BEE 271 Digital circuits and systems

Spring 2017

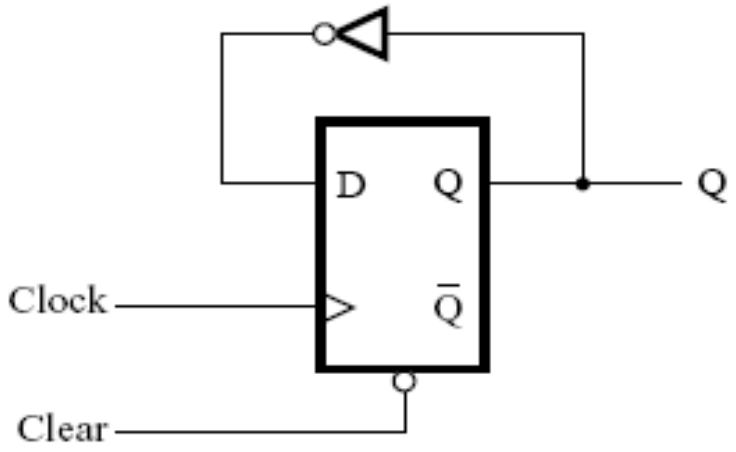
Lecture 12: State machines

Nicole Hamilton

https://faculty.washington.edu/kd1uj

Today's topics

1. Review
2. State machines



Assume:

t_{su} = 0.6 ns

t_h = 0.4 ns

0.8 ns <= t_{cQ} <= 1.0 ns

t_{gate} = 1.0 + 0.1k

where

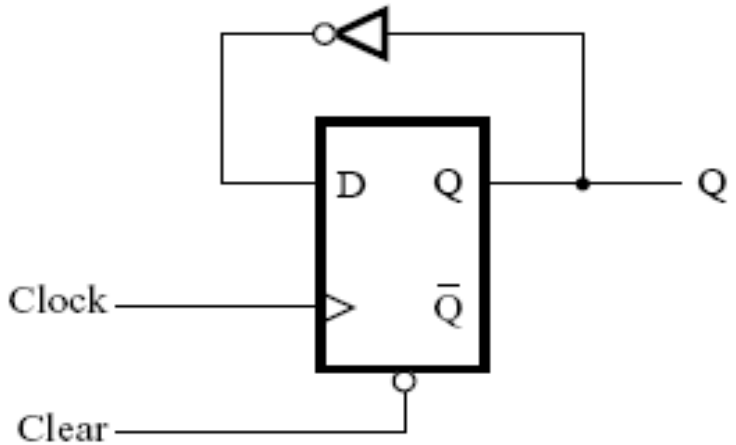
k = number of inputs

$T_{min} = t_{cQmax} + t_{NOT} + t_{su}$

$= 1.0 + 1.1 + 0.6 = 2.7$ ns

F_{max} = 1/2.7 ns = 370.37 MHz.

Figure 5.66. A simple flip-flop circuit.



Checking hold time

Assume:

t_{su} = 0.6 ns

t_h = 0.4 ns

0.8 ns <= t_{cQ} <= 1.0 ns

t_{gate} = 1.0 + 0.1k

where

k = number of inputs

Shortest delay = t_{cQmin} + t_{NOT}
= 0.8 + 1.1 = 1.9 ns

Since 1.9 ns > t_h = 0.4 ns,

no hold violation.

Figure 5.66. A simple flip-flop circuit.

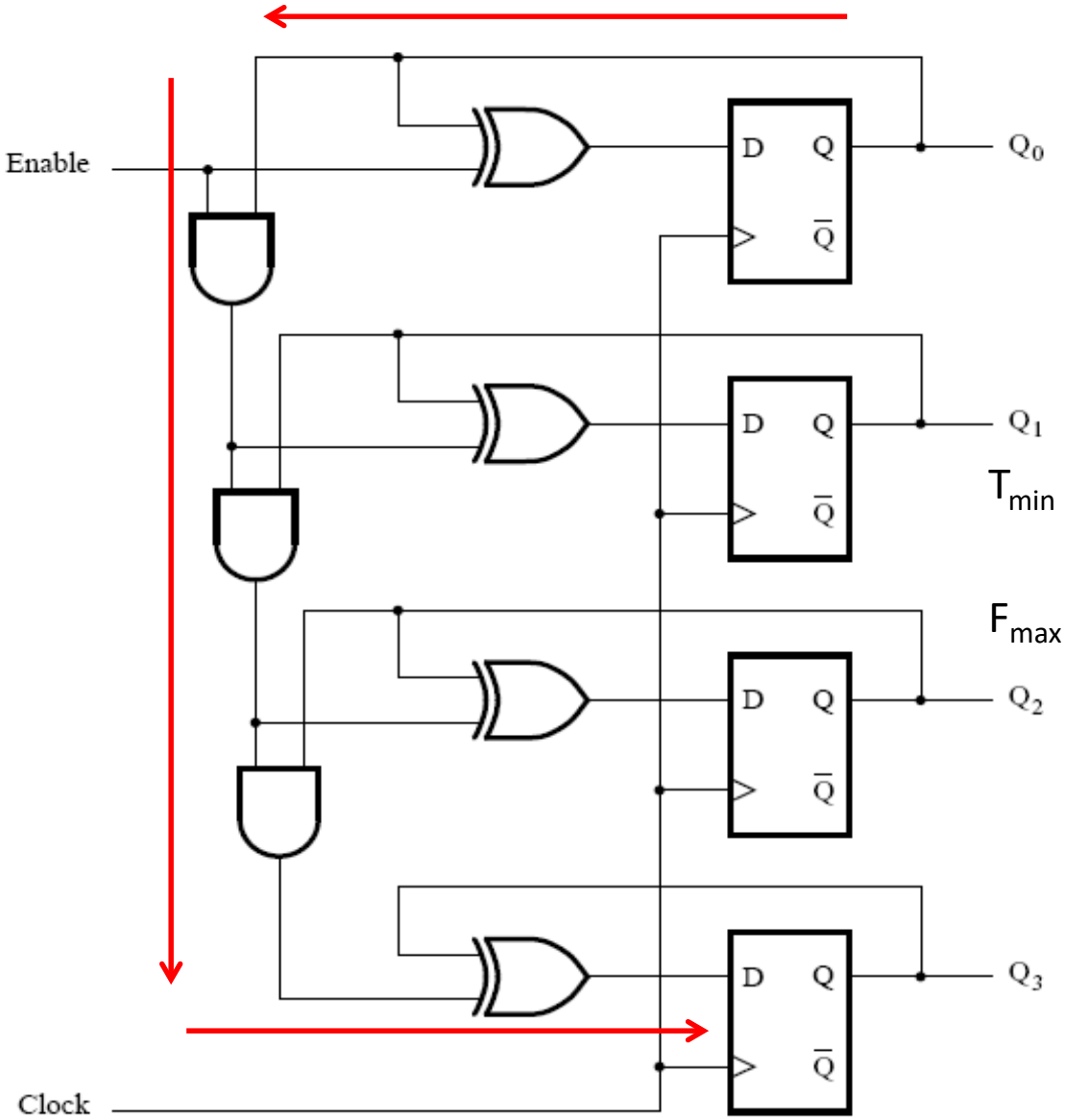


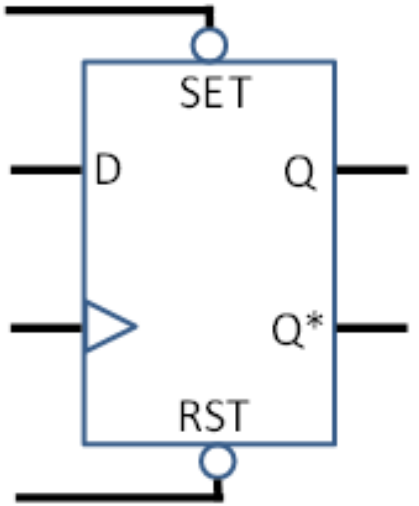
Figure 5.67. A 4-bit counter *critical path*.

$T_{min} = t_{cQmax} + 3\,t_{AND} + t_{XOR} + t_{su}$

$= 1.0 + 3(1.2) + 1.2 + 0.6 = 6.4$ ns

$F_{max} = 1/6.4$ ns $= 156.25$ MHz.

```
module Dflipflop( input clock,
      reset, D, output reg Q );

   always @( posedge clock )
      Q <= reset ? 0 : D;

endmodule
```



Edge-triggered

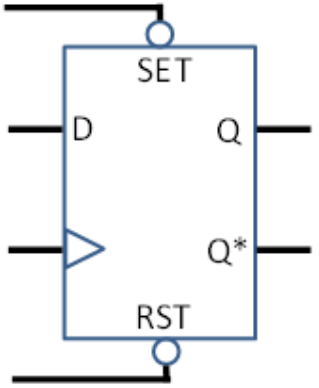
Code for a D flip-flop using a <= non-blocking assignment.

```
module Dflipflop( input clock,
      reset, D, output reg Q );

   always @( posedge clock )
      Q <= reset ? 0 : D;

endmodule
```

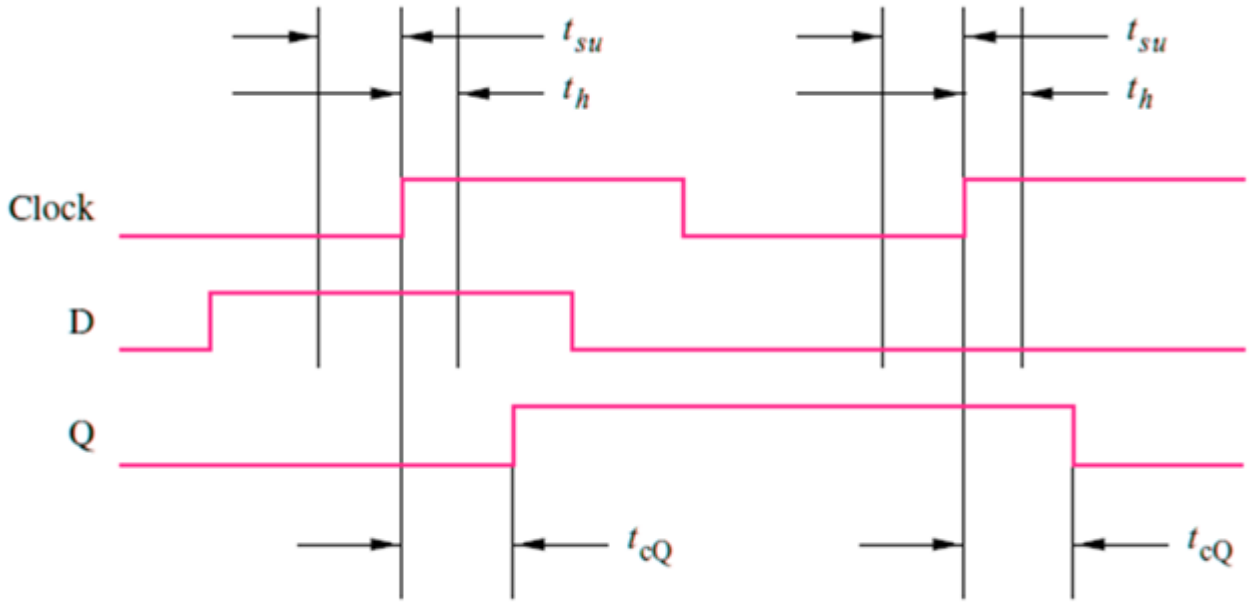
The RHS of the <= operator is evaluated just before the clock edge and the assignment is made just after the clock edge.

Code for a D flip-flop using a <= non-blocking assignment.

```
module Dflipflop( input clock,
      reset, D, output reg Q );

   always @( posedge clock )
      Q <= reset ? 0 : D;

endmodule
```



Edge-triggered

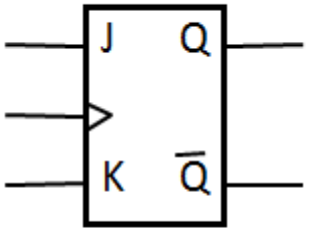


(b) Timing diagram

```
module JK( input clock, J, K, reset,
      output reg Q );

   always @( posedge reset,
         posedge clock )
      casex ( { reset, J, K } )
         'b1xx: Q <= 0;
         'b000: Q <= Q;
         'b001: Q <= 0;
         'b010: Q <= 1;
         'b011: Q <= ~Q;
      endcase

endmodule
```

J	K	Q (t+1)
0	0	Q (t)
0	1	0
1	0	1
1	1	$\overline{Q}$ (t)

(b) Truth table



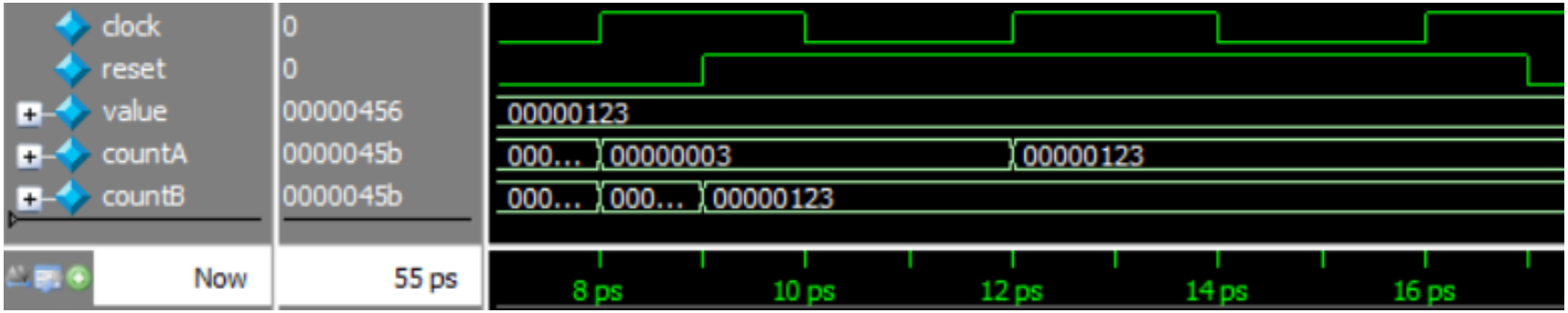
(c) Graphical symbol

```
module SynchReset( input clock, reset, D,
      output Q );

   // Synchronous reset (synchronized to the clock)
   always @( posedge clock )
      Q <= reset ? 0 : D;

endmodule


module AsyncReset( input clock, reset, D, output Q );

   // Asynchronous reset (not synchronized to the clock)
   always @( posedge reset, posedge clock )
      Q <= reset ? 0 : D;

endmodule
```

D flip-flops with synchronous and asynchronous resets.

```
module CounterA(
      input clock, reset,
      input [ 31:0 ] resetValue,
      output reg [ 31:0 ] count = 0 );

   // Synchronous reset (synchronized to the clock)
   always @( posedge clock )
      count <= reset ? resetValue : count + 1;

endmodule


module CounterB(
      input clock, reset,
      input [ 31:0 ] resetValue,
      output reg [ 31:0 ] count = 0 );

   // Asynchronous reset (not synchronized to the clock)
   always @( posedge reset, posedge clock )
      count <= reset ? resetValue : count + 1;

endmodule
```

From simulation, reset in *synchronous* in CounterA, changing only *with* the clock, and *asynchronous* in CounterB.



CounterA

```
always @( posedge clock )
   count <= reset ?
      resetValue : count + 1;
```

CounterB

```
always @( posedge reset, posedge clock )
   count <= reset ?
      resetValue : count + 1;
```

```
module ShiftRegister #( parameter n = 4 )
      ( input clock, reset, D,
      output reg [0 : n - 1] Q );

   always @( posedge clock )
      if ( reset )
         Q <= 0;
      else
         begin
         integer i;
         for ( i = n - 1; i != 0; i = i - 1 )
            Q[ i ] <= Q[ i - 1 ];
         Q[ 0 ] <= D;
         end

endmodule
```

```
module ShiftRegister2 #( parameter n = 4 )
      ( input clock, reset, D,
      output reg [ 0 : n - 1 ] Q );

   always @( posedge clock )
      if ( reset )
         Q <= 0;
      else
         Q <= { D, Q[ 0 : n - 2 ] };
endmodule
```

Finite state machines

Machines whose next state depends on the inputs and the previous state.

Called finite state machines because they have only a ***finite*** number of states.

In a combinational circuit, the values of the outputs are determined solely by the present values of its inputs.

