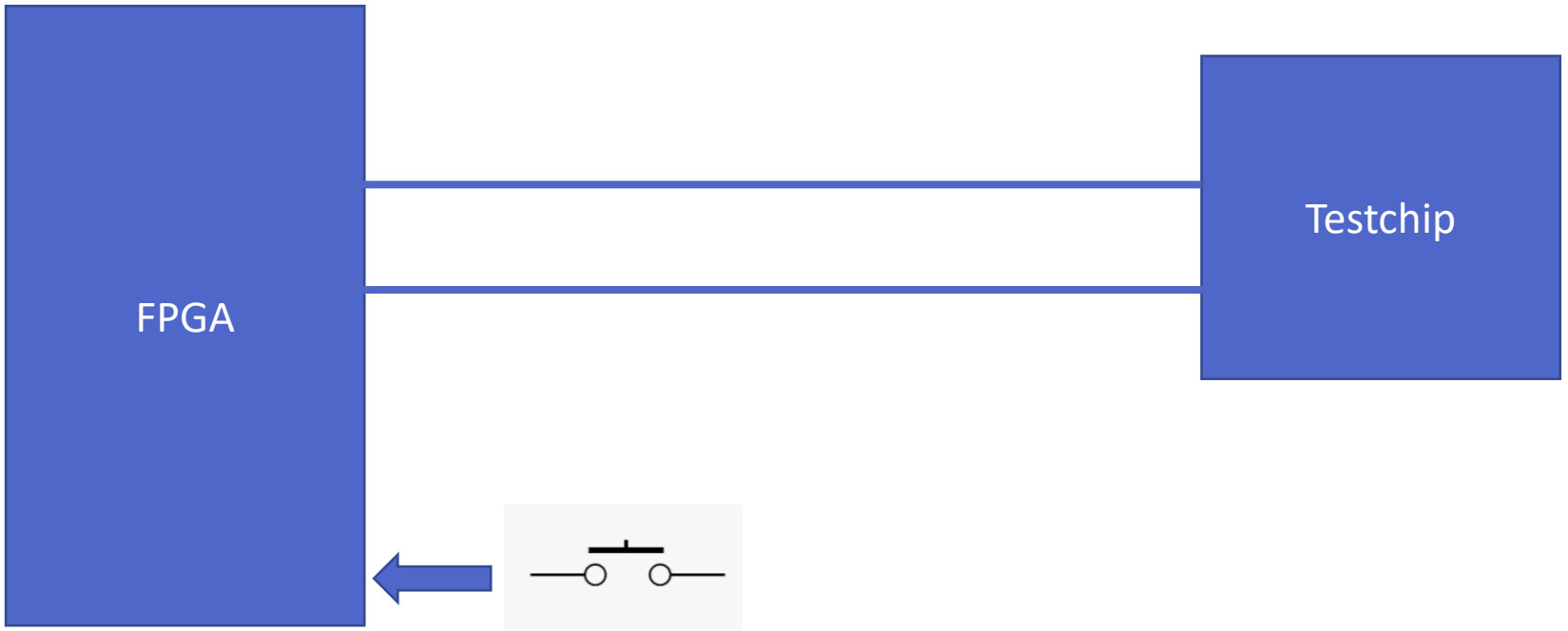
Question: build circuits that outputs 1 pulse, whenever receives “110”

```
1  module PasscodeDetector (
2      input   clk, data_in, rstb,
3      output reg data_out
4  );
5
6      reg [1:0] state;
7      // Declare states
8      parameter STAT_IDLE = 0,
9                STAT_R1 = 1,
10               STAT_R2 = 2,
11               STAT_R3 = 3;
12     // Output depends only on the state
13     always @ (state) begin
14         if(state == STAT_R3) begin
15             data_out <= 1; // alarming!
16         end
17         else begin
18             data_out <= 0;
19         end
20     end
```

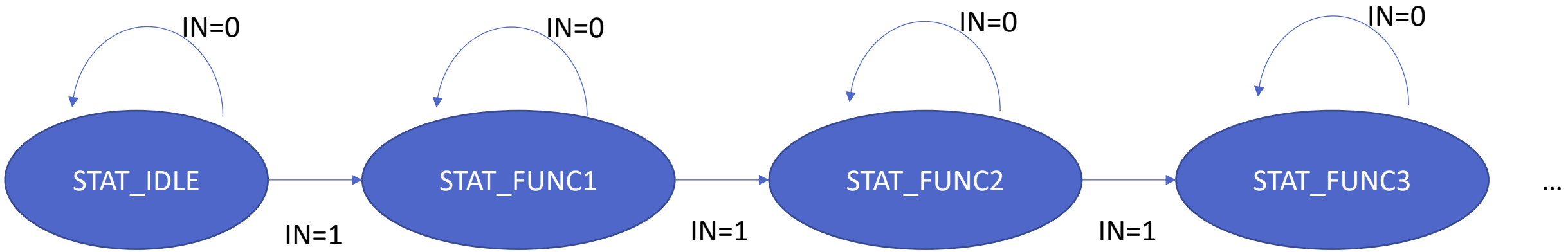
```
22     // Determine the next state
23     always @ (posedge clk) begin
24         if (~rstb)
25             state <= STAT_IDLE;
26         else
27             case (state)
28                 STAT_IDLE: begin
29                     if(data_in==1) begin
30                         state <= STAT_R1;
31                     end
32                 end
33                 STAT_R1: begin
34                     if (data_in==1)
35                         state <= STAT_R2;
36                     else
37                         state <= STAT_IDLE;
38                 end
39                 STAT_R2: begin
40                     if (data_in==0)
41                         state <= STAT_R3;
42                     else
43                         state <= STAT_R2;
44                 end
45                 STAT_R3: begin
46                     if(data_in==1) begin
47                         state <= STAT_R1;
48                     end
49                     else begin
50                         state <= STAT_IDLE;
51                     end
52                 end
53             endcase
54     end
55 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!

```
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:**

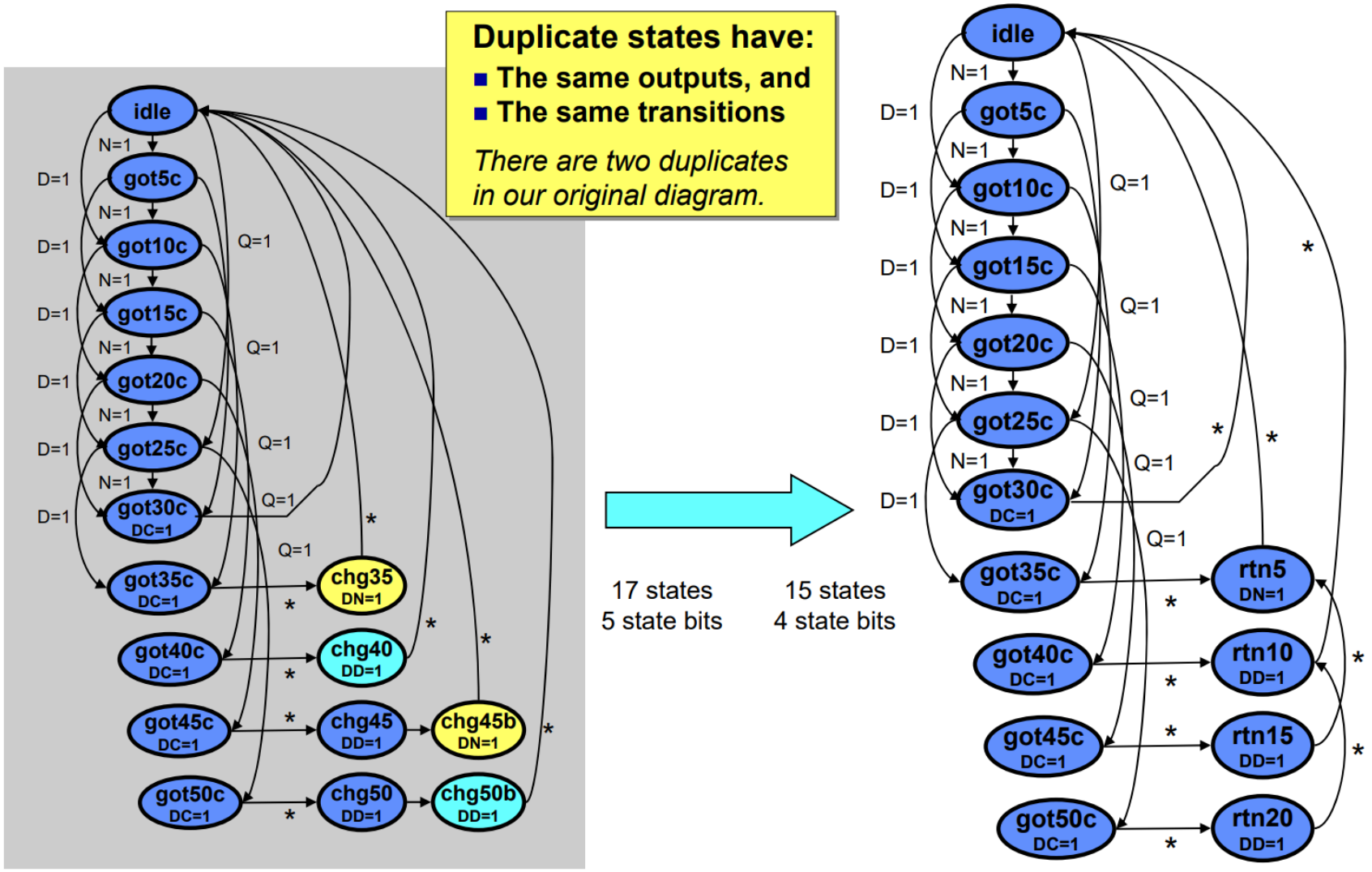
- **What's the maximum amount of money captured before purchase?**
  25 cents (just shy of a purchase) + one quarter (largest coin)

- **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.



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.