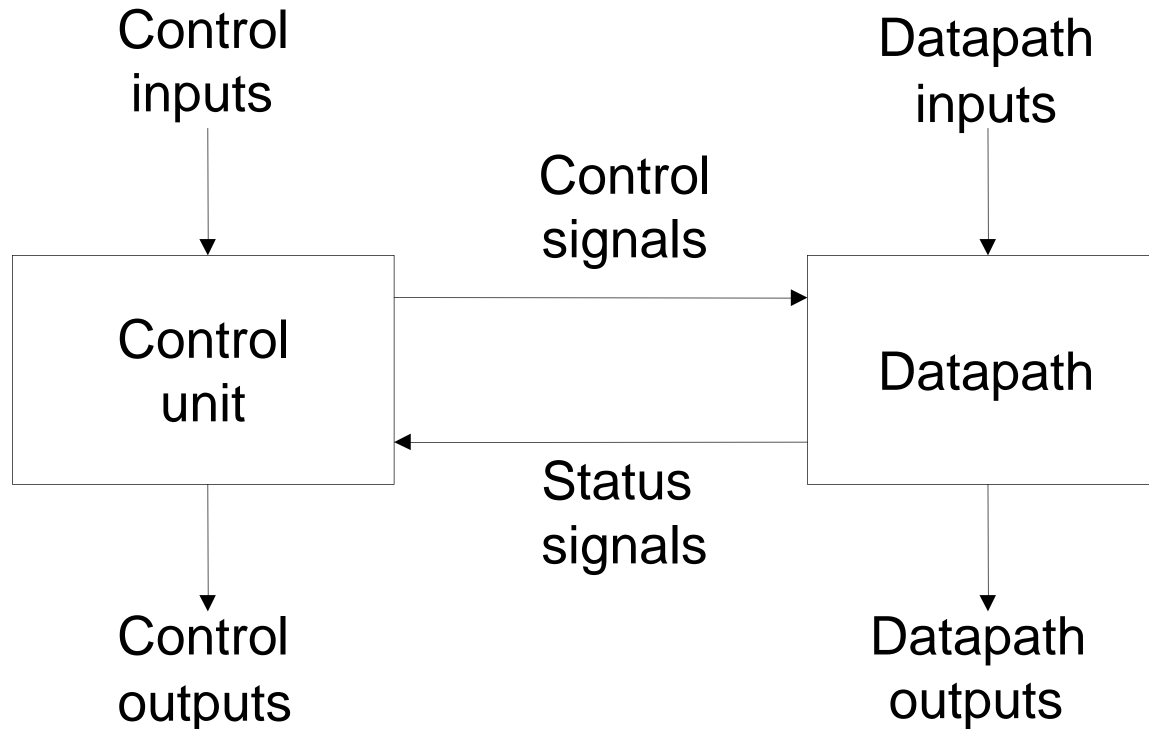
FPGA系統設計

Control Unit

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

Control inputs

Datapath inputs

Control signals

Status signals

Next-state logic (C)

State register (S)

D Q

D Q

Output logic (C)

Selector (C)

Register (S)

RF

Mem (S)

Bus_1

Bus_2

ALU (C)

* / ÷ (C)

Register

Control Unit

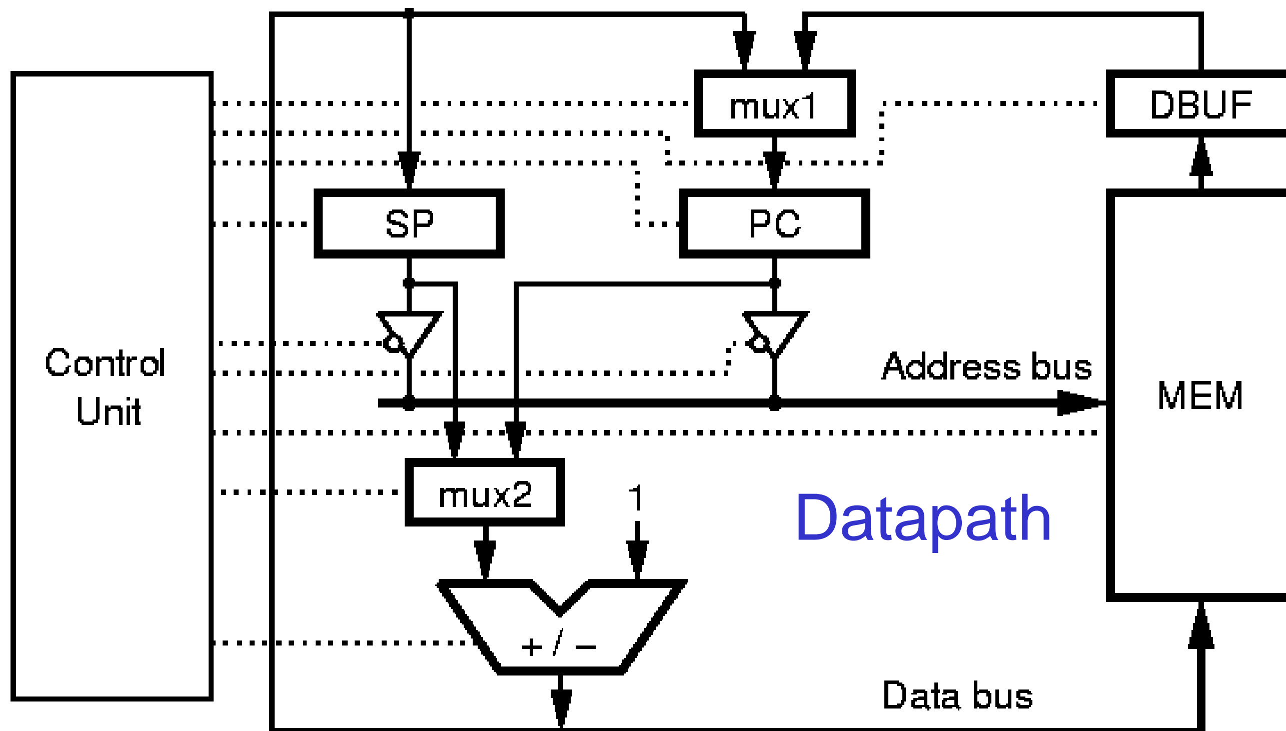
Control outputs

Datapath outputs

Datapath

Modern Design (3/3)

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

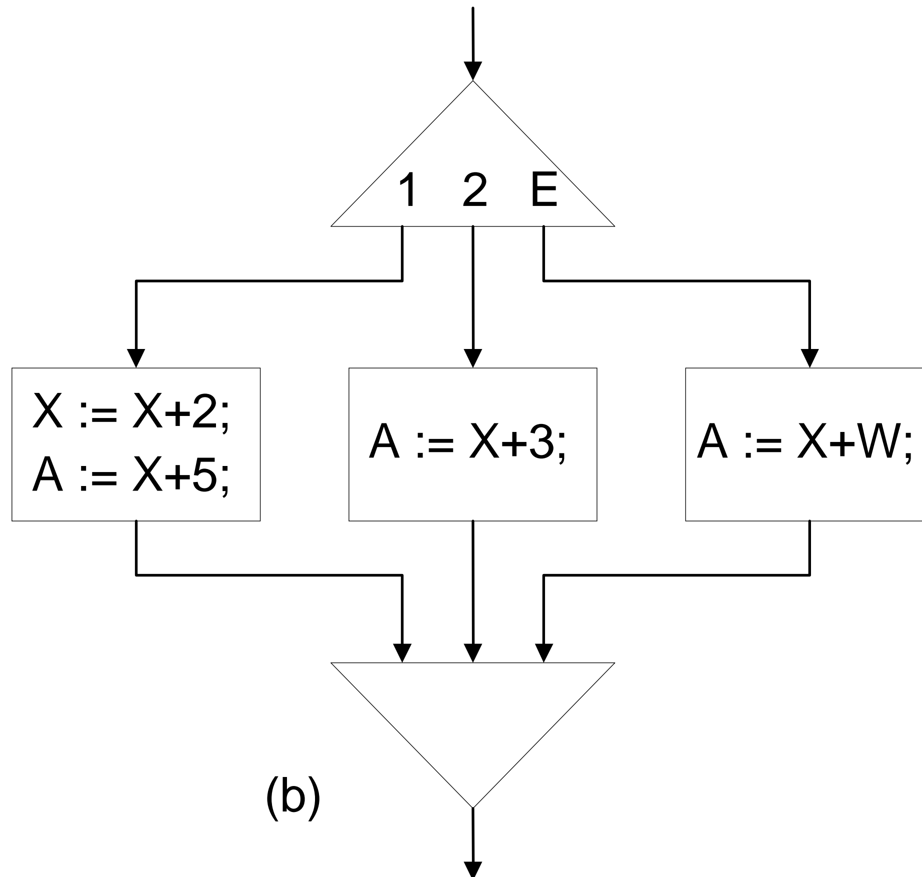


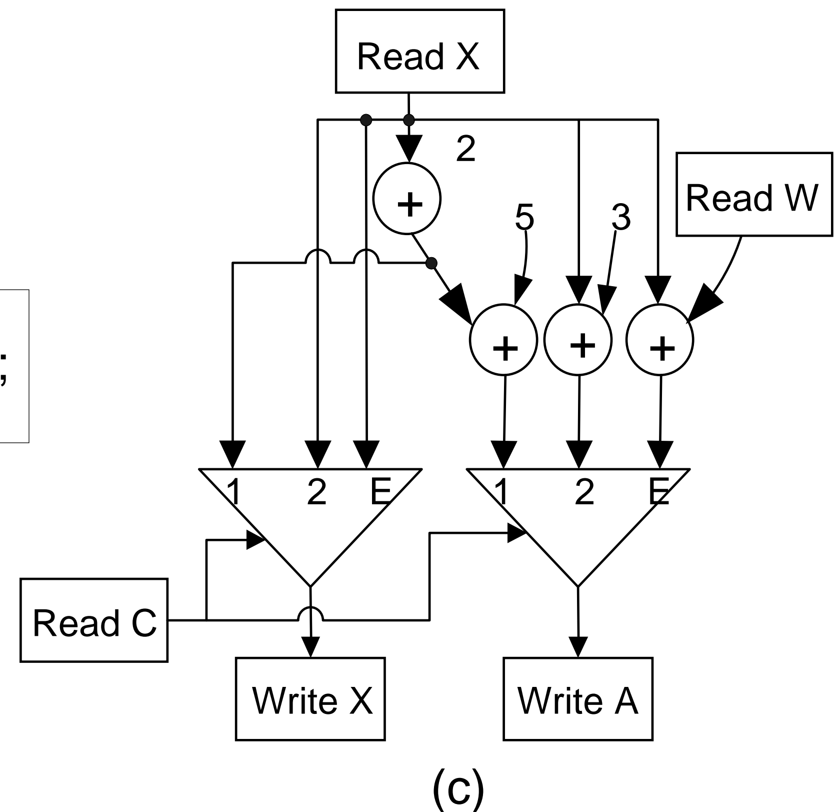
An synthesis example of case statement

- **(a) HDL description**
- **(b) Control-flow representation**
- **(c) Data-flow representation**

(a)

```
Case C is
    When 1=>      X : = X+2;
                  A : = X+5;
    When 2=>      A : = X+3;
    When others => A : = X+W;
end case;
```

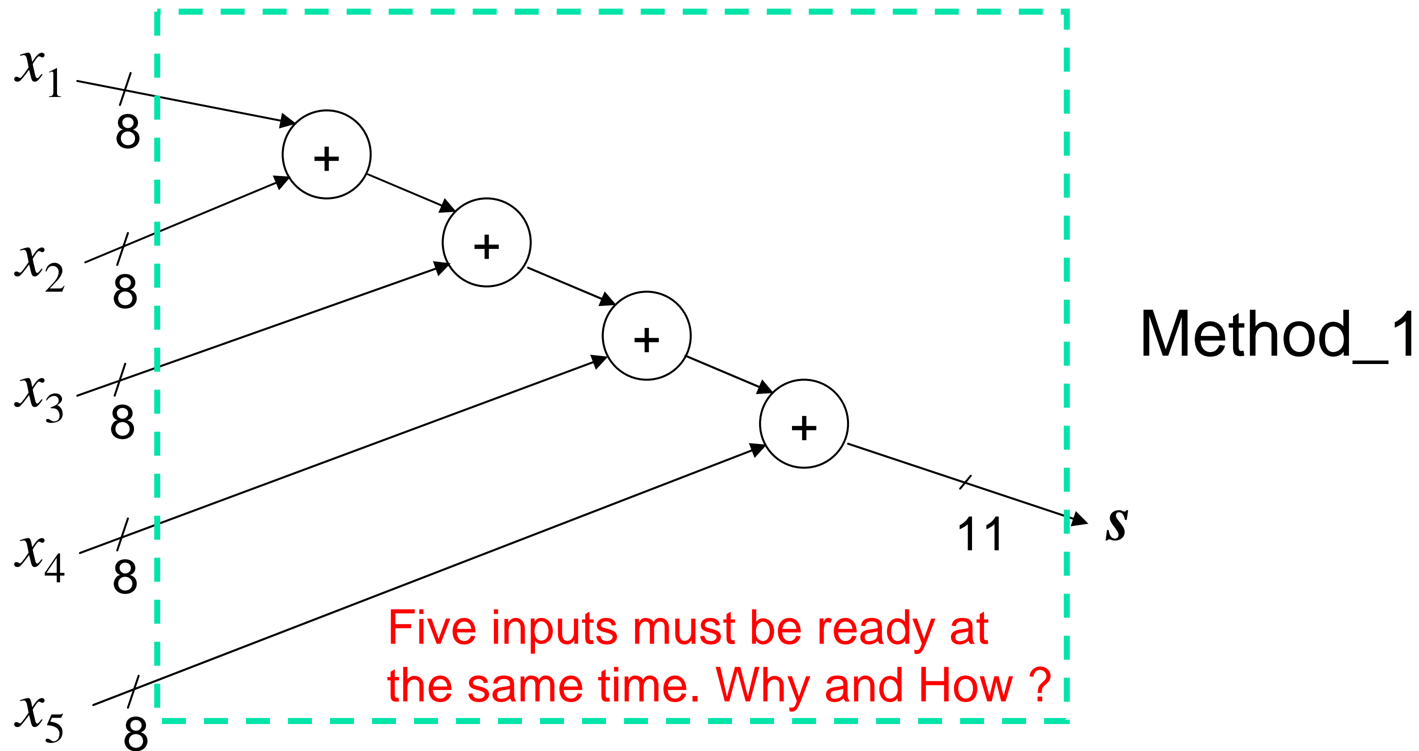




Summation Problem (1/4)

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

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



Method_1

a. How many input pins and output pins ?

b. What is the resolutions of x_i ?

c. How fast you need ?

d. What is your design cost ?

Summation Problem (2/4)

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

2. Sum up five inputs in the different time units by using only 1 adders

Initial $S = 0$

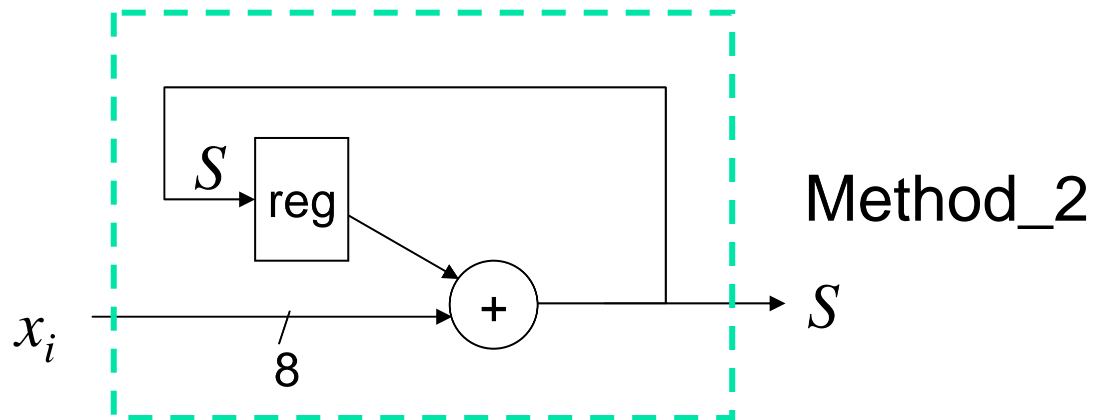
Time unit _1 $S <= S + x_1$

Time unit _2 $S <= S + x_2$

Time unit _3 $S <= S + x_3$

Time unit _4 $S <= S + x_4$

Time unit _5 $S <= S + x_5$

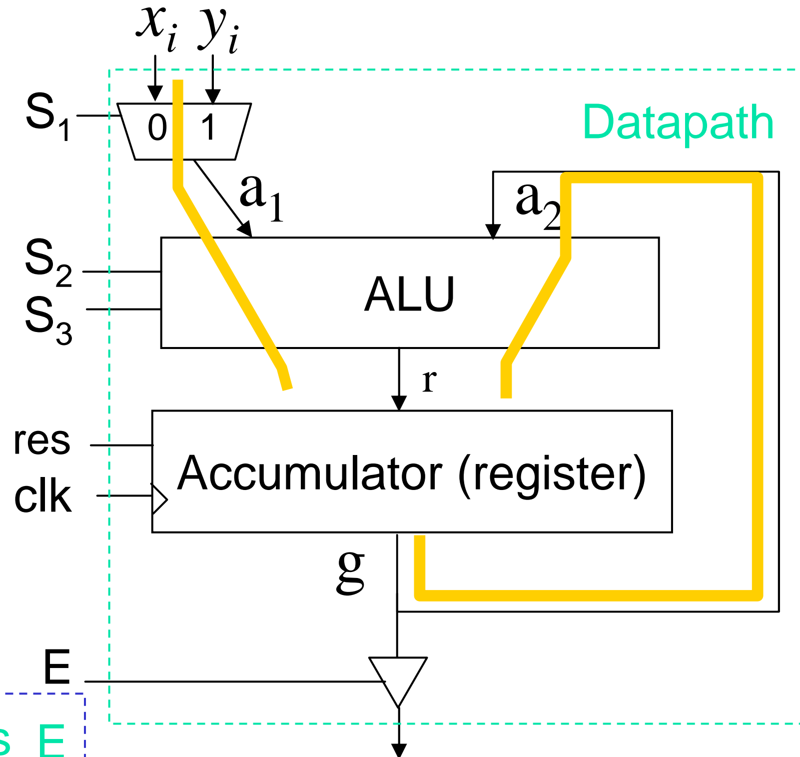


Cost is lower and critical path is shorter than the Method_1 .

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

Summation Problem (3/4)

Problem: **Calculate** $S = x_1 + x_2 + x_3 + x_4 + \ldots + x_{50}$ $\quad S = \sum_{i=1}^{50} x_i$



If $S_1=0$, $a_1=x_i$

If $S_1=1$, $a_1=y_i$

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

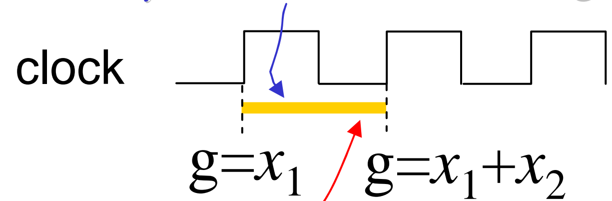
If $S_2=0$ $S_3=1$, $r = a_1+a_2$

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	res	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

New x_i must come 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

⇨ 倒數為clock rate

Summation Problem (4/4)

Control unit should send out proper control signals at each state.

There are two ways to generate those control signals:

(1) Microgramming control

a. Store control signals of each state at memory (ROM)

b. Read out the control signals one by one

word →

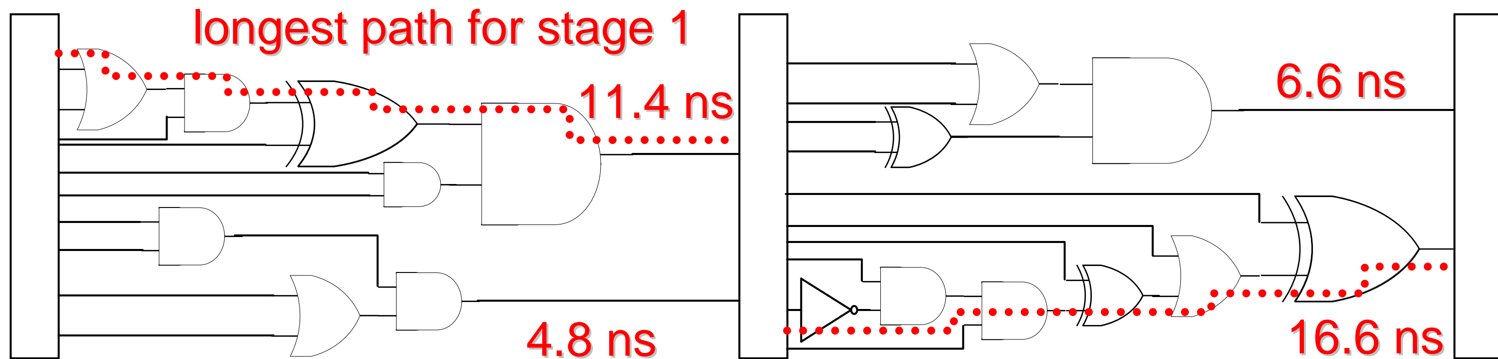
	S_1	S_2	S_3	res	E
T_0	0	0	0	1	0
T_1	0	1	1	0	0
$T_{2\sim49}$	0	0	1	0	0
…..					
T_51	X	0	0	0	1

(2) Hardwired control

Use dedicate logic gates to generate the proper signals state by state (one by one)

Clock Period (1/4)

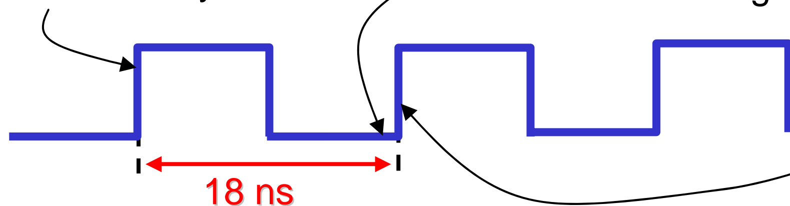
Gate:	not	and	or	xor
Delay:	1ns	2.4ns	2.4 ns	4.2ns



R1(Registers,p-trigger)　R2(p-trigger)　R3(p-trigger)

16.6 ns > 11.4ns

critical path=16.6 ns, so the clock period must be more than 16.6 ns (e.g., 18ns), why ?



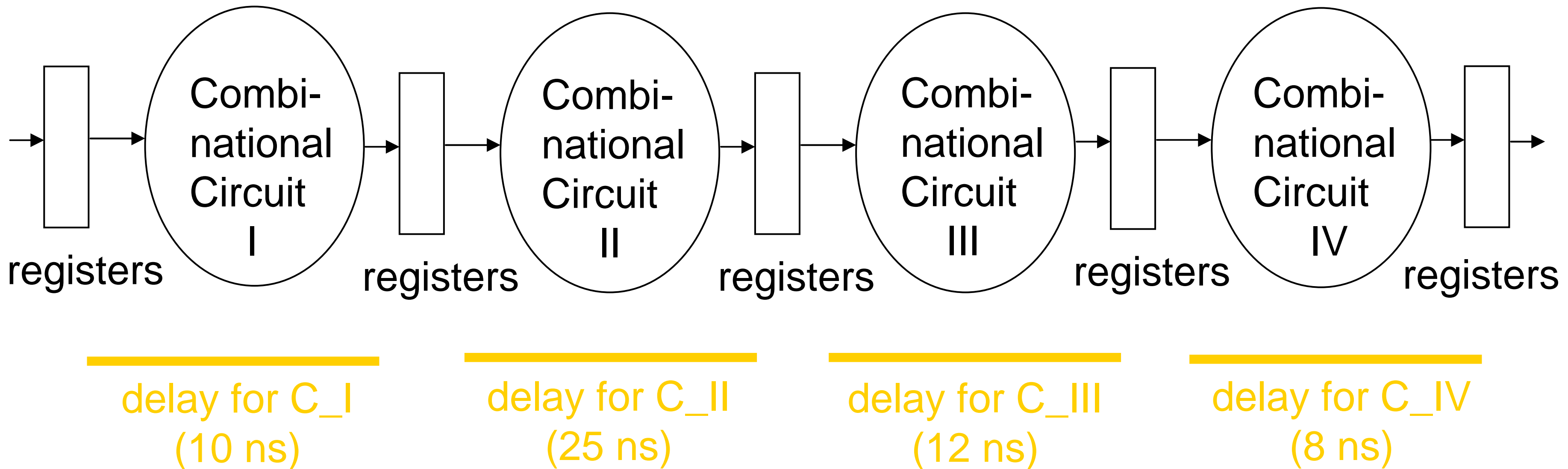
New value for R1 is ready here
New value for R2 is ready here

need 11.4 ns
Correct result for stage 1 must be ready here before next p-edge
Correct result for stage 2 must be ready here before next p-edge
need 16.6 ns

New value for R2 is ready here
New value for R3 is ready here

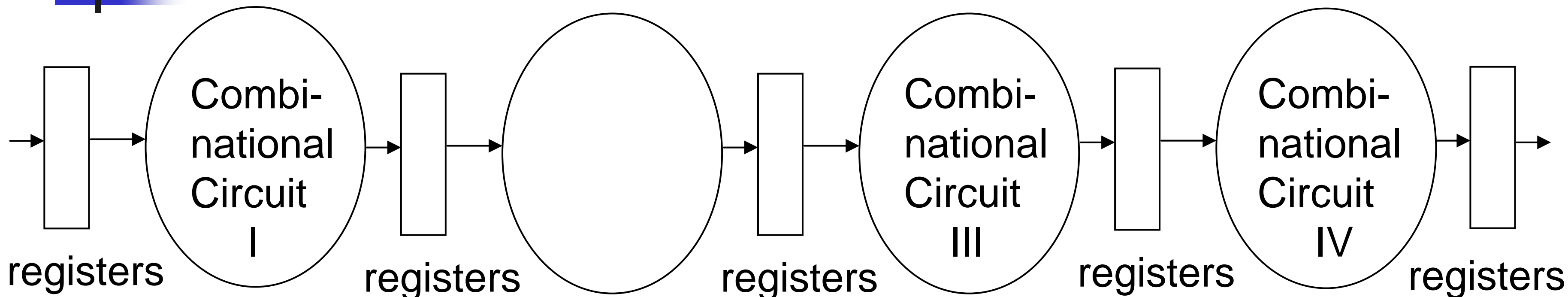
Clock Period (2/4)

How to decide the clock period in a system?



1. Find out the longest delay among combinational circuits C_I, C_II, C_III and C_IV.
2. The longest delay is named as the critical path (here is 25 ns).
3. The clock period can be set as little longer than the critical path, why?
4. $\text{clock frequency} = \dfrac{1}{\text{clock period}}$ $\left(\text{here } \dfrac{1}{25\text{ns}} = \dfrac{1}{25\times 10^{-9}} = 40 \text{ MHz}\right)$

Clock Period (3/4)

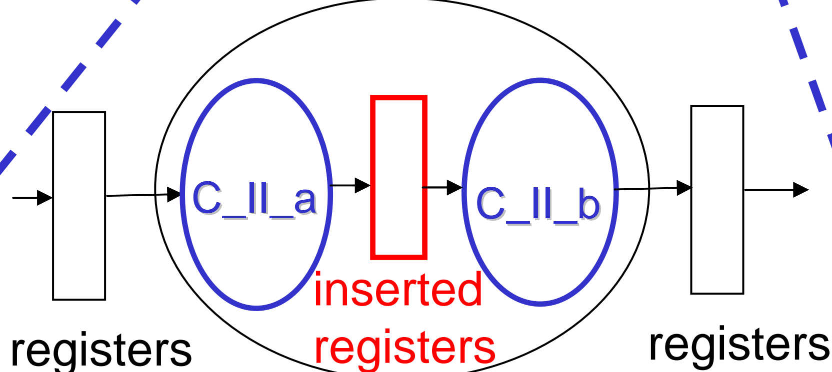


delay for C_I
(10 ns)

delay for C_II
(25 ns)

delay for C_III
(12 ns)

delay for C_IV
(8 ns)



delay for C_II_a
(14 ns)

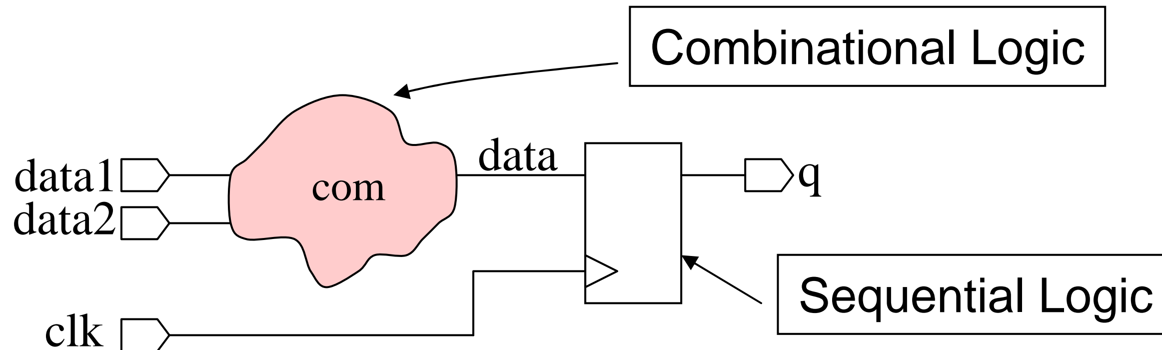
delay for C_II_b
(14 ns)

Separate C_II into two parts (C_II_a and C_II_b) by inserting proper registers to achieve faster clock frequency

Now, clock frequency is 71.4 MHz

Clock Period (4/4)

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 (always @(data1 or data2) block)

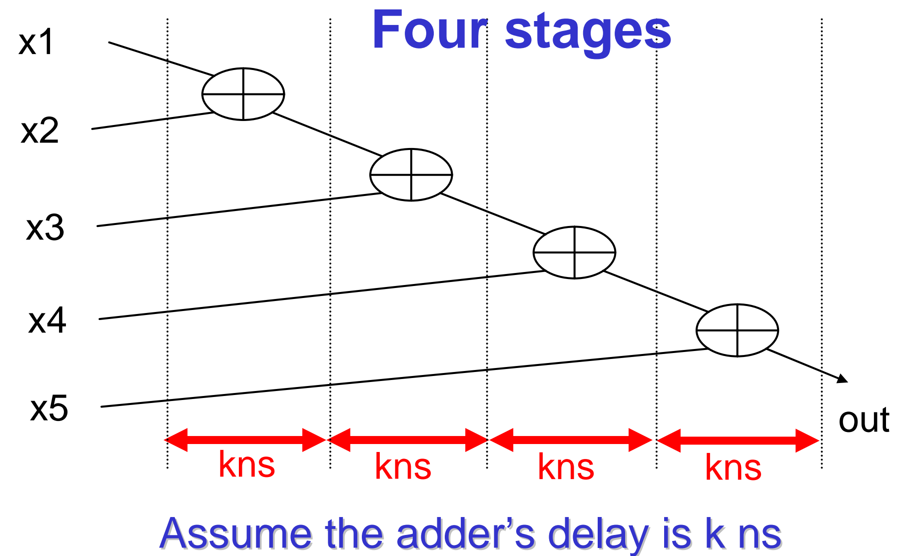
Sequential Logic (always @(posedge clk) block)

Design for Summation Problem (1/7)

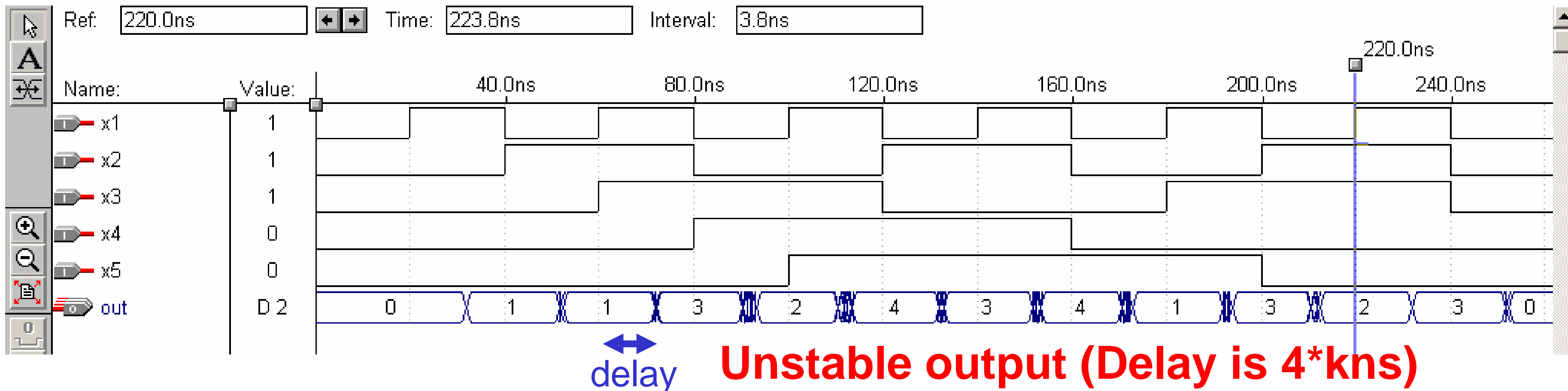
Method_1

```
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

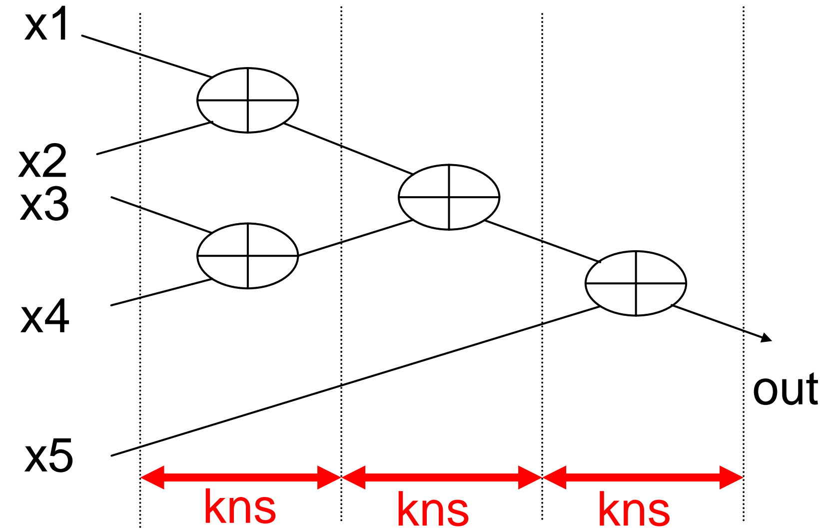


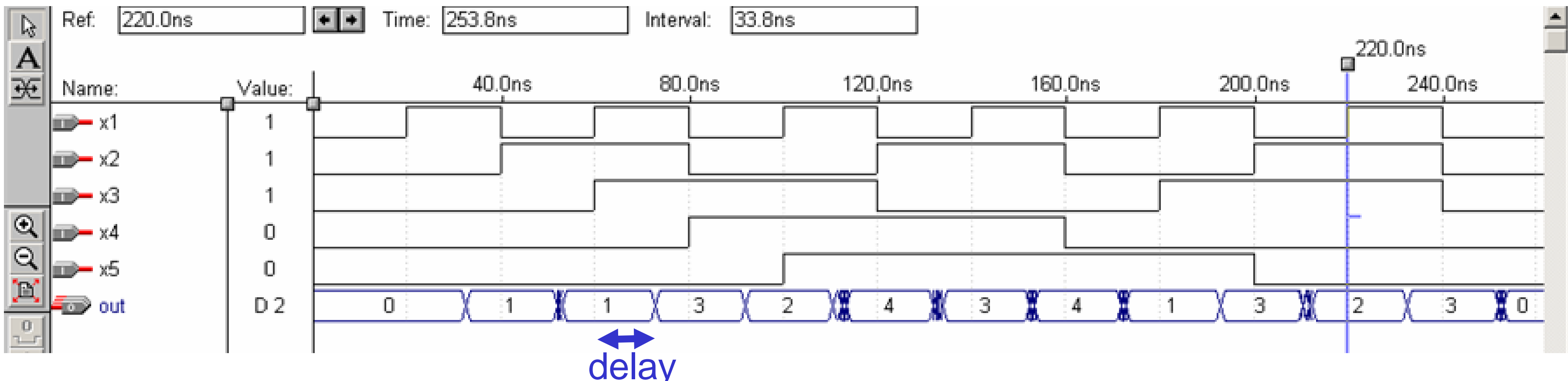
Design for Summation Problem (2/7)

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
```

Three stages



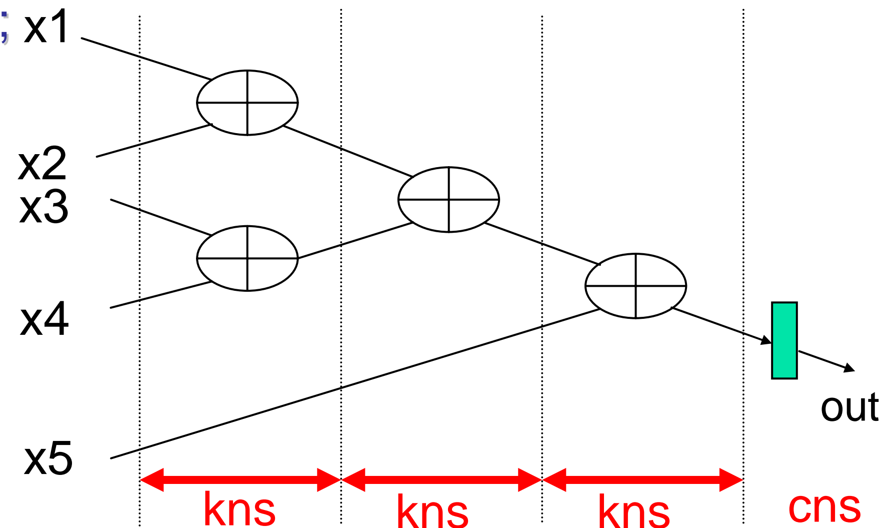


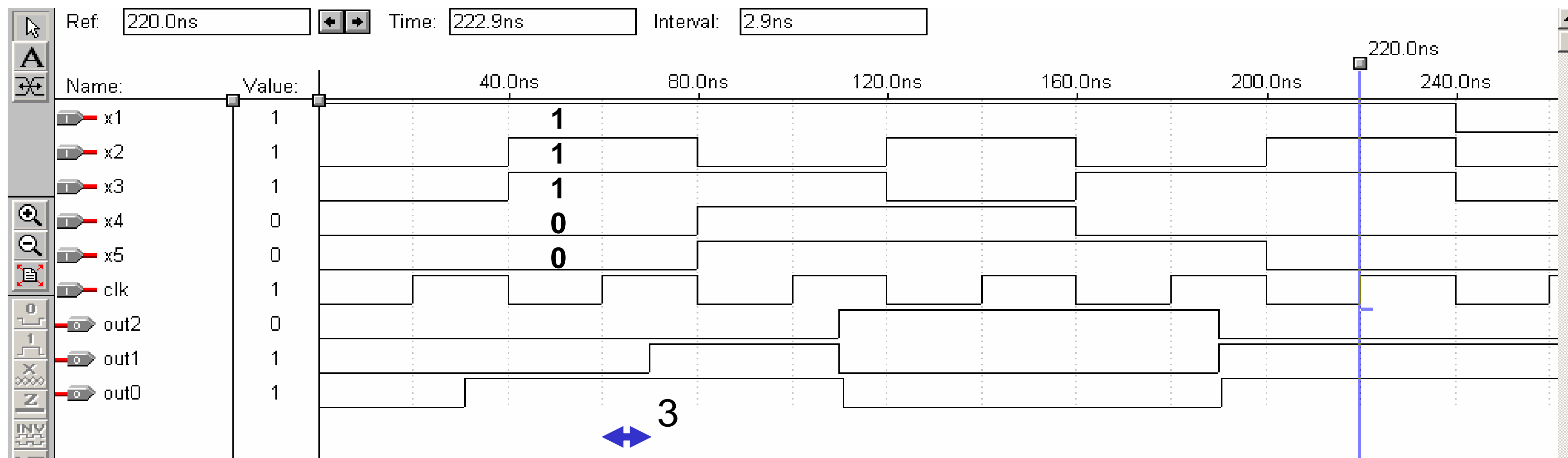
Unstable output (Delay is 3*kns -- less than Method_1)

Design for Summation Problem (3/7)

Method_3

```
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
```



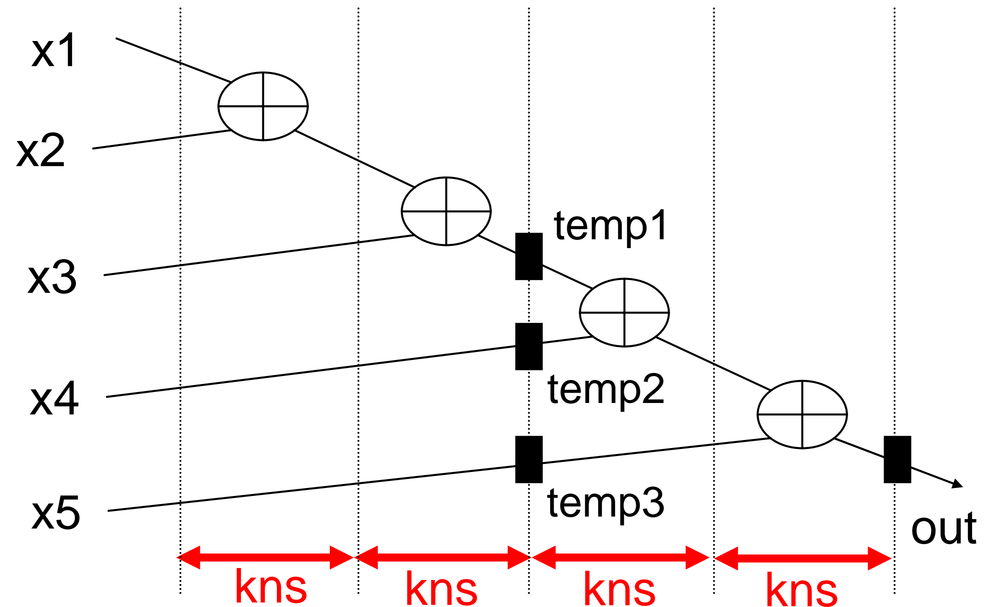


Stable output with register (3-bit flip-flop)   Delay is 3*kns+cns (reg assign delay)

Design for Summation Problem (4/7)

Method_4

```
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
```

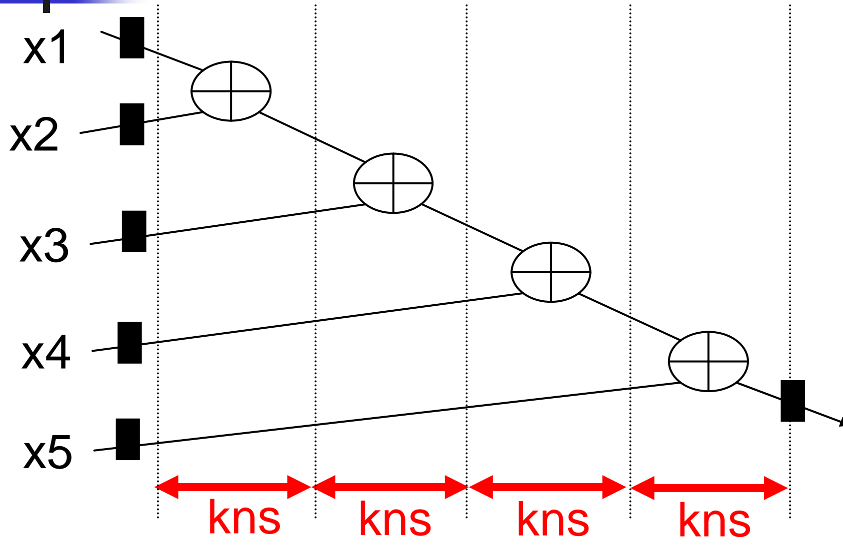


Delay is 2*kns+cns which is less than Method_1 (4kns), Method_2 (3kns) and Method_3 (3kns+cns)

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)

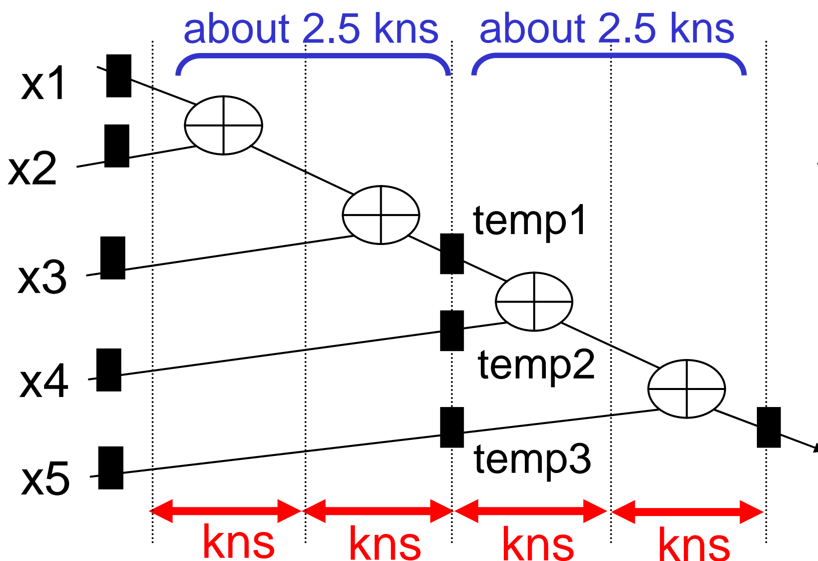
1. Wire delay
2. Register assignment delay



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?

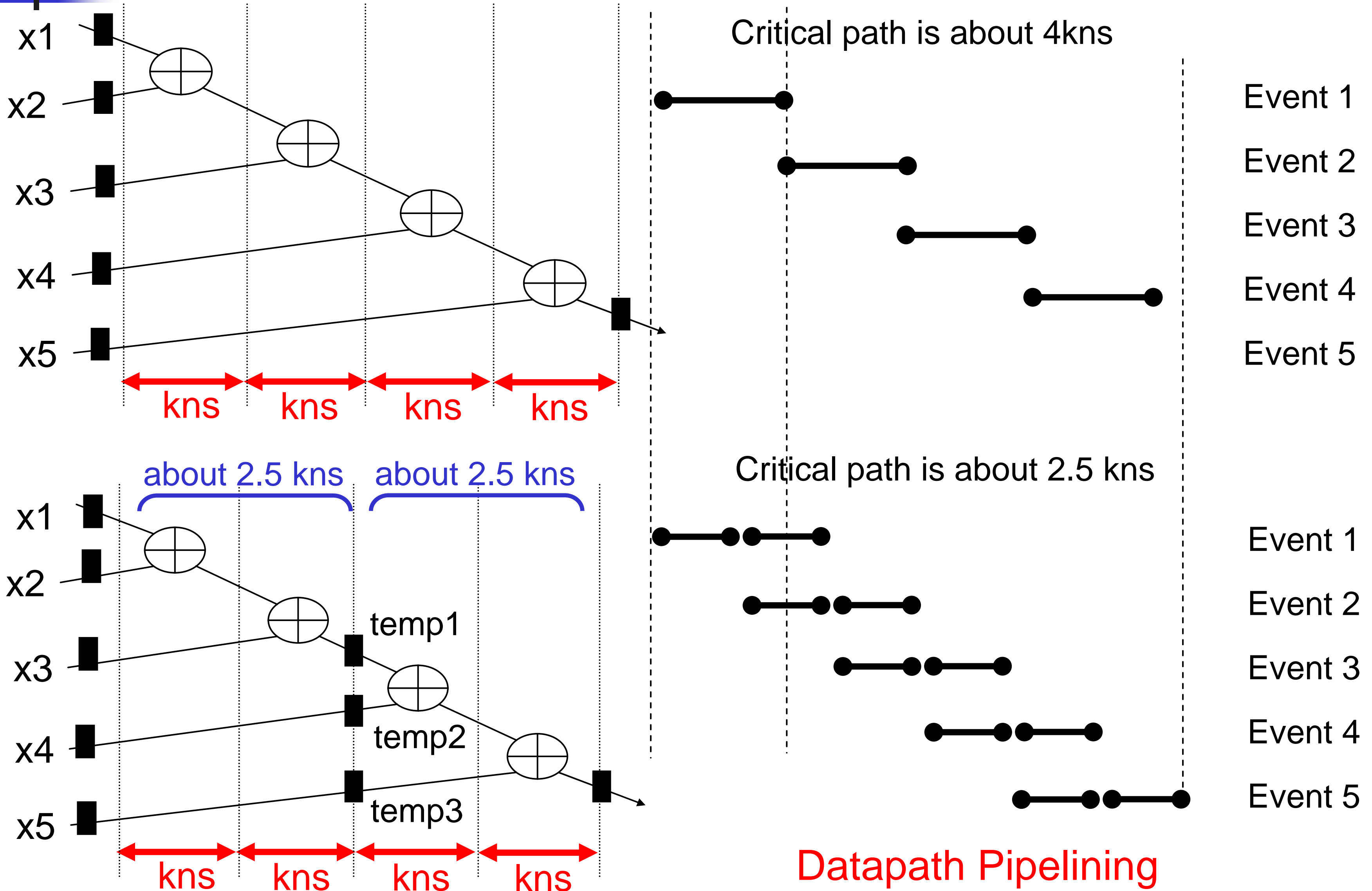
A correct output is generated after two clock cycles

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

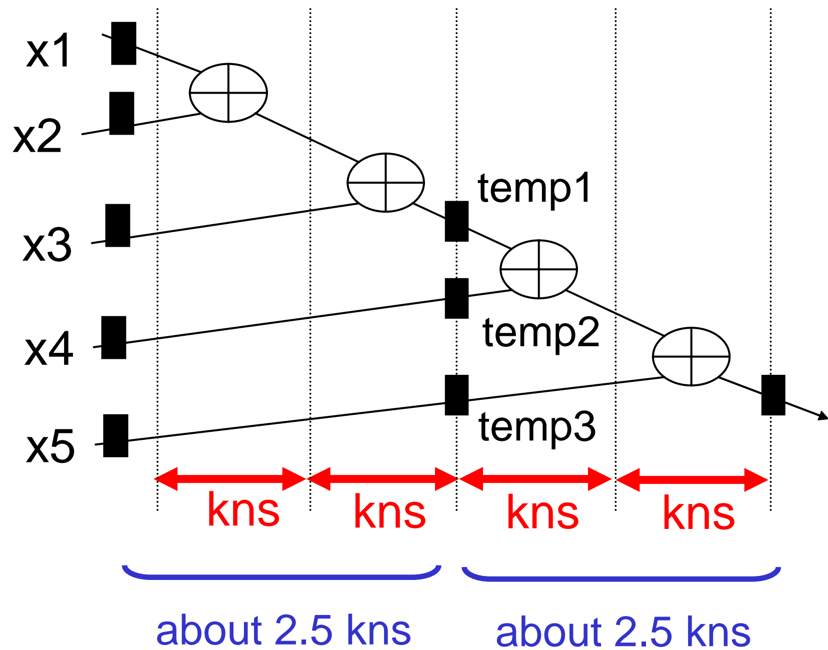
Two events are parallel processed in the unit.