In a sequential circuit, the values of the outputs depend on the past behavior of the circuit, as well as the present values of its inputs.

A sequential circuit has states, which in conjunction with the present values of inputs determine its behavior.

Sequential circuits can be:

- Synchronous – where flip-flops are used to implement the states, and a clock signal is used to control the operation

- Asynchronous – where no clock is used

Terminology

Sequential circuits: Outputs depend on the past behavior as well as the present inputs.

Synchronous sequential: Controlled by a clock.

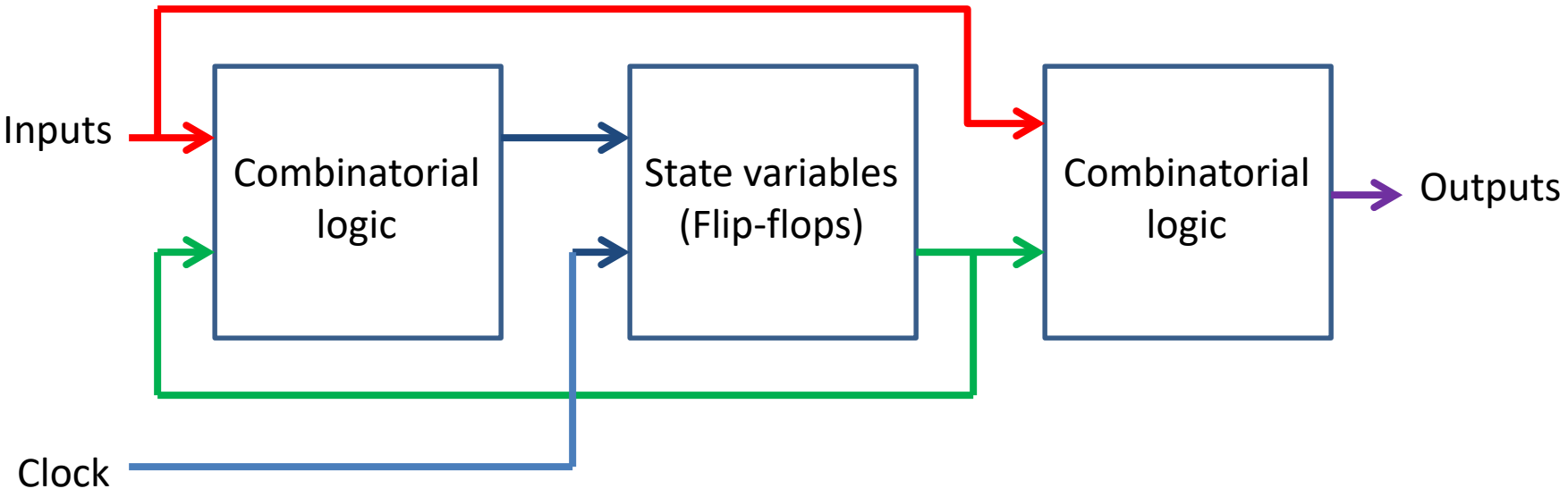
Asynchronous sequential: No clock is used.

State: The stored values of any flip-flops.

Active edge: The edge of the clock that causes the outputs to change.

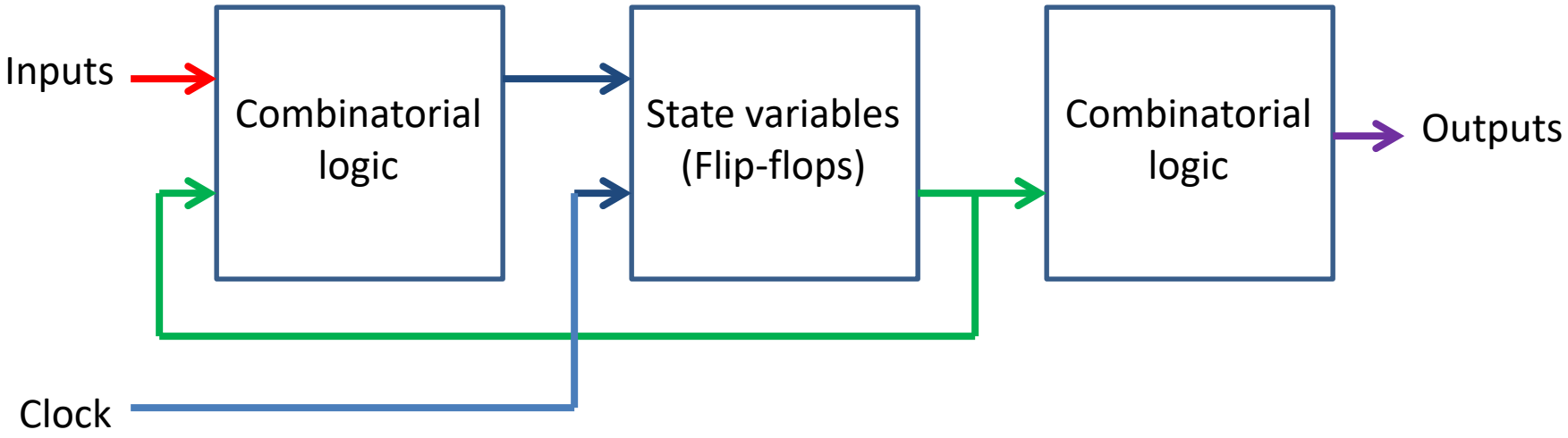
Finite state machine (FSM): Formal name for a sequential circuit.

Generalized form of a sequential circuit

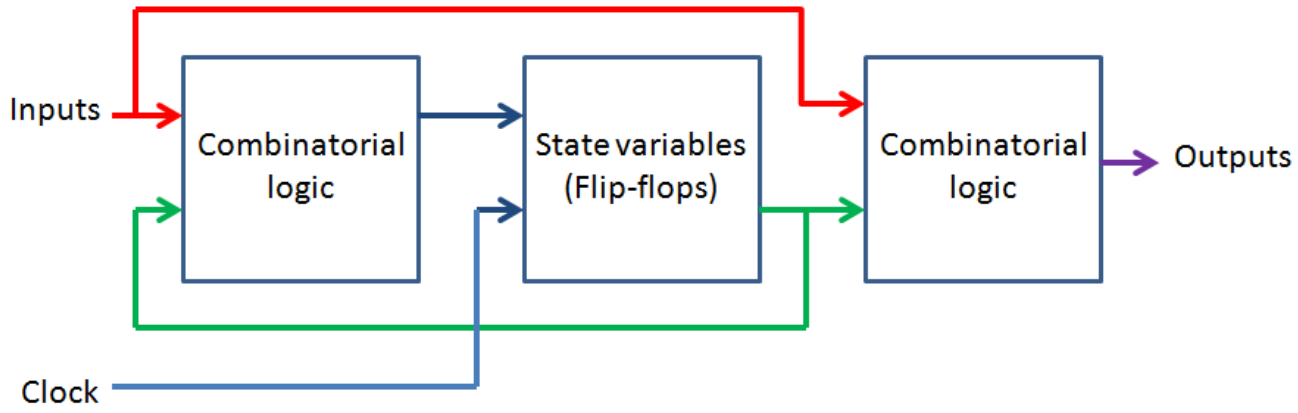


If the outputs depend on both the current state and the current inputs, it is called a ***Mealy*** machine, named after George Mealy, who invented the concept in 1955.

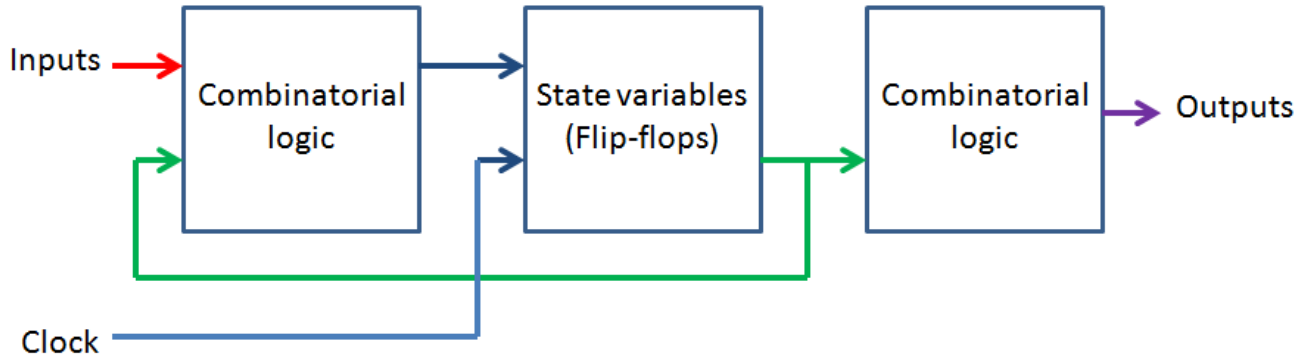
Moore machine



If the outputs depend only on the current state, it is called a ***Moore*** machine. *("Moore is less.")* It's named after Edward Moore, who invented the concept in 1956.



Mealy machines require fewer states but if the inputs change asynchronously, the outputs can change asynchronously as well.



Moore machines require more state variables and the outputs are delayed by one clock. But all the outputs are guaranteed to be synchronous.

Asynchronous inputs and metastability

Two related problems.

1. ***Asynchronous inputs.*** An input may change be between clocks.
2. ***Metastability.*** The input might not actually be a 1 or a 0.

If an input signal with one of these problems is a combinatorial factor in your output, your output will have the same problems.

Solution

We can condition the input somewhat with analog circuitry, e.g., Schmidt triggers.

The digital solution is that we can sample it with a clock and save the result in a flip-flop.

We expect that no matter what the input, the flip-flop will certainly settle to 1 or 0 and that's usually true.

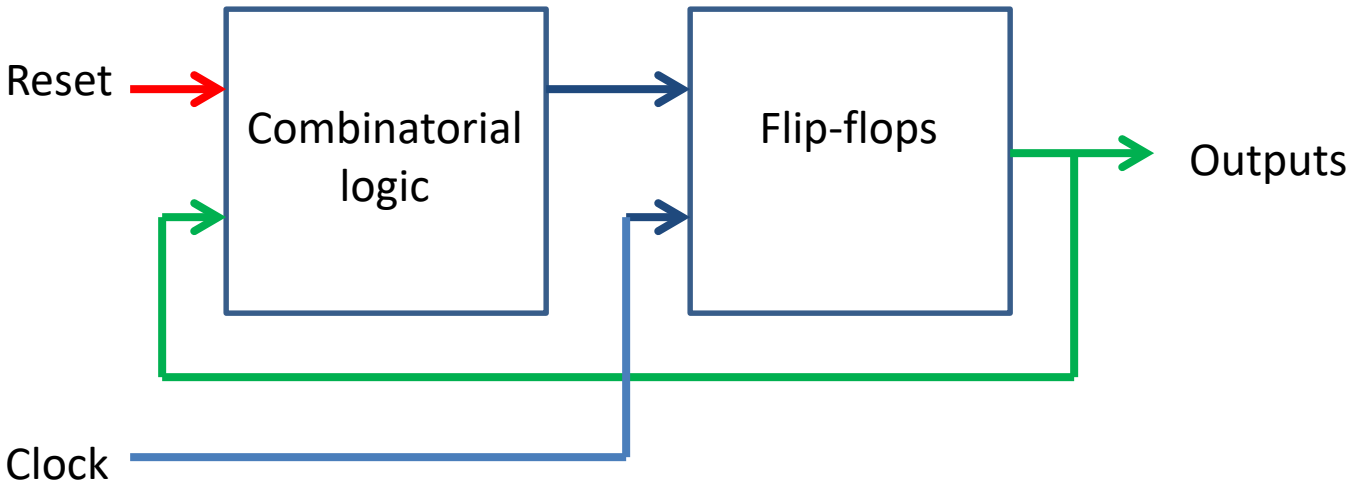
But note

Sampling doesn't always work.

It's possible for a flip-flop to get stuck in a metastable state because of a bad input, never settling into a 0 or 1.

If you add additional flip-flops to create a shift register, you can reduce the likelihood of having a metastable circuit but you cannot make it go away.

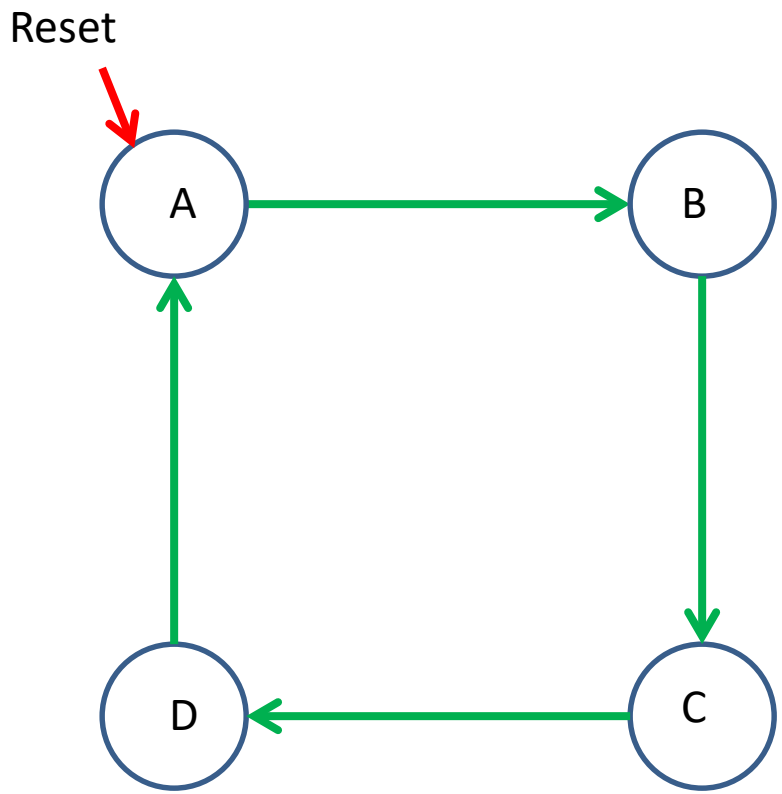
Simplest example: A counter

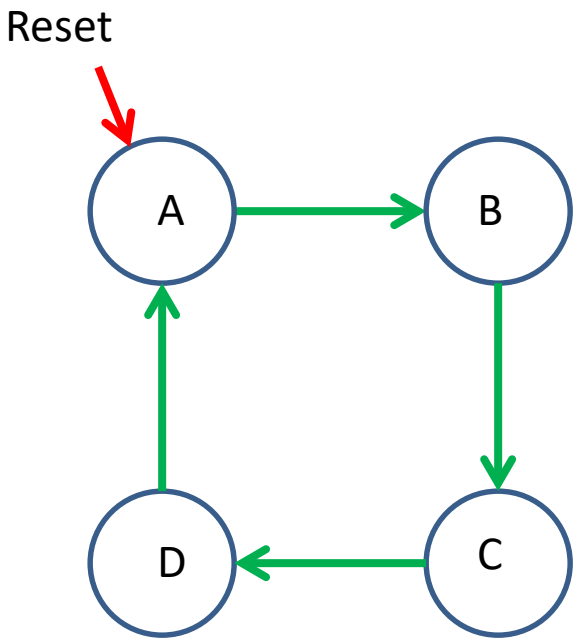


State diagram for a counter with 4 states.

At each clock, it always moves to the next state.

The arrows between states are called ***edges*** or ***transitions***.





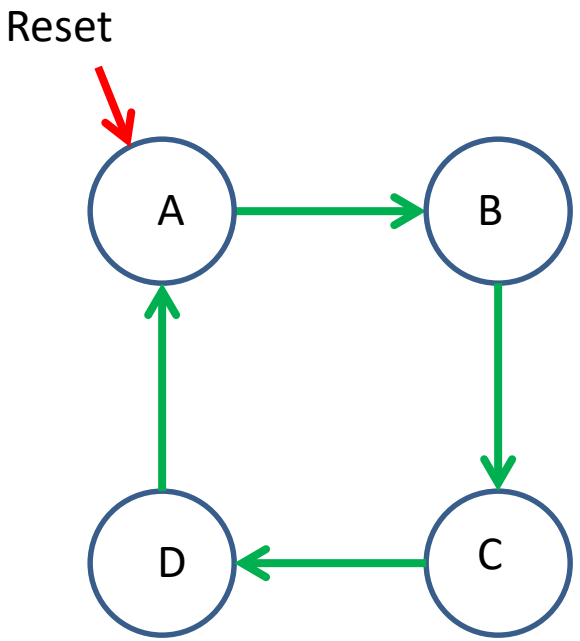
State table

Present state	Next state
A	B
B	C
C	D
D	A

***State-assigned* table**

	Present state y2 y1	Next state Y2 Y1
A	00	01
B	01	10
C	10	11
D	11	00

We pick any state assignments we like, though some could be better than others.



Having picked the assignments, we can use Karnaugh maps to derive the equations for the next state variables.

***State-assigned* table**

	Present state y2 y1	Next state Y2 Y1
A	00	01
B	01	10
C	10	11
D	11	00

Y1		y1	
		0	1
y2	0	1	0
	1	1	0

Y1 = y1'

Y2		y1	
		0	1
y2	0	0	1
	1	1	0

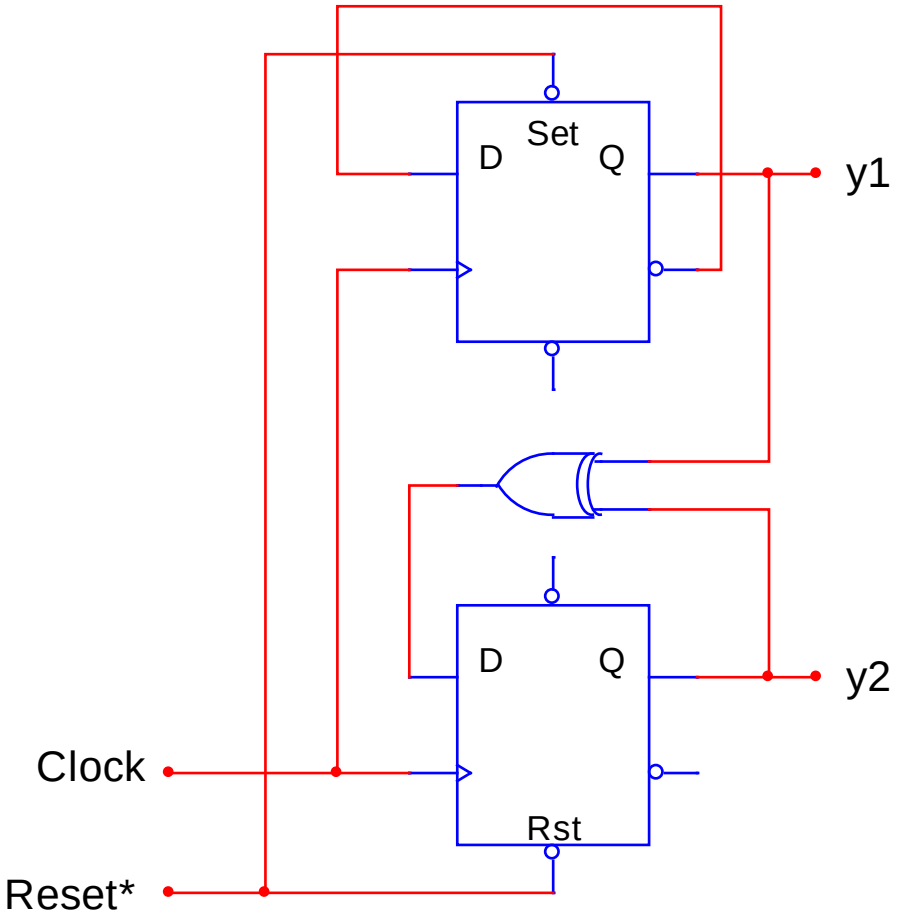
Y2 = y1 ^ y2

Y1		y1	
		0	1
y2	0	1	0
	1	1	0

Y1 = y1'

Y2		y1	
		0	1
y2	0	0	1
	1	1	0

Y2 = y1 ^ y2



Y1		y1	
		0	1
y2	0	1	0
	1	1	0

Y1 = y1'

Y2		y1	
		0	1
y2	0	0	1
	1	1	0

Y2 = y1 ^ y2

```
module Counter( input clock,
      reset,
      output reg y1, y2 );

   always @( posedge reset,
         posedge clock )
      if ( reset )
         begin
         y1 <= 0;
         y2 <= 0;
         end
      else
         begin
         y1 <= ~y1;
         y2 <= y1 ^ y2;
         end
endmodule
```

In Verilog, the <= “non-blocking” assignment means all the assignments happen synchronously at exit from the always block.

In Verilog, we would probably skip the Karnaugh maps write the code directly from the state-assigned table.

***State-assigned* table**

	Present state y2 y1	Next state Y2 Y1
A	00	01
B	01	10
C	10	11
D	11	00

```
module Counter2( input clock,
      reset,
      output reg [ 1:0 ] y );

   always @( posedge reset,
         posedge clock )
      y <= reset ? 0 : y + 1;

endmodule
```

If the cases were more complex or not in order, we might write it like this with a case statement.

***State-assigned* table**

	Present state y2 y1	Next state Y2 Y1
A	00	01
B	01	10
C	10	11
D	11	00

```
module Counter3( input clock,
      reset,
      output reg [ 1:0 ] y );

   always @( posedge reset,
      posedge clock )
         if ( reset )
            y <= 0;
         else
            case ( y )
               0:  y <= 1;
               1:  y <= 2;
               2:  y <= 3;
               3:  y <= 0;
            endcase
endmodule
```

Or we might parameterize the assignments.

***State-assigned* table**

	Present state y2 y1	Next state Y2 Y1
A	00	01
B	01	10
C	10	11
D	11	00

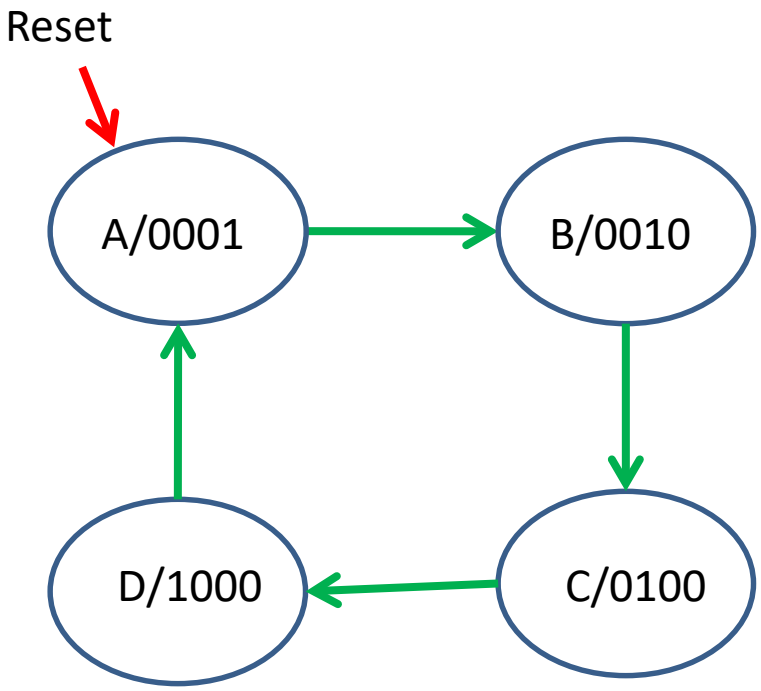
```
module Counter4( input clock,
      reset,
      output reg [ 1:0 ] y );

   parameter A = 0, B = 1,
      C = 2, D = 3;

   always @( posedge reset,
         posedge clock )
      if ( reset )
         y <= 0;
      else
         case ( y )
            A:  y <= B;
            B:  y <= C;
            C:  y <= D;
            D:  y <= A;
         endcase

endmodule
```

Verilog makes it really easy to pick any assignments you like and walk from one arbitrary state to another.

Example: Create a 3-bit counter in Verilog that cycles through this sequence: 4, 7, 0, 3, 2, 6, 1, 5.

Example: Create a 3-bit counter in Verilog that cycles through this sequence: 4, 7, 0, 3, 2, 6, 1, 5.

```
module PseudoRandom( input clock,
    output reg [ 2:0 ] Q );

  always @( posedge clock )
    case ( Q )
      4:  Q <= 7;
      7:  Q <= 0;
      0:  Q <= 3;
      3:  Q <= 2;
      2:  Q <= 6;
      6:  Q <= 1;
      1:  Q <= 5;
      5:  Q <= 4;
    endcase

endmodule
```

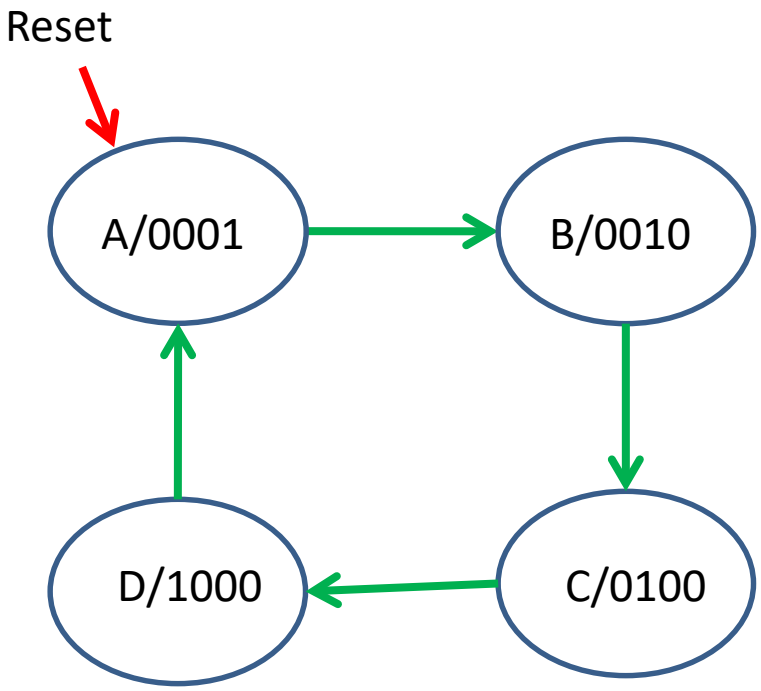


In the previous example, we didn't specify the outputs, merely that it had to count.

But consider this state diagram, where the outputs have been specified.

The "/*nnnn*" part specifies the desired outputs at each state.

Since the outputs depend only on the state, this is a Moore machine.



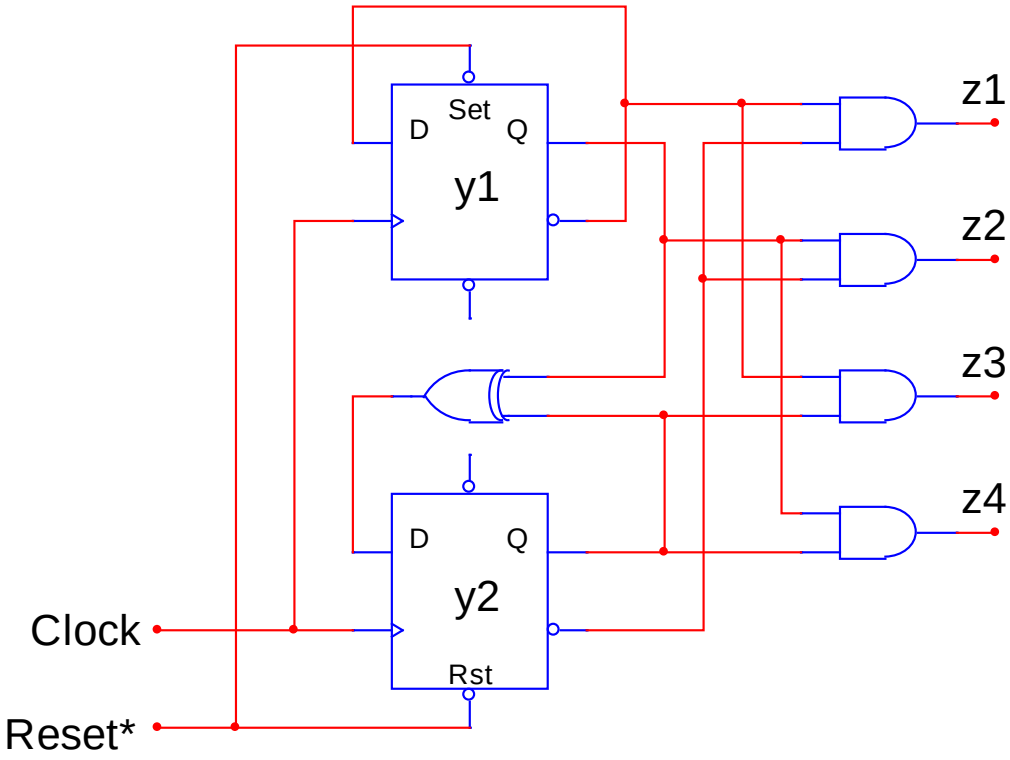
State table

Present state	Next state	Output
A	B	0001
B	C	0010
C	D	0100
D	A	1000

Suppose we use the previous state assignments.

State-assigned table

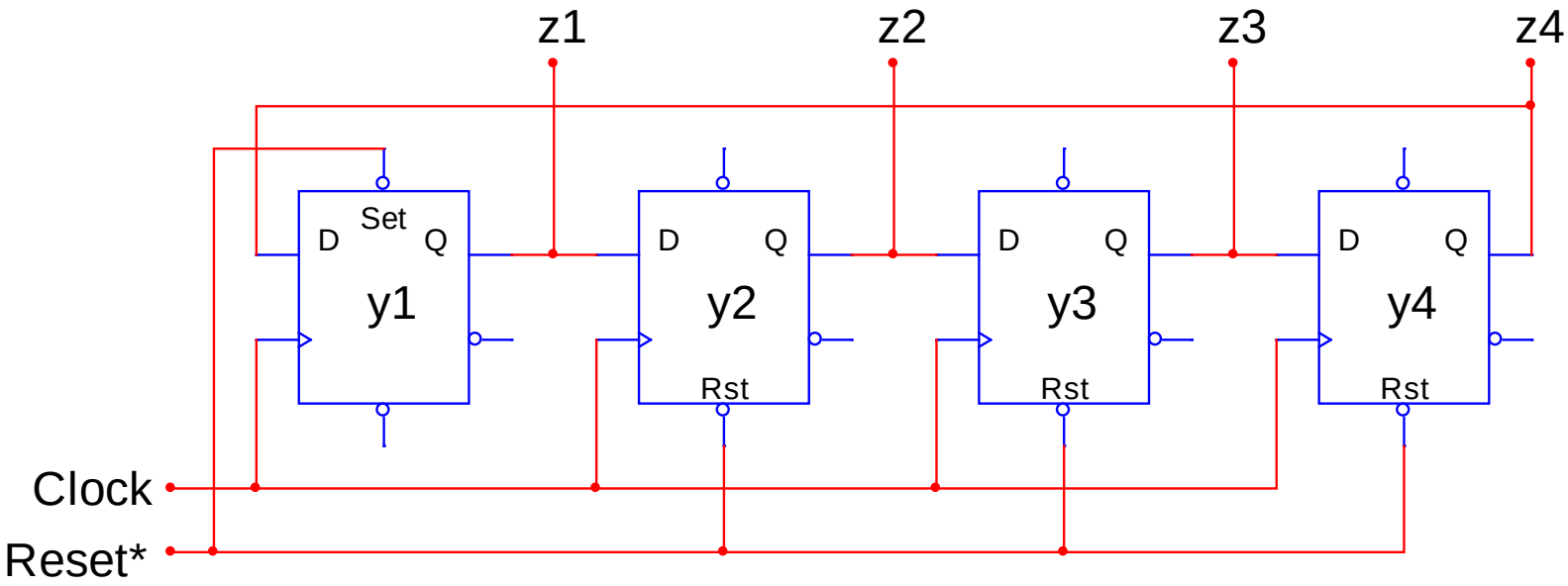
	Present state y2 y1	Next state Y2 Y1	Output z4 z3 z2 z1
A	00	01	0001
B	01	10	0010
C	10	11	0100
D	11	00	1000



State-assigned table

With these assignments, we need a decoder to produce the outputs.

	Present state y2 y1	Next state Y2 Y1	Output z4 z3 z2 z1
A	00	01	0001
B	01	10	0010
C	10	11	0100
D	11	00	1000



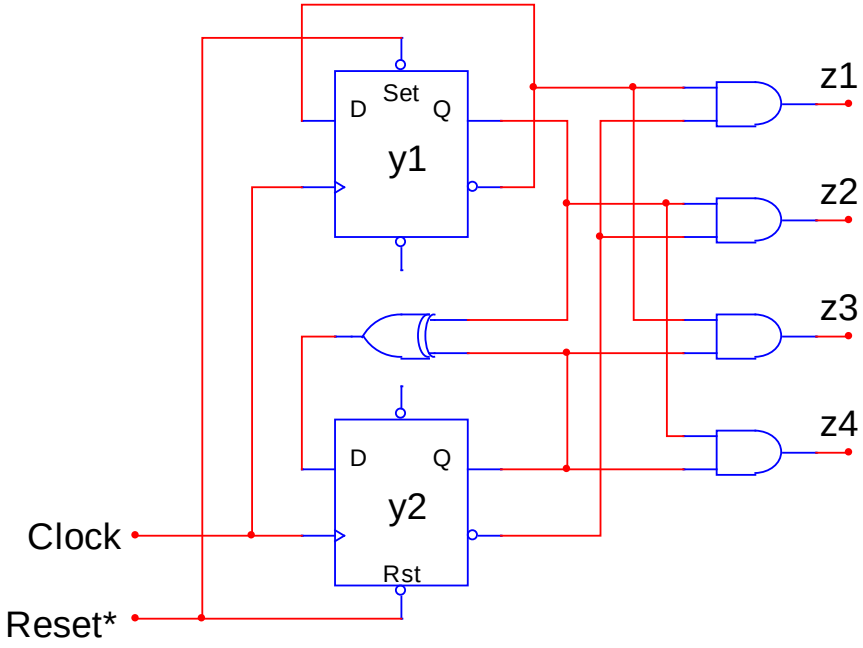
Here is an alternate assignment as a "one-hot" ring counter.

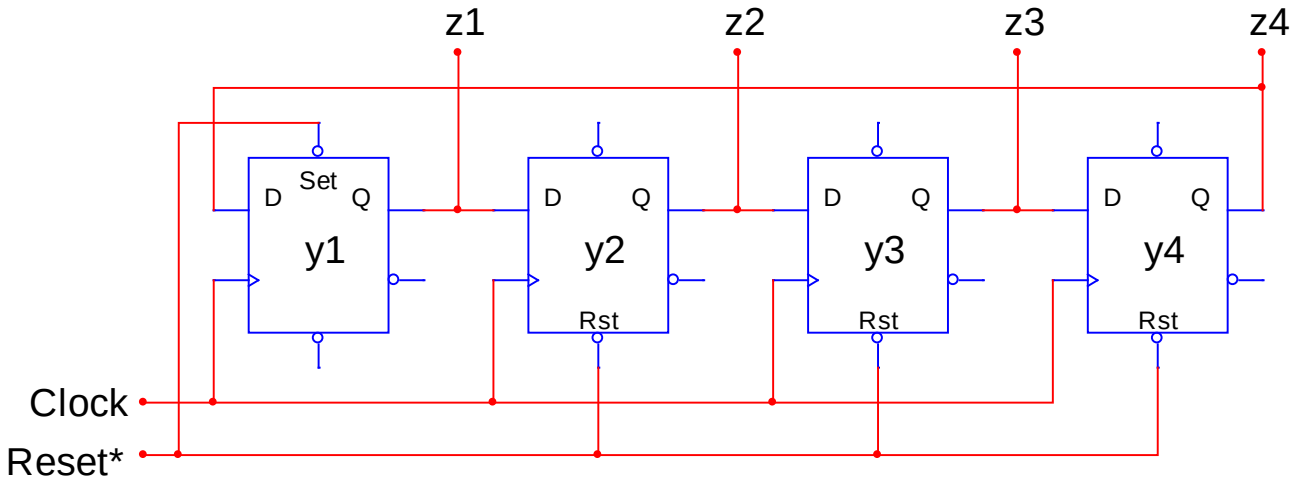
"One hot" state-assigned table

	Present state y4 y3 y2 y1	Next state Y4 Y3 Y2 Y1	Output z4 z3 z2 z1
A	0001	0010	0001
B	0010	0100	0010
C	0100	1000	0100
D	1000	0001	1000

So which is the better design? Both are correct.

Picking a good set of state assignments usually involves consideration of both cost and elegance.





Example: A simple speed control

Outputs 1 if a vehicle's speed is excessive for 2 or more clocks.

Example: A speed governor that limits a vehicle's top speed.

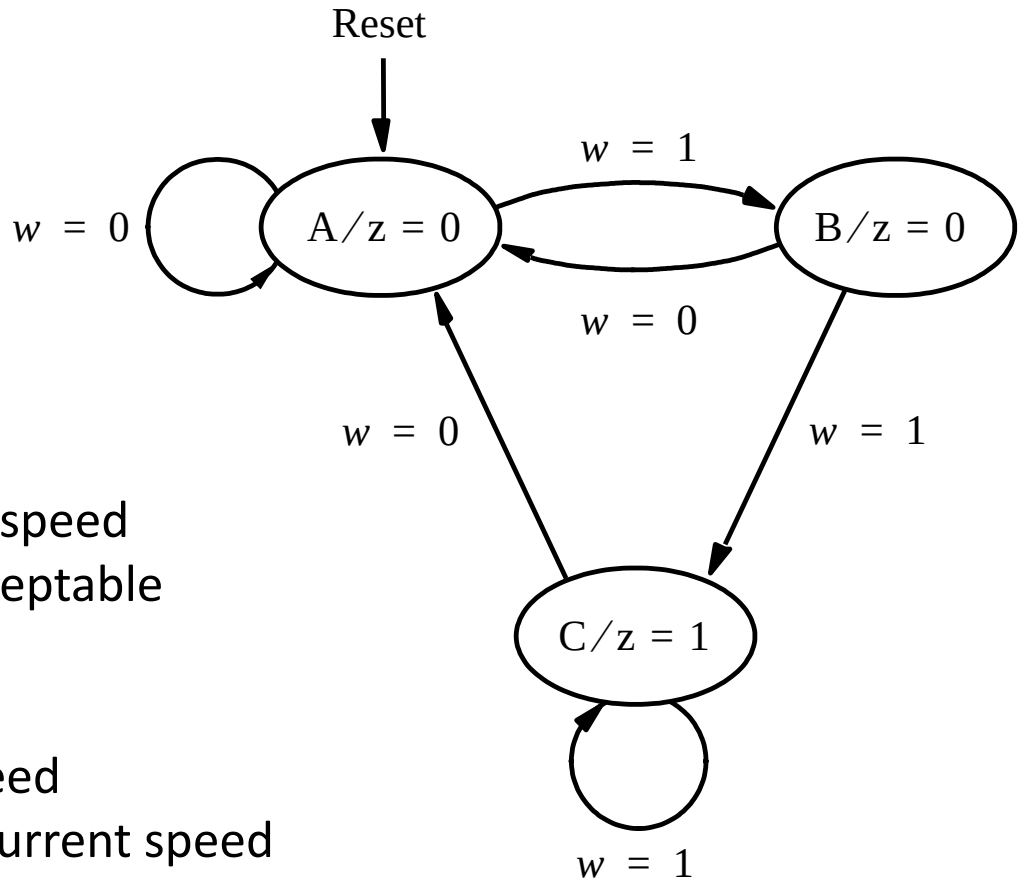
Input:
w == 1 → excessive speed
w == 0 → speed acceptable

Output:
z == 1 → reduce speed
z == 0 → maintain current speed

w == 1 for 2+ clocks → z == 1

Clockcycle:	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
w:	0	1	0	1	1	0	1	1	1	0	1
z:	0	0	0	0	0	1	0	0	1	1	0

Figure 6.2. Sequences of input and output signals.

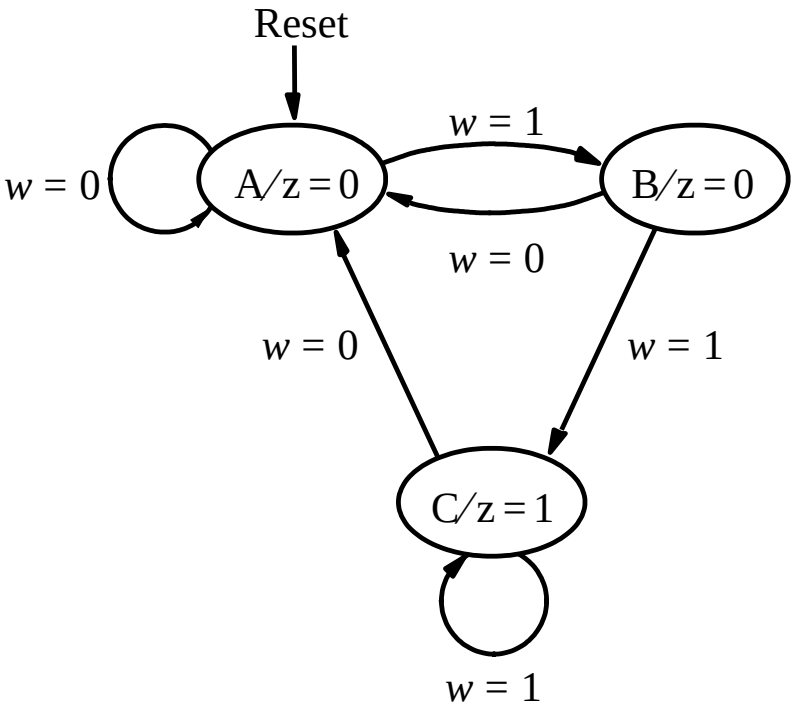


Input:
w == 1 → excessive speed
w == 0 → speed acceptable

Output:
z == 1 → reduce speed
z == 0 → maintain current speed

w == 1 for 2+ clocks → z == 1

Figure 6.3. State diagram for the speed control.



Present state	Next state		Output z
	w = 0	w = 1	
A	A	B	0
B	A	C	0
C	A	C	1

Figure 6.4. State table for the speed control.

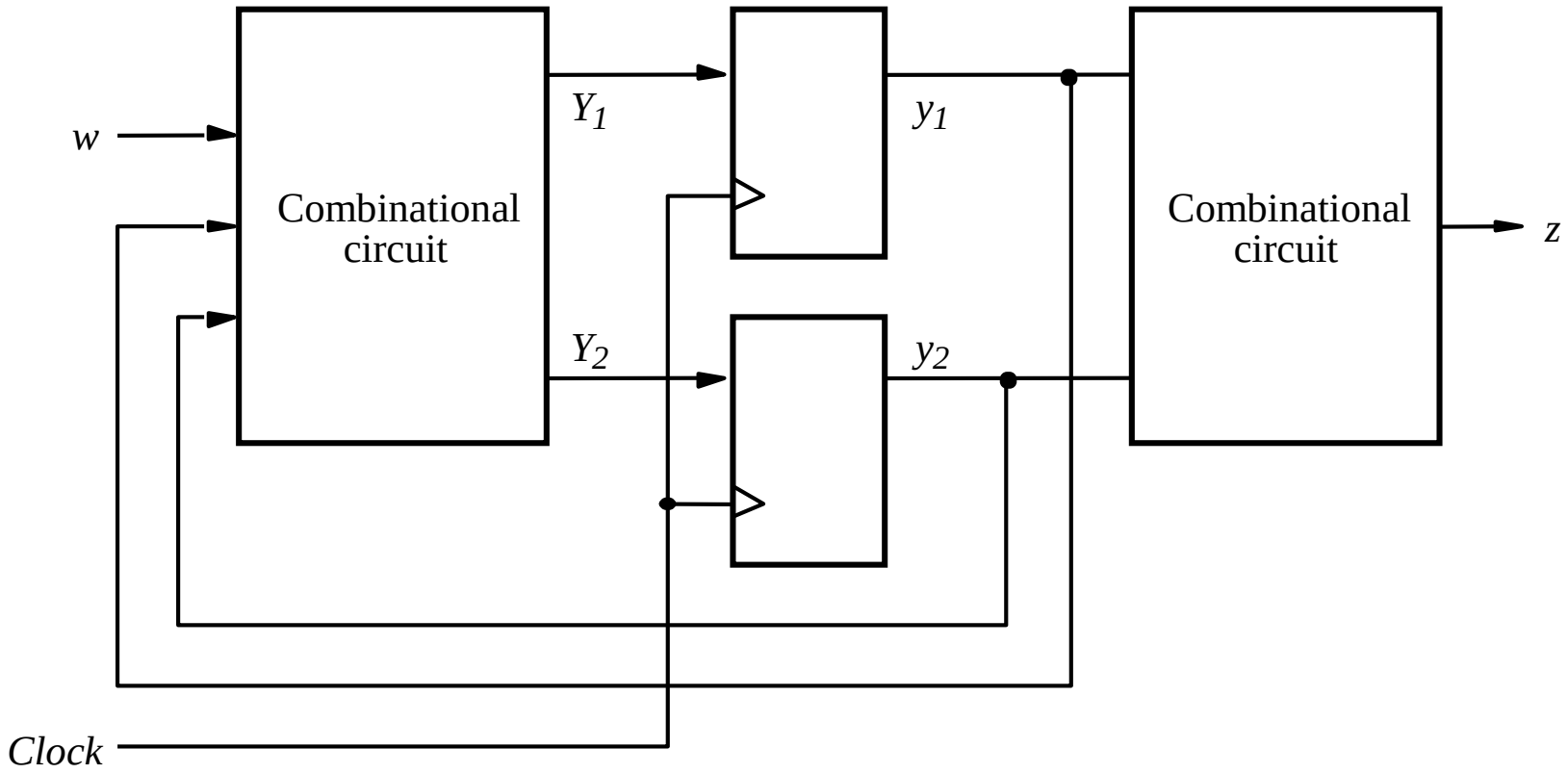
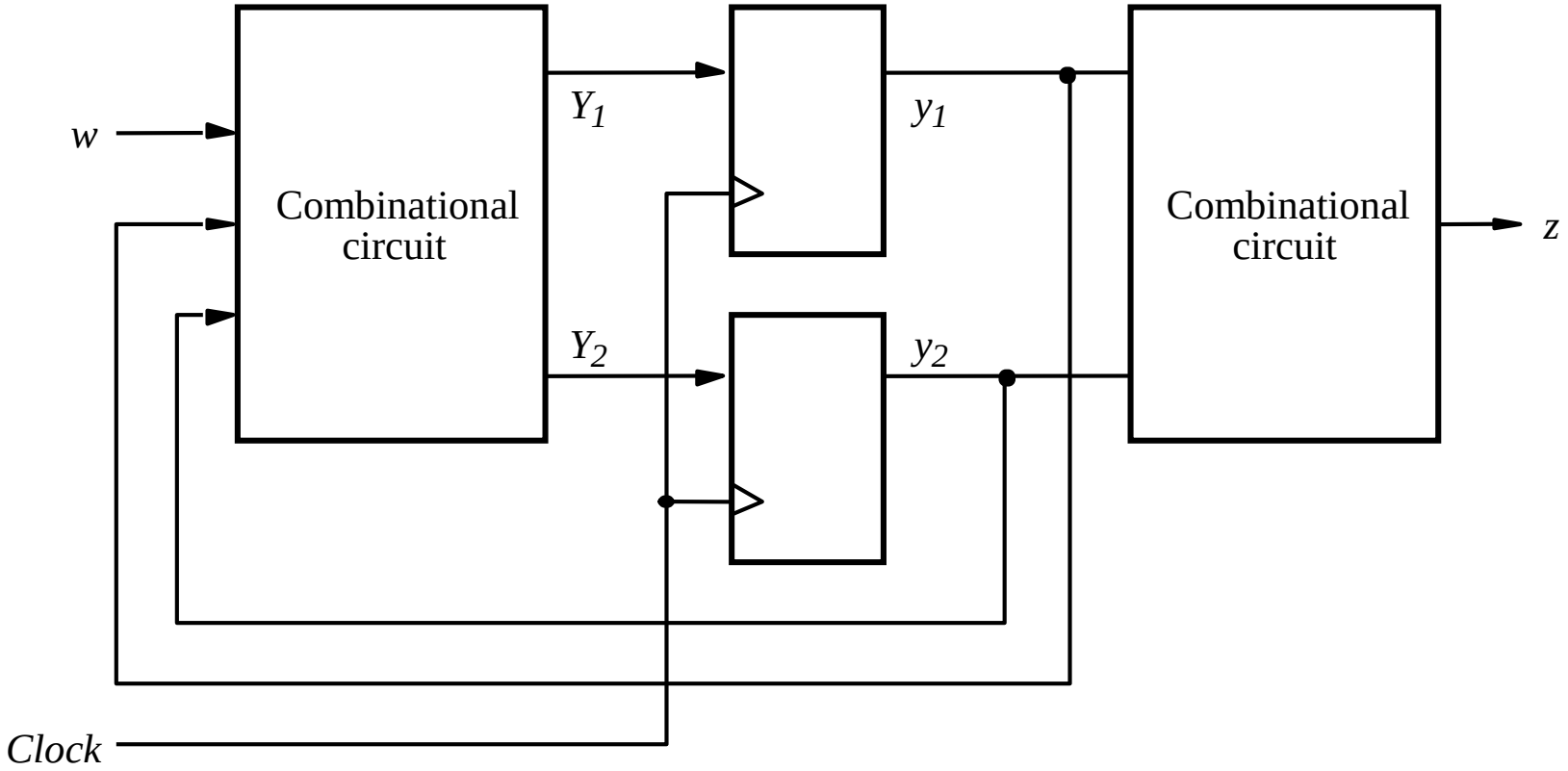


Figure 6.5. A generalized solution to the speed controller, with input *w*, output *z*, and two flip-flops for the three states.



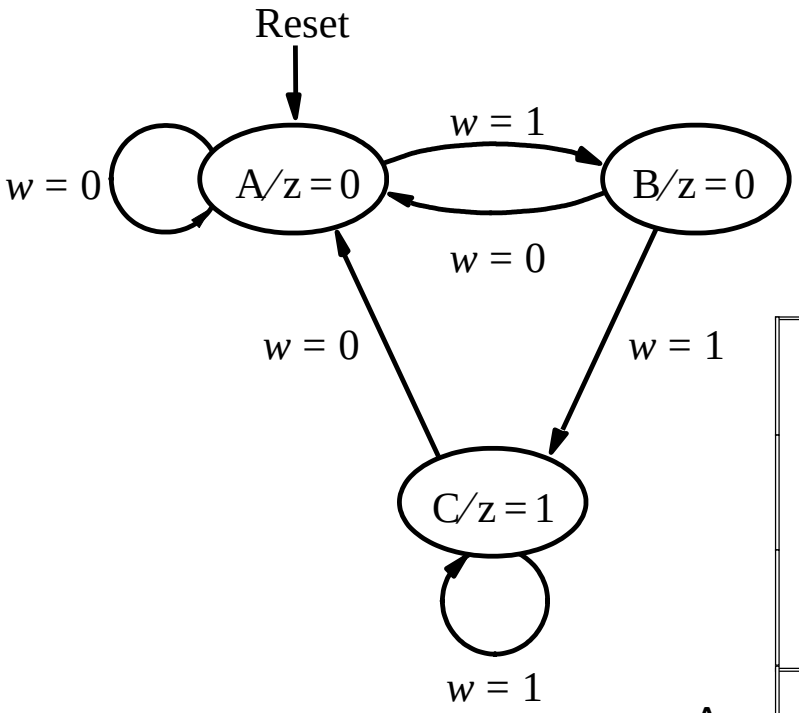
The present state variables, y_1 and y_2, determine the present state of the circuit.

The next state variables, Y_1 and Y_2, determine the state into which the circuit will go after the next active edge of the clock signal.

State variable assignments

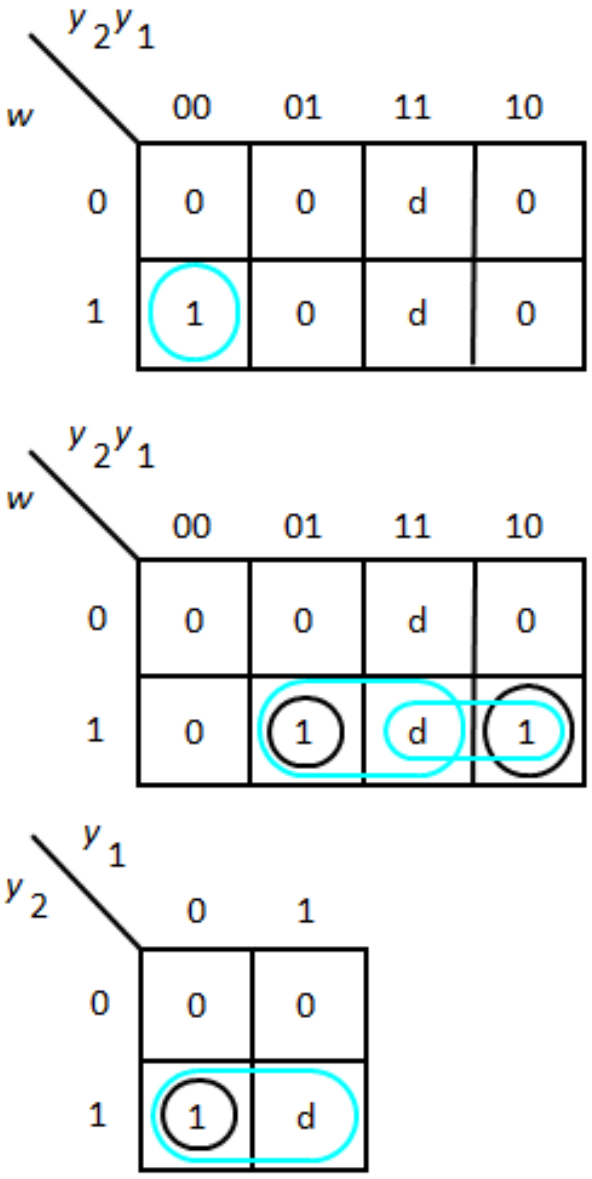
Each of the states in a state diagram or a state table must be represented by some unique combination of 1’s and 0’s.

We have to pick those assignments and some assignments are better than others.



	Present state y_2y_1	Next state $w = 0$ Y_2Y_1	Next state $w = 1$ Y_2Y_1	Output z
A	00	00	01	0
B	01	00	10	0
C	10	00	10	1
	11	*dd*	*dd*	*d*

Figure 6.6. One possible state assignment for the speed controller.



Ignoring don't cares

Y1 = w y1' y2'

Y2 = w y1 y2' + w y1' y2

z = y1' y2

Using don't cares

Y1 = w y1' y2'

Y2 = w y1 + w y2 = w (y1 + y2)

z = y2

Figure 6.7. Karnaugh maps for the next state variables in the speed controller.

	Present state	Next state		Output
		$w = 0$	$w = 1$	
	y_2y_1	Y_2Y_1	Y_2Y_1	z
A	00	00	01	0
B	01	00	10	0
C	10	00	10	1
	11	*dd*	*dd*	*d*

Y1 = w y1' y2'

Y2 = w (y1 + y2)

z = y2

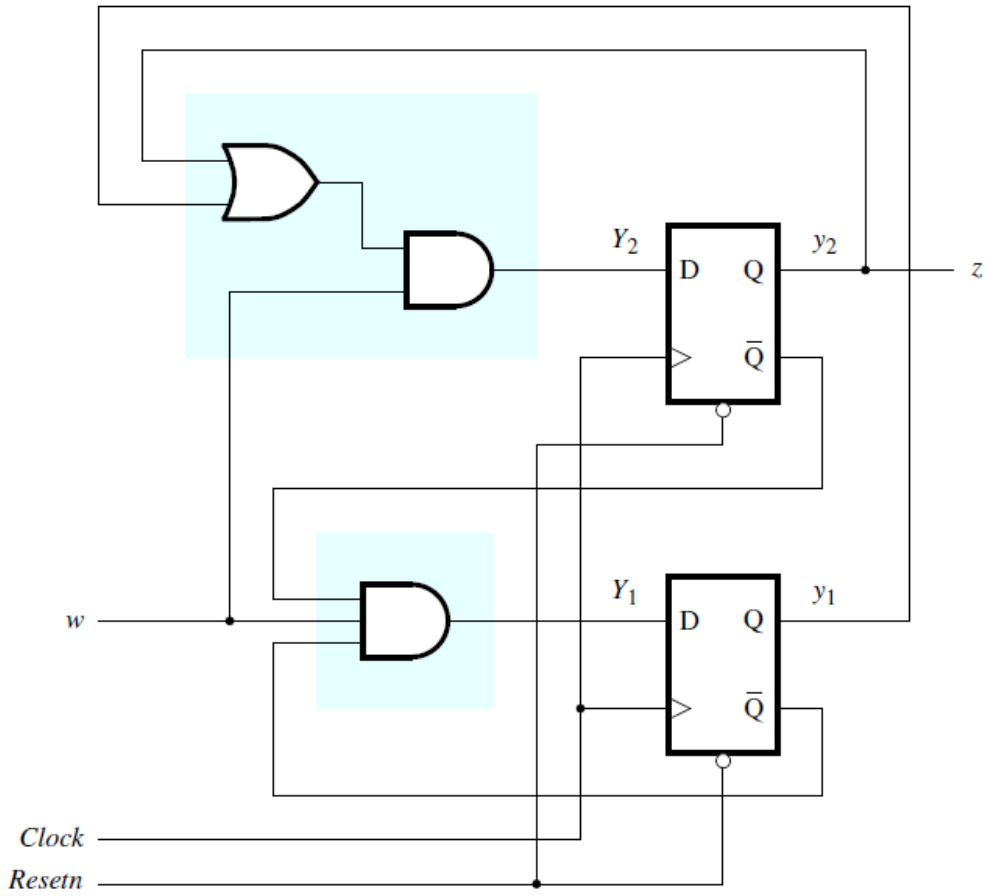


Figure 6.8. Final implementation of the speed controller using the don't cares.

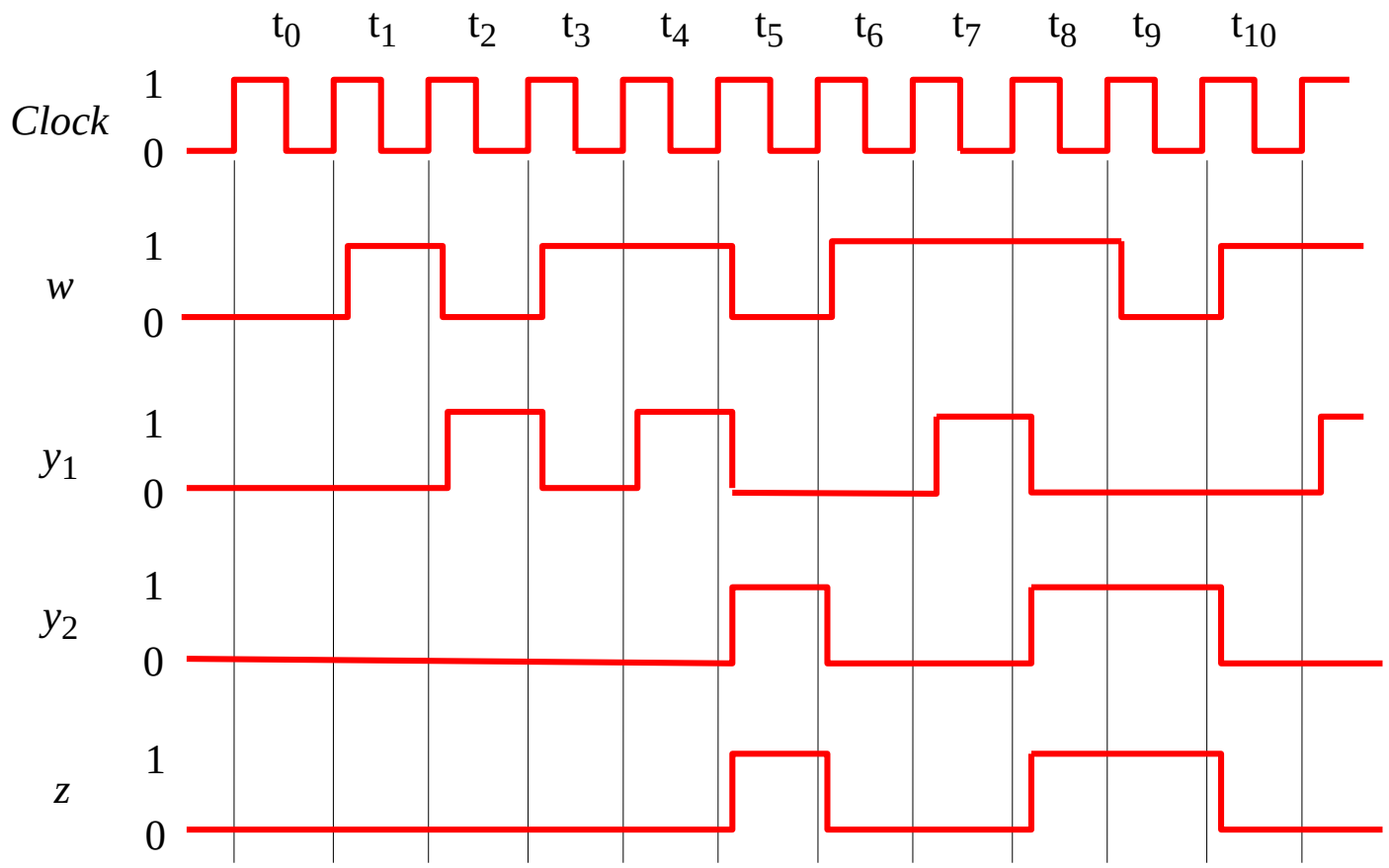


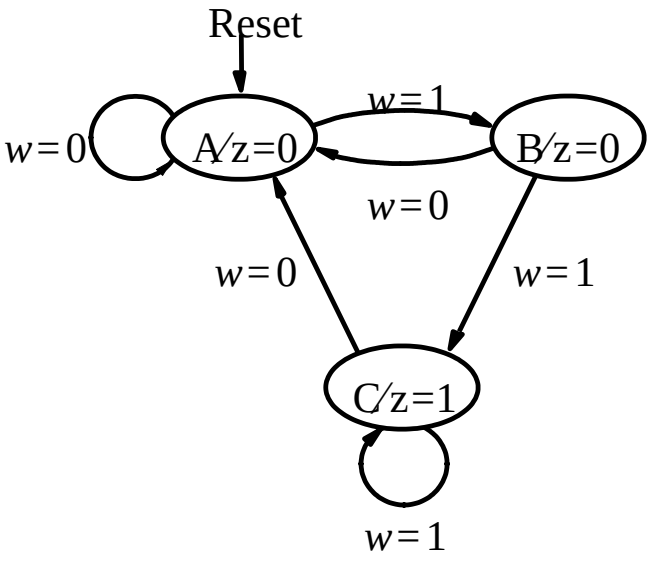
Figure 6.9. Timing diagram for the speed controller.

Design steps:

1. Obtain the specification of the desired circuit.
2. Derive a state diagram.
3. Derive the corresponding state table.
4. Reduce the number of states if possible.
5. Decide on the number of state variables.
6. Choose the type of flip-flops to be used.
7. Derive the logic expressions needed to implement the circuit.

Are all state assignments equivalent?

No, they are not.



	Present state	Next state		Output
		$w = 0$	$w = 1$	
	y_2y_1	Y_2Y_1	Y_2Y_1	z
A	00	00	01	0
B	01	00	11	0
C	11	00	11	1
	10	*dd*	*dd*	*d*

State C is 11 instead of 10.

Figure 6.16. Improved state assignment for the speed controller.

	Present state	Next state		Output
		$w = 0$	$w = 1$	
	y_2y_1	Y_2Y_1	Y_2Y_1	z
A	00	00	01	0
B	01	00	11	0
C	11	00	11	1
	10	*dd*	*dd*	*d*

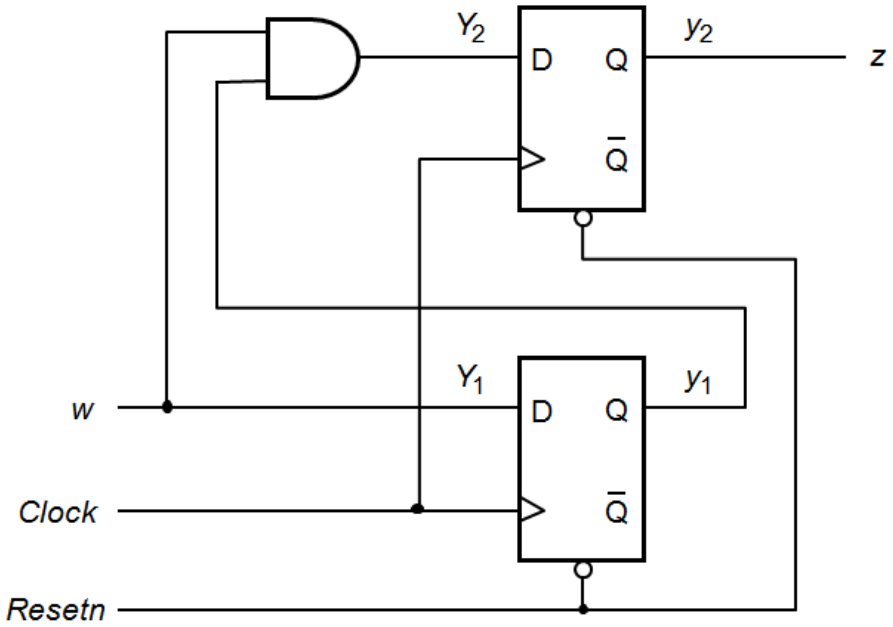
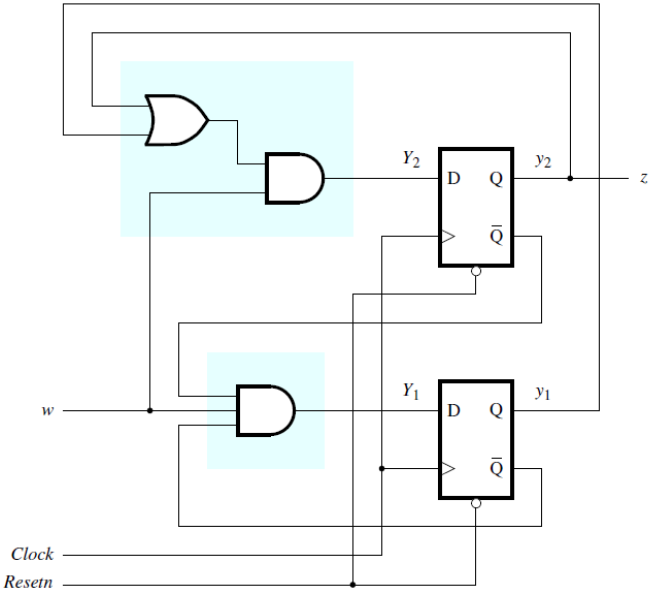
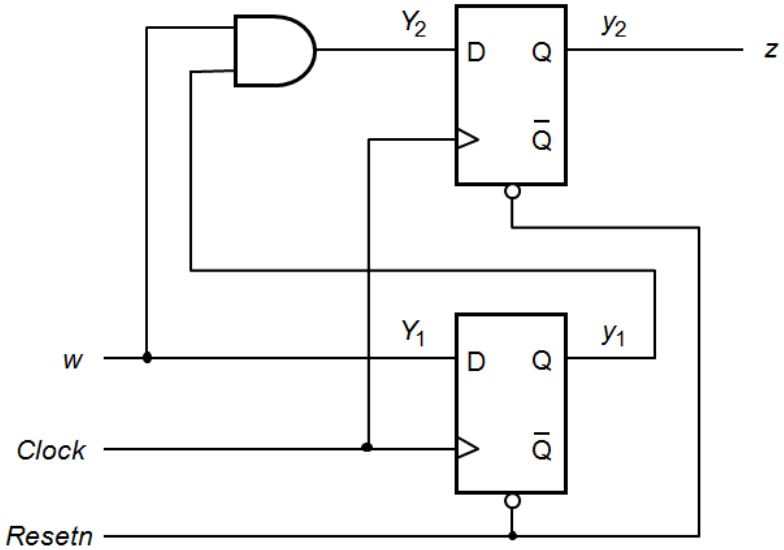


Figure 6.17. Final circuit for the improved state assignment for the speed controller.

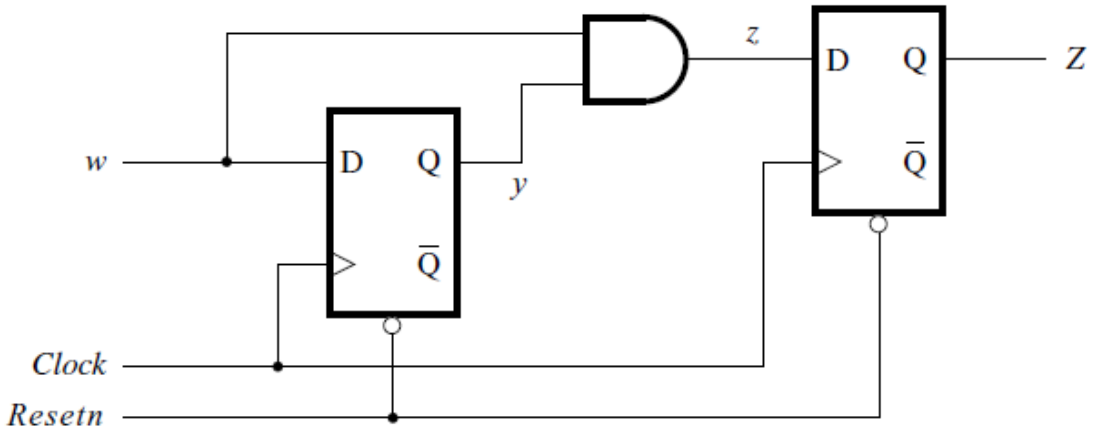
	Present state	Next state		Output
		$w = 0$	$w = 1$	
	y_2y_1	Y_2Y_1	Y_2Y_1	z
A	00	00	01	0
B	01	00	10	0
C	10	00	10	1
	11	*dd*	*dd*	*d*

	Present state	Next state		Output
		$w = 0$	$w = 1$	
	y_2y_1	Y_2Y_1	Y_2Y_1	z
A	00	00	01	0
B	01	00	11	0
C	11	00	11	1
	10	*dd*	*dd*	*d*





Original versus improved state assignment for the speed controller.



(a) Circuit

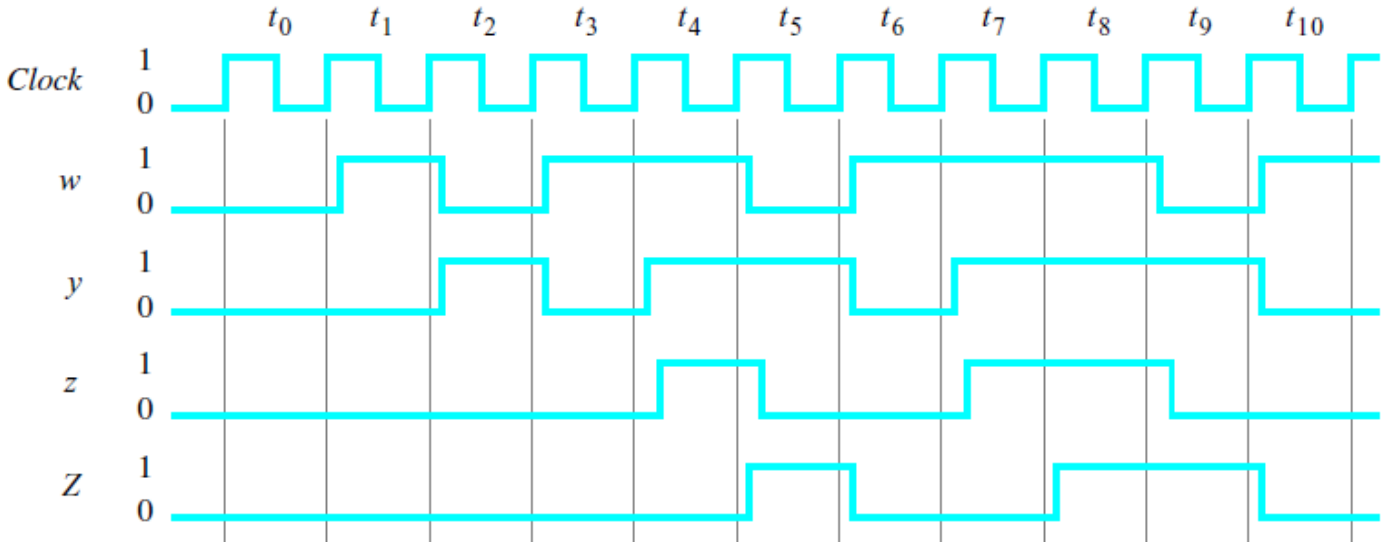


Figure 6.27. Improved speed controller is still a Moore design.

```
module simple1 ( input clock,
      reset, w, output z );
   reg [ 2:1 ] y, Y;
   parameter [ 2:1 ] A = 2'b00,
      B = 2'b01, C = 2'b10;

   // Define the next state
   always @( w, y )
      case ( y )
         A: if ( w ) Y = B;
            else Y = A;
         B: if ( w ) Y = C;
            else Y = A;
         C: if ( w ) Y = C;
            else Y = A;
         default: Y = 2'bxx;
      endcase

   // Define the sequential block
   always @( posedge reset,
         posedge clock)
      if ( reset ) y <= A;
      else y <= Y;

   // Define output
   assign z = y == C;

endmodule
```

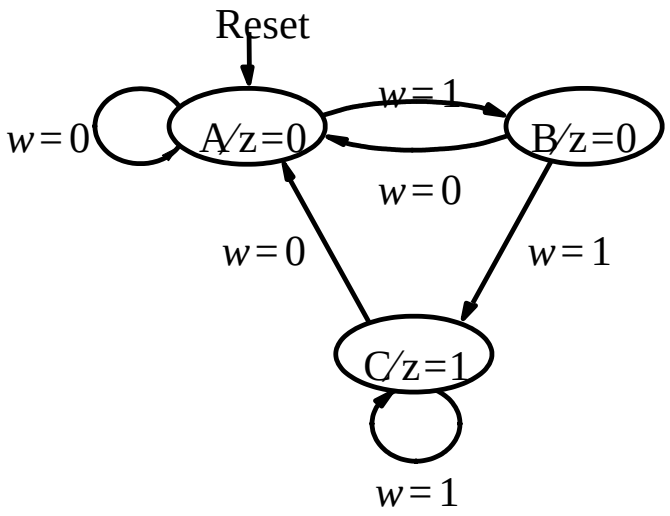


Figure 6.29. Verilog code for the speed controller.

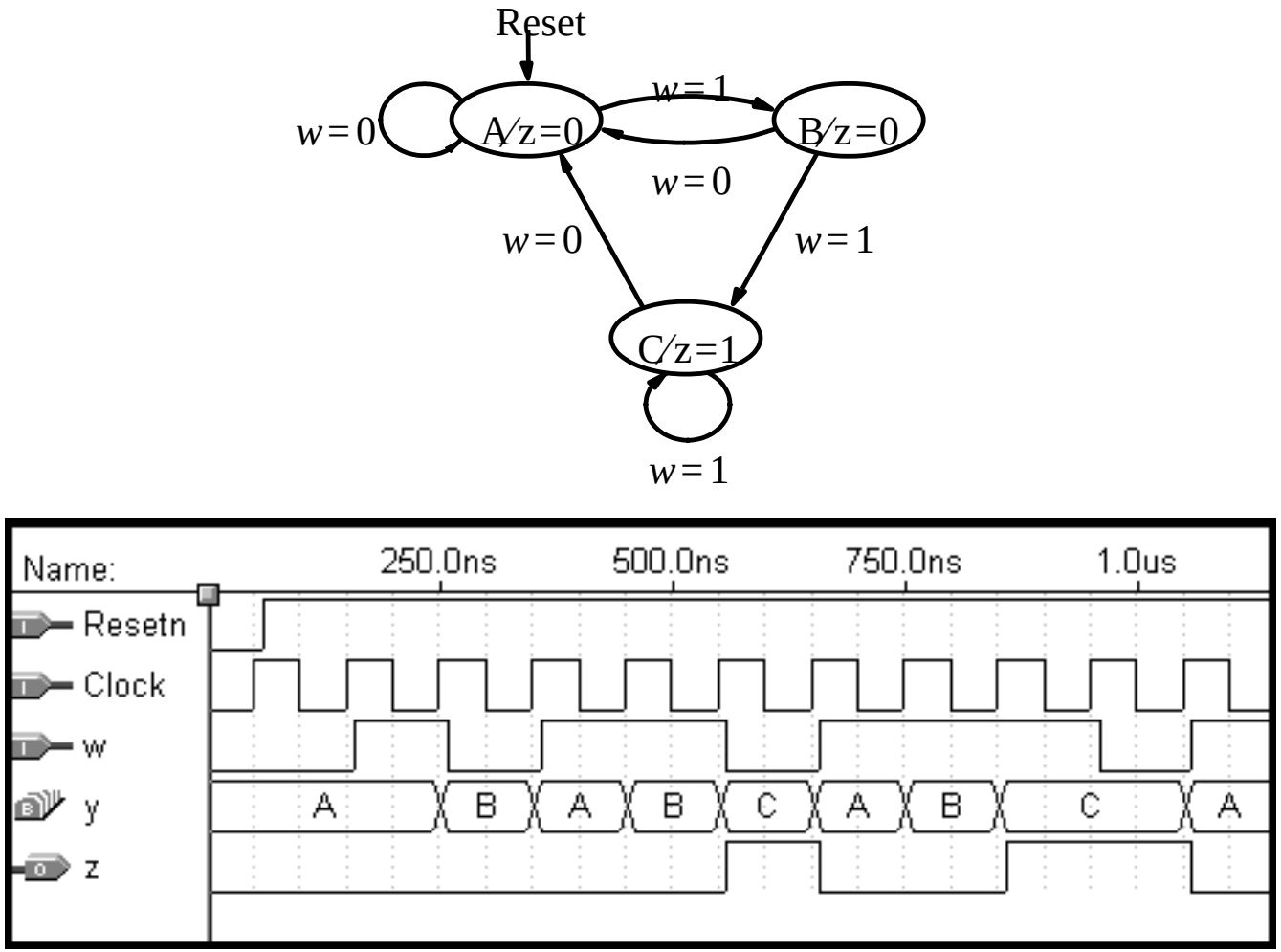
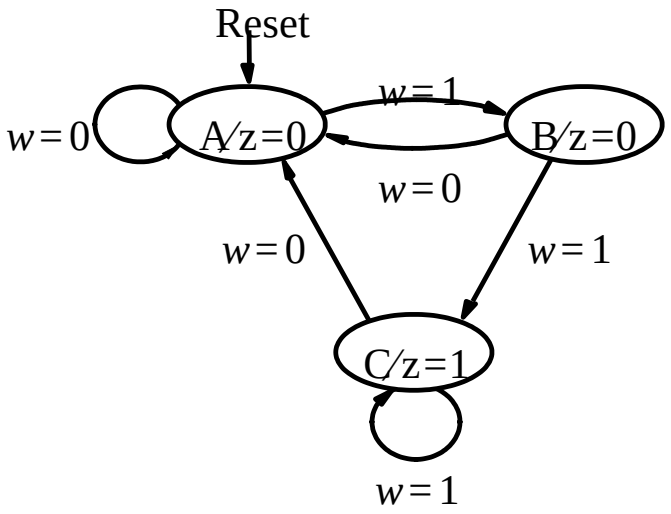


Figure 6.32. Simulation results for the speed controller.

```
module simple2 ( input clock,
      reset, w, output reg z );

   reg [ 2:1 ] y, Y;
   parameter [ 2:1 ] A = 2'b00,
      B = 2'b01, C = 2'b10;

   // Define the next state
   always @( w, y )
      begin
      case ( y )
         A: if ( w ) Y = B;
            else Y = A;
         B: if ( w ) Y = C;
            else Y = A;
         C: if ( w ) Y = C;
            else Y = A;
         default: Y = 2'bxx;
      endcase
      z = y == C; // Define output
      end
```

```
   // Define the sequential block
      always @( posedge reset,
            posedge clock)
         if ( reset ) y <= A;
         else y <= Y;

endmodule
```



Output assignment moved to the always block.

Figure 6.33

```
module simple3 ( input clock,
      reset, w, output z );

   reg [ 2:1 ] y, Y;
   parameter [ 2:1 ] A = 2'b00,
      B = 2'b01, C = 2'b10;

   // Define the next state
   always @( posedge reset,
         posedge clock)
      if ( reset )
         y <= A;
      else
         case ( y )
            A: if ( w ) y <= B;
               else y <= A;
            B: if ( w ) y <= C;
               else y <= A;
            C: if ( w ) y <= C;
               else y <= A;
            default: y <= 2'bxx;
         endcase

   // Define output
      assign z = y == C;

endmodule
```

Next state and reset calculations moved into the always block.

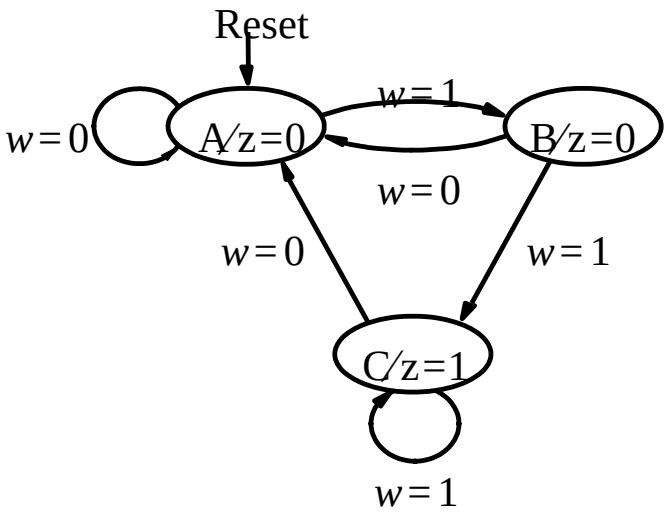
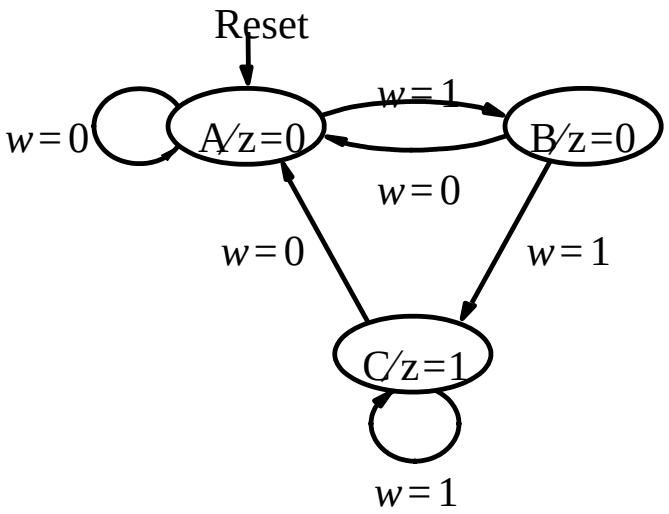
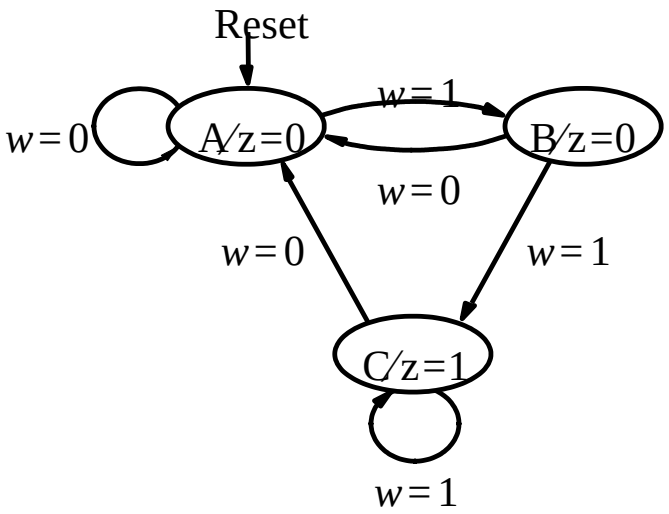


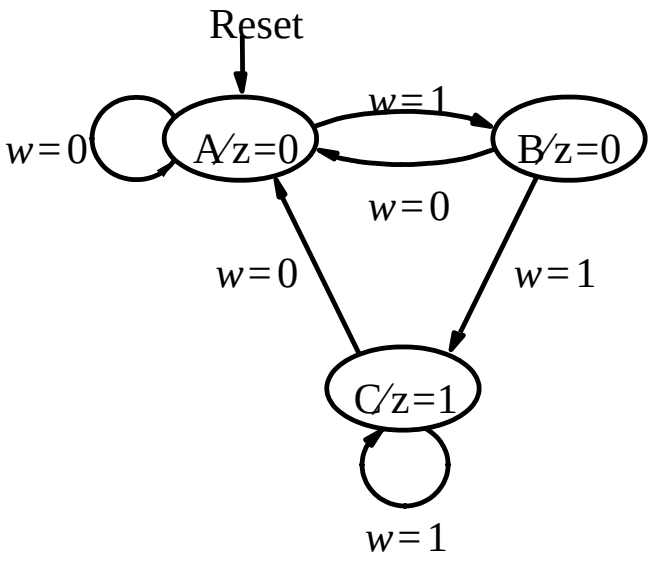
Figure 6.34

```
module simple4 ( input clock,
      reset, w, output reg z );

   reg [ 2:1 ] y, Y;
   parameter [ 2:1 ] A = 2'b00,
      B = 2'b01, C = 2'b10;

   // Define the next state
   always @( posedge reset,
         posedge clock)
      casex ( { reset, w, y } )
         3'b1xxx: y <= A;
         3'b00xx: y <= A;
         { 2'b01, A }: y <= B;
         { 2'b01, B }: y <= C;
         { 2'b01, C }: y <= C;
         default: y <= 2'bxx;
      endcase

   // Define output
     assign z = y == C;

endmodule
```



```
module simple5 ( input clock,
      reset, w, output reg z );

   reg [ 2:1 ] y, Y;
   parameter [ 2:1 ] A = 2'b00,
      B = 2'b01, C = 2'b10,
      x = 2'bxx;

   // Define the next state
   always @( posedge reset,
         posedge clock )
      casex ( { reset, w, y } )
         { 2'b1x, x }:  y <= A;
         { 2'b00, x }:  y <= A;
         { 2'b01, A }:  y <= B;
         { 2'b01, B }:  y <= C;
         { 2'b01, C }:  y <= C;
         default:       y <= x;
      endcase

   // Define output
     assign z = y == C;

endmodule
```





	Present state	Nextstate		Output
		$w = 0$	$w = 1$	
	$y_3y_2y_1$	$Y_3Y_2Y_1$	$Y_3Y_2Y_1$	z
A	001	001	010	0
B	010	001	100	0
C	100	001	100	1

Figure 6.20. Anther alternative: A one-hot state assignment for the speed controller.

A Mealy alternative

Clockcycle:	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
w:	0	1	0	1	1	0	1	1	1	0	1
z:	0	0	0	0	0	1	0	0	1	1	0

The original Moore design

Clock cycle:	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
w:	0	1	0	1	1	0	1	1	1	0	1
z:	0	0	0	0	1	0	0	1	1	0	0

The alternative Mealy design

Figure 6.22. Sequence detector for an alternate speed controller. This one recognizes w = 1 on two cycles immediately.

Clock cycle:	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
w:	0	1	0	1	1	0	1	1	1	0	1
z:	0	0	0	0	1	0	0	1	1	0	0

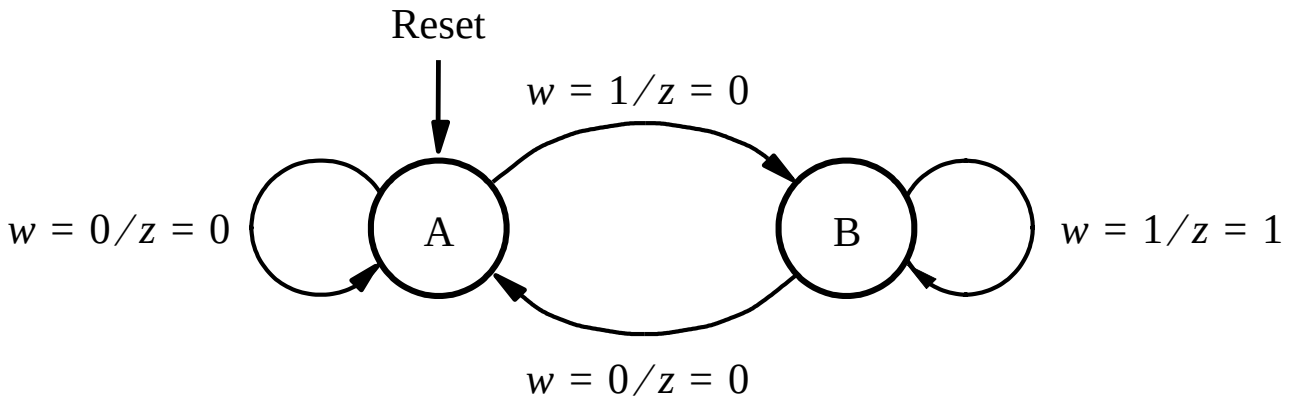
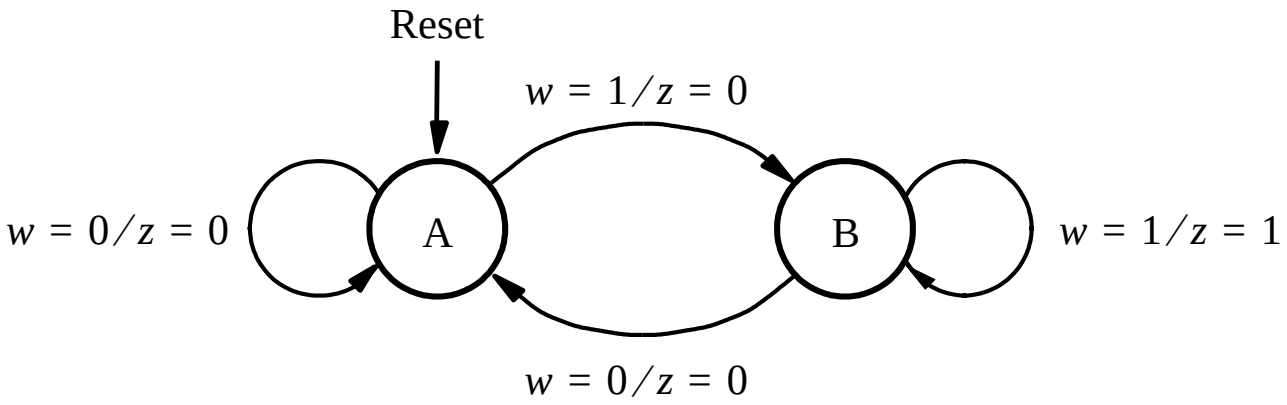


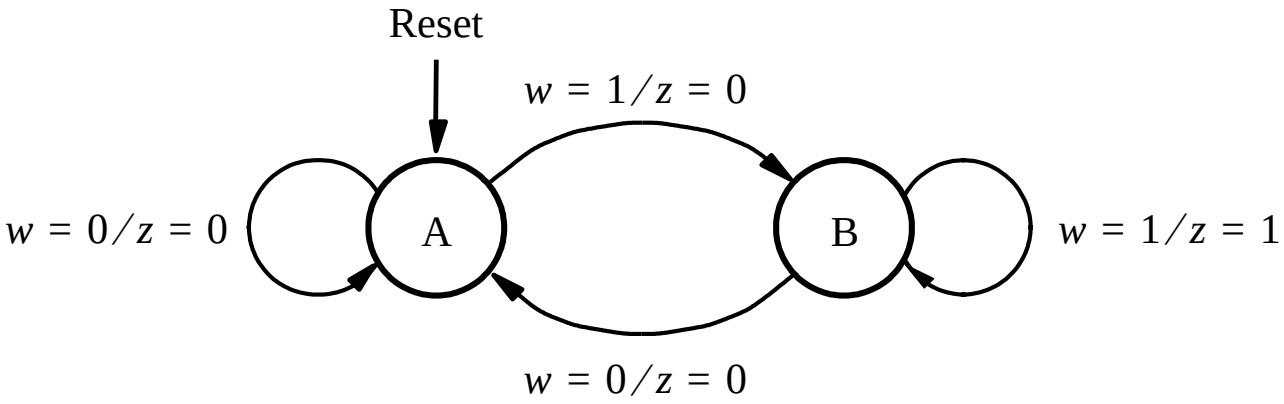
Figure 6.23. State diagram of the alternate speed controller.

Clock cycle:	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
w:	0	1	0	1	1	0	1	1	1	0	1
z:	0	0	0	0	1	0	0	1	1	0	0



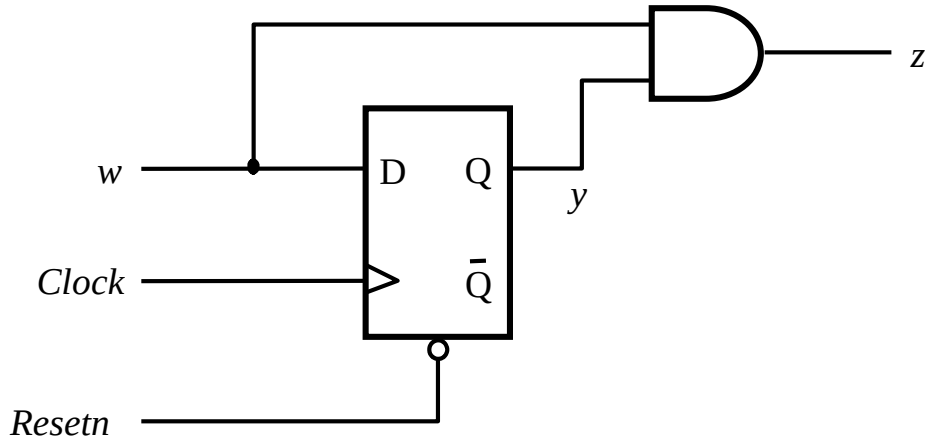
Present state	Next state		Output *z*	
	w = 0	*w* = 1	*w* = 0	*w* = 1
A	A	B	0	0
B	A	B	0	1

Figure 6.24. State table for the alternate speed controller.

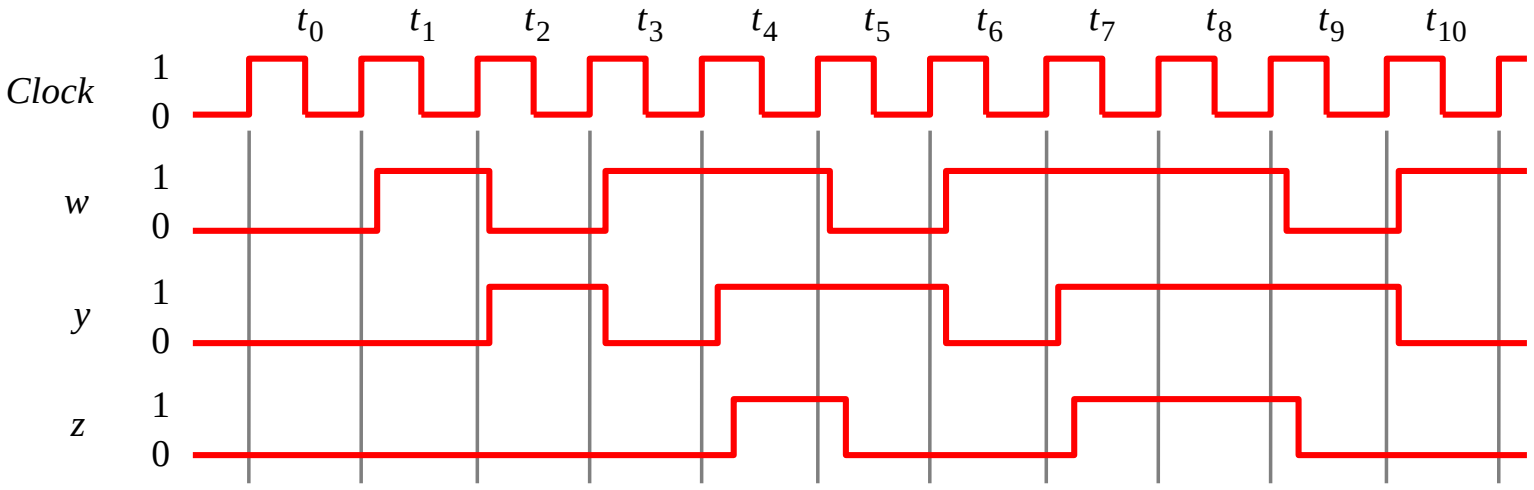


	Present state	Next state		Output	
		$w = 0$	$w = 1$	$w = 0$	$w = 1$
	y	Y	Y	z	z
A	0	0	1	0	0
B	1	0	1	0	1

Figure 6.25. State-assigned table for an alternate Mealy speed controller.

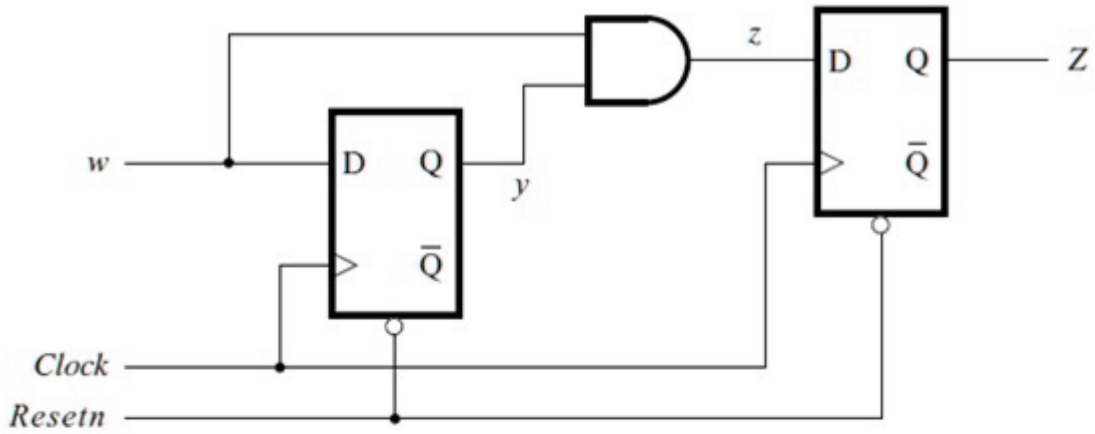


(a) Circuit



(b) Timing diagram

Figure 6.26. The alternate Mealy speed controller.



(a) Circuit

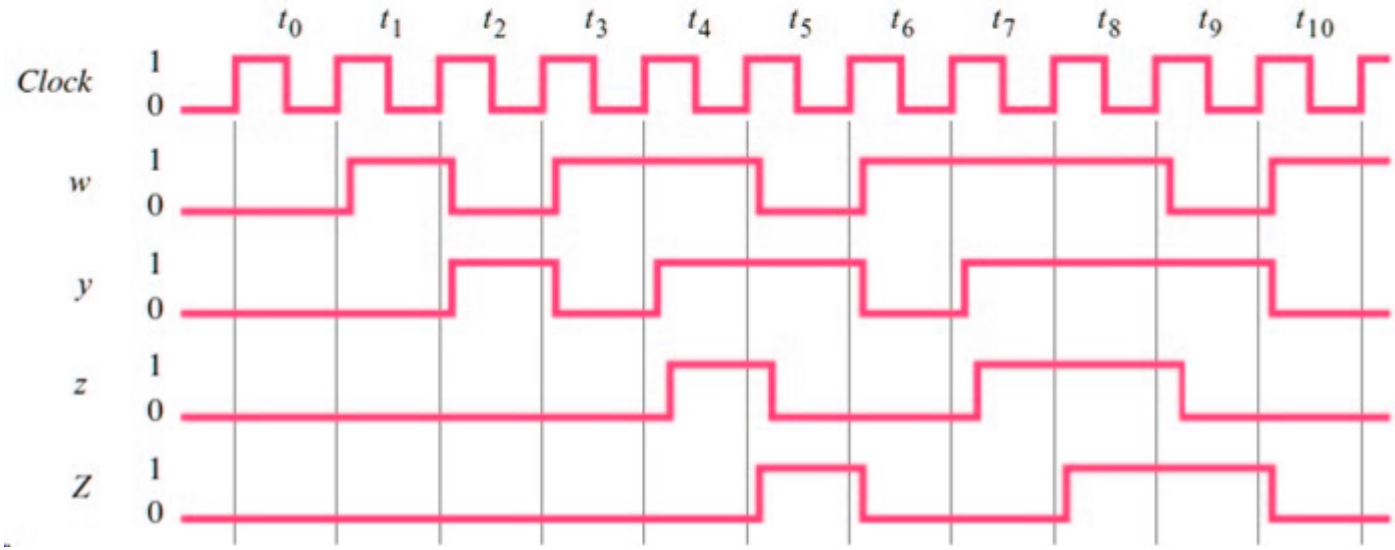
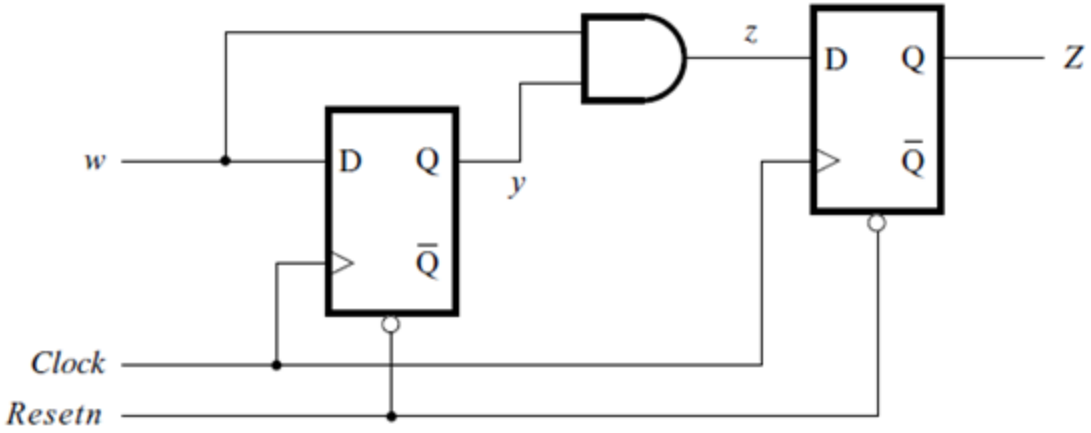
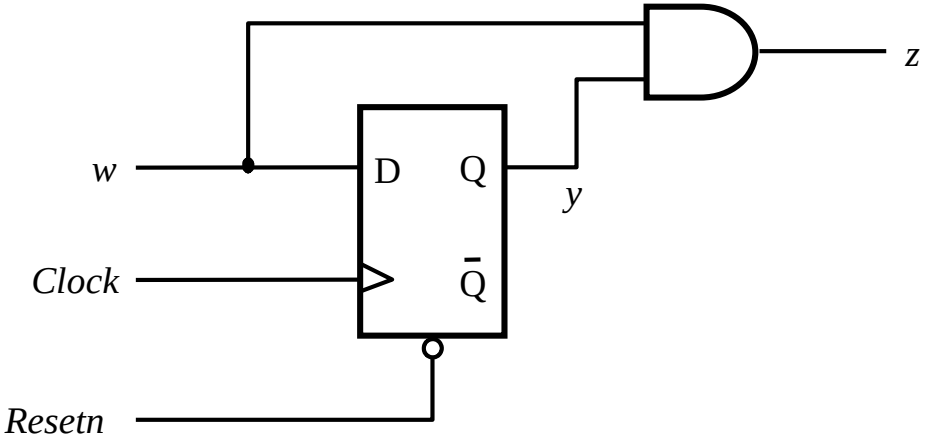


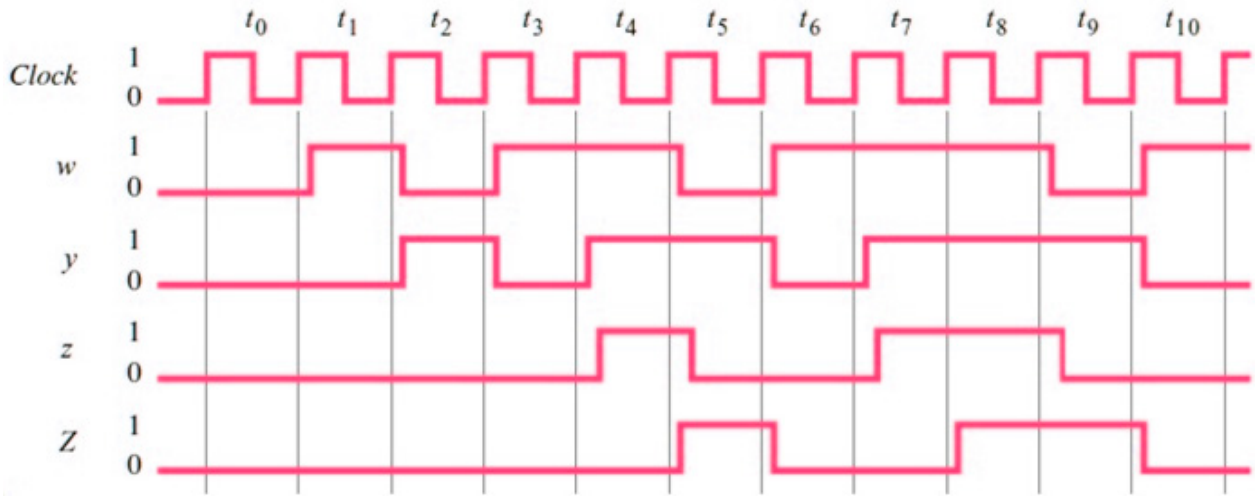
Figure 6.27. The original Moore speed controller.



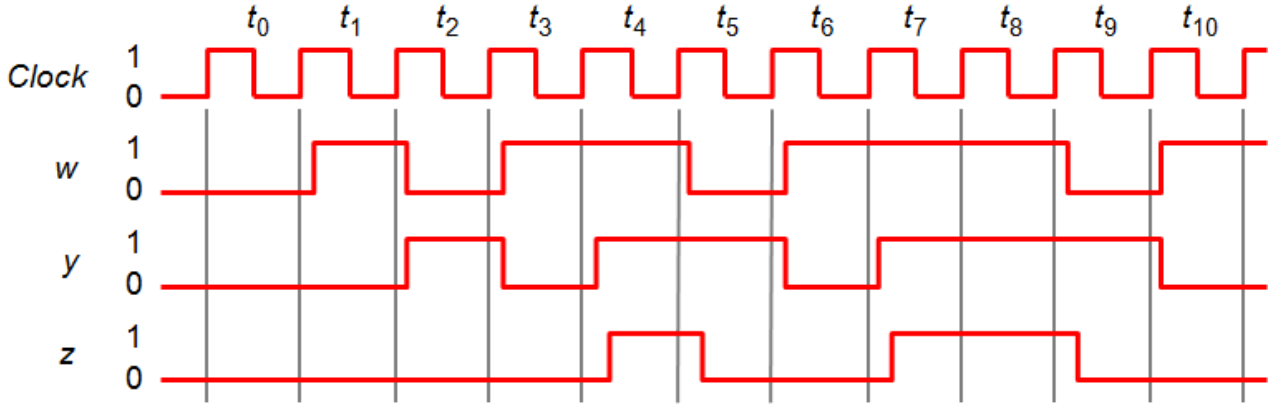
The original Moore design



The alternative Mealy design



The original Moore design



The alternative Mealy design

```
module mealy1( input clock, reset, w,
      output reg z );

   reg y, Y;
   parameter A = 0, B = 1;

   // Define the next state and outputs
   always @( * )
      case ( y )
         A: if ( w )
               begin
               z = 0;
               Y = B;
               end
            else
               begin
               z = 0;
               Y = A;
               end
         B: if ( w )
               begin
               z = 1;
               Y = B;
               end
            else
               begin
               z = 0;
               Y = A;
               end
      endcase

   // Define the sequential block
   always @( posedge reset,
         posedge clock )
      if ( reset )
         y <= A;
      else
         y <= Y;

endmodule
```

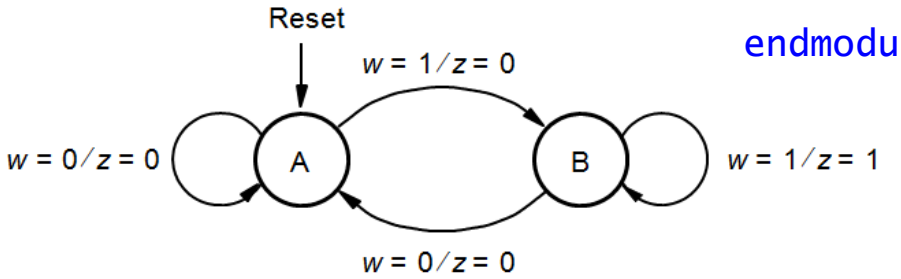
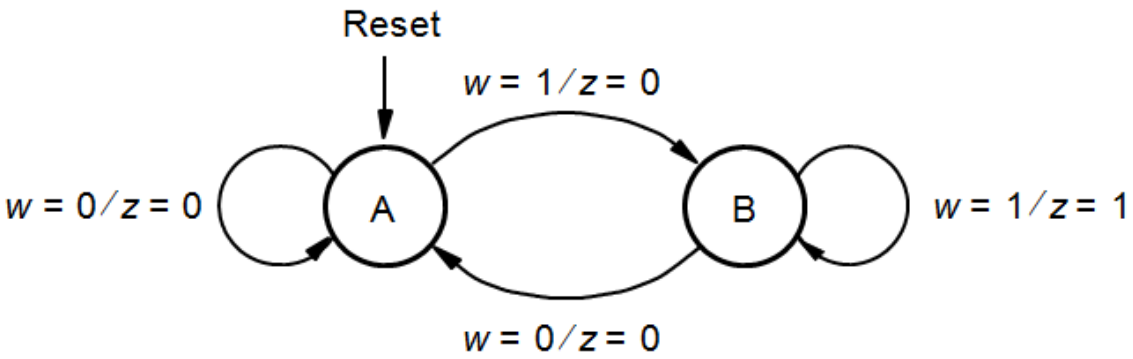


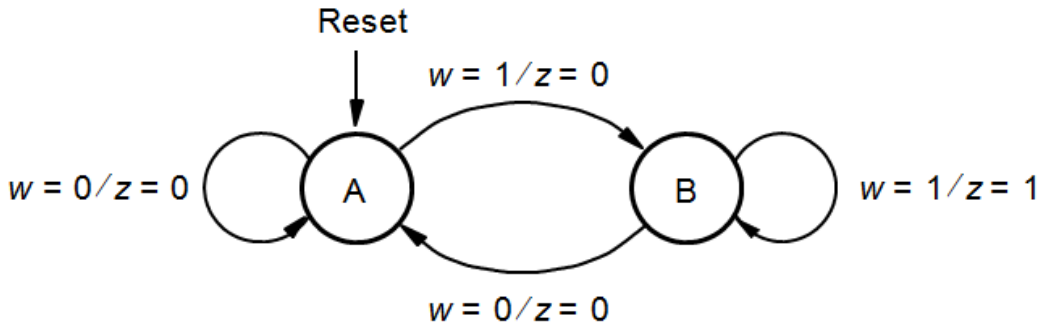
Figure 6.36

```
module mealy2( input clock, reset, w,
      output reg z );

   reg y, Y;
   parameter A = 0, B = 1;

   // Define the next state and outputs
   always @( * )
      if ( ~w )
         begin
         z = 0;
         Y = A;
         end
      else
         case ( y )
            A: begin
               z = 0;
               Y = B;
               end
            B: begin
               z = 1;
               Y = B;
               end
         endcase
```

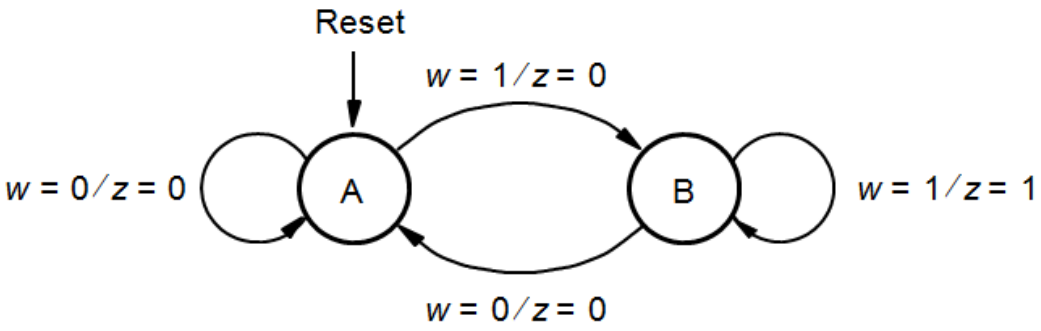
```
   // Define the sequential block
     always @( posedge reset,
           posedge clock )
        if ( reset )
           y <= A;
        else
           y <= Y;

endmodule
```



```
module mealy3( input clock, reset, w,
      output z );

   reg y, Y;
   parameter A = 0, B = 1;

   // Define the next state
   always @( * )
      if ( ~w )
         Y = A;
      else
         case ( y )
            A: Y = B;
            B: Y = B;
         endcase

   assign z = y == B && w;

   // Define the sequential block
   always @( posedge reