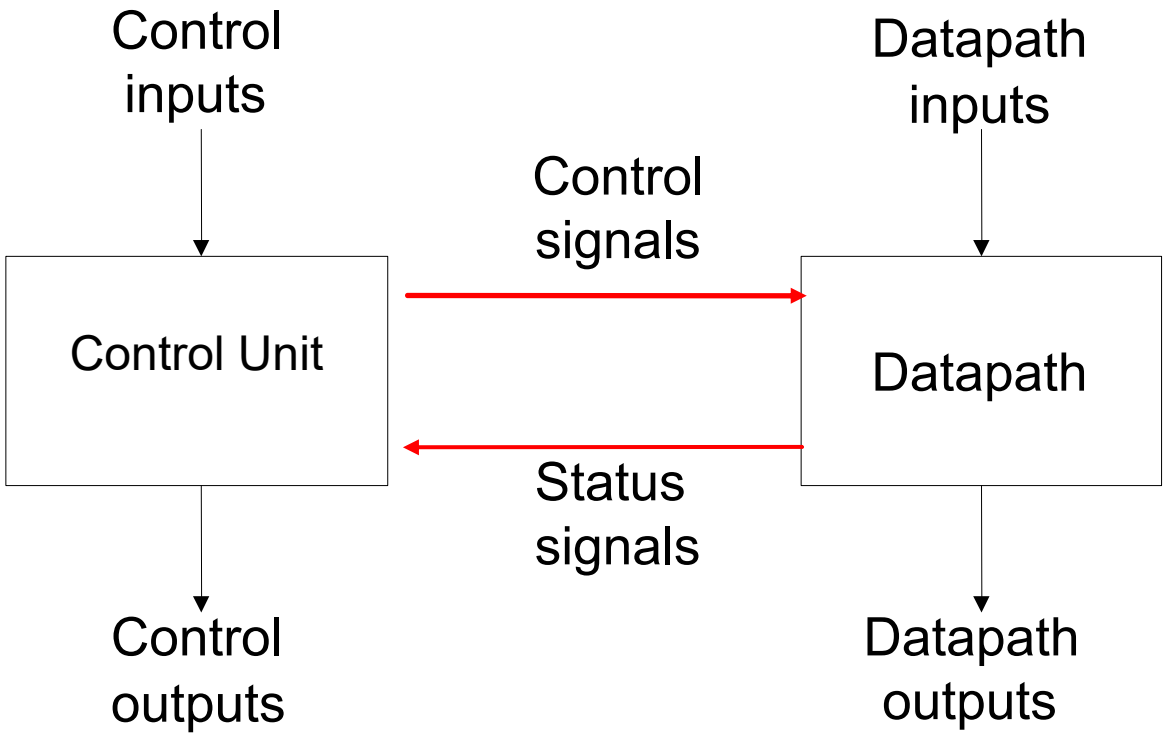
Control Unit

Slides modified from Digital IC Design by Pei-yin Chen and Digital System and Designs and Practices by Ming-bo Lin and

Modern Design (1/3)

Modern design is composed of (1) Datapath and

(2) Controller (control unit or control path)



(Note)

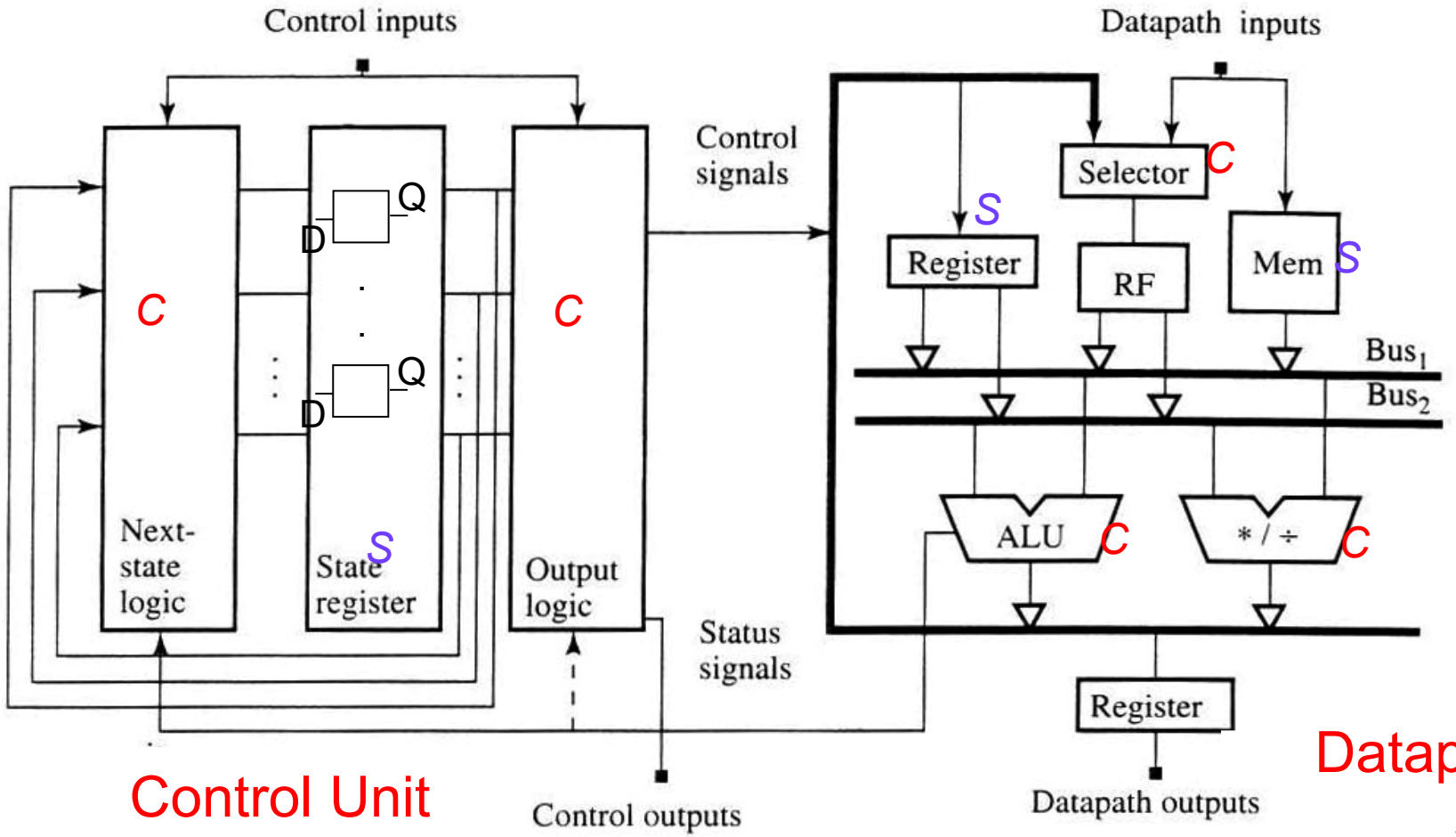
High-level block diagram

Modern Design (2/3)

Register-transfer-level block diagram

C: Combinational circuit

S: Sequential circuit



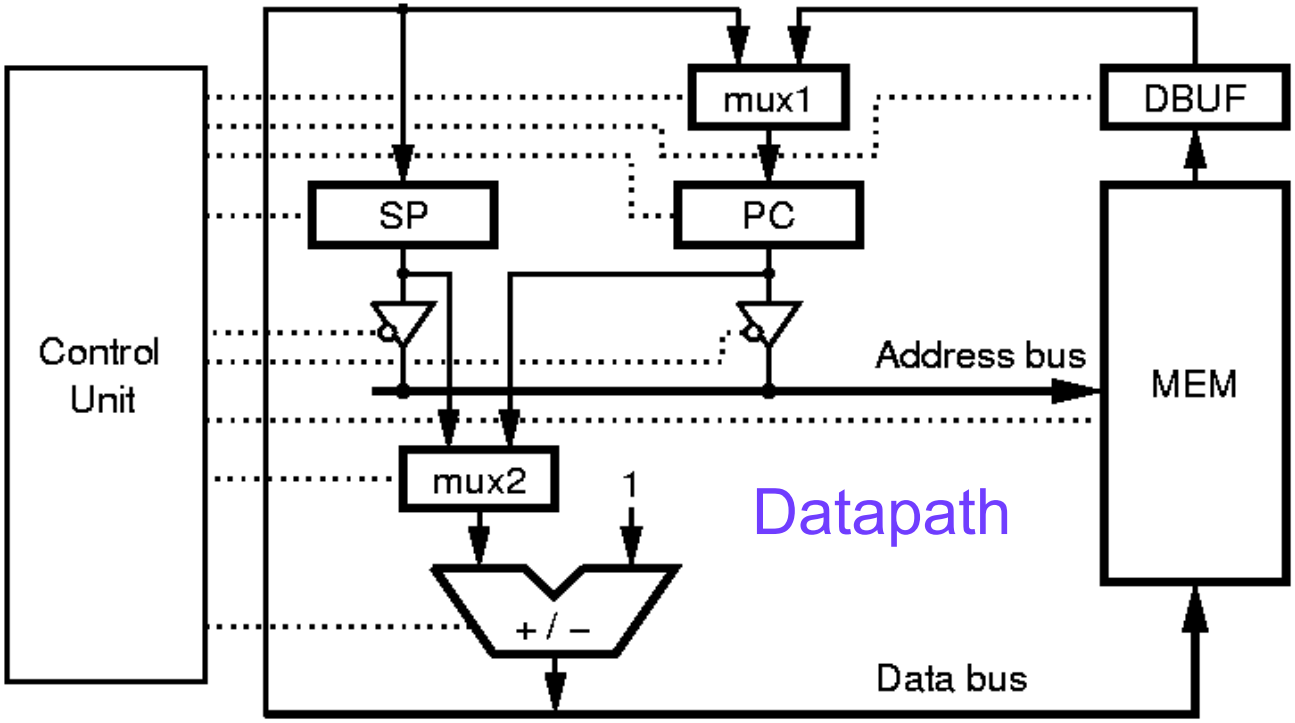
Modern Design (3/3)

How to design the following system??

```
if IR(3) = '0' then
    PC          := PC + 1;
else
    DBUF        := MEM(PC);
    MEM(SP)     := PC + 1;
    SP          := SP – 1;
    PC          := DBUF;
end if;
```

SP: Stack Pointer

IR: Instruction Register



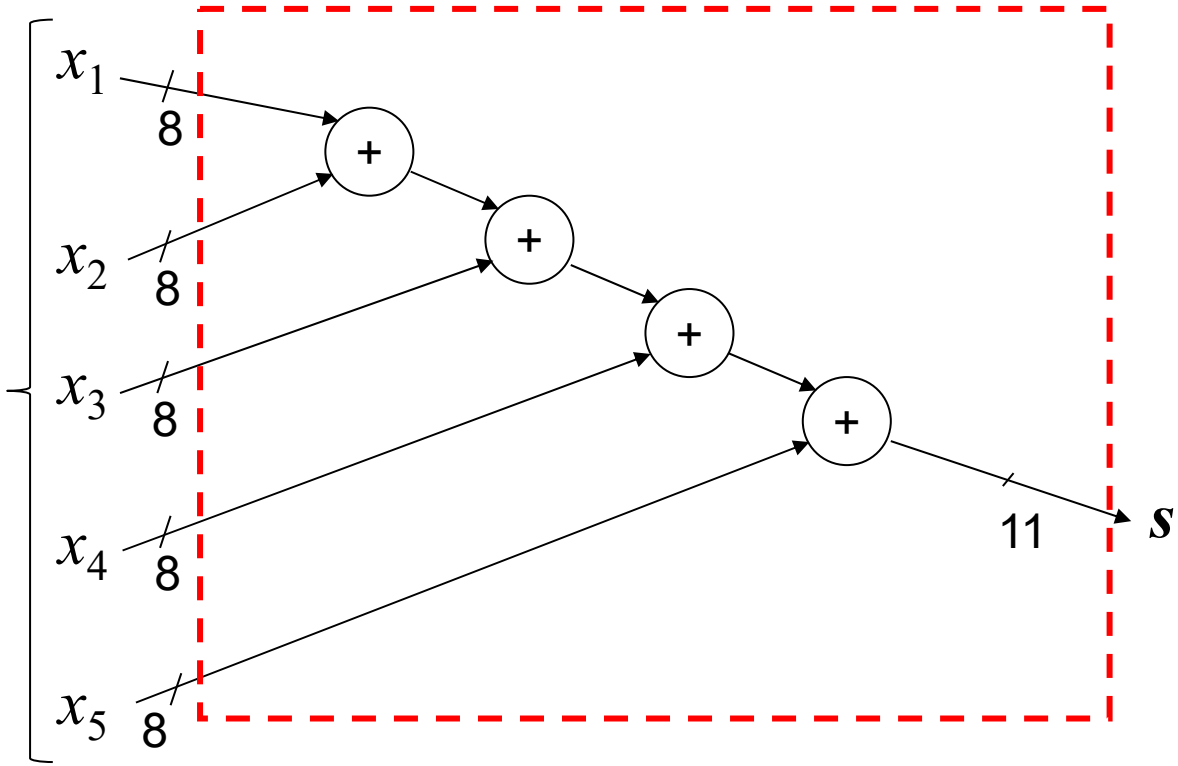
Datapath

Summation Problem (1/4)

Question: Calculate $S = x_1 + x_2 + x_3 + x_4 + x_5$ with a ASIC chip

Method 1: Sum up five inputs in the same period by using 4 adders

Five inputs must be ready at the same time. Why and How ?



a. How many input pins and output pins ?

b. What is the resolutions (bit width) of x_i ?

c. How fast is the circuit (critical path)?

d. What is your design cost ?

Summation Problem (2/4)

Calculate $S = x_1 + x_2 + x_3 + x_4 + x_5$

Method_2: Sum up five inputs in the different time units by using only 1 adders

Initial $S = 0$

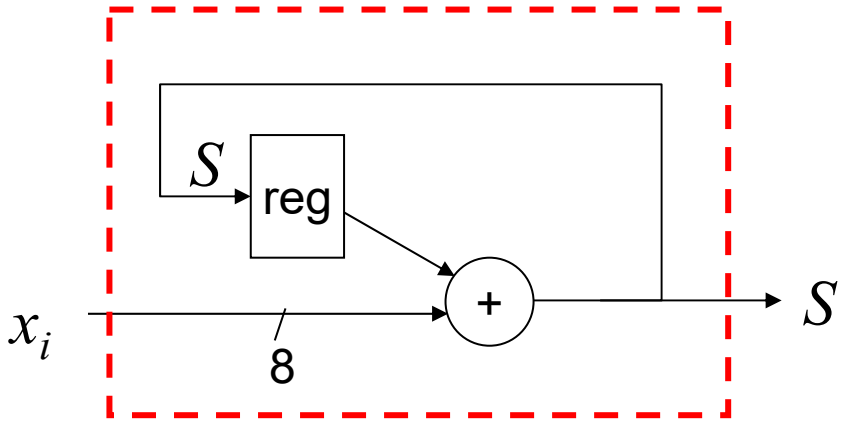
Time unit _1 $S <= S + x_1$

Time unit _2 $S <= S + x_2$

Time unit _3 $S <= S + x_3$

Time unit _4 $S <= S + x_4$

Time unit _5 $S <= S + x_5$



Only one input must be ready at a time. Why?

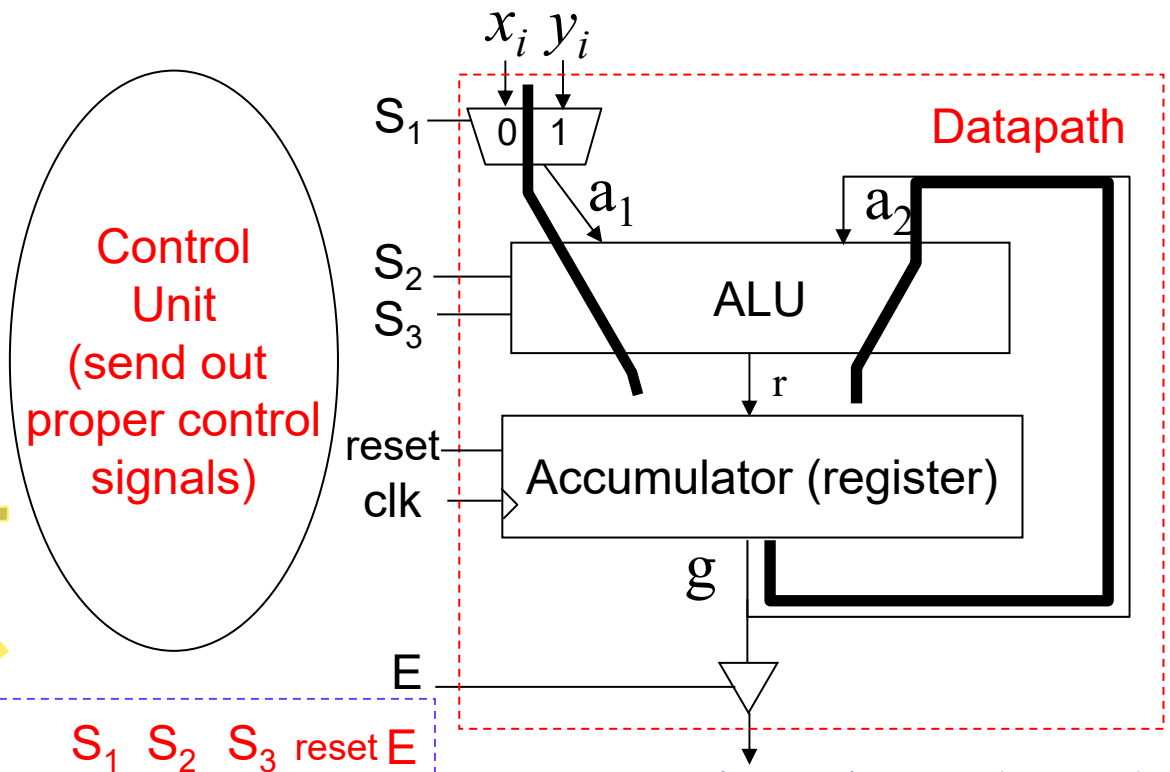
Cost is lower and critical path is shorter than the Method_1 .

But its working rate is slower than Method_1. (5 clock cycles for 1 summation result)

Summation Problem (3/4)

Problem: Calculate $S = x_1 + x_2 + x_3 + x_4 + \ldots + x_{50}$

$$S = \sum_{i=1}^{50} x_i$$



If S_1=0, $a_1 = x_i$

If S_1=1, $a_1 = y_i$

If S_2=0 S_3=0, $r = a_2$

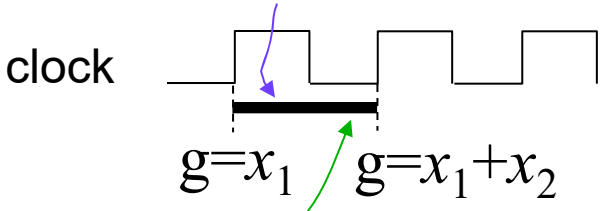
If S_2=0 S_3=1, $r = a_1 + a_2$

If S_2=1 S_3=0, $r = a_1 - a_2$

If S_2=1 S_3=1, $r = a_1$

	S_1	S_2	S_3	reset	E
T_0	0	0	0	1	0
T_1	0	1	1	0	0
T_2~49	0	0	1	0	0
.....					
T_51	X	0	0	0	1

Next x_i must be ready now (e.g., x_2)



r must be ready before the next positive edge comes

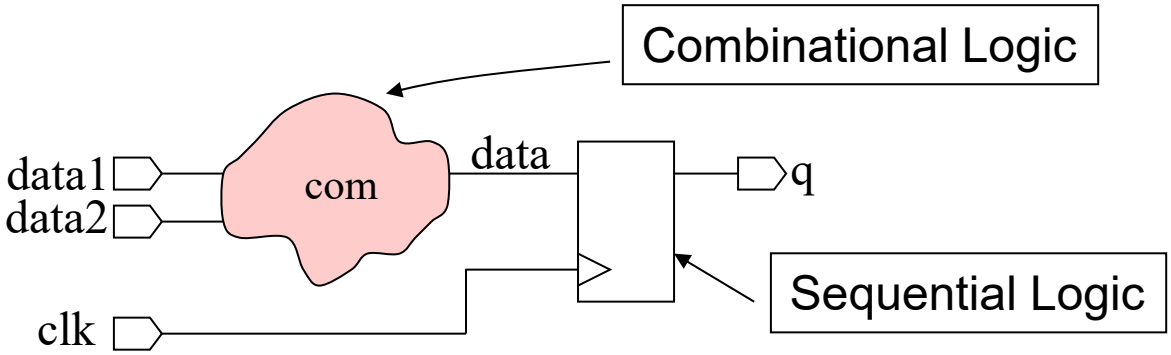
What is the length of clock period (_____)?

Critical (longest) path delay

⇨ Reciprocal is clock rate

Better HDL style

Separating combinational and sequential circuits




```
module EXAMPLE(data1,data2,clk,q);
input     data1, data2, clk;
output    q;
reg       data,q;
always @(data1 or data2)
   data = com(data1,data2);

always @(posedge clk)
   q <= data;
endmodule
```

Combinational Logic — Blocking Assignment

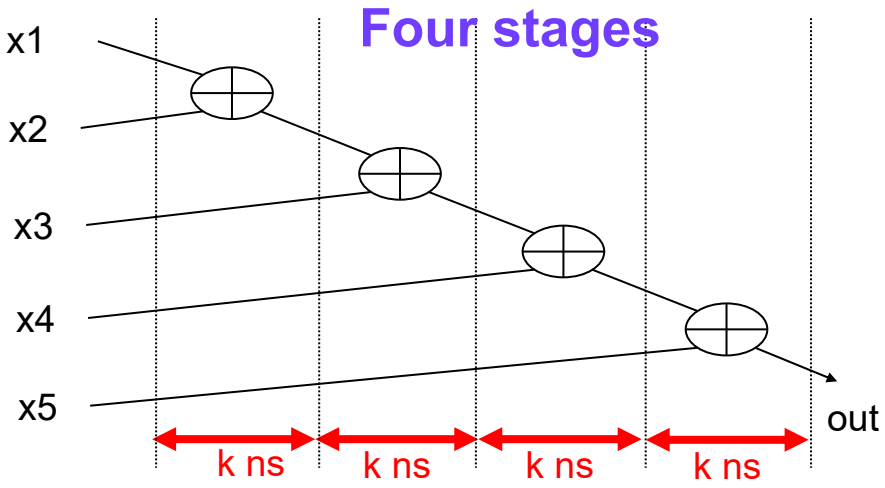
Sequential Logic — Nonblocking Assignment

Design for Summation Problem (1/7)

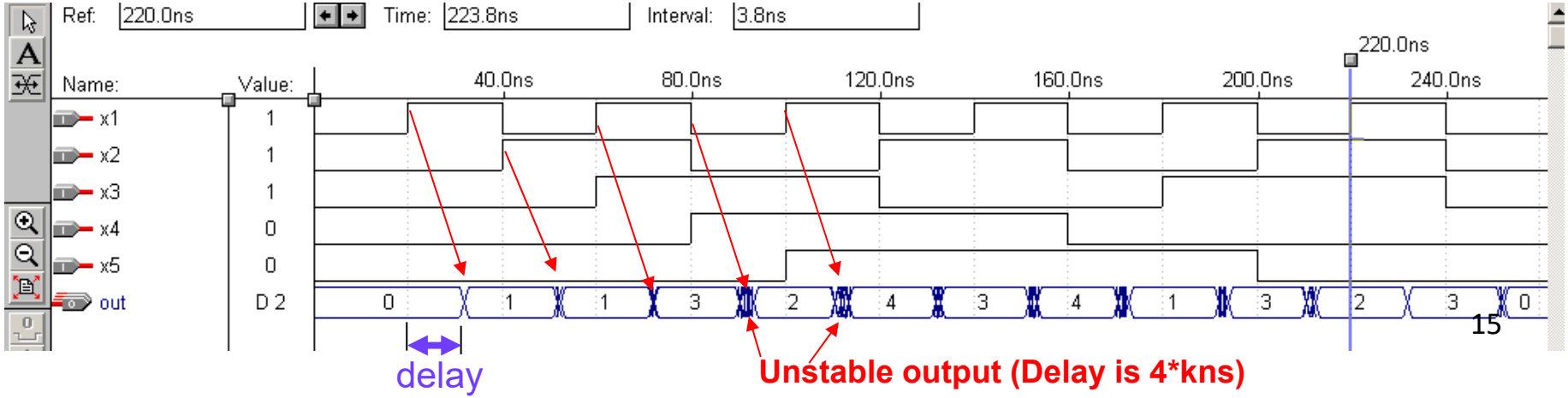
Method_1

```
module adder1(x1, x2, x3, x4, x5, out);
input  x1, x2, x3, x4, x5;
output [2:0] out;
reg    [2:0] out;

always@(x1 or x2 or x3 or x4 or x5)

   out=(((x1+x2)+x3)+x4)+x5;

endmodule
```

Calculate $S = x_1 + x_2 + x_3 + x_4 + x_5$



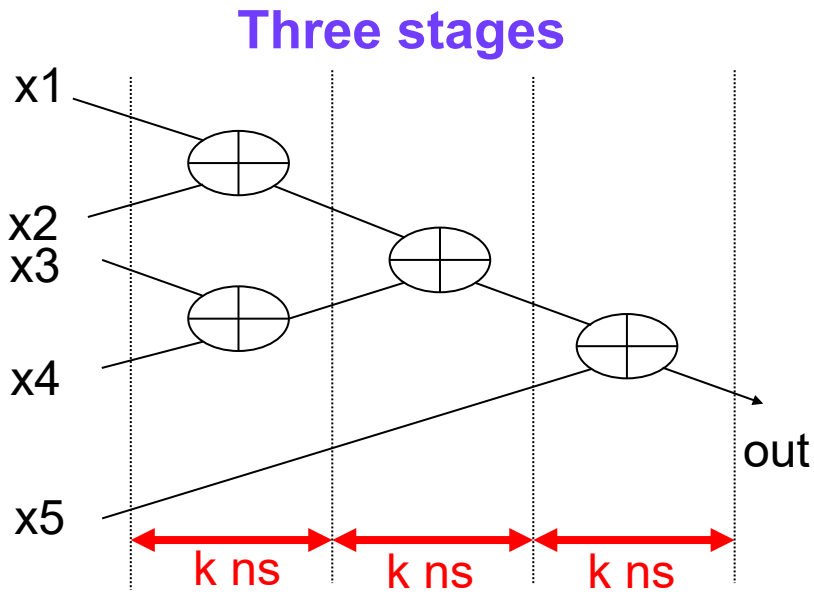
Assume the adder's delay is k ns

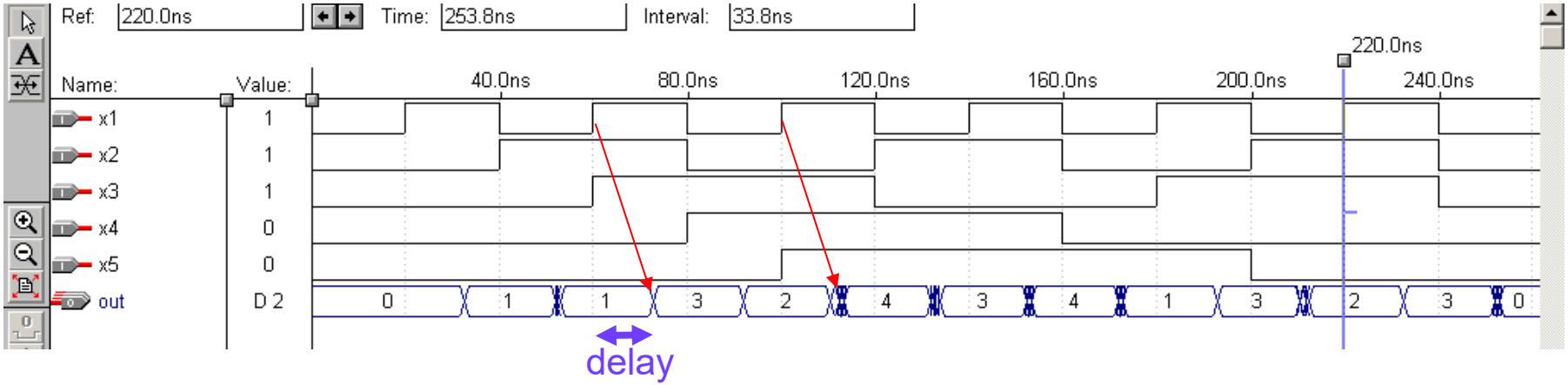


Design for Summation Problem (2/7)

Method_2 (shorter delay)

```
module adder2(x1, x2, x3, x4, x5, out);
input  x1, x2, x3, x4, x5;
output [2:0] out;
reg    [2:0] out;

always@(x1 or x2 or x3 or x4 or x5)
   out=((x1+x2)+(x3+x4))+x5;
endmodule
```



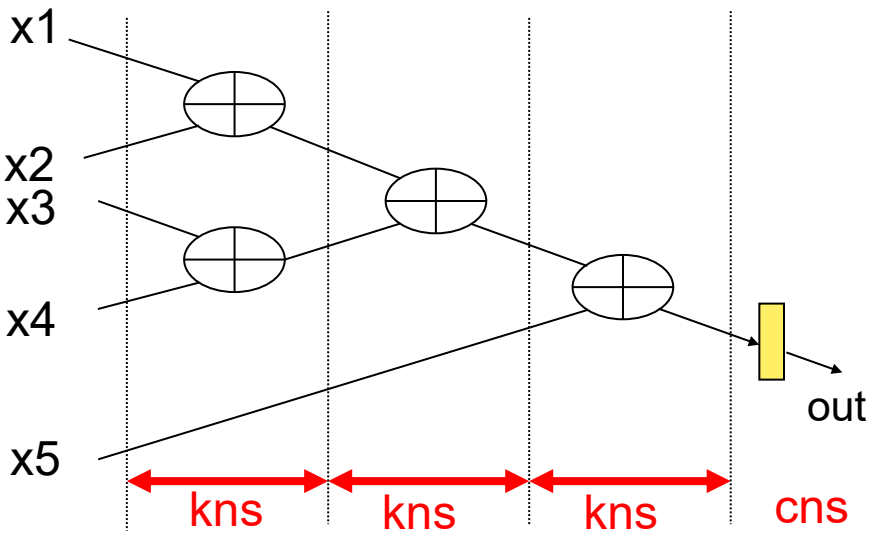


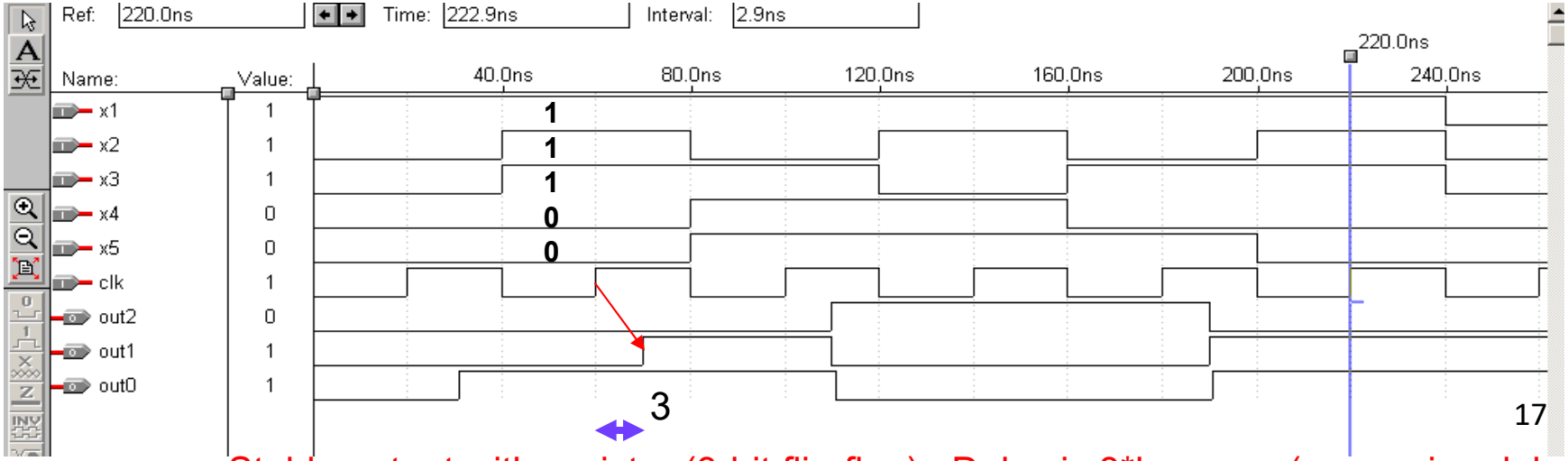
Unstable output (Delay is 3*kns -- less than Method_1)

Design for Summation Problem (3/7)

Method_3 (Using registered output)

```
module adder3(x1, x2, x3, x4, x5, clk, out);
input  x1, x2, x3, x4, x5, clk;
output [2:0] out;
reg    [2:0] out;

always@(posedge clk)

   out=((x1+x2)+(x3+x4))+x5;

endmodule
```



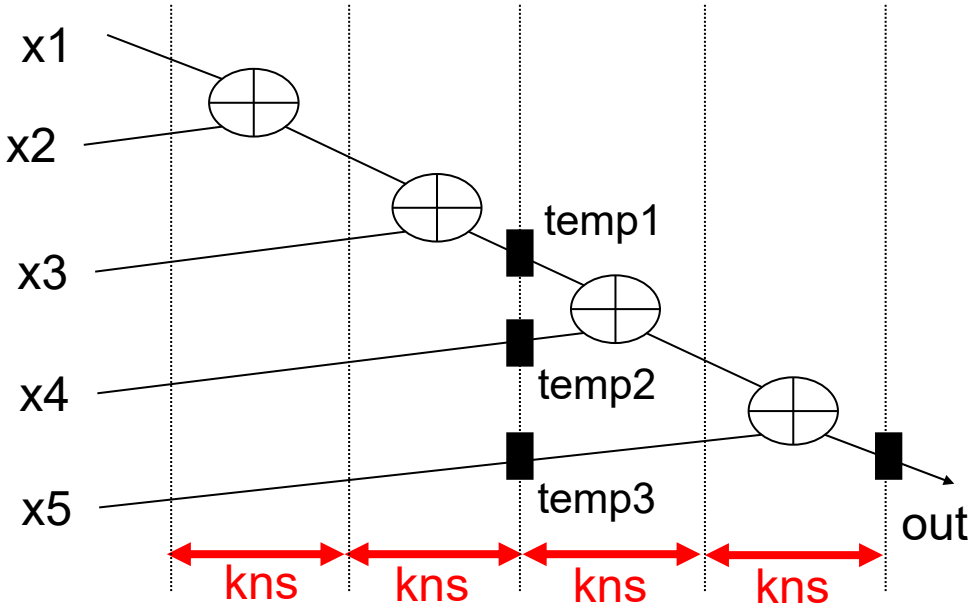


Stable output with register (3-bit flip-flop) Delay is 3*kns+cns (reg assign delay)

Design for Summation Problem (4/7)

Method_4: Using pipelined register

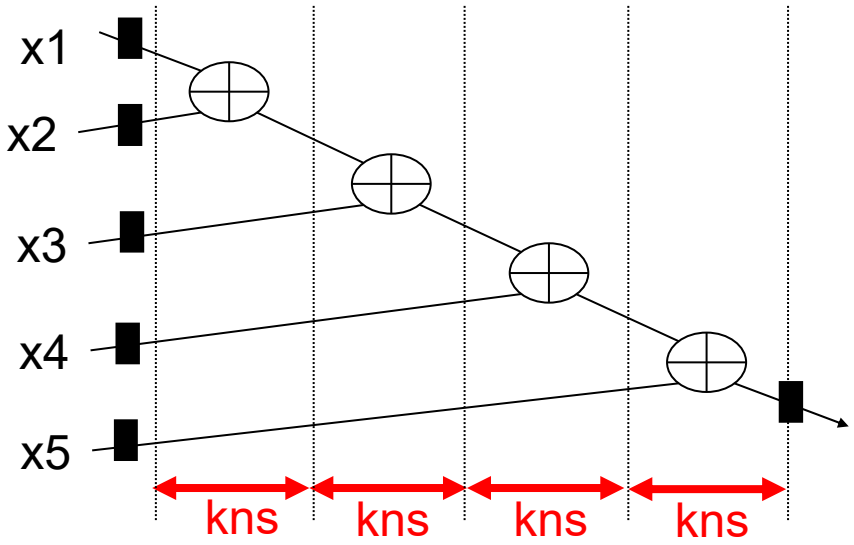
```
module adder4(clk, x1, x2, x3, x4,
x5, out);
input  clk,x1, x2, x3, x4, x5;
output [2:0] out;
reg [2:0] out, temp1, temp2,temp3;
always@(posedge clk)
   begin
      temp1<=(x1+x2)+x3;
      temp2<=x4;    temp3<=x5;
      out<=temp1+temp2+temp3;
   end
endmodule
```



Delay is 2*kns+cns, which is less than Method_1 (4kns), Method_2 (3kns) and Method_3 (3kns+cns)

So, this method can achieve the best (fastest) clock rate because its critical path is shortest. However, the correct out is generated after two clock cycles not just one (also named as datapath pipelining)

Design for Summation Problem (5/7)

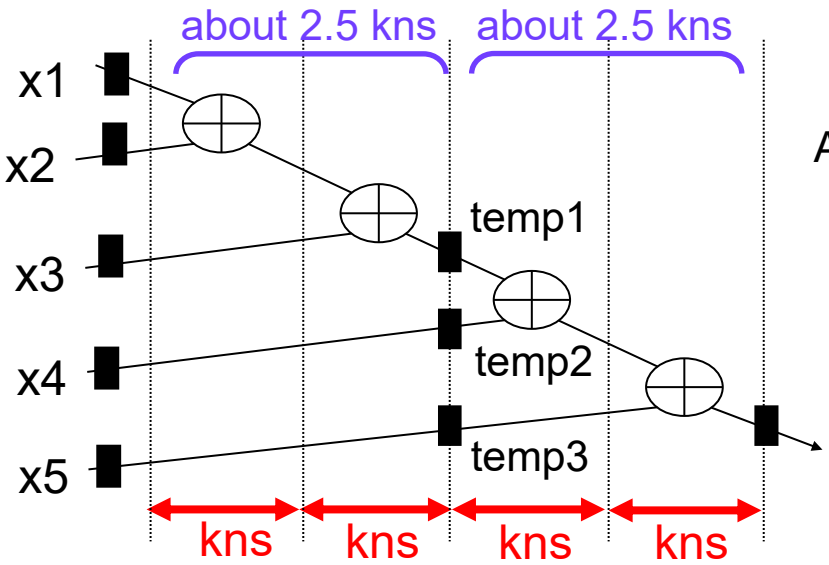


1. Wire delay
2. Register assignment delay

Critical path is about 4kns

A correct output is generated every clock cycle

Event	1	2	3	4	5	6
Completed time	4k	8k	12k	16k	20k	24k



faster clock rate

Critical path is about 2.5 kns, why?

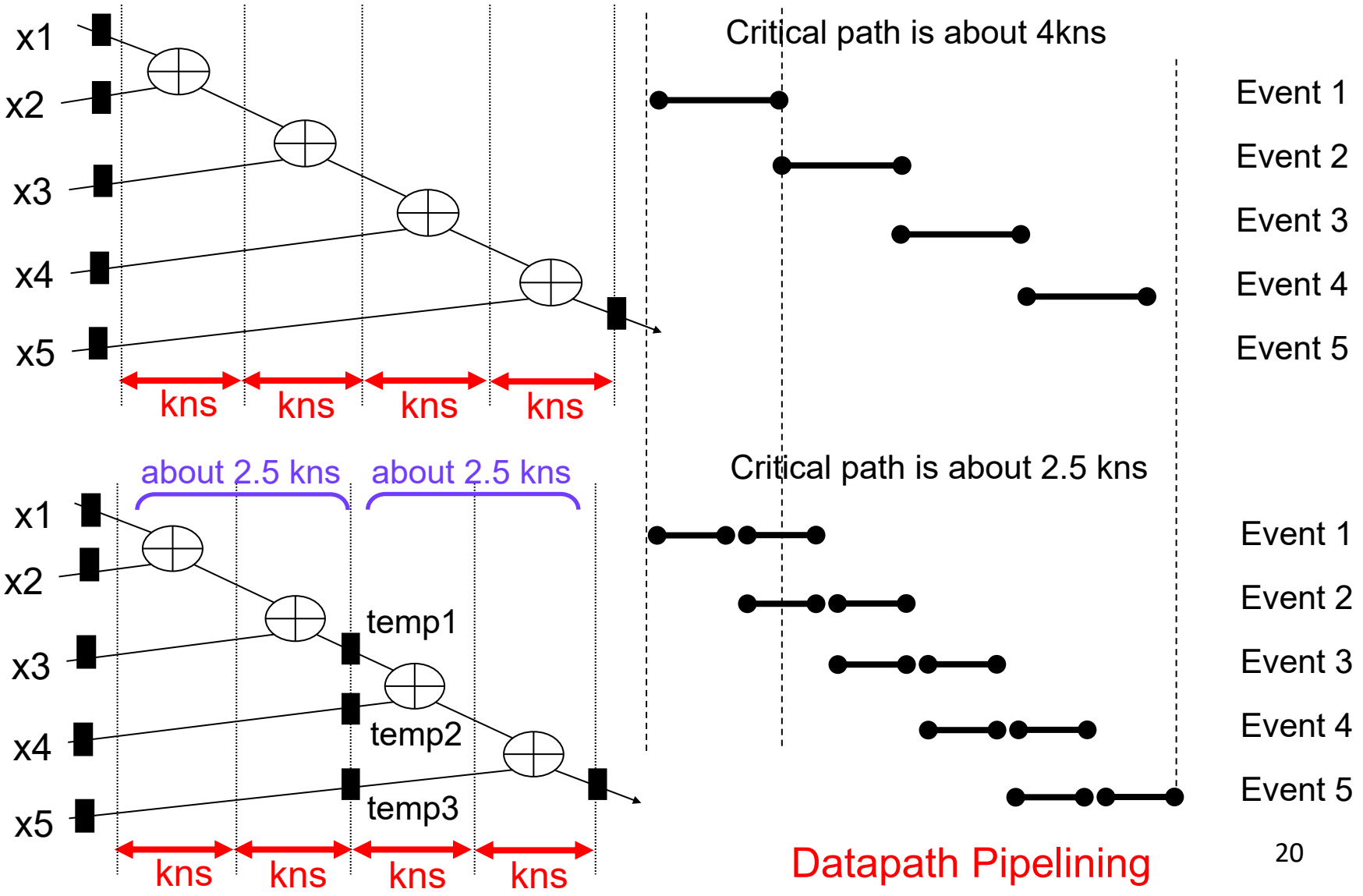
A correct output is generated after two clock cycles

Event	1	2	3	4	5	6
Completed time	5k	7.5k	10k	12.5k	15k	17.5k

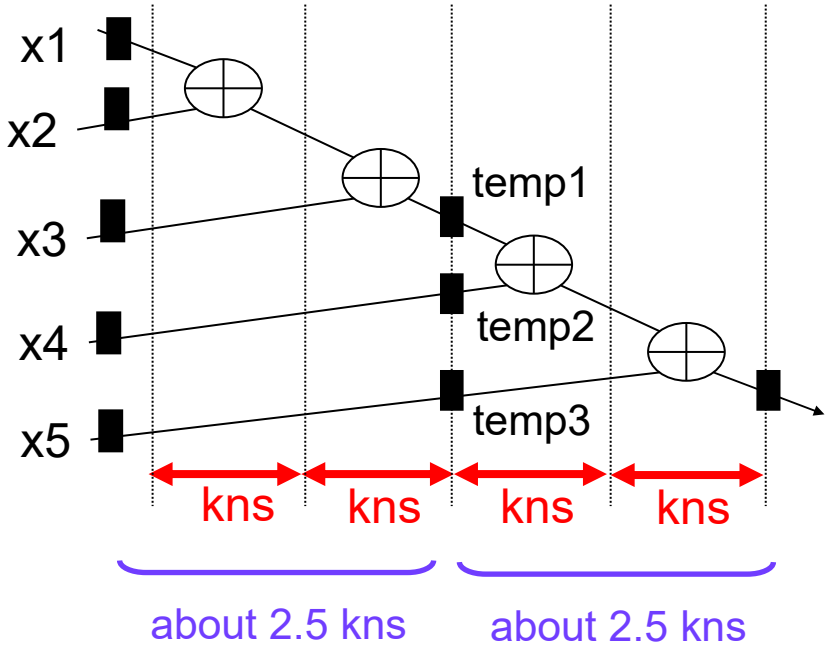
Two events are parallel processed in the unit.

Faster clock rate but higher cost (3 extra regs)

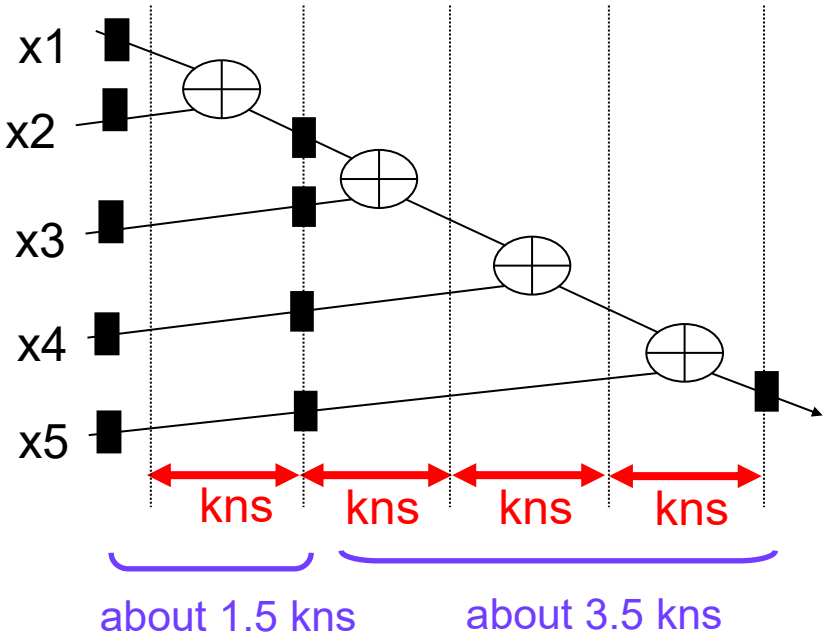
Design for Summation Problem (6/7)



Design for Summation Problem (7/7)



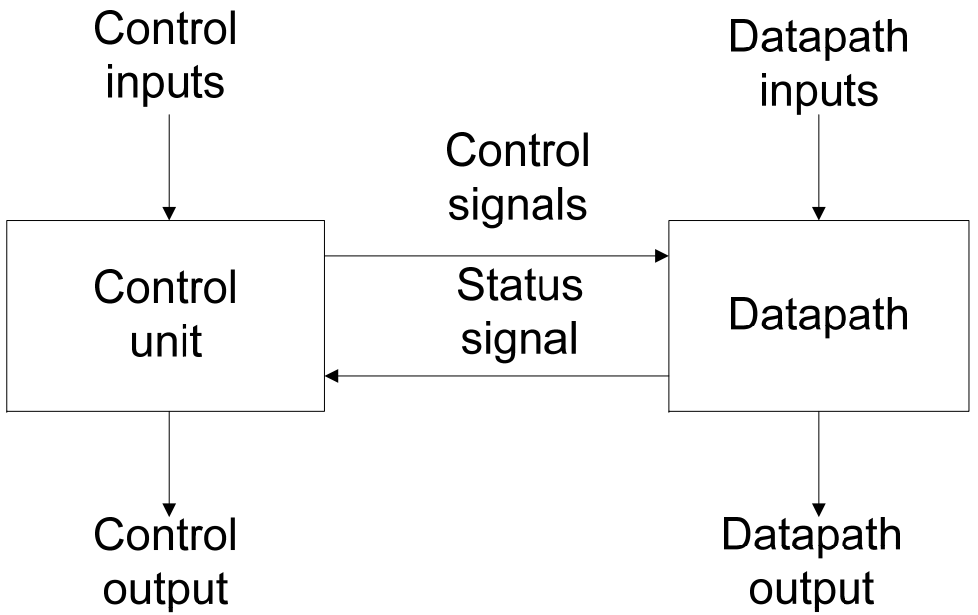
Critical path is about 2.5 kns



Critical path is about 3.5 kns

Which one is better ? Balance is important

Optimization for RTL Design



Optimization for control unit:

1. As suggestion by most textbooks of "Logic System Design"
2. Write HDL descriptions with good styles, and then optimized by EDA tools

Optimization for datapath:

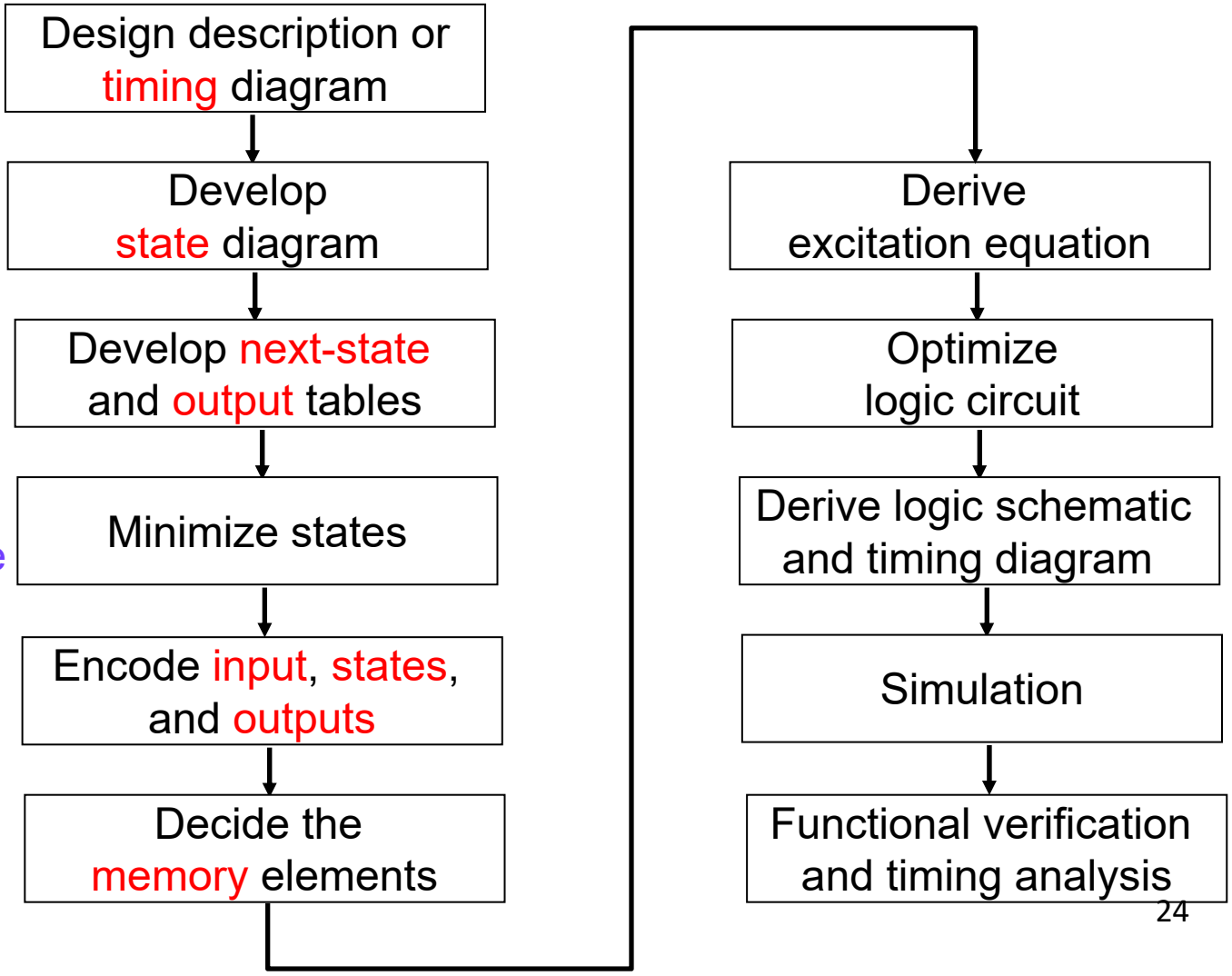
1. Resource optimization
2. Time optimization

Optimization for Control Unit

Traditional Optimization Flow for Control Unit
=> Use Finite State Machine to design Control unit

Control Unit

Finite State Machine (FSM)



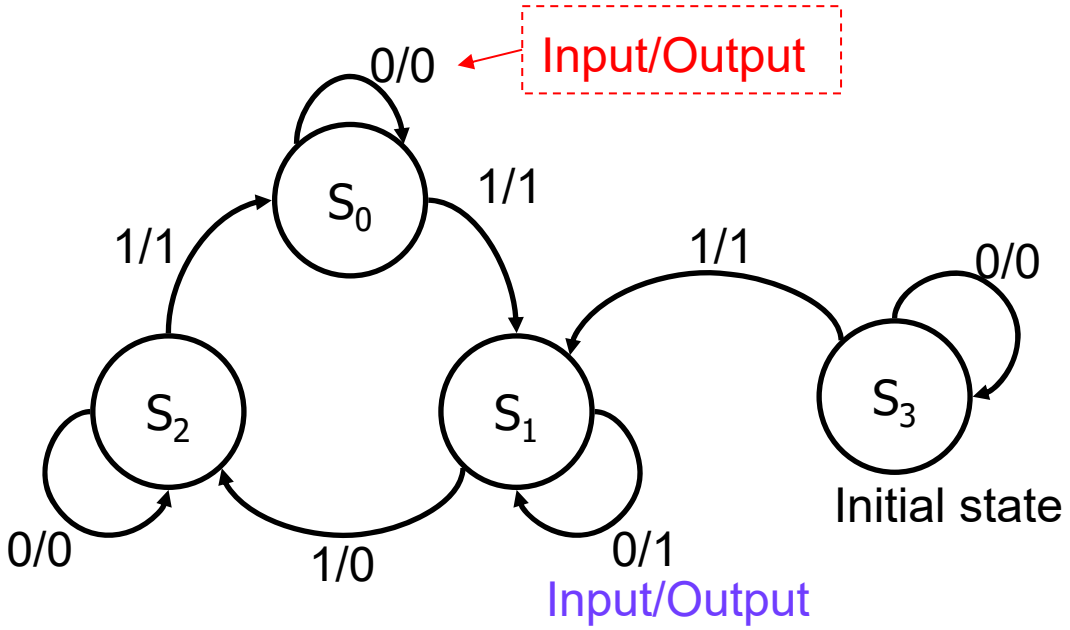
Finite State Machine (1/4)

Moore machine: $S \rightarrow O$ (output is dependent only on current state)

Mealy machine: $S \times I \rightarrow O$ (output is dependent on input and state)

State diagram for a Mealy machine

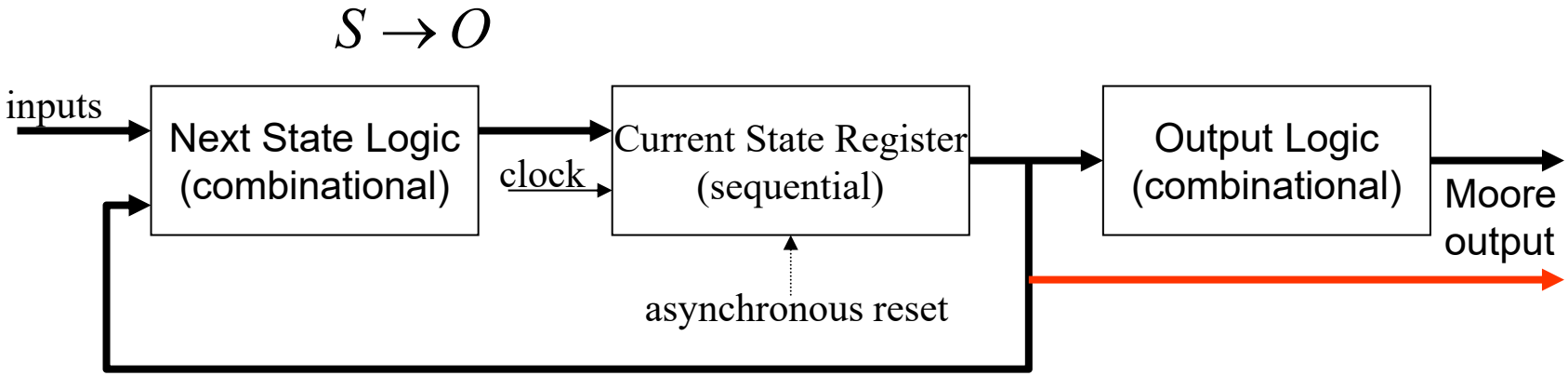
Four states: S_0, S_1, S_2, S_3



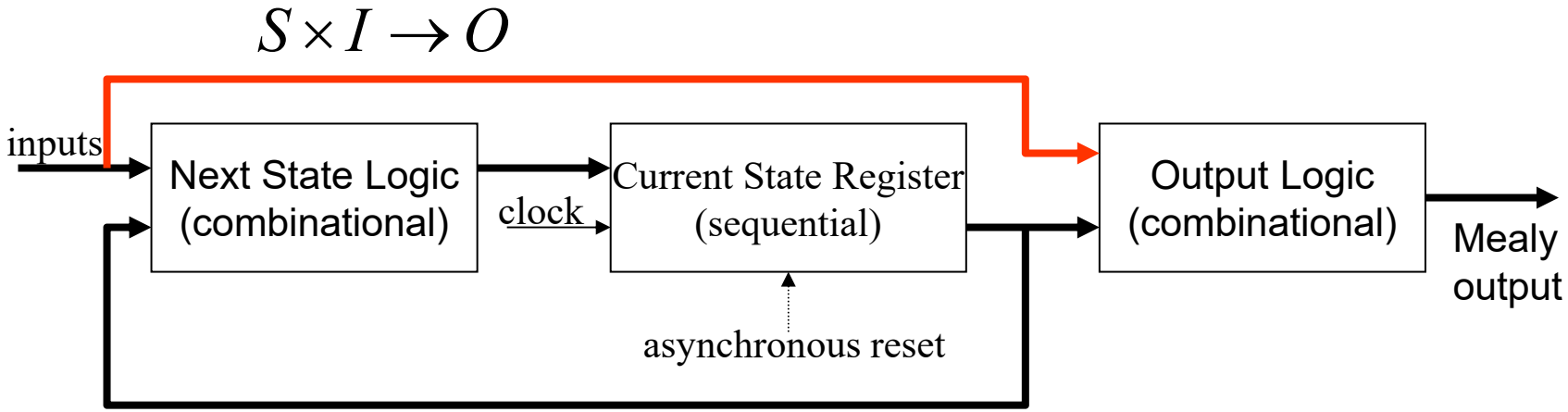
Next-state and output tables (I=input)

Present State	Next State		Output	
	I=0	I=1	I=0	I=1
S_0	S_0	S_1	0	1
S_1	S_1	S_2	1	0
S_2	S_2	S_0	0	1
S_3	S_3	S_1	0	1

Finite State Machine (2/4)



Moore Machine (state-based machine)

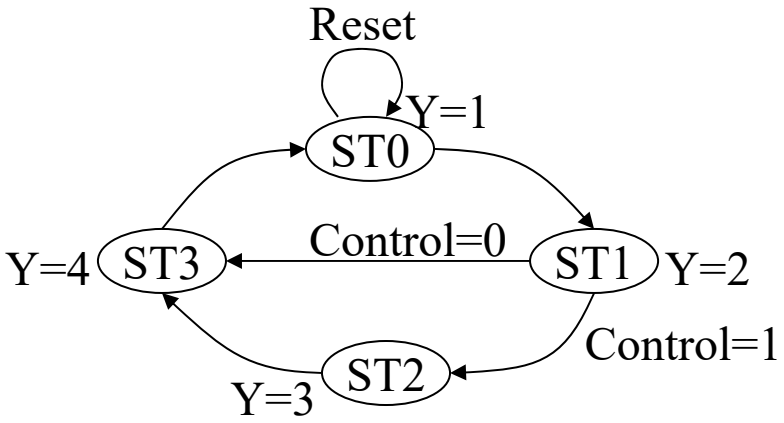


Mealy Machine (input-based machine)

Finite State Machine (3/4)

- For best legibility, describe FSM using two or three ***always@*** statements
  - (1) current state or state register (sequential circuit)
  - (2) next state logic (combinational circuit)
  - (3) output logic (combinational circuit)
    - Two combinational logic can be merged
- Use ***parameter*** to describe the state name

Example: Write Verilog for the FSM



Review: State Encoding

- Common FSM encoding options:
  - One-hot code
  - Binary code
  - Gray code
  - Random code

State	Binary	Gray	One hot
A	00	00	1000
B	01	01	0100
C	10	11	0010
D	11	10	0001

Finite State Machine (4/4)

```
module FSM(Clock, Reset, Control, Y)
input Clock, Reset, Control;
output [2:0] Y;

reg [1:0] CurrentState, Nextstate;
reg [2:0] Y;

parameter [1:0] ST0 = 2'b00,
                ST1 = 2'b01,
                ST2 = 2'b10,
                ST3 = 2'b11;

always @(posedge Clock or posedge Reset)
   if (Reset)
      CurrentState <= ST0;
   else
      CurrentState <= NextState;

always @(Control or Currentstate)
   begin
      NextState = ST0;
      case (CurrentState)
         ST0: NextState = ST1;
         ST1:  if (Control)
                  NextState = ST2;
               else
                  NextState = ST3;
         ST2: NextState = ST3;
         ST3: NextState = ST0;
      endcase    end

always @(CurrentState)
   begin
      case(CurrentState)
         ST0: Y = 1;   ST1: Y= 2;
         ST2: Y = 3;   ST3: Y = 4;
      endcase
   end      endmodule
```

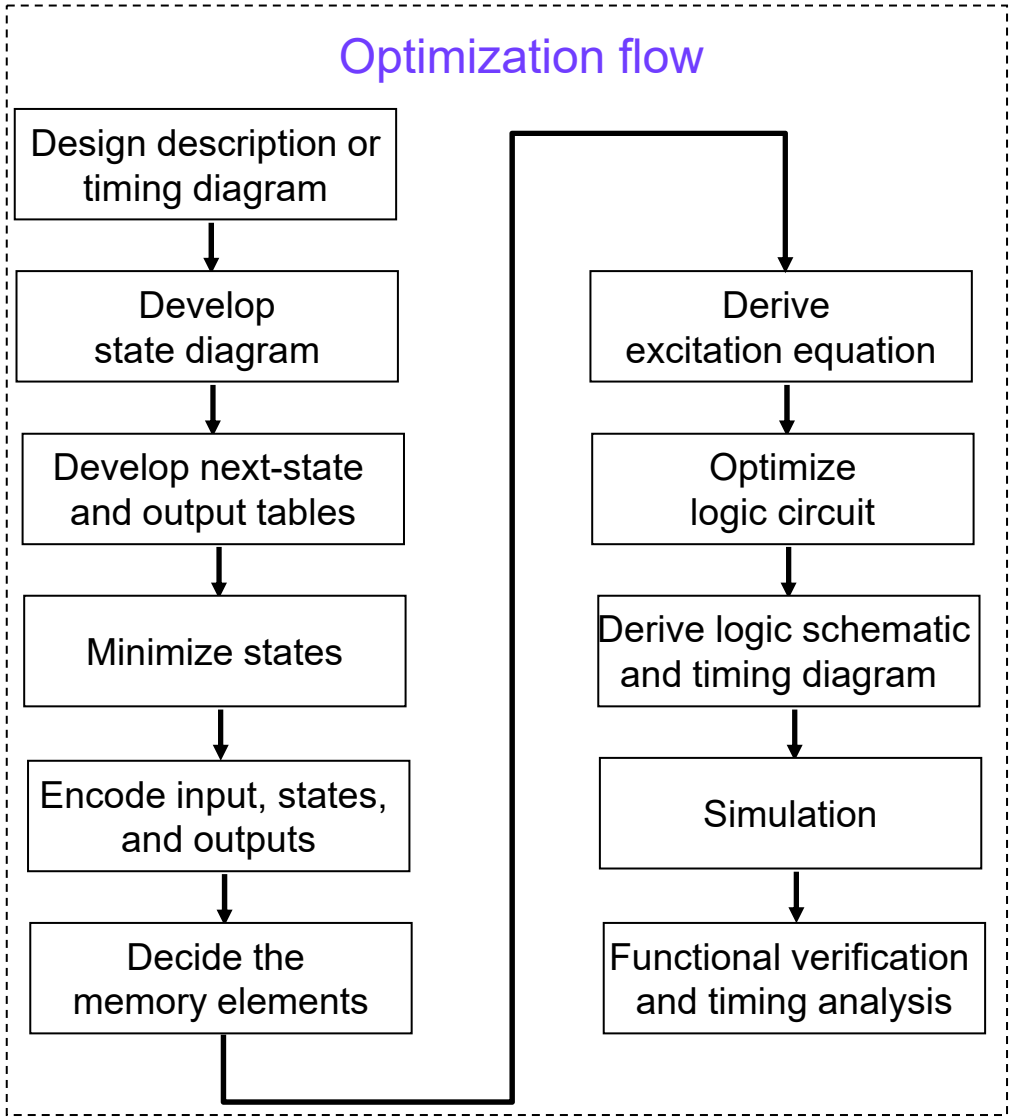
Next state logic (Comb.C.)

State name (parameter)

State register (Seq.C.)

Output logic (Comb.C.)

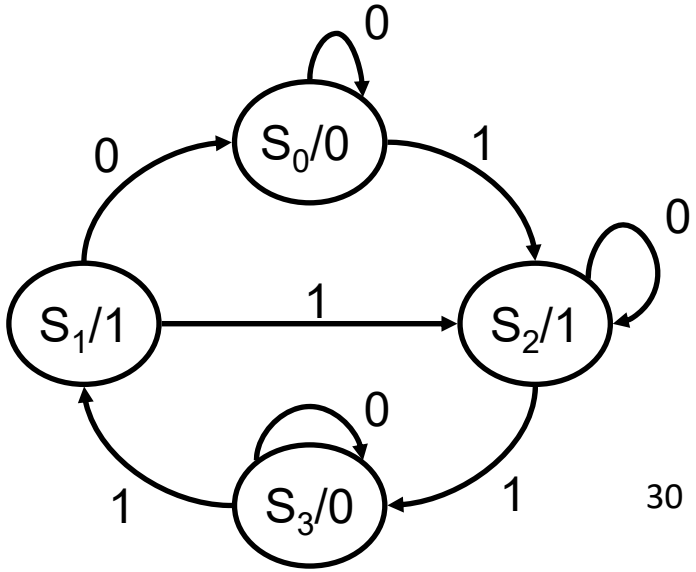
Moore Machine (1/8)



$S \rightarrow O \quad S:\text{state} \quad O:\text{output}$

Next-state and output tables (I=input)

Present State	Next State		Output
	I=0	I=1	I=0 or 1
S_0	S_0	S_2	0
S_1	S_0	S_2	1
S_2	S_2	S_3	1
S_3	S_3	S_1	0



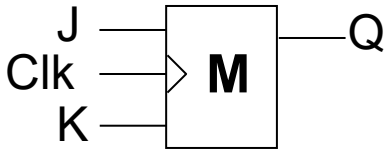
Moore Machine (2/8)

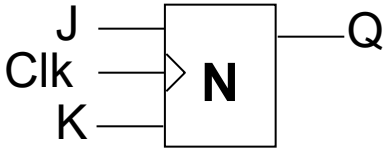
original state table

Present State	Next State		Output	
	I=0	I=1	I=0	I=1
S_0	S_0	S_2	0	0
S_1	S_0	S_2	1	1
S_2	S_2	S_3	1	1
S_3	S_3	S_1	0	0

Assume that we use JK flip-flops for storage

4 states ⇨ need 2 flip-flops (named M and N)





characteristic table

J	K	Q(t+1)
0	0	Q(t)
0	1	0
1	0	1
1	1	Q'(t)

excitation table

Q(t)	Q(t+1)	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

I	Present State		Next State		M(JK)		N(JK)		Output
	M(t)	N(t)	M(t+1)	N(t+1)	MJ	MK	NJ	NK	
0	0	0	0	0	0	X	0	X	0
1	0	0	1	0	1	X	0	X	0
0	0	1	0	0	0	X	X	1	1
1	0	1	1	0	1	X	X	1	1
0	1	0	1	0	X	0	0	X	1
1	1	0	1	1	X	0	1	X	1
0	1	1	1	1	X	0	X	0	0
1	1	1	0	1	X	1	X	0	0

Moore Machine (3/8)

I \ MN	00	01	11	10
0	0	0	X	X
1	1	1	X	X

MJ=I

I \ MN	00	01	11	10
0	0	X	X	0
1	0	X	X	1

NJ=MI

I \ MN	00	01	11	10
0	X	X	0	0
1	X	X	1	0

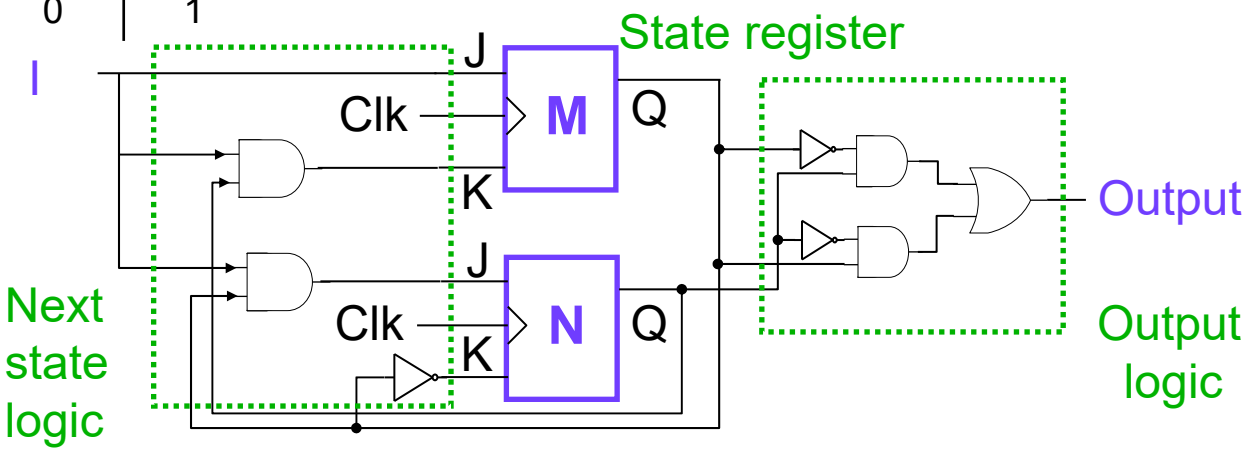
MK=NI

I \ MN	00	01	11	10
0	X	1	0	X
1	X	1	0	X

NK=M’

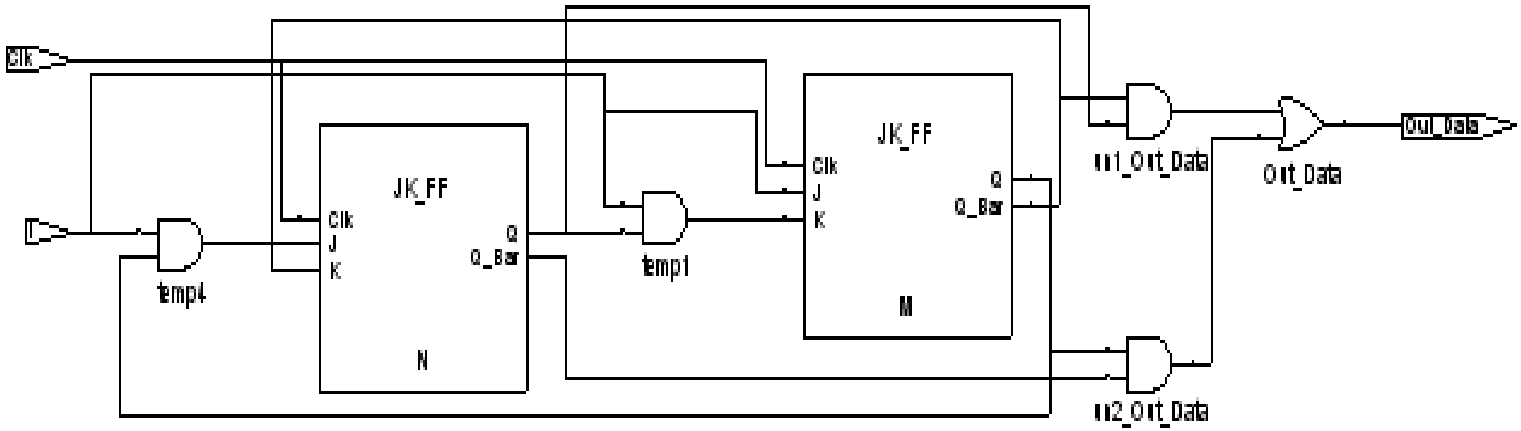
I \ MN	00	01	11	10
0	0	1	0	1
1	0	1	0	1

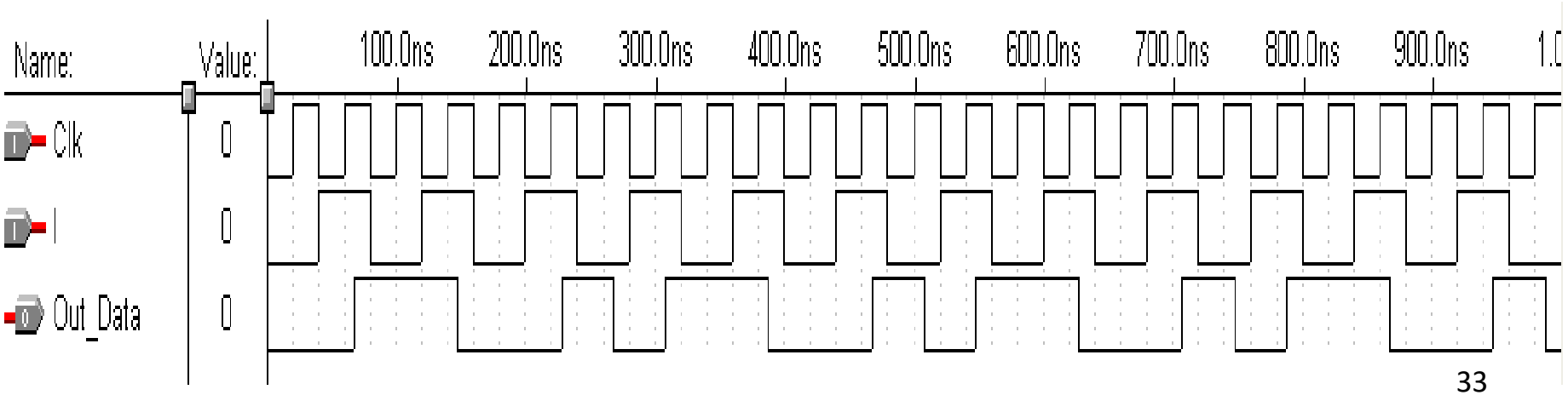
Output=M’N+MN’



Moore Machine (4/8)

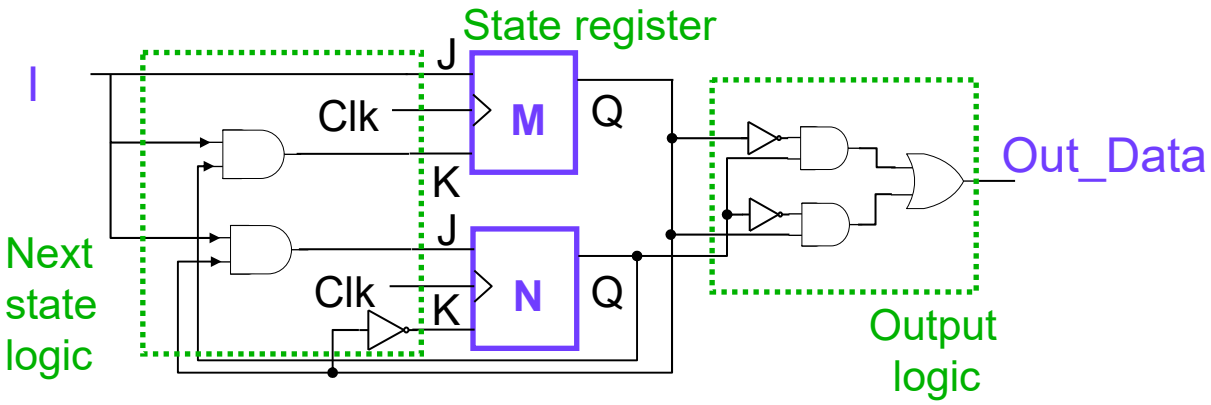
Synthesis Result





Implement the circuit with structural HDL, See circuit in the previous slide

Moore Machine (5/8)




```
module moore_JK(Clk, I, Out_Data);
input   Clk, I; output Out_Data;
wire    temp1, temp2, temp3, temp4, temp5, temp6;
assign temp1 = I & temp5;
assign temp4 = I & temp2;
assign Out_Data = (temp3 &  temp5) | (temp2 &
temp6);
JK_FF M(Clk, I, temp1, temp2,  temp3);
JK_FF N(Clk, temp4, temp3, temp5, temp6);
endmodule
```

```
module JK_FF(Clk, J,
K, Q, Q_Bar);
input   Clk, J, K;
output Q, Q_Bar;
reg      Q, Q_Bar;
always @(posedge Clk)
begin
  case({J,K})
    2'b00:
      Q=Q;
    2'b01:
      Q=0;
    2'b10:
      Q=1;
    2'b11:
      Q=~Q;
  endcase
end
endmodule
```

The better way is to write behavioral HDL directly and let the EDA tool do the whole optimization job (including Karnaugh Map and logic minimization)

Moore Machine-Bad Example (6/8)

Both State and Out_Data are implemented with flip-flops

```
module moore_bad(Clk,
Reset, In_Data, Out_Data);
input   Clk, Reset, In_Data;
output [1:0] Out_Data;
reg      [1:0] Out_Data;
reg      [1:0] State;
parameter S0=2'b00,
S1=2'b01, S2=2'b11,
S3=2'b10;
always @(posedge Clk)
begin
  if(Reset)
     State=S0;
  else begin
     case(State)
```

```
S0: begin
          Out_Data = 0;
          if(In_Data == 1)
               State = S2;
          else
               State = S0;
       end
S1: begin
        Out_Data = 1;
        if(In_Data == 1)
             State = S2;
        else
             State = S0;
      end
```

```
S2: begin
        Out_Data = 1;
        if(In_Data == 1)
             State = S3;
        else
             State = S2;
      end
S3: begin
        Out_Data = 0;
        if(In_Data == 1)
             State = S1;
        else
             State = S3;
      end
endcase
end
end
endmodule
```

Note: This is a bad-style HDL

Moore Machine-Good Example (7/8)

```
module moore_good(Clk,
    Reset, In_Data, Out_Data);

input Clk, Reset, In_Data;
output [1:0] Out_Data;
reg [1:0] Out_Data;
reg [1:0] State, NextState;
parameter S0=2'b00, S1=2'b01,
          S2=2'b10, S3=2'b11;
```

```
always @(posedge Clk or
            posedge Reset)
begin
  if(Reset)
    State <= S0;
  else
    State <= NextState;
end
```

State register (flip-flops)

```
always @(In_Data or State)
begin
  case(State)
    S0: begin
          if(In_Data == 1)
            NextState = S2;
          else
            NextState = S0;
        end
    S1:begin
          if(In_Data == 1)
            NextState = S2;
          else
            NextState = S0;
        end
    S2: begin
          if(In_Data == 1)
            NextState = S3;
          else
            NextState = S2;
        end
    S3: begin
          if(In_Data == 1)
            NextState = S1;
          else
            NextState = S3;
        end
  endcase
end
```

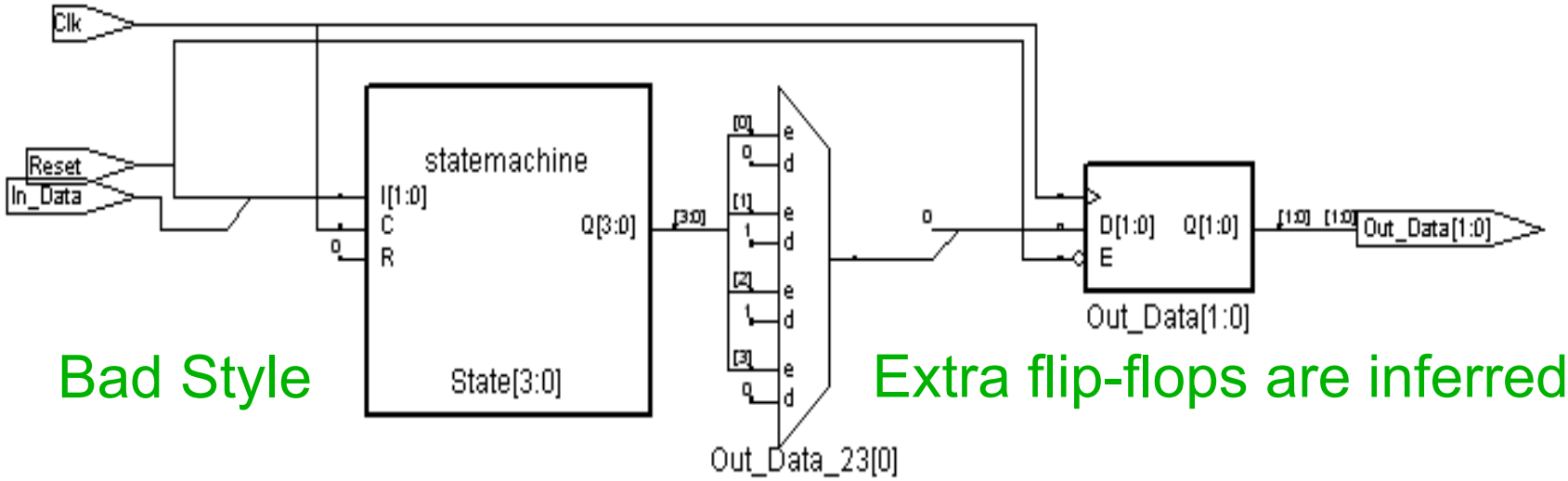
Next state logic

```
always @(State)
begin
  case(State)
    S0:Out_Data = 0;
    S1:Out_Data = 1;
    S2:Out_Data = 1;
    S3:Out_Data = 0;
    endcase
end
endmodule
```

Output logic

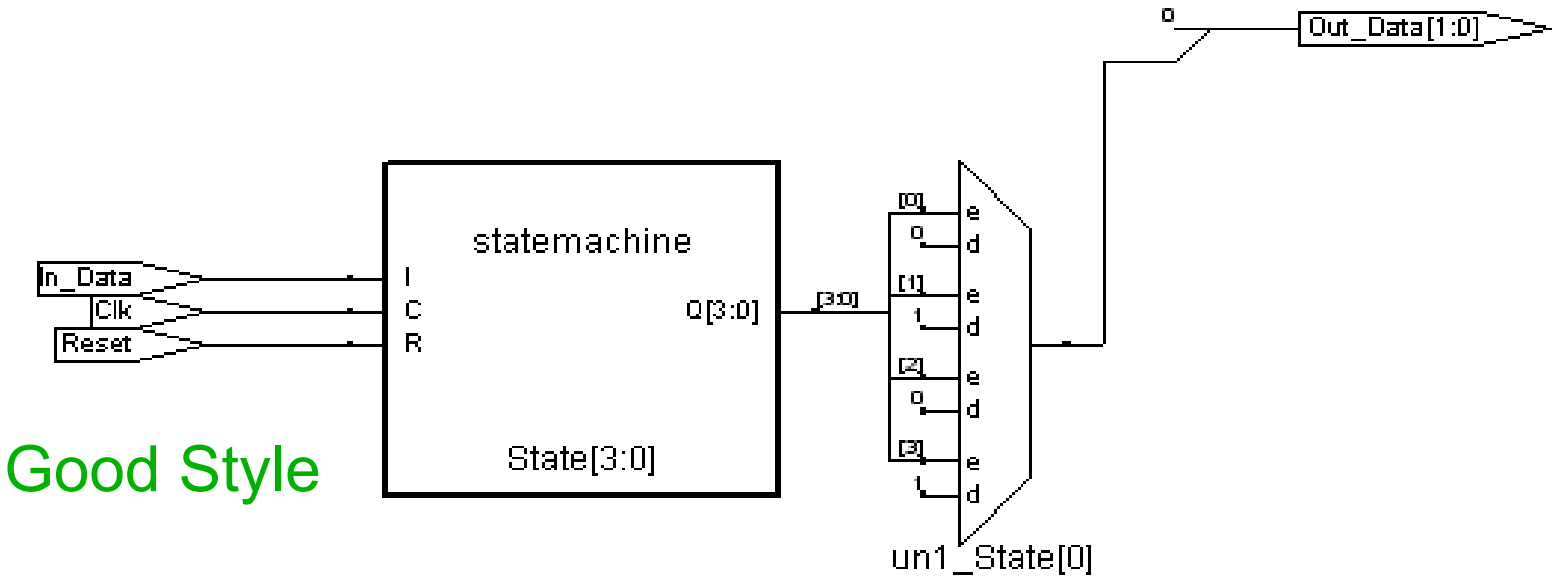
Note: This is a good-style HDL (only "State" is implemented with flip-flops)

Moore Machine-Good Example (8/8)



Bad Style

Extra flip-flops are inferred



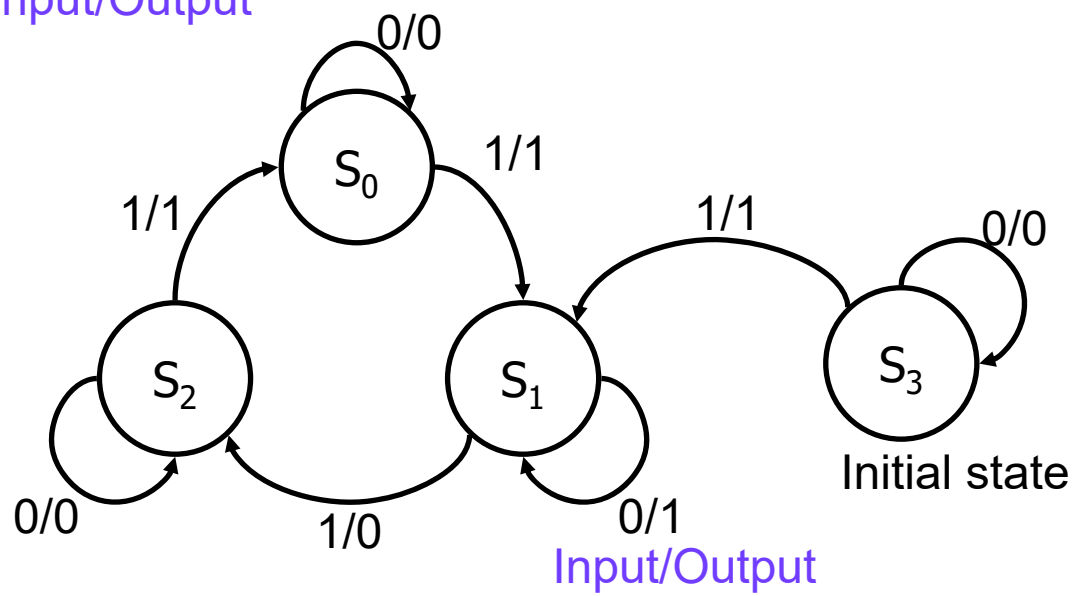
Good Style

Mealy Machine (1/2)

State diagram

Four states: S_0, S_1, S_2, S_3

Input/Output

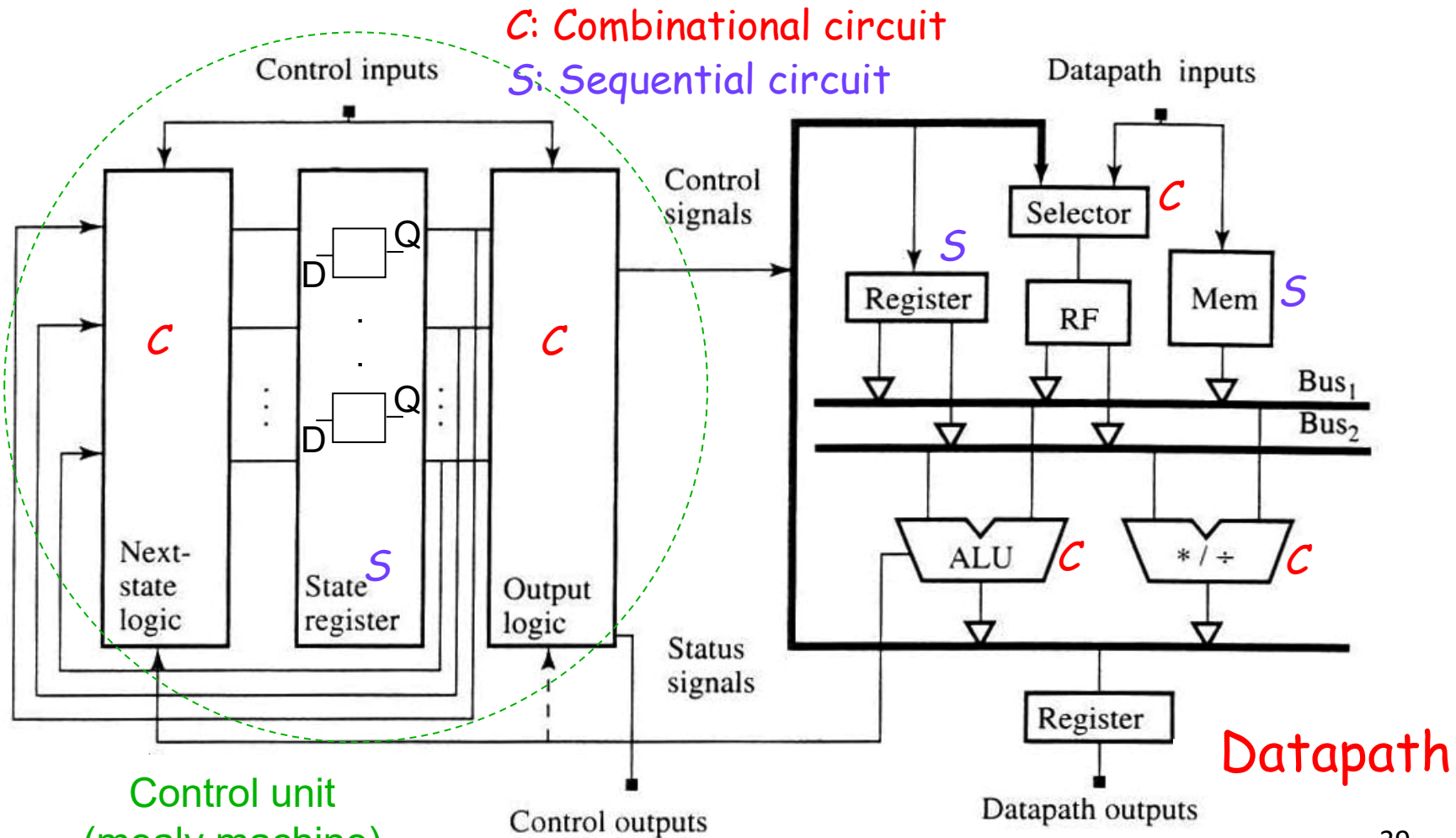


Next-state and output tables (I=input)

Present State	Next State		Output	
	I=0	I=1	I=0	I=1
S_0	S_0	S_1	0	1
S_1	S_1	S_2	1	0
S_2	S_2	S_0	0	1
S_3	S_3	S_1	0	1

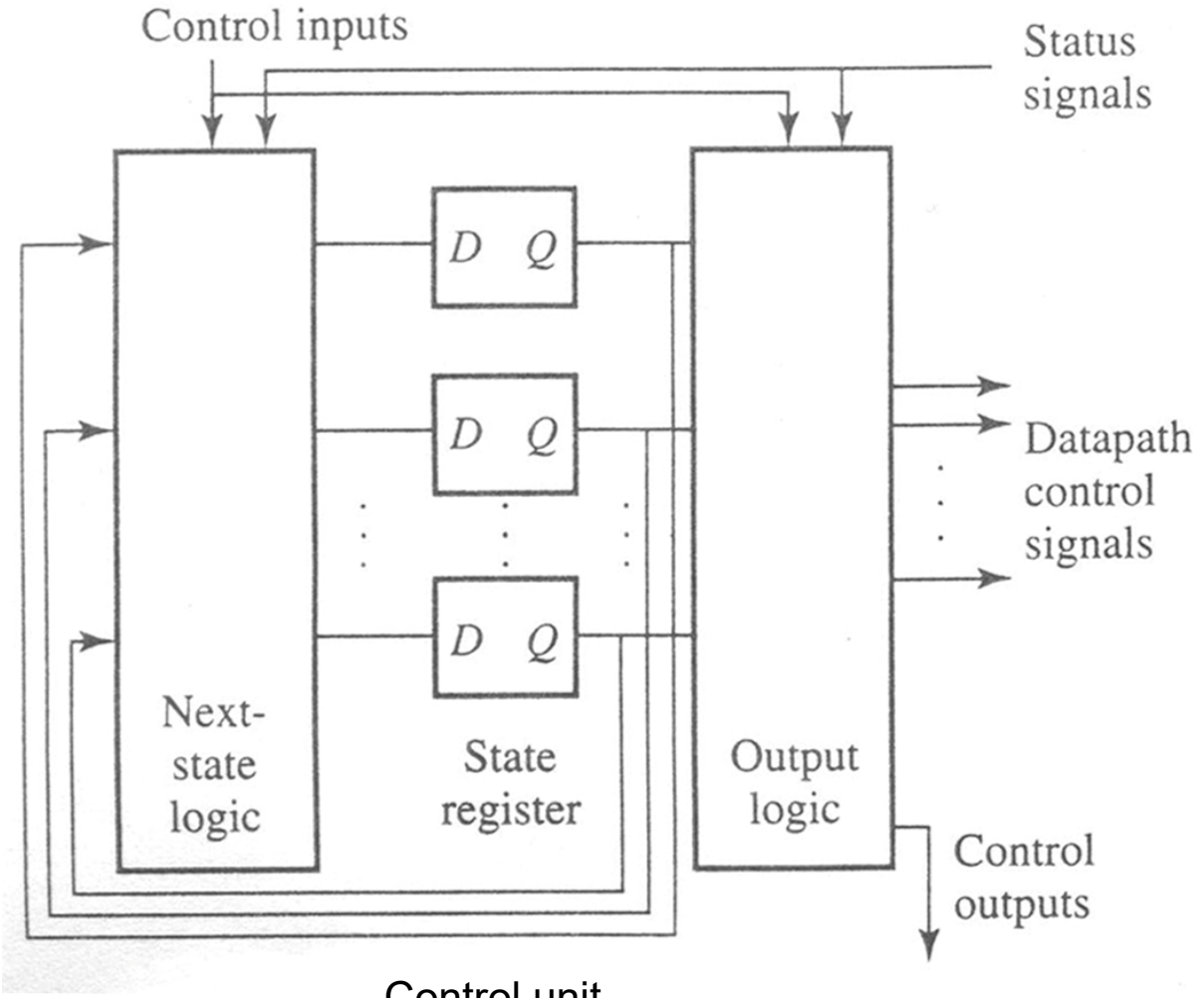
Mealy Machine (2/2)

Remember to write your mealy machine by using the good-style HDL



Control-Unit Implementation Styles (1/3)

Hardwired Control

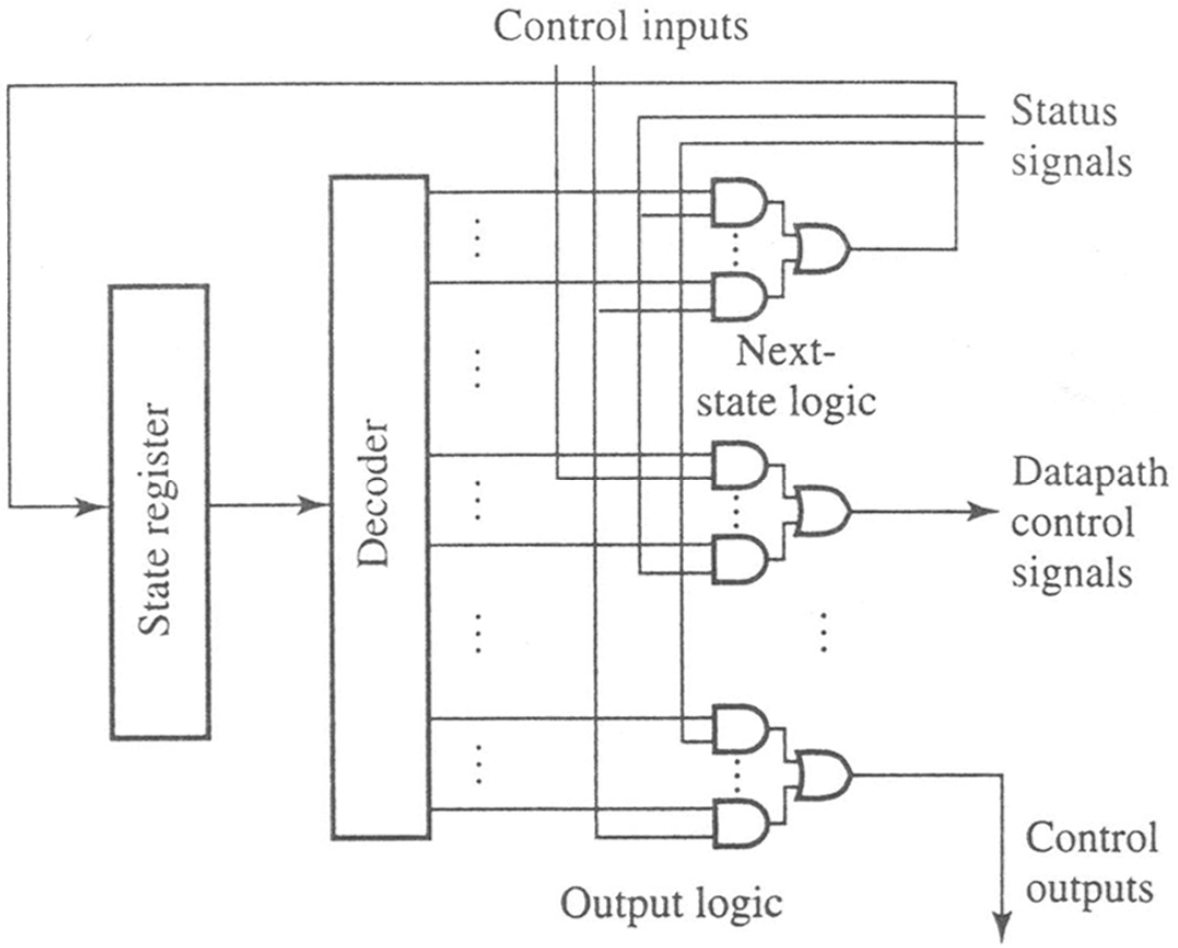


Control unit

Control-Unit Implementation Styles (2/3)

Hardwired Control

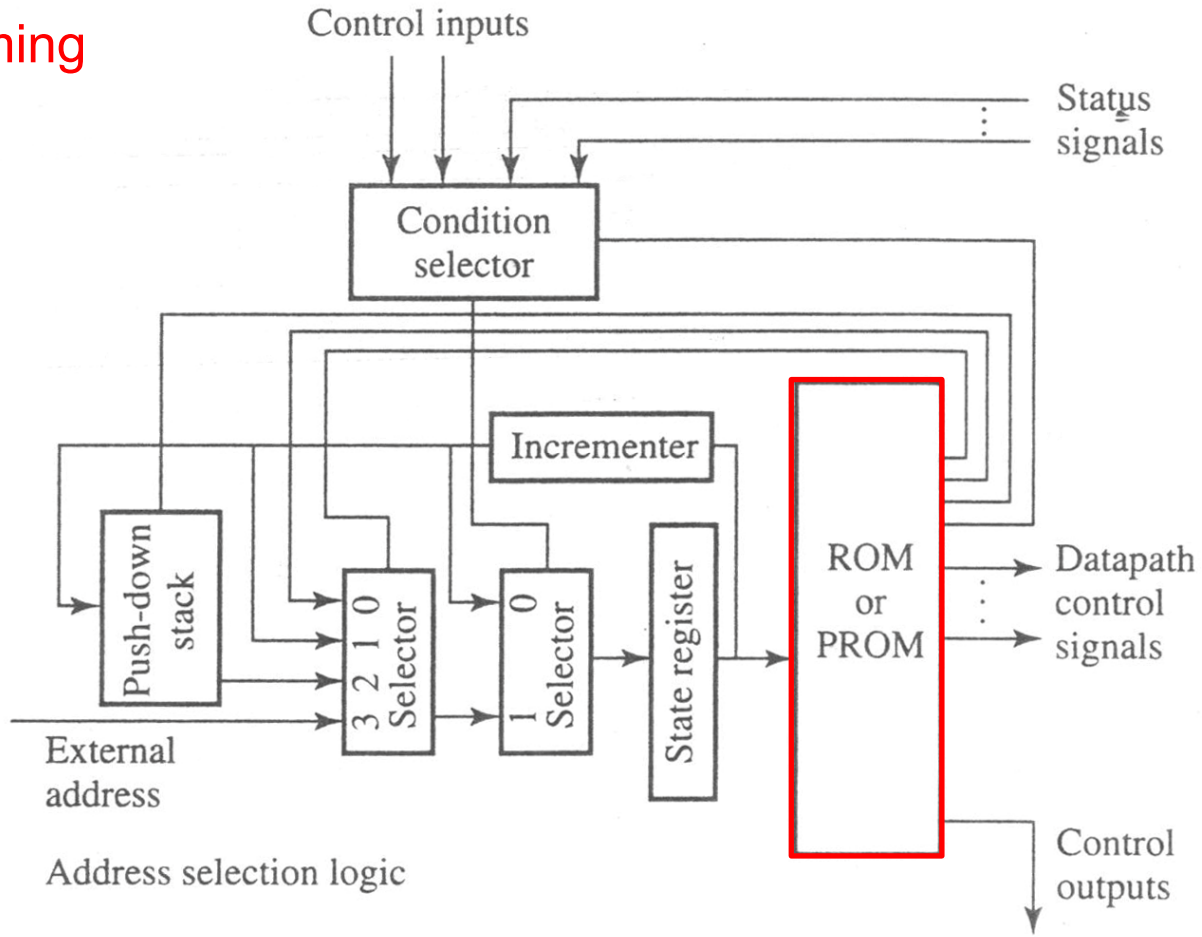
Control unit with state-register and decoder



Control unit with state-register and decoder

Control-Unit Implementation Styles (3/3)

Microprogramming Control

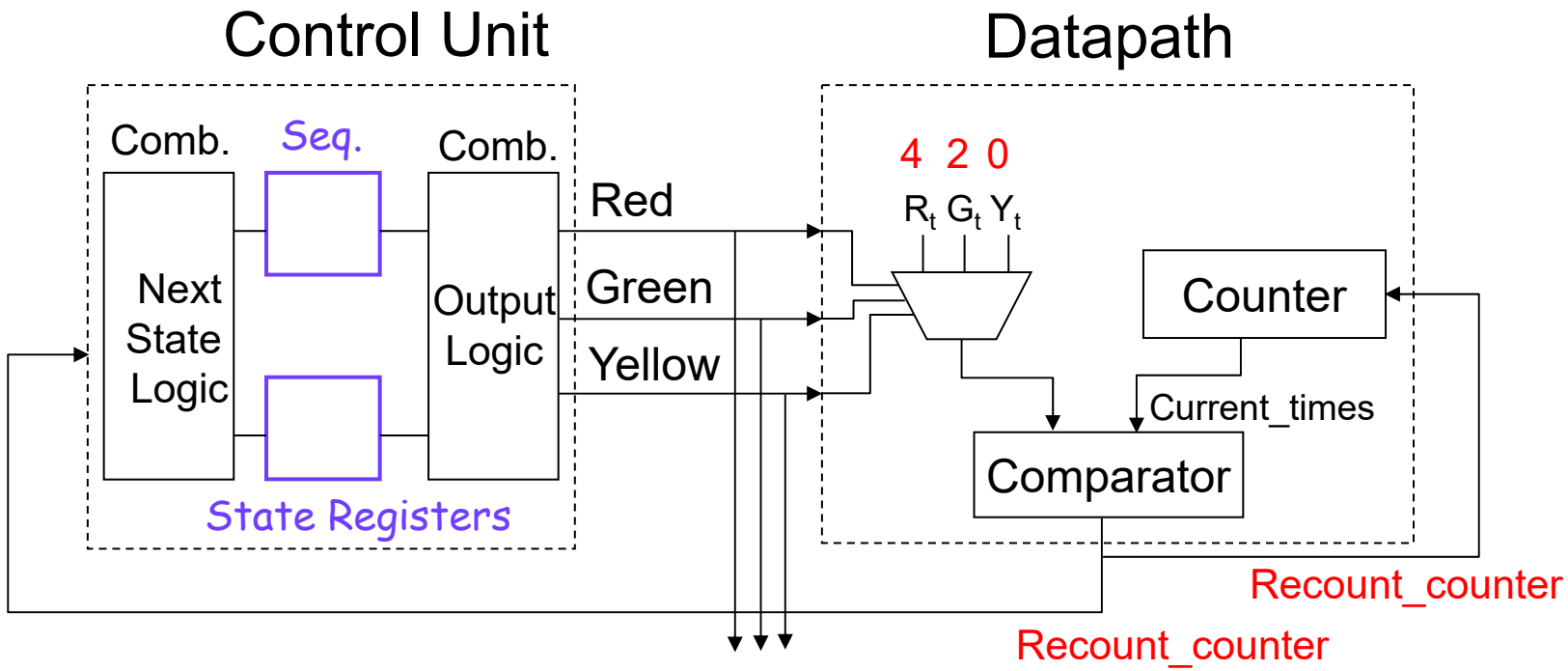


Control unit with state-register and ROM

Example

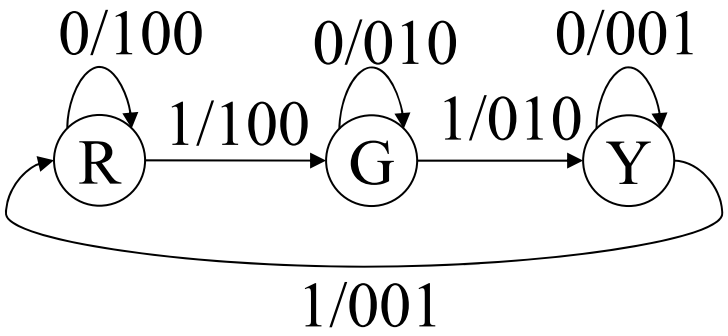
- Design a traffic light controller which has the following behavior
  - Two input signals (Clock, Reset) are for clock in and reset signal.
  - Three output signals (Red, Yellow, Green) enable each of red light, yellow light and green light.
  - The time with each traffic light is 5 seconds for red light, 3 seconds for green light and 1 second for yellow light.
  - The sequence of traffic light is in specific order (red, yellow, green and repeat).

Traffic Light Controller (1/7)



Input/Output

Recount_Counter16/Red Green Yellow



R_time: 4+1=5 cycles
G_time: 2+1=3 cycles
Y_time: 0+1=1 cycles

Traffic Light Controller (2/7)

```
module  traffic(Clock,Reset,Red,Green,Yellow);
input Clock,Reset;  output Red,Green,Yellow;
wire Recount_conter;  wire [3:0]
Counter_Number;

Traffic_Control (.Clock(Clock),.Reset(Reset),
.Recount_Counter16(Recount_conter),.Red(Re
d), .Green(Green),.Yellow(Yellow));

Datapath
(.Clock(Clock), .Reset(Reset), .RGY({Red,Gree
n,Yellow}), .Recount(Recount_conter));
endmodule
```

```
module Datapath(Clock, Reset, RGY,
Recount);
input Clock, Reset; input [2:0] RGY;
output Recount; wire [3:0] Counter_Number;

 Compare A1
(.current_times(Counter_Number),
  .RGY(RGY), .Recount_conter16(Recount)
);
Counter16 A2 (.Clock(Clock),.Reset(Reset),
   .Recount_Counter16(Recount), .Count_O
ut(Counter_Number));

endmodule
```

Traffic Light Controller (3/7)

```
module  Counter16(Clock,Reset,Recount_Counter16,
                  Count_Out);
input Clock,Reset,Recount_Counter16;
output [3:0] Count_Out;
reg [3:0] Count_Out;

always@(posedge Clock)
begin
    if(Reset)
      Count_Out=0;
    else
      begin
          if(Recount_Counter16)
            Count_Out=0;
          else
            Count_Out=Count_Out+1;
      end
end
endmodule
```

Traffic Light Controller (4/7)

```
module compare(current_times,
RGY, Recount_conter16);
input [2:0] RGY;
input [3:0] current_times;
output Recount_conter16;
reg Recount_conter16;
parameter  R_times=4, G_times=2,
Y_times=0;

always @(RGY)
begin
  case(RGY)
    3'b100:begin
            if(current_times ==R_times)
              Recount_conter16=1;
            else
              Recount_conter16=0;
            end
    3'b001:begin
            if(current_times ==Y_times)
              Recount_conter16=1;
            else
              Recount_conter16=0;
            end
3'b010:begin
            if(current_times ==G_times)
              Recount_conter16=1;
            else
              Recount_conter16=0;
    end
    default: Recount_conter16=1;
  endcase
end
endmodule
```

Traffic Light Controller (5/7)

```
module  Traffic_Control(Clock,Reset,
     Recount_Counter16,Red,Green,Yellow);
input Clock, Reset,Recount_Counter16;
output Red, Green, Yellow;
reg Red, Green, Yellow;
reg [1:0]   currentstate,nextstate;

parameter [1:0] Red_Light=0, Green_Light=1,
                      Yellow_Light=2;

always@(posedge Clock)
begin
    if(Reset)
      currentstate = Red_Light;
    else
      currentstate = nextstate;
end
```

State Register (Seq. C.)

```
always@(currentstate)
begin
    case(currentstate)
       Red_Light:begin
            if(Recount_Counter16)
                 nextstate=Green_Light;
            else
                nextstate=Red_Light;  end
       Green_Light:begin
            if(Recount_Counter16)
                nextstate=Yellow_Light;
            else
                 nextstate=Green_Light; end
       Yellow_Light:begin
            if(Recount_Counter16)
                 nextstate=Red_Light;
            else
                 nextstate=Yellow_Light; end
        default: nextstate=Red_Light;
    endcase
end
```

Next State Logic (Comb. C.)

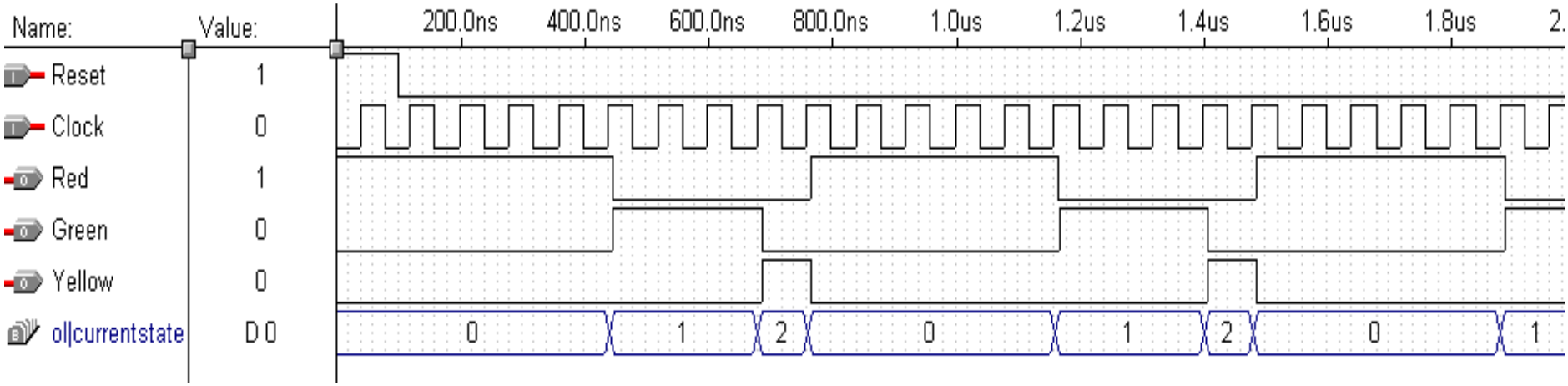
Traffic Light Controller (6/7)

```
always @(currentstate)
begin
  case(currentstate)
    Red_Light:begin
              Red=1'b1;
              Green=1'b0;
              Yellow=1'b0;
    end
    Green_Light:begin
              Red=1'b0;
              Green=1'b1;
              Yellow=1'b0;
    end
    Yellow_Light:begin
              Red=1'b0;
              Green=1'b0;
              Yellow=1'b1;
    end
    default:begin
              Red=1'b0;
              Green=1'b0;
              Yellow=1'b0;
    end
  endcase
end
endmodule
```

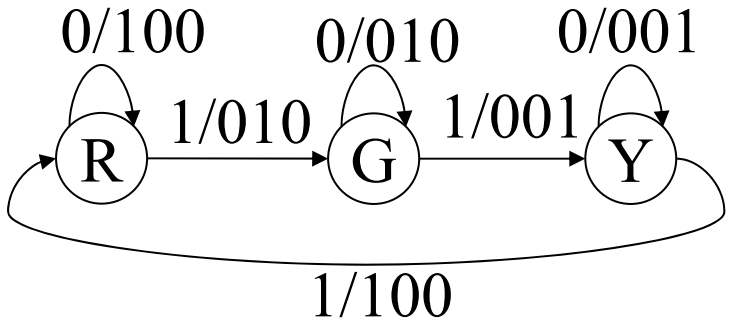
Output Logic (Comb. C.)

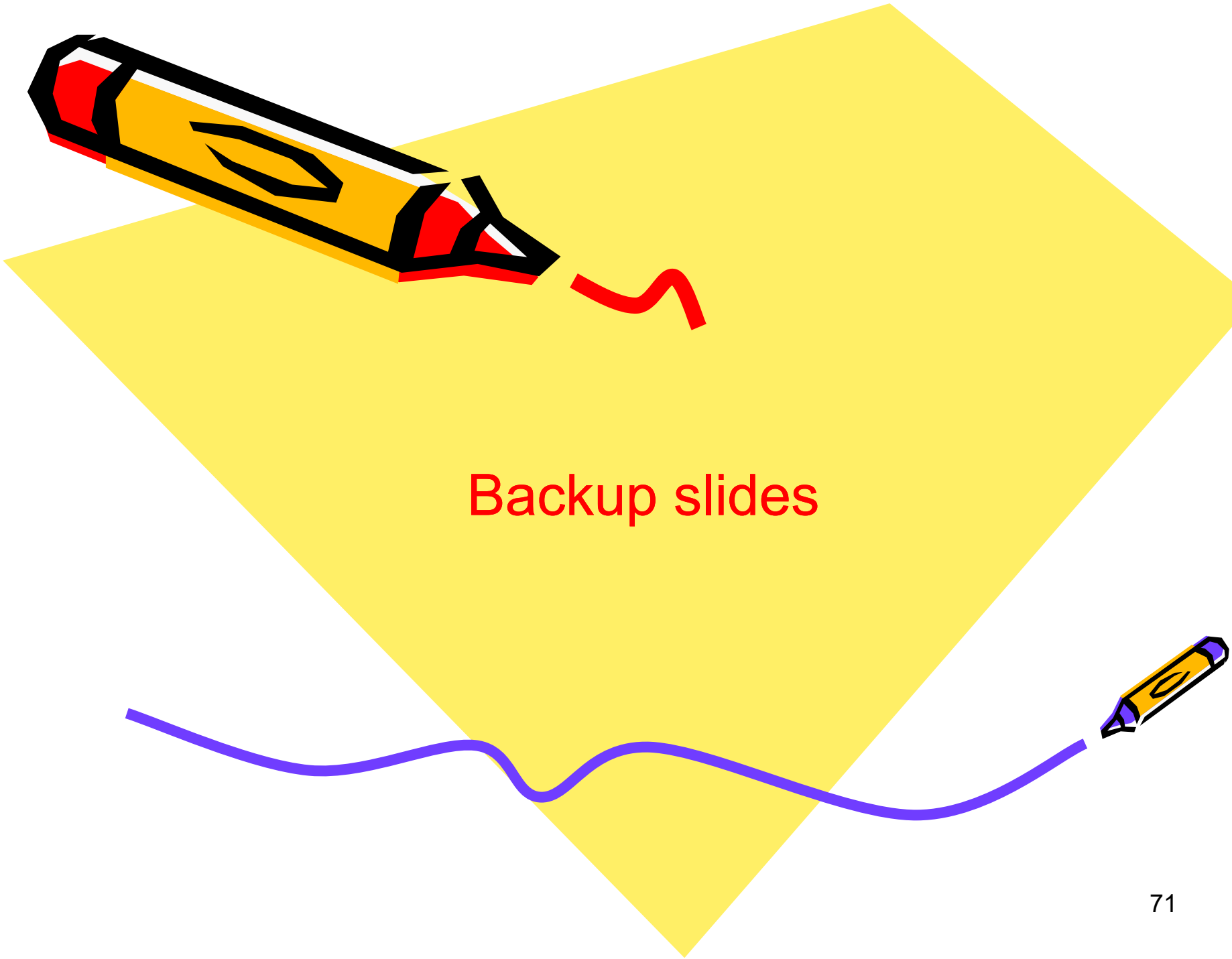
Traffic Light Controller (7/7)

R_time: 4+1=5 cycles
G_time: 2+1=3 cycles
Y_time: 0+1=1 cycles



Input/Output Recount_Counter16/Red Green Yellow




Backup slides