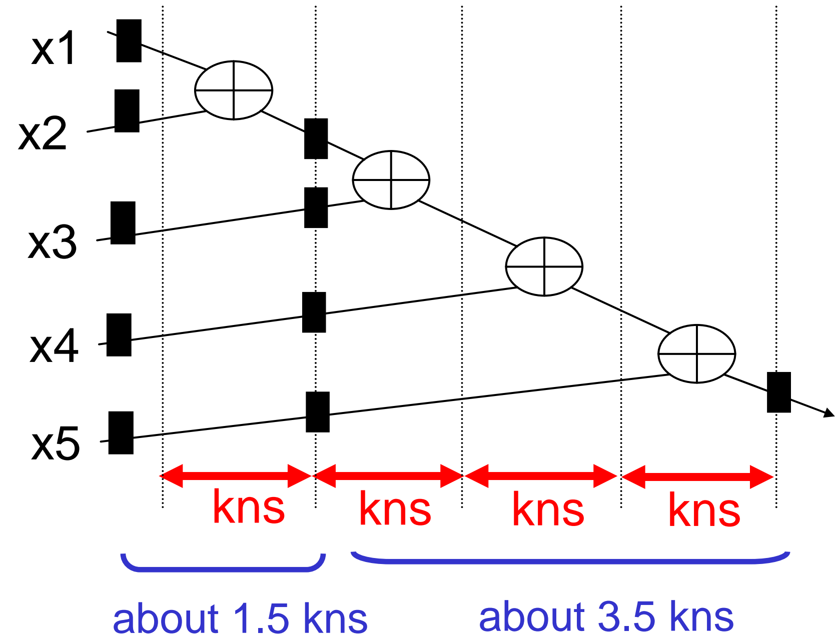
Faster clock rate but higher cost (3 extra regs)

Design for Summation Problem (6/7)



Design for Summation Problem (7/7)



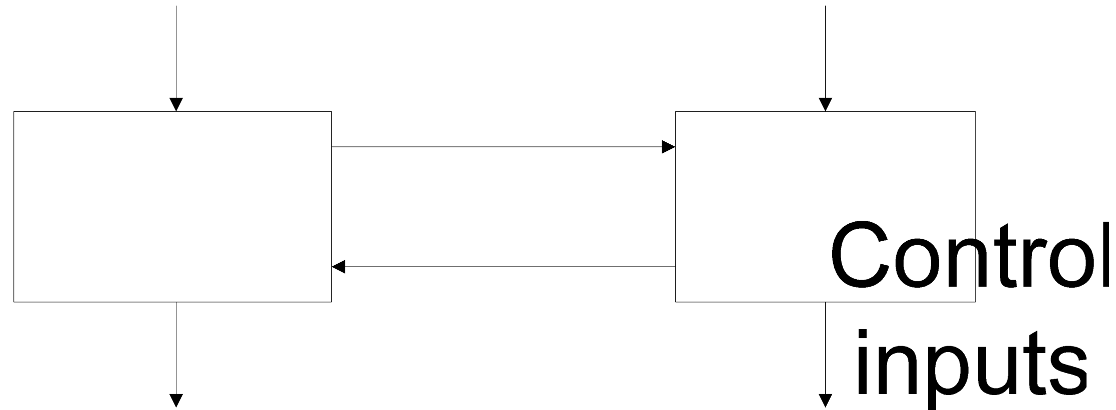


Critical path is about 2.5 kns

Critical path is about 3.5 kns

Which one is better ? Balance is important

Optimization for RTL Design



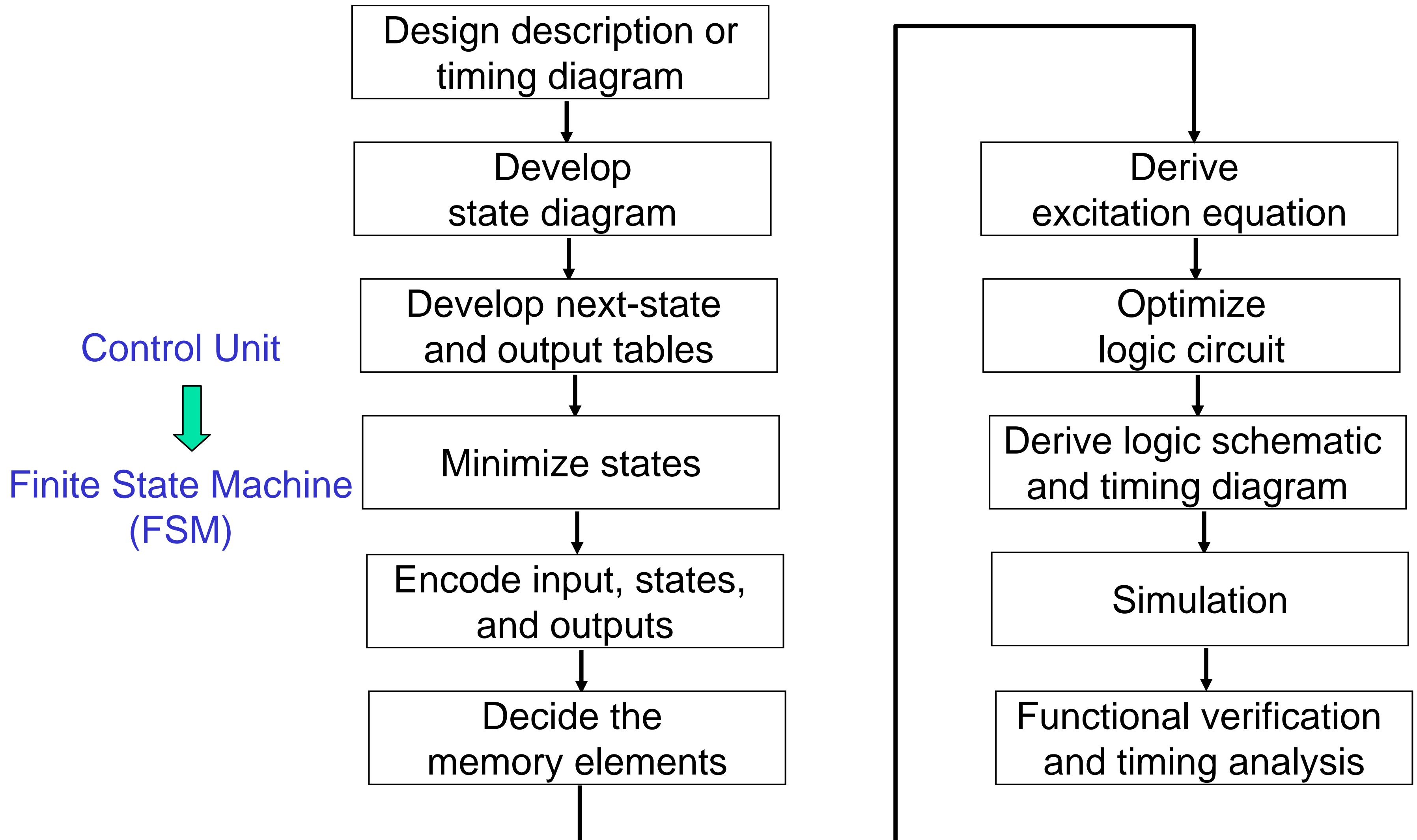
Optimization for control unit:

1. As suggestion by most textbooks of “Logic System Design”
2. Write a good-style HDL descriptions which are optimized by EDA tools

Optimization for datapath:

1. Resource optimization
2. Time optimization

Optimization for Control Unit



Finite State Machine (1/4)

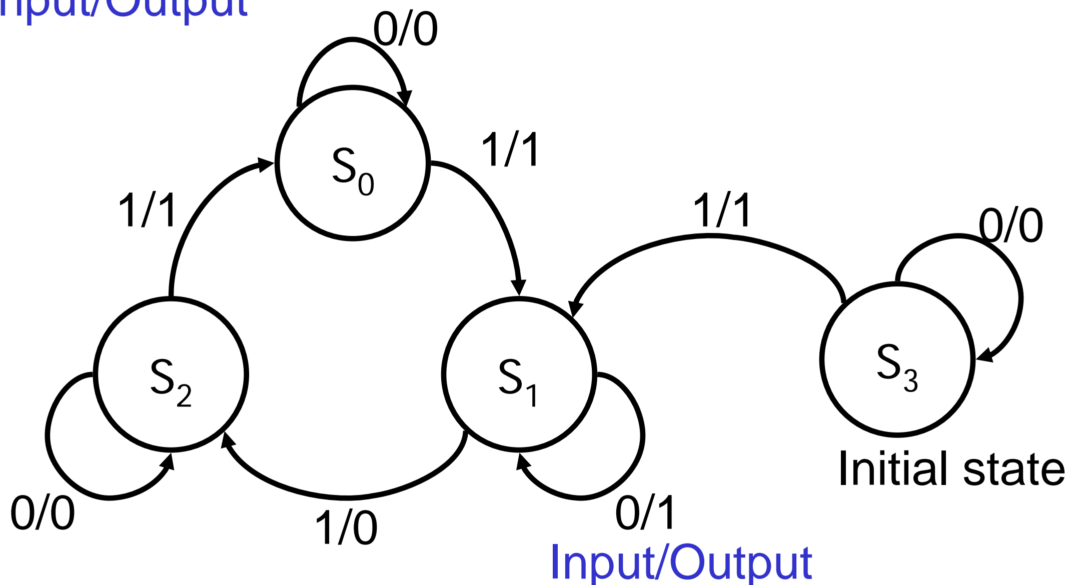
Moore machine: $S \rightarrow O$ (output is dependent only on current state)

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

State diagram

Four states: S_0, S_1, S_2, S_3

Input/Output



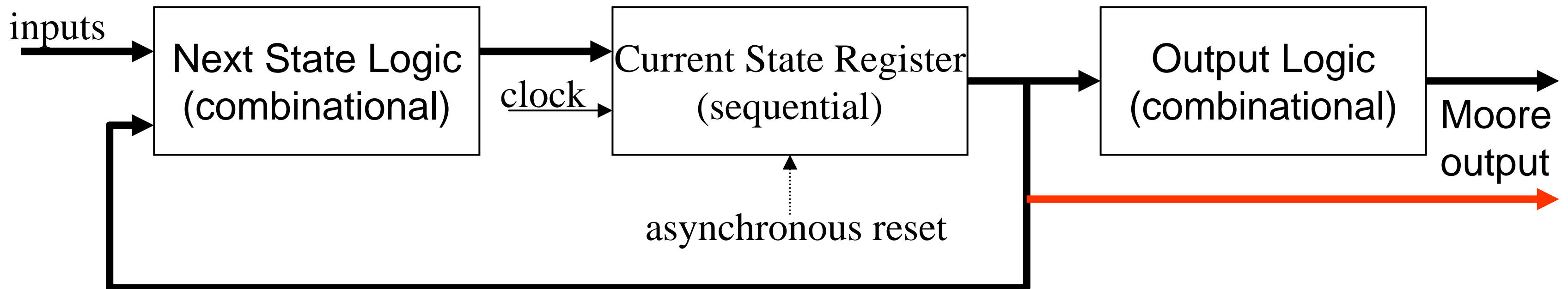
Next-state and output tables (I=input)

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

A mealy machine

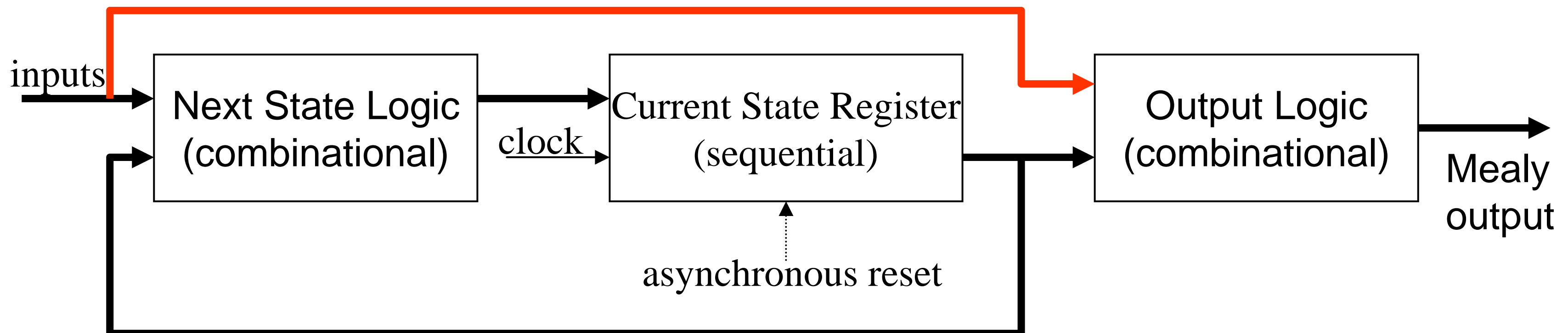
Finite State Machine (2/4)

$S \rightarrow O$



Moore Machine (state-based machine)

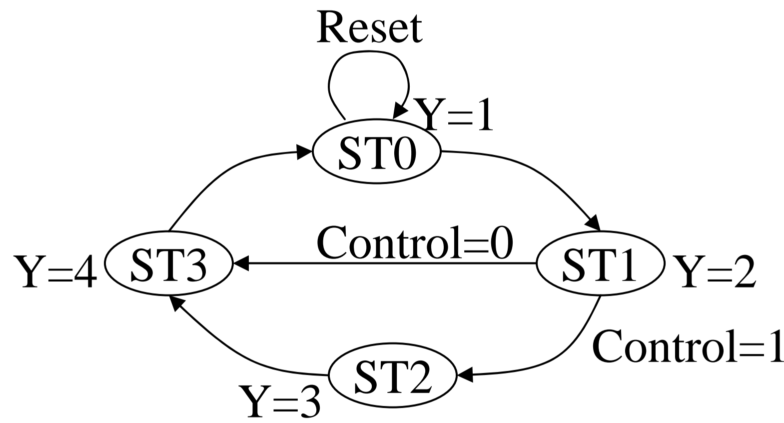
$S \times I \rightarrow O$



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



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;
```

State name (parameter)

State register (Seq.C.)

```
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.)

Output logic (Comb.C.)

Moore Machine (1/8)

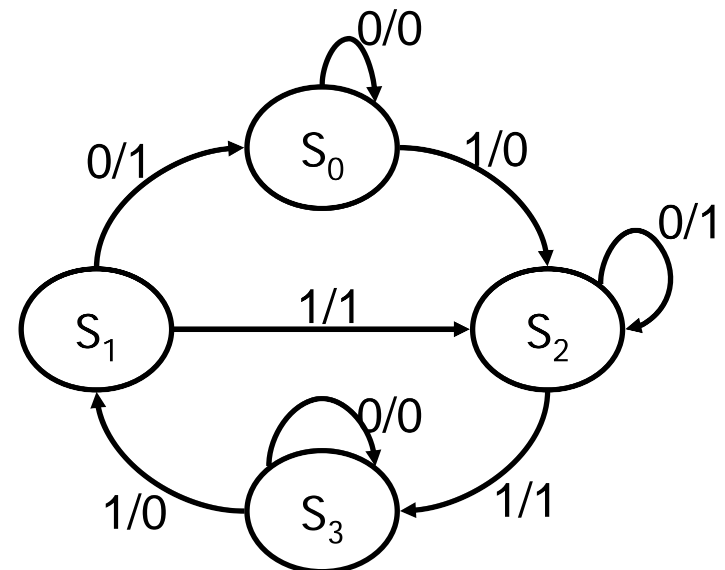
Optimization flow

Design description or timing diagram → Develop state diagram → Develop next-state and output tables → Minimize states → Encode input, states, and outputs → Decide the memory elements → Derive excitation equation → Optimize logic circuit → Derive logic schematic and timing diagram → Simulation → Functional verification and timing analysis

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

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_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



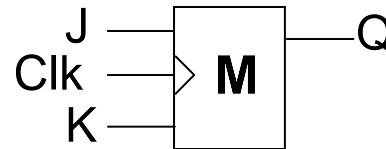
Moore Machine (2/8)

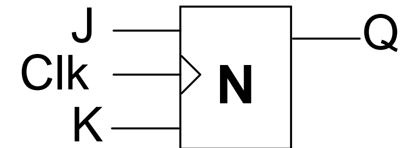
original state table

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)

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

MJ=I

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

NJ=MI

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

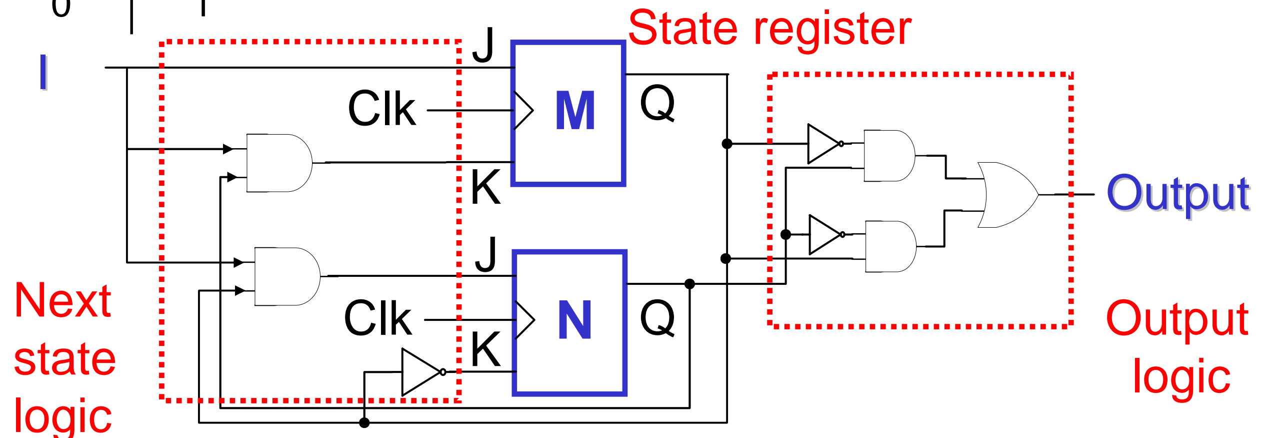
MK=NI

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

NK=M'

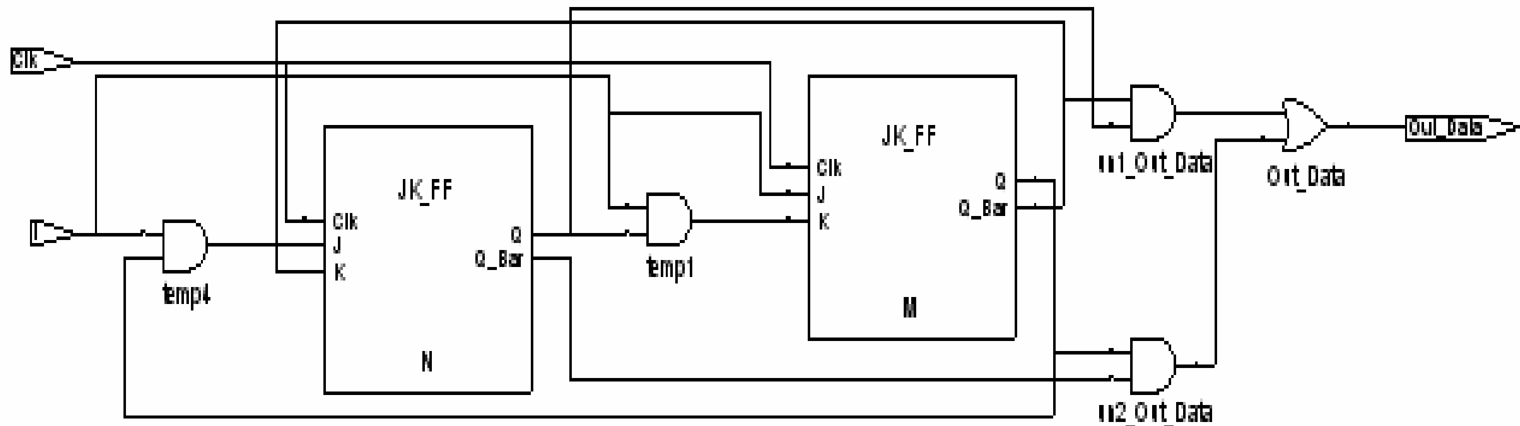
X \ MN	00	01	11	10
0	0	1	0	1
1	0	1	0	1

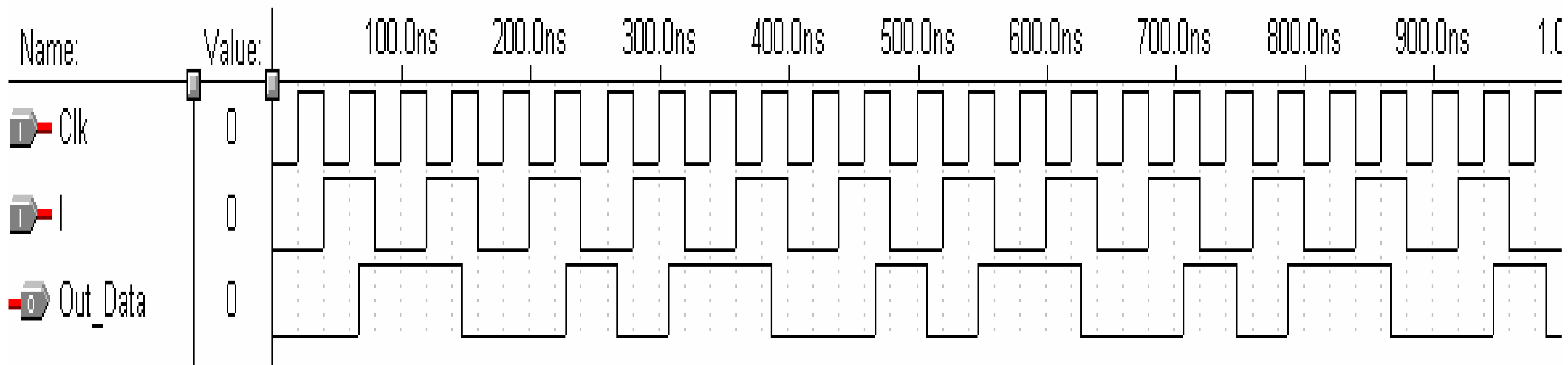
Output=M'N+MN'



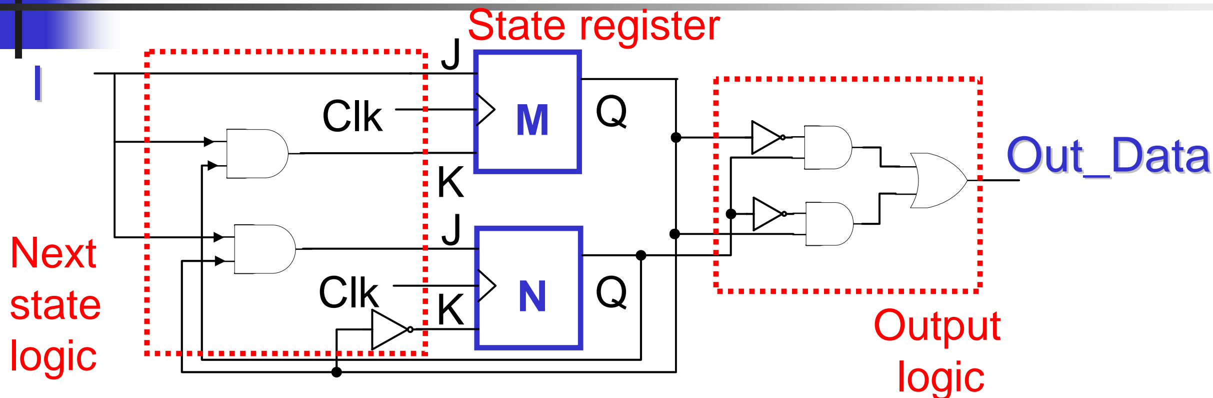
Moore Machine (4/8)

Synthesis Result





Moore Machine (5/8)



Implement the circuit with structural HDL

```
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);
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
```

Moore Machine-Bad Example (6/8)

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)

```
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
```

Both State and Out_Data are implemented with flip-flops

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
```

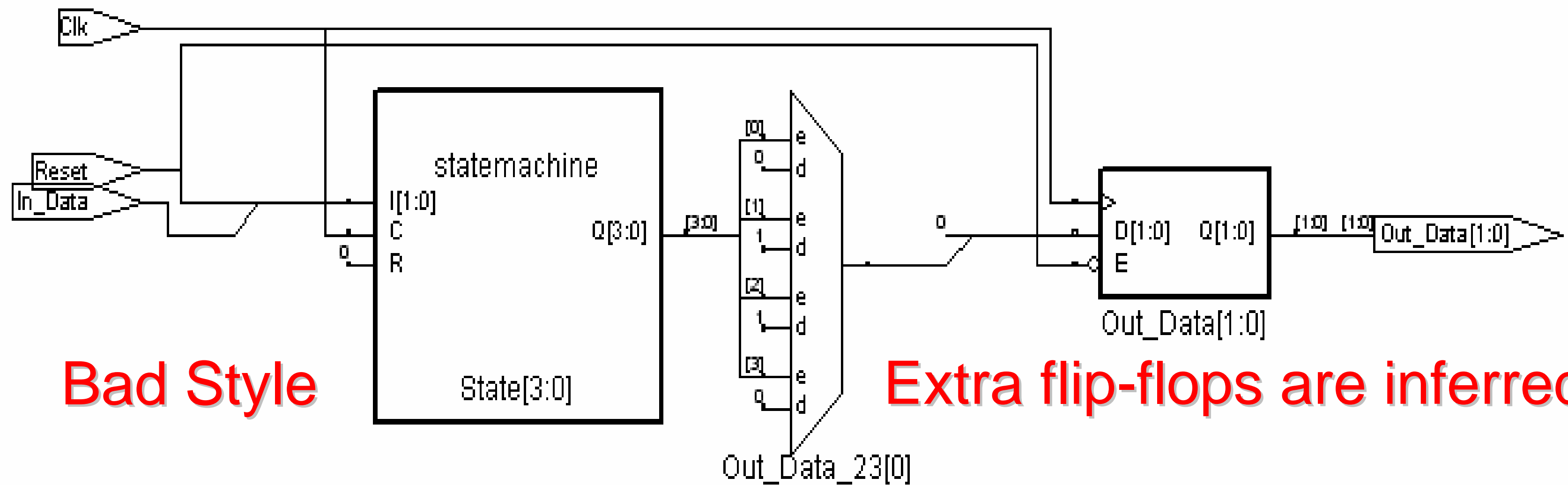
Next state logic

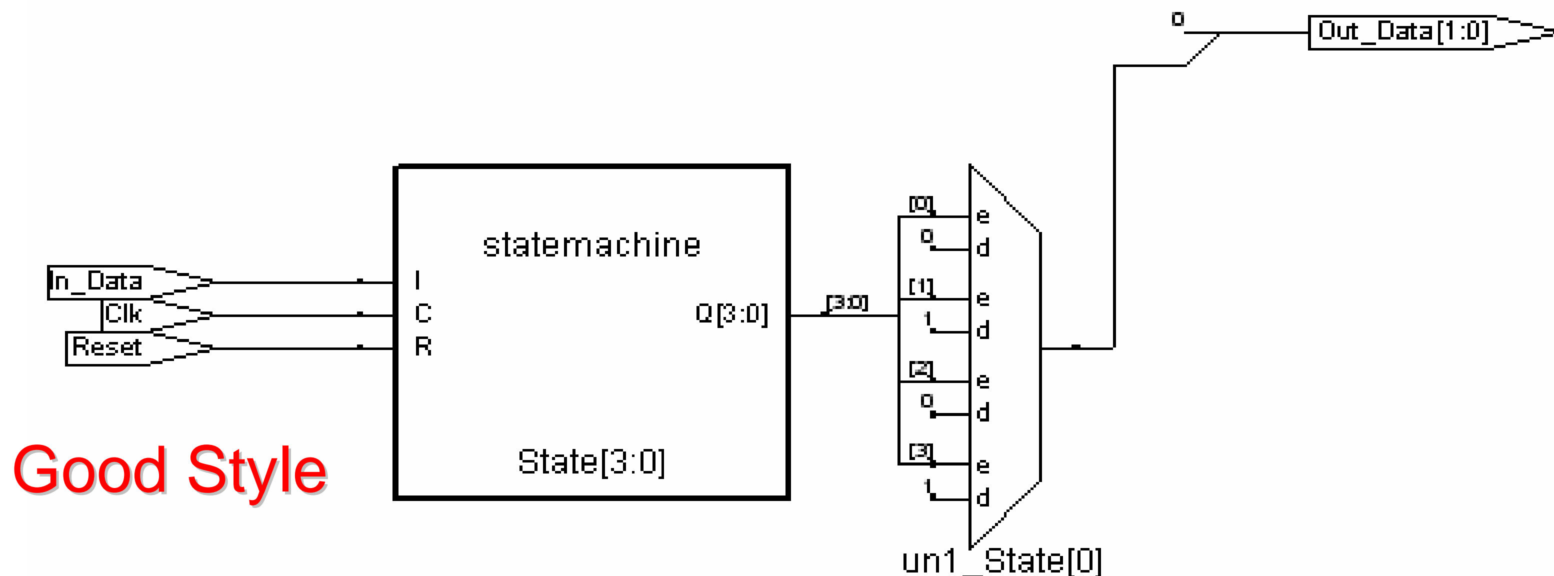
```
always @(State)
begin
  case(State)
     S0:Out_Data = 0;
     S1:Out_Data = 1;
     S2:Out_Data = 1;
     S3:Out_Data = 0;
     endcase
end
endmodule
```

Output logic

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

Moore Machine-Good Example (8/8)



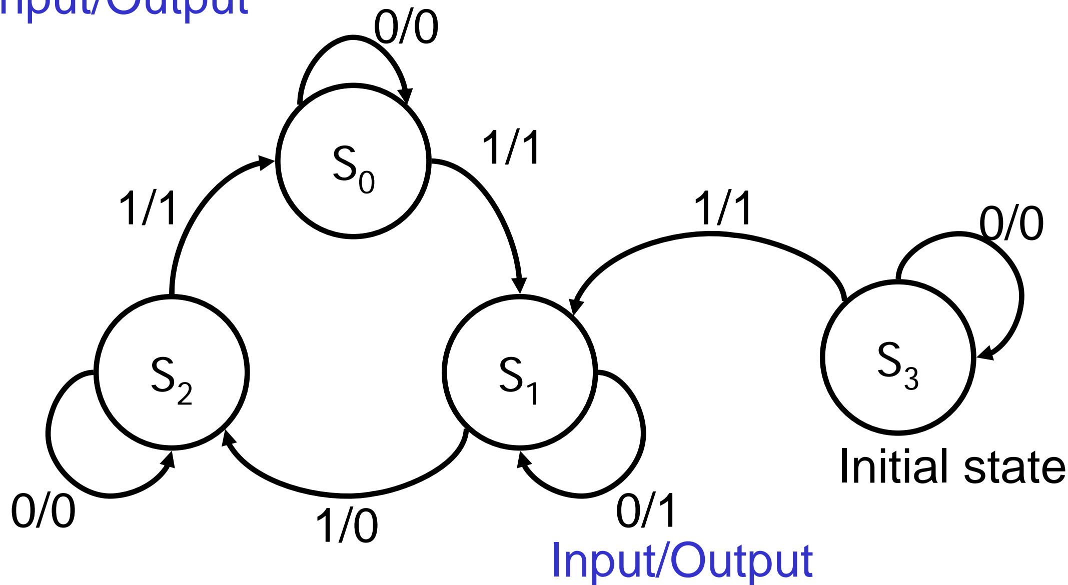


Mealy Machine (1/2)

State diagram

Four states: S_0, S_1, S_2, S_3

Input/Output

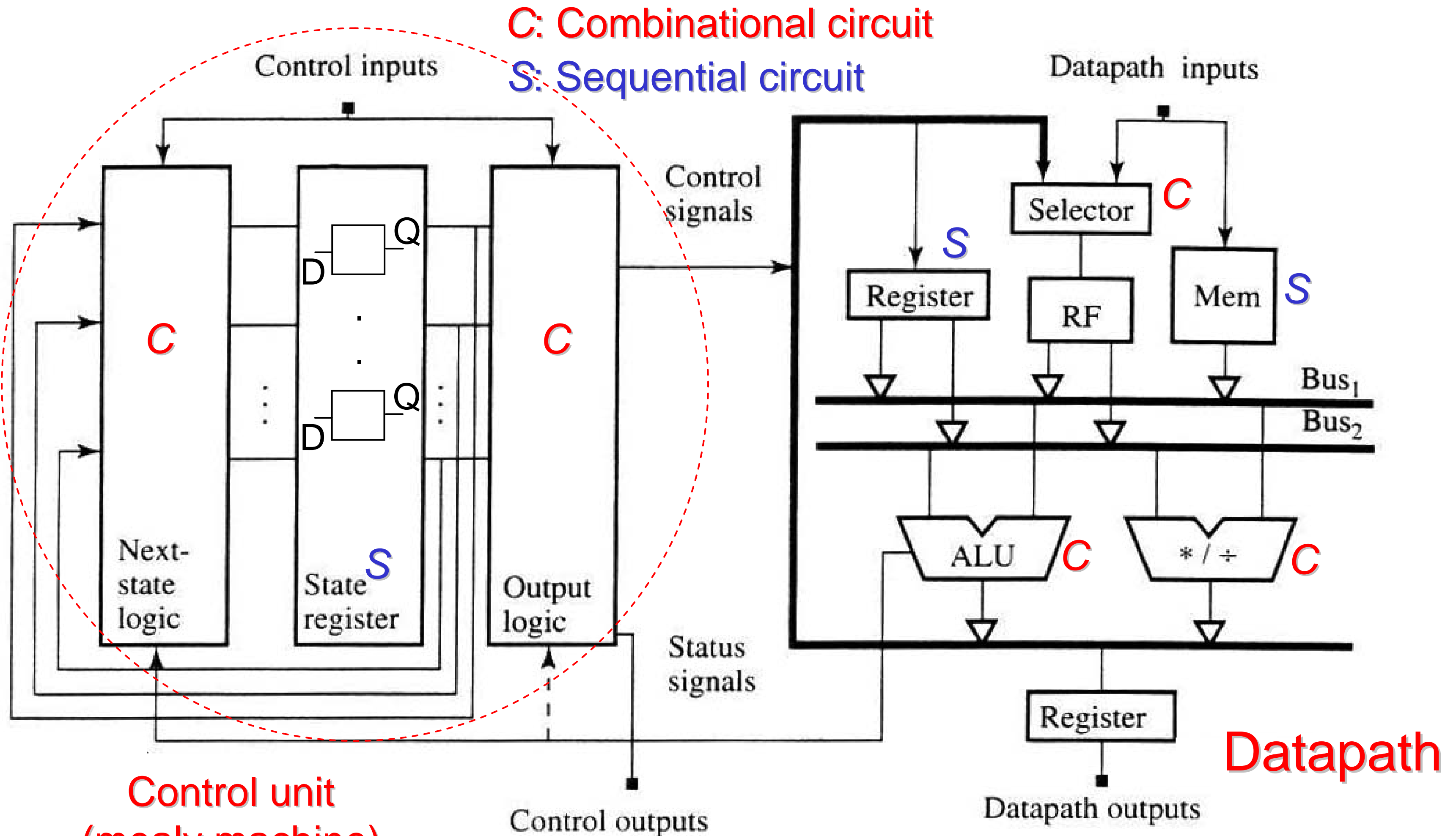


Next-state and output tables (I=input)

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)

Please do remember to write your mealy machine by using the good-style HDL



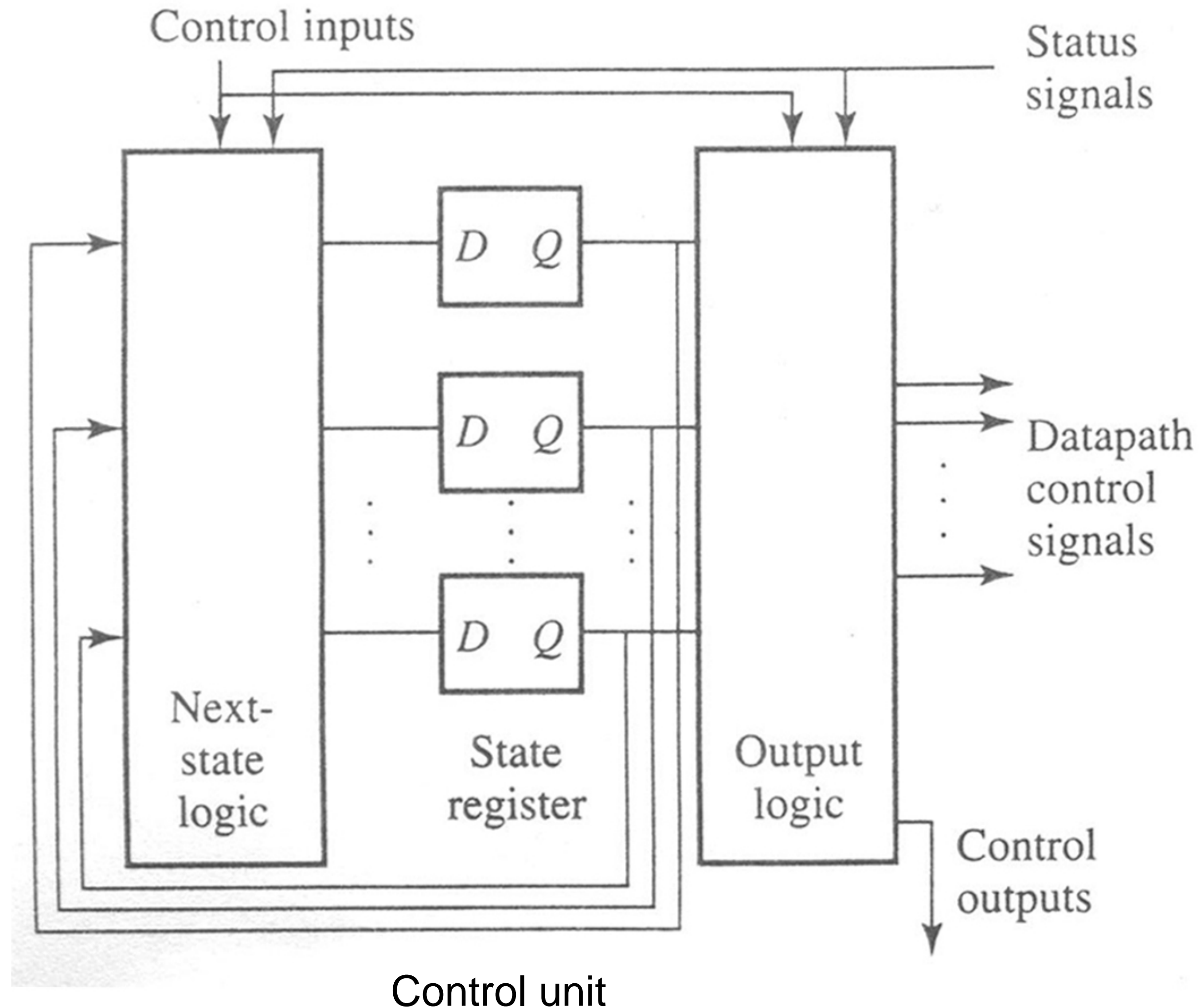
Control unit
(mealy machine)
Using three always statements

Homework: implement a mealy machine

Control-Unit Implementation Styles (1/3)

Hardwired

Control

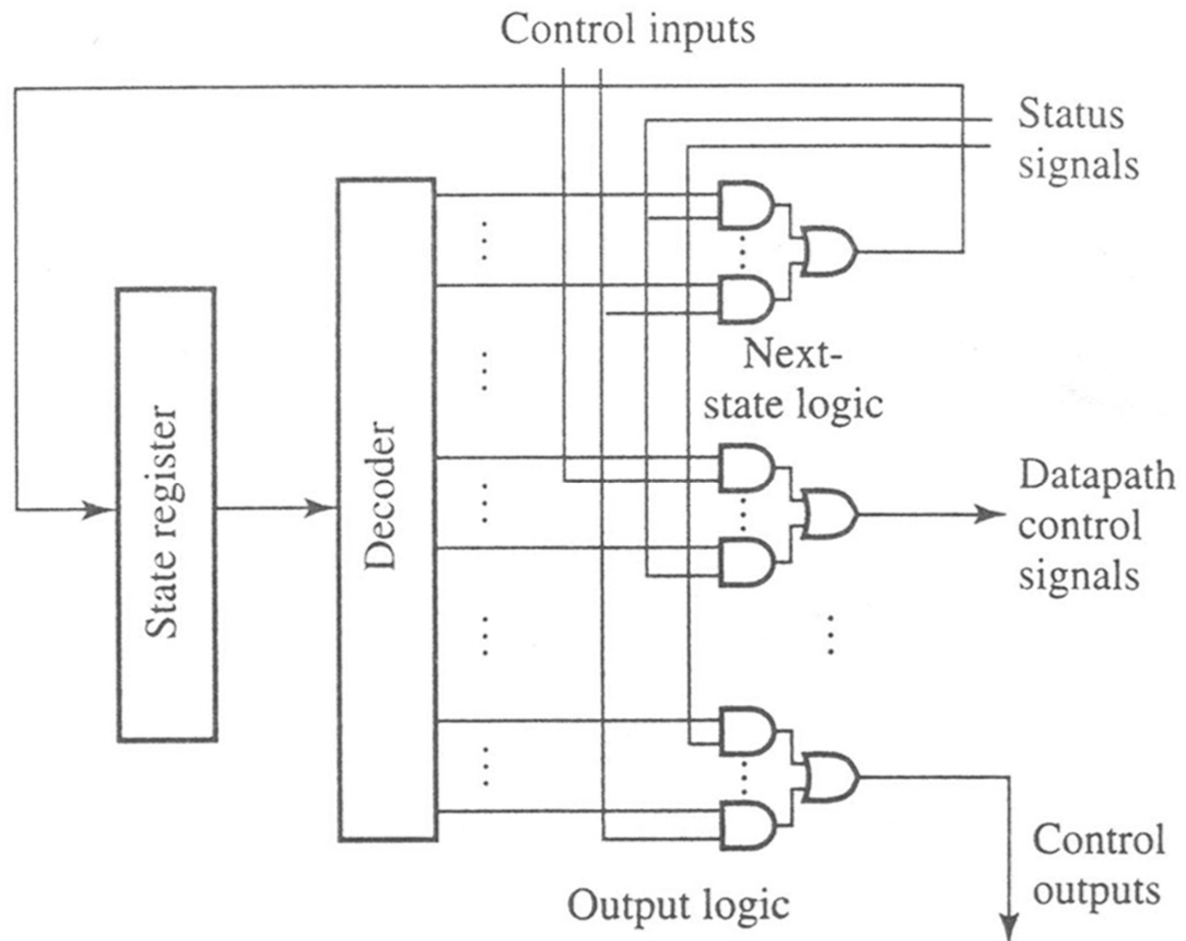


Control unit

Control-Unit Implementation Styles (2/3)

Hardwired

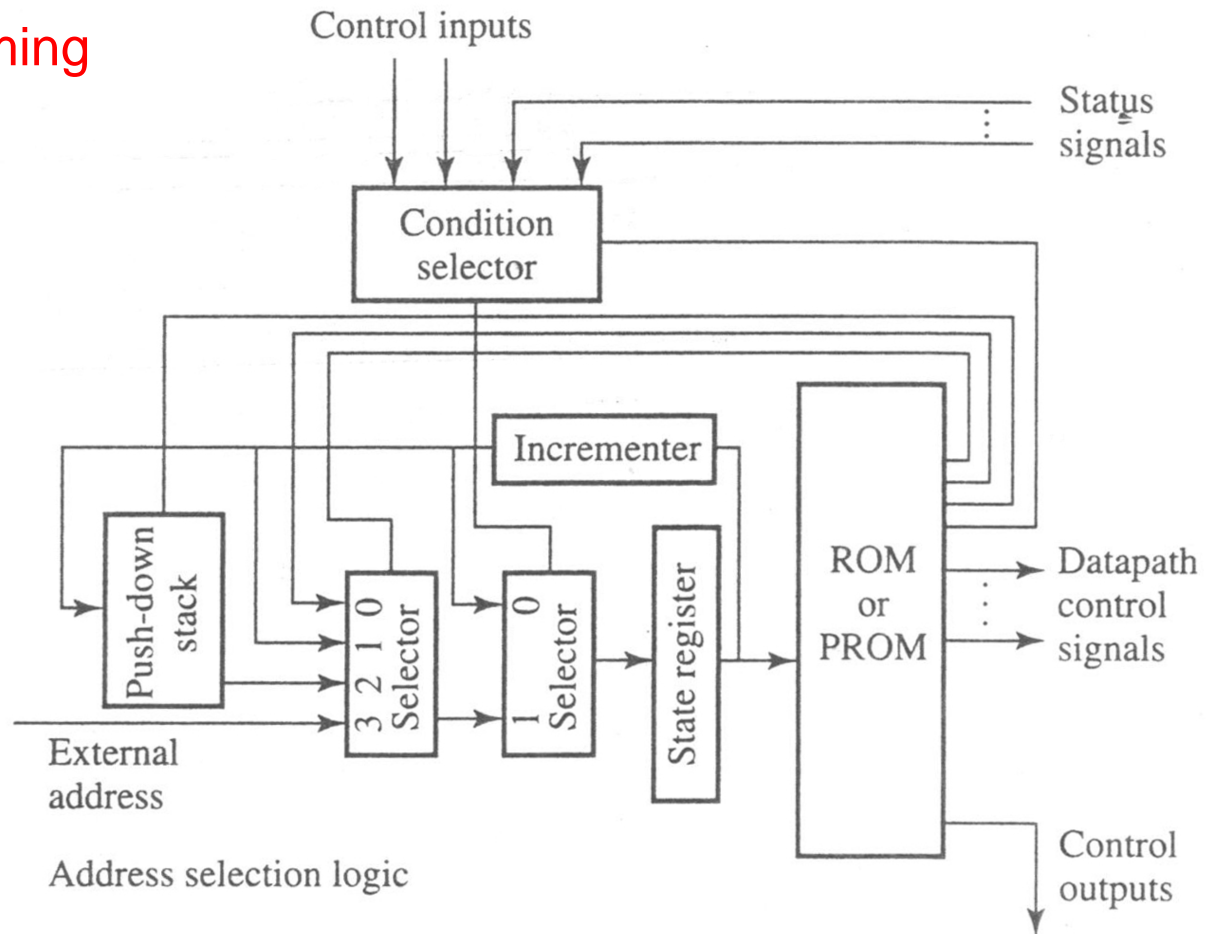
Control



Control unit with state-register and decoder

Control-Unit Implementation Styles (3/3)

Microprogramming Control



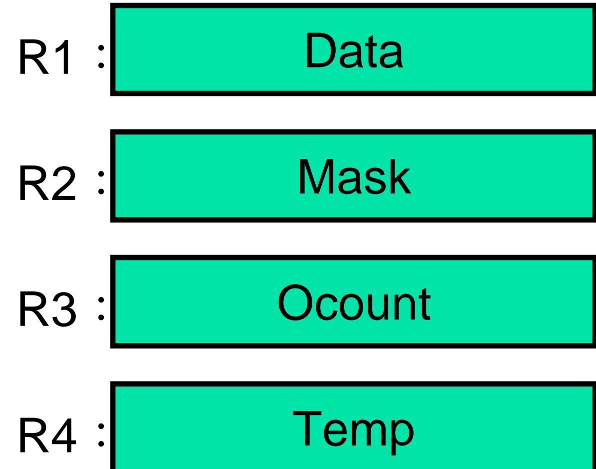
Control unit with state-register and ROM

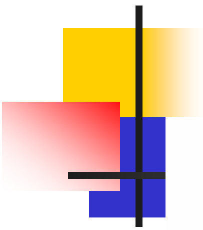
One's Count Problem (1/2)

One's – counter implementation

Problem：Using a datapath with a 3 port register-file (2 read port and 1 write port), design a one's counter that count the number of ones in an input dataword, and return the result after completion

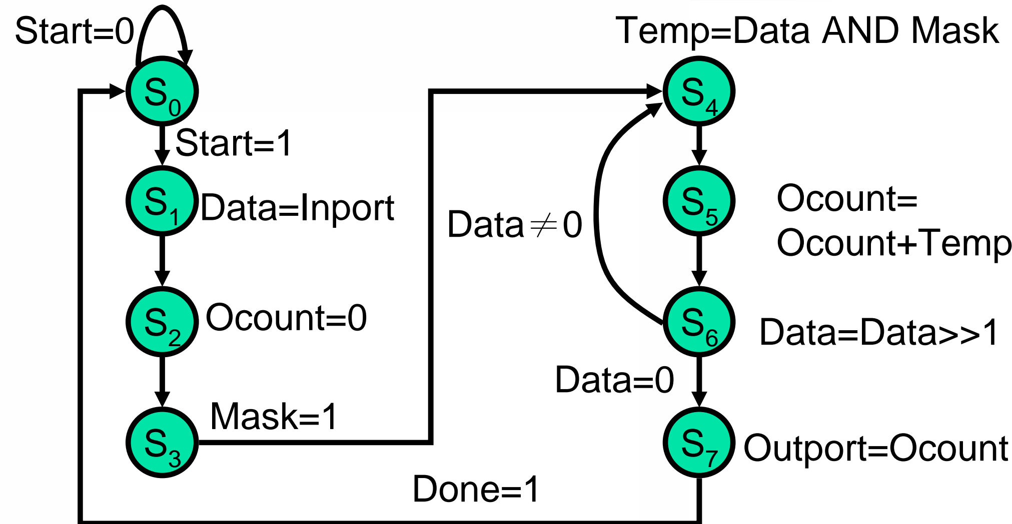
Data := Input
Ocount := 0
Mask := 1
while Data :≠ 0 **repeat**
Temp := Data AND Mask
Ocount := Ocount + Temp
Data := Data >> 1
end while
Outport := Ocount





One’s count Problem (2/2)

Inport
19
S
1
0
Selector
16-18
3
WA
15
WE
Clk
8*m register file
12-14
3
RAA
11
REA
8-10
3
RAB
REB
7
Bus A
Bus B
Control Unit
6
5
4
M
S_1
S_0
A
B
ALU
control signals
"0"
"0"
3
2
1
S_2
S_1
S_0
I_L
I_R
Shifter
Result bus
0
Outport
Datapath
(a) Datapath schematic



	IE	WT AD	READ AD A	READ AD B	ALU	SHIFT	OE
1	1	R_1	X	X	X	X	0
2	0	R_3	0	0	Add	Pass	0
3	0	R_2	0	X	Inc	Pass	0
4	0	R_4	R_1	R_2	AND	Pass	0
5	0	R_3	R_3	R_4	Add	Pass	0
6	0	R_1	R_1	0	Add	Shift right	0
7	0	None	R_3	0	Add	Pass	1

Datapath of One’s-Counter (1/4)

Optimized by EDA tool

```
module    data_path(clock,reset,control_word,inport,outport,data);
input     clock,reset;
input     [19:0] control_word;
input     [7:0] inport;
output    [7:0] outport,data;
wire      [7:0] line1,line2,line3,line4;

selector O1(.inp_A(inport), .inp_B(data), .select(control_word[19]), .outp(line1));

register NO2(.clock(clock), .reset(reset), .WA(control_word[17:15]),
.WE(control_word[18]), .RAA(control_word[13:11]), .REA(control_word[14]),
.RAB(control_word[9:7]), .REB(control_word[10]), .Data_in(line1), .Bus_A(line2),
.Bus_B(line3));

alu NO3(.Datain_A(line2), .Datain_B(line3), .select(control_word[6:4]), .outp(line4));

shifter NO4(.inp(line4), .select(control_word[3:1]), .outp(data));

buffer NO5(.OE(control_word[0]), .inp(data), .outp(outport));
endmodule
```

Datapath of One's-Counter (2/4)

```
module    selector(inp_A,inp_B,select,outp);
input     [7:0] inp_A,inp_B;
input     select;
output    [7:0] outp;
reg       [7:0] outp;

always@(select or inp_A or inp_B)
begin
    if(select)
      outp = inp_A;
    else
      outp = inp_B;
end
endmodule
```

M	S_1	S_0	ALU OPERATIONS
0	0	0	Complement A
0	0	1	AND
0	1	0	EX-OR
0	1	1	OR
1	0	0	Decrement A
1	0	1	Add
1	1	0	Subtract
1	1	1	Increment A

```
module    alu(Datain_A,Datain_B,select,outp);
input     [7:0] Datain_A,Datain_B;
input     [2:0] select;
output    [7:0] outp;  reg      [7:0] outp;

always@(select or Datain_A or Datain_B)
begin
    case(select)
      3'b000:outp = ~Datain_A;
      3'b001:outp = Datain_A & Datain_B;
      3'b010:outp = Datain_A ^ Datain_B;
      3'b011:outp = Datain_A | Datain_B;
      3'b100:outp = Datain_A - 1;
      3'b101:outp = Datain_A + Datain_B;
      3'b110:outp = Datain_A - Datain_B;
      3'b111:outp = Datain_A + 1;
    endcase
end    endmodule
```

Datapath of One's-Counter (3/4)

```
module    shifter(inp,select,outp);
input     [7:0] inp;
input     [2:0] select;
output    [7:0] outp;
reg       [7:0] outp;
reg       temp;
always@(select or inp)
begin
    case(select)
       3'b000:outp = inp;
       3'b001:outp = inp;
       3'b100:outp = inp << 1;
       3'b101:
            begin
               temp = inp[7];
               outp = inp << 1;
               outp[0] = temp;
            end
       3'b110:outp = inp >> 1;
3'b111:
          begin
            temp = inp[0]; outp = inp >> 1;
            outp[7] = temp;
          end
default: outp=8'hxx;
    endcase
end
endmodule
```

```
module    buffer(OE,inp,outp);
input     OE;
input     [7:0] inp;
output    [7:0] outp;
reg       [7:0] outp;
always@(OE or inp)
begin
    if(OE)
      outp = inp;
    else  outp=8'bz;
end endmodule
```

S_2	S_1	S_0	SHIFT OPERATIONS
0	0	0	Pass
0	0	1	Pass
0	1	0	Not used
0	1	1	Not used
1	0	0	Shift left
1	0	1	Rotate left
1	1	0	Shift right
1	1	1	Rotate right

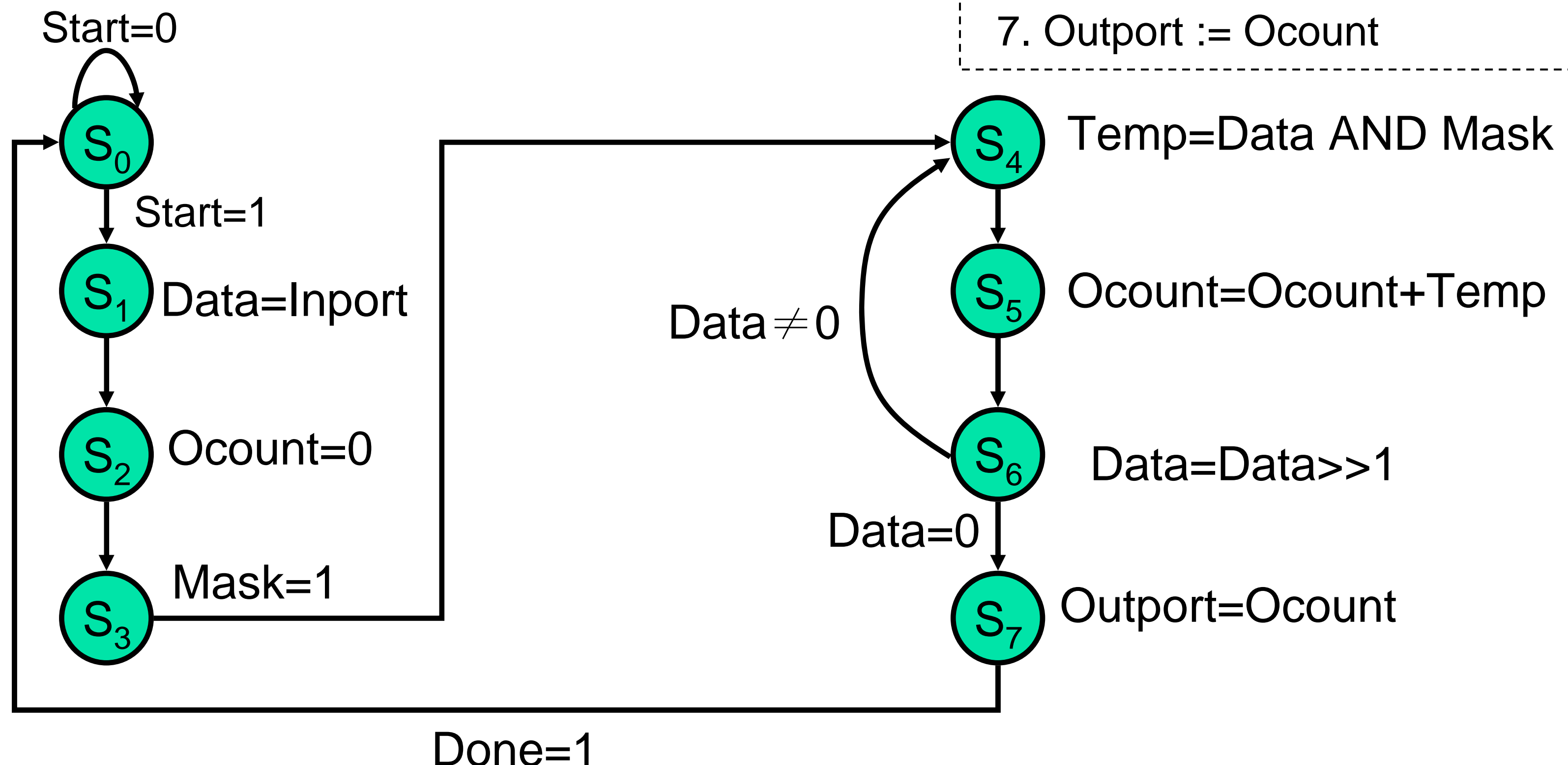
Datapath of One’s-Counter (4/4)

```
module    register(clock,reset,WA,WE,RAA,
REA,RAB,REB,Data_in,Bus_A,Bus_B);
input     clock,reset,WE,REA,REB;
input     [2:0] WA,RAA,RAB;
input  [7:0] Data_in; output  [7:0] Bus_A,Bus_B;
reg [7:0] reg_array [7:0];

always@(posedge clock)
begin
    if(reset)
      begin
      reg_array[0]=8'h00; reg_array[1]=8'h00;
      reg_array[2]=8'h00; reg_array[3]=8'h00;
      reg_array[4]=8'h00; reg_array[5]=8'h00;
      reg_array[6]=8'h00; reg_array[7]=8'h00; end
     else
      begin
          if(WE)
            reg_array[WA]=Data_in;
      end
end

assign Bus_A=REA?reg_array[RAA]:8’h00;
assign Bus_B=REB?reg_array[RAB]:8’h00;
endmodule
```

Controller of One's-Counter (1/8)

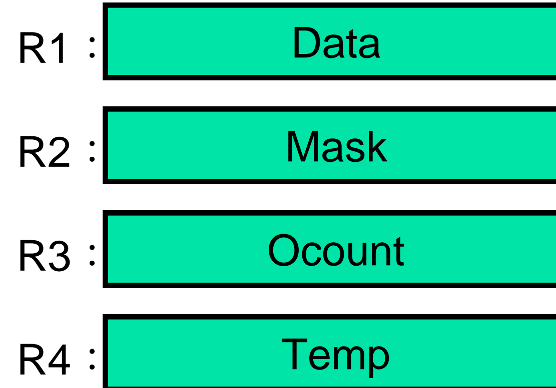
- State lasts for a clock cycle
- In each state the datapath executes the statement indicated on its right side.

1. Data := Input
2. Ocount := 0
3. Mask := 1
   while Data :≠ 0 **repeat**
4. Temp := Data AND Mask
5. Ocount := Ocount + Temp
6. Data := Data >> 1
   end while
7. Outport := Ocount



Controller of One's-Counter (2/8)

1. Data := Input
2. Ocount := 0
3. Mask := 1
   while Data :≠ 0 **repeat**
4. Temp := Data AND Mask
5. Ocount := Ocount + Temp
6. Data := Data >> 1
   end while
7. Outport := Ocount



Control words for one's counter

Control WORDS	IE	WRITE ADDRESS	READ ADDESS A	READ ADDRESS B	ALU OPERATION	SHIFTER OPERATION	OE
1	1	R_1	X	X	X	X	0
2	0	R_3	0	0	Add	Pass	0
3	0	R_2	0	X	Increment	Pass	0
4	0	R_4	R_1	R_2	AND	Pass	0
5	0	R_3	R_3	R_4	Add	Pass	0
6	0	R_1	R_1	0	Add	Shift right	0
7	0	None	R_3	0	Add	Pass	1

← word

Controller of One's-Counter (3/8)

State	$Q_2Q_1Q_0$	IE	Write address				Read address A				Read address B				ALU operations			Shift operations			OE
			WE	WA_2	WA_1	WA_0	REA	RAA_2	RAA_1	RAA_0	REB	RAB_2	RAB_1	RAB_0	M	S_1	S_0	S_2	S_1	S_0	
s_0	000	0	0	X	X	X	0	X	X	X	0	X	X	X	X	X	X	X	X	X	0
s_1	001	1	1	0	0	1	0	X	X	X	0	X	X	X	X	X	X	X	X	X	0
s_2	010	0	1	0	1	1	0	X	X	X	0	X	X	X	1	0	1	0	0	0	0
s_3	011	0	1	0	1	0	0	X	X	X	0	X	X	X	1	1	1	0	0	0	0
s_4	100	0	1	1	0	0	1	0	0	1	1	0	1	0	0	0	1	0	0	0	0
s_5	101	0	1	0	1	1	1	0	1	1	1	1	0	0	1	0	1	0	0	0	0
s_6	110	0	1	0	0	1	1	0	0	1	0	X	X	X	1	0	1	1	1	0	0
s_7	111	0	0	X	X	X	1	0	1	1	0	X	X	X	1	0	1	0	0	0	1

(a) Output logic table

Optimized by hand

$$IE = Q_2'Q_1'Q_0$$
$$WA_2 = Q_1'Q_0'$$
$$WA_1 = Q_2Q_0 + Q_2'Q_1$$
$$WA_0 = Q_1'Q_0 + Q_1Q_0'$$
$$WE = Q_2Q_1' + Q_2'Q_0 + Q_1Q_0'$$

$$RAB_2 = Q_0$$
$$RAB_1 = Q_0'$$
$$RAB_0 = 0$$
$$REB = Q_2Q_1'$$

$$RAB_2 = Q_0$$
$$RAB_1 = Q_0'$$
$$RAB_0 = 0$$
$$REB = Q_2Q_1'$$

$$M = Q_1 + Q_0$$
$$S_1 = Q_2'Q_0$$
$$S_0 = 1$$
$$S_2 = S_1 = Q_2Q_1Q_0'$$
$$S_0 = 0$$
$$OE = Q_2 + Q_1 + Q_0$$

(b) Output equations

Controller of One's-Counter (4/8)

States	$Q_2Q_1Q_0$	Start .(Data≠0) 00	01	10	11
S_0	000	000	000	001	001
S_1	001	010	010	010	010
S_2	010	011	011	011	011
S_3	011	100	100	100	100
S_4	100	101	101	101	101
S_5	101	110	110	110	110
S_6	110	100	111	100	111
S_7	111	000	000	000	000

(a) Next-state table

Start Data≠0 \ Q_1Q_0	Q_2=0 00	01	11	10	Q_2=1 00	01	11	10
00	000	010	100	011	101	110	000	100
01	000	010	100	011	101	110	000	111
11	001	010	100	011	101	110	000	111
10	001	010	100	011	101	110	000	100

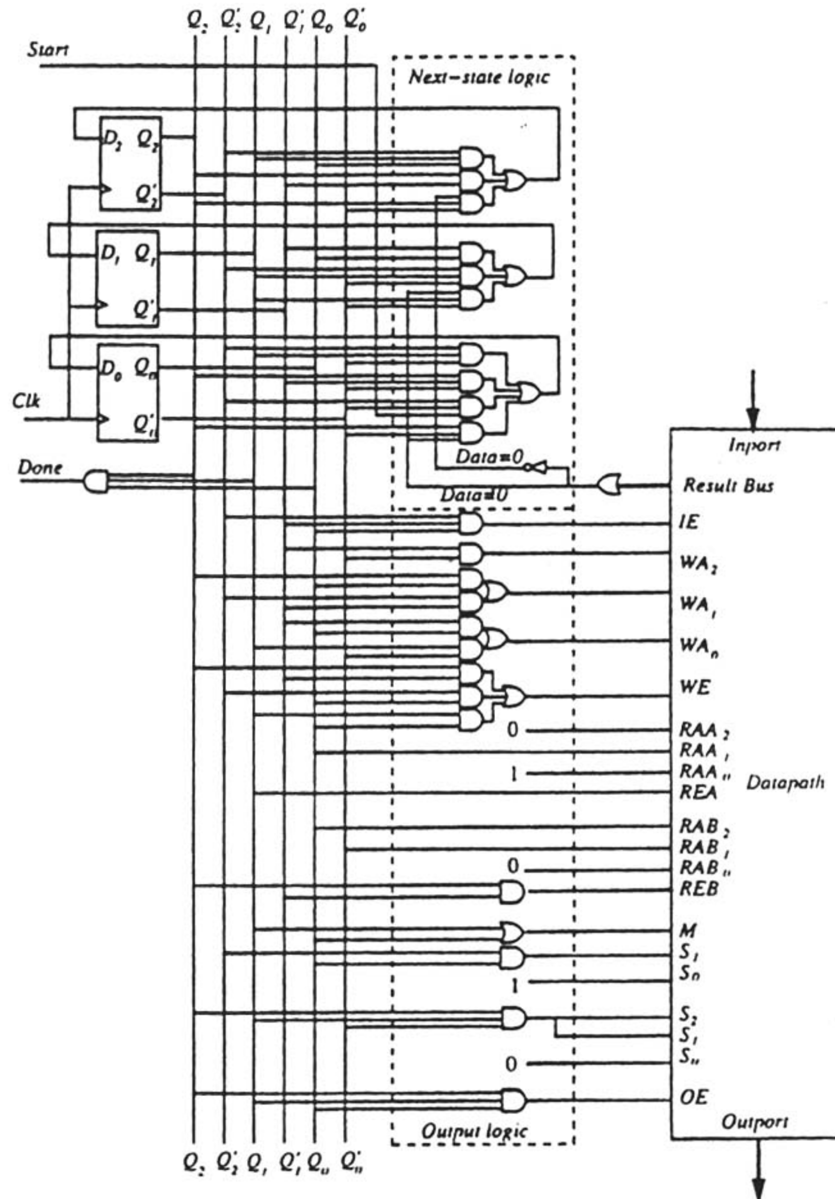
(b) Karnaugh map

Optimized by hand

$$Q_2(next) = Q'_2Q_1Q_0 + Q_2Q'_1 + (Data \neq 0)Q_2Q'_0$$
$$Q_1(next) = Q'_1Q_0 + Q'_2Q'_1Q'_0 + (Data \neq 0)Q_1Q'_0$$
$$Q_0(next) = Q'_2Q_1Q'_0 + Start\ Q'_2Q'_0 + (Data \neq 0)Q_2Q'_0$$

(c) Next-state equations

Controller of One's-Counter (5/8)



Optimized by hand

Controller of One's-Counter (6/8)

Optimized by EDA tool

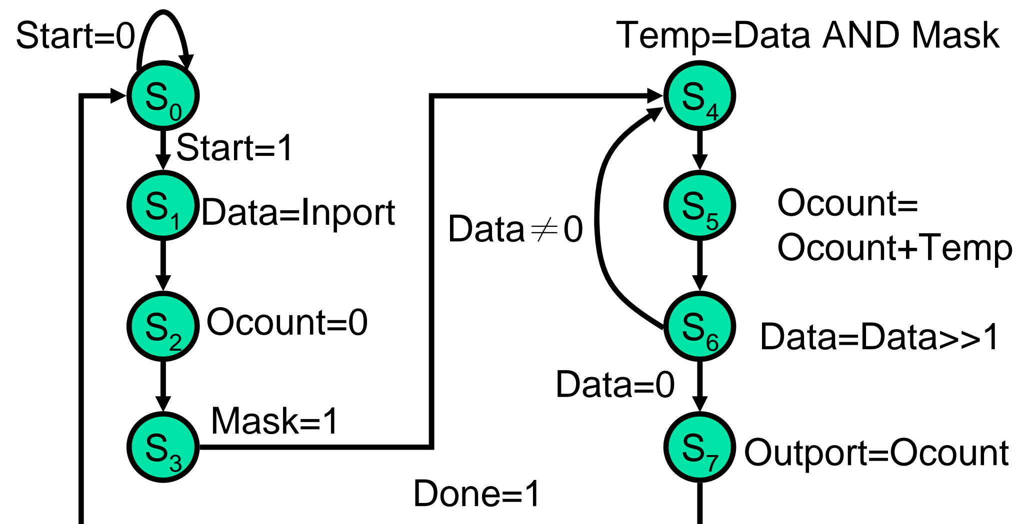
```
module    one_counter(clock,reset,start,inport,done,outport);
input     clock,reset,start; input     [7:0] inport;
output    done;  output    [7:0] outport;
wire      [7:0] data;  wire      [19:0] control_word;
control_unit NO1(.clock(clock), .reset(reset), .start(start), .data(data),
                 .control_word(control_word), .done(done));
data_path NO2(.clock(clock), .reset(reset), .control_word(control_word),
              .inport(inport), .outport(outport), .data(data));
endmodule

module    control_unit(clock,reset,start,data,control_word,done);
input     clock,reset,start;  input     [7:0] data;
output    done;  output    [19:0] control_word;
parameter  S0=0,S1=1,S2=2,S3=3,S4=4,S5=5,S6=6,S7=7;
reg       [2:0] currentstate,nextstate;
reg       done;  reg       [19:0] control_word;
always@(posedge clock)
begin
    if(reset)
      currentstate=S0;
    else
      currentstate=nextstate;
end
```

State Register (Seq. C.)

Controller of One's-Counter (7/8)

Next State Logic (Comb. C.)

```
always@(currentstate or start or data)
begin
    case(currentstate)
        S0:
          begin
              if(start==0)
                nextstate=S0;
              else
                nextstate=S1;
          end
        S1:
          nextstate=S2;
        S2:
          nextstate=S3;
        S3:
          nextstate=S4;
        S4:
          nextstate=S5;
        S5:
          nextstate=S6;
S6:
          begin
              if(data!=8'h00)
                nextstate=S4;
              else
                nextstate=S7;
          end
        S7:
          nextstate=S0;
    endcase
end
```



Controller of One's-Counter (8/8)

	IE	WR AD	READ AD A	READ AD B	ALU	SHIFT	OE
1	1	R_1	X	X	X	X	0
2	0	R_3	0	0	Add	Pass	0
3	0	R_2	0	X	Inc	Pass	0
4	0	R_4	R_1	R_2	AND	Pass	0
5	0	R_3	R_3	R_4	Add	Pass	0
6	0	R_1	R_1	0	Add	Shift right	0
7	0	None	R_3	0	Add	Pass	1

Output Logic (Comb. C.)

```
always@(currentstate)
begin
    done=0;
    case(currentstate)
      S0:
        control_word=20'b00XXX0XXX0XXXXXXXXX0;
      S1:
        control_word=20'b110010XXX0XXXXXXXXX0;
      S2:
        control_word=20'b010110XXX0XXX1010000;
      S3:
        control_word=20'b010100XXX1XXX1110000;
      S4:
        control_word=20'b01100100110100010000;
      S5:
        control_word=20'b01011101111001010000;
      S6:
        control_word=20'b0100110010XXX1011100;
      S7:
        begin
            control_word=20'b00XXX10110XXX1010001;
            done=1;
        end      endcase   end   endmodule
```

Traffic Light Controller (1/7)

Control Unit

Datapath

Comb. Seq. Comb.

Next State Logic

Output Logic

State Registers

Red

Green

Yellow

4 2 0

R_t G_t Y_t

Counter

Current_times

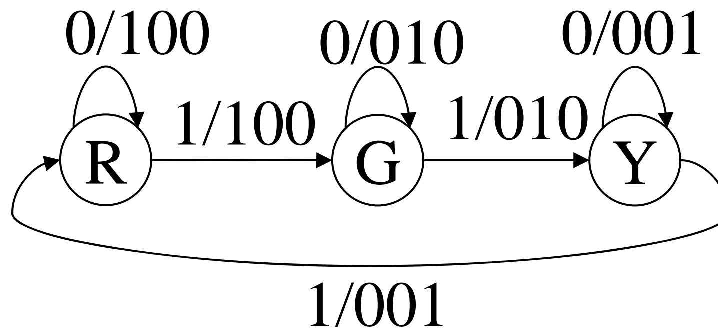
Comparator

Recount_counter

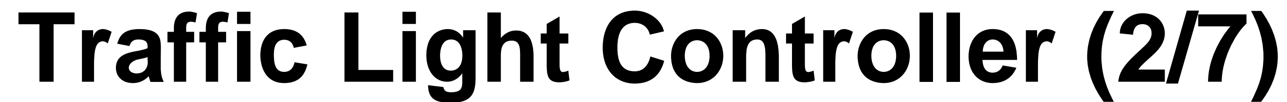
Recount_counter

Input/Output

Recount_Counter16/Red Green Yellow



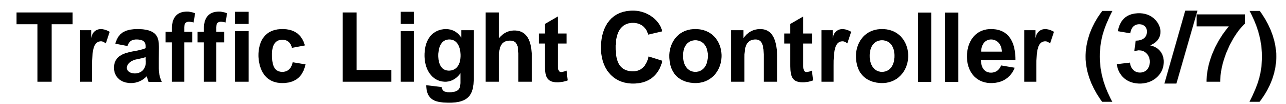
R_time: 4+1=5 cycles
G_time: 2+1=3 cycles
Y_time: 0+1=1 cycles



Traffic Light Controller (2/7)

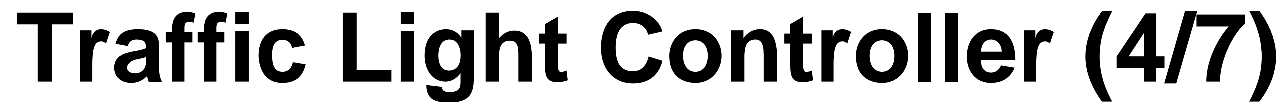
```
module  traffic(Clock,Reset,Red,Green,Yellow);
input Clock,Reset;  output Red,Green,Yellow;
wire Recount_conter;  wire [3:0] Counter_Number;

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

Datapath (.Clock(Clock), .Reset(Reset), .RGY({Red,Green,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_Out(Counter_Number));
endmodule
```



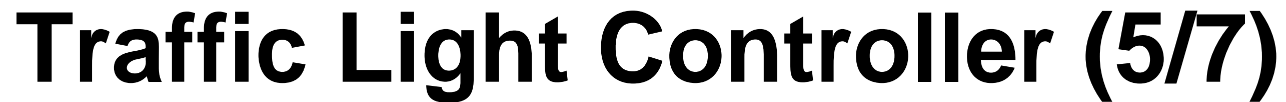
Traffic Light Controller (3/7)

```
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
```



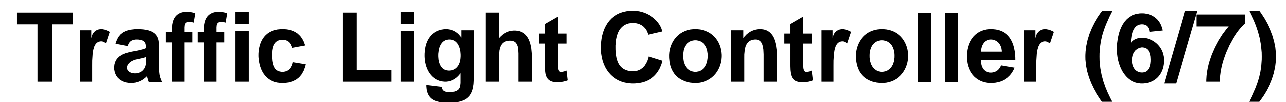
Traffic Light Controller (5/7)

```
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.)



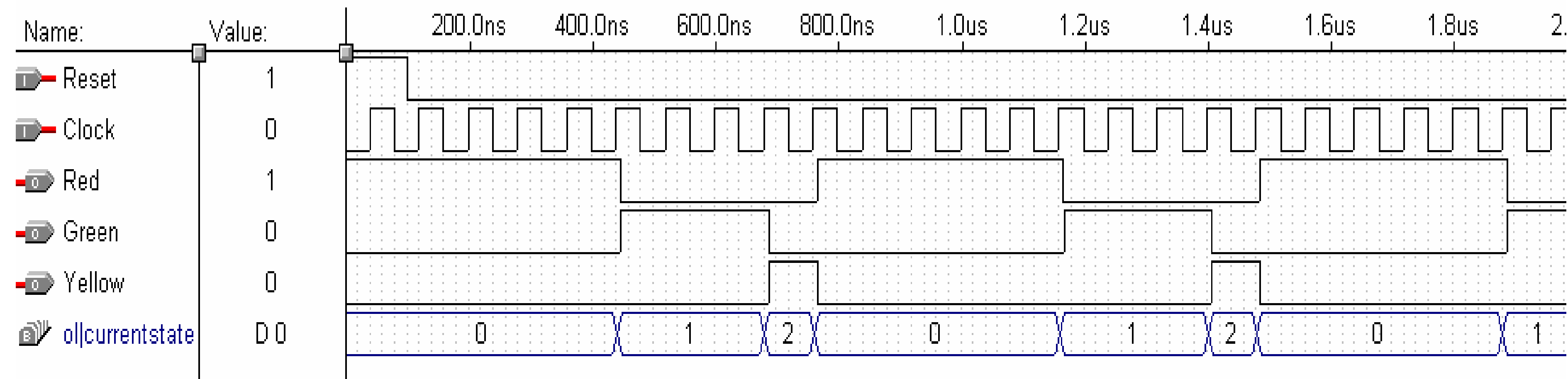
Traffic Light Controller (6/7)

```
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.)

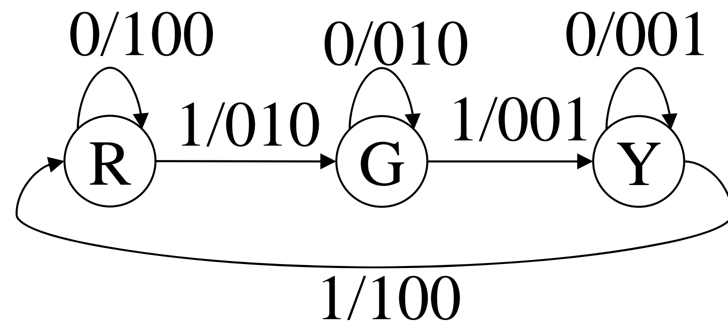
Traffic Light Controller (7/7)

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

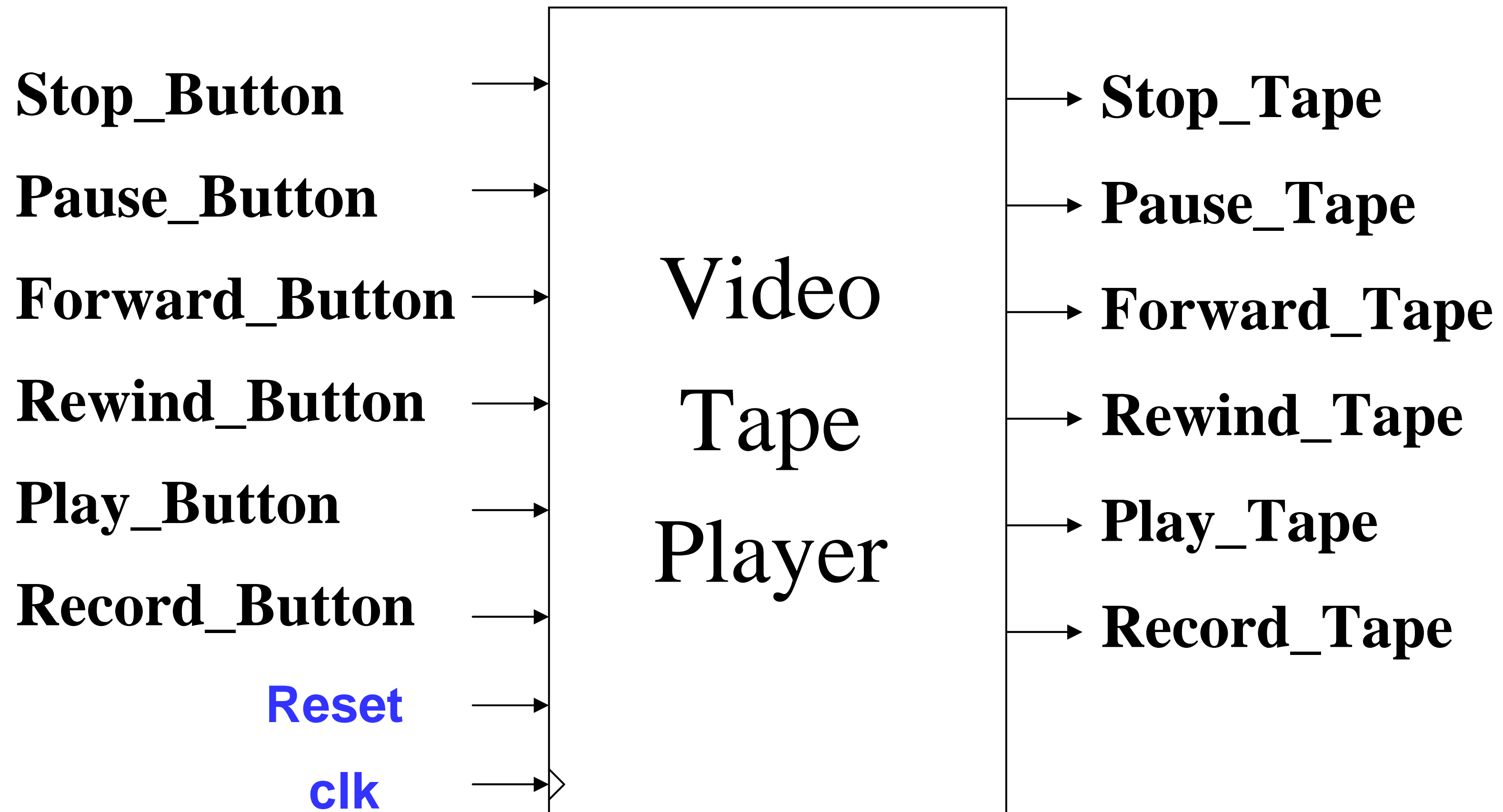


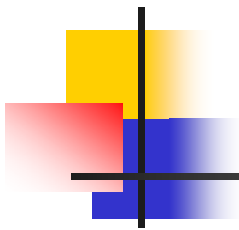
Input/Output Recount_Counter16/Red Green Yellow

Homework: Design traffic controller with Mealy machine

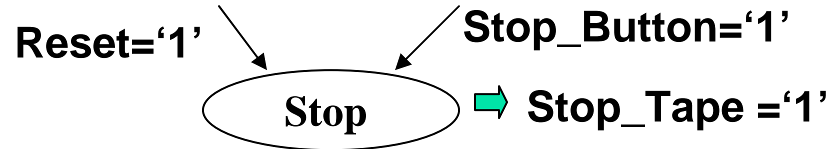


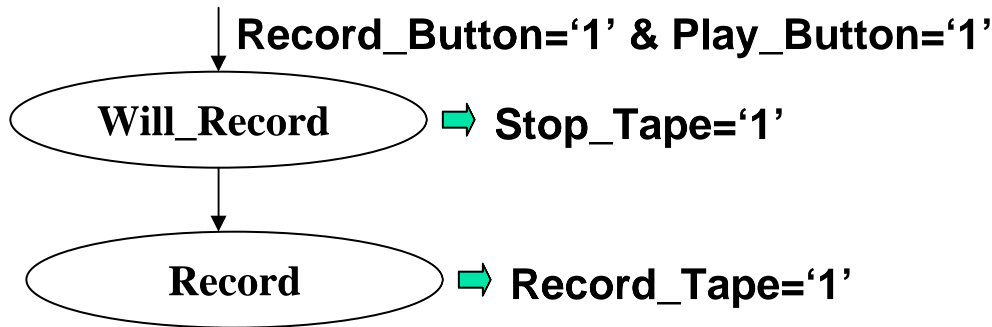
Video Tape Player (1/4)

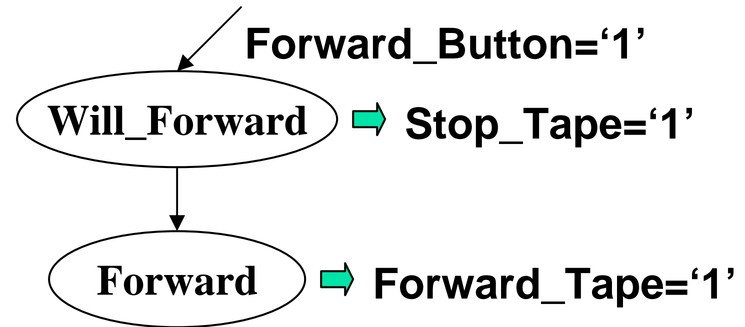


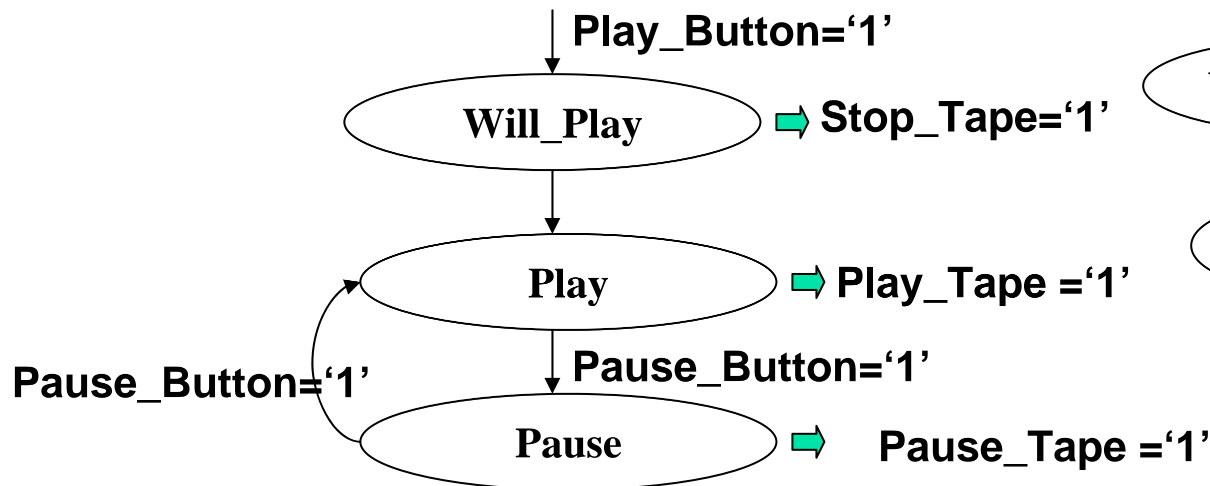


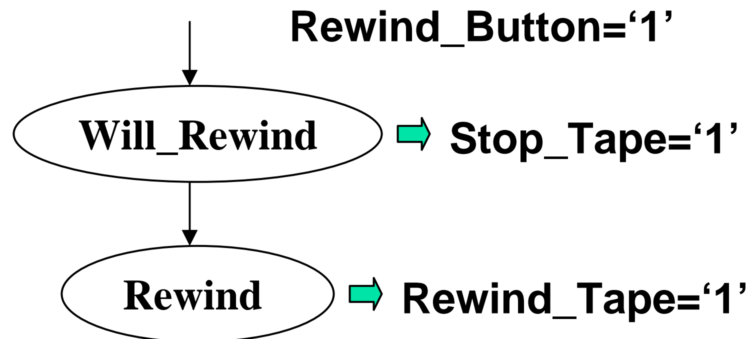
Video Tape Player (2/4)











Why needs will-XX state ?

Video Tape Player (3/4)

```
parameter STOP = 4'b0000, WILL_FORWARD = 4'b0001,
FORWARD = 4'b0010, WILL_REWIND = 4'b0011,
REWIND = 4'b0100, WILL_PLAY = 4'b0101,
PLAY = 4'b0110, PAUSE = 4'b0111,
WILL_RECORD = 4'b1000, RECORD = 4'b1001;

assign {Stop_Tape, Pause_Tape, Forward_Tape,
Rewind_Tape, Play_Tape, Record_Tape} = outTape;
always @(posedge clk or posedge reset)
begin
        if(reset)
          curState = 0;
        else
          curState = nextState;

end

always @(curState)
begin
  case(curState)
  STOP: outTape = 6'b100000;
  WILL_FORWARD: outTape = 6'b100000;
  FORWARD: outTape = 6'b001000;
  WILL_REWIND: outTape = 6'b100000;
  REWIND:  outTape = 6'b000100;
  WILL_PLAY:  outTape = 6'b100000;
  PLAY:    outTape = 6'b000010;
  PAUSE:  outTape = 6'b010000;
  WILL_RECORD: outTape = 6'b100000;
  RECORD:  outTape = 6'b000001;
  default:  outTape = 6'b100000;
  endcase
end
```

Video Tape Player (4/4)

```
always @(curState or Stop_Button or
Pause_Button or Forward_Button or
Rewind_Button or Play_Button or Record_Button)
begin
    nextState = STOP;
    case(curState)
    STOP:
            nextState = STOP;
    WILL_FORWARD:
            nextState = FORWARD;
    FORWARD:
            nextState = FORWARD;
    WILL_REWIND:
            nextState = REWIND;
    REWIND:
            nextState = REWIND;
    WILL_PLAY:
            nextState = PLAY;
     PLAY:
            if(Pause_Button)   nextState = PAUSE;
            else               nextState = PLAY;
PAUSE:   if(Pause_Button)  nextState = PLAY;
         else   nextState = PAUSE;
WILL_RECORD:
         nextState = RECORD;
RECORD:
         nextState = RECORD;
endcase

if(STOP || FORWARD || REWIND
|| PLAY || PAUSE || RECORD)
begin
      if(Stop_Button)
       nextState = STOP;
      else if(Record_Button && Play_Button)
       nextState = WILL_RECORD;
      else if(Play_Button)
        nextState = WILL_PLAY;
      else if(Forward_Button)
        nextState = WILL_FORWARD;
      else if(Rewind_Button)
        nextState = WILL_REWIND;
end
end
```

will_forward, will_rewind, will_play and will_record are not affected by any input buttons.
All input buttons are pushed order-sensitive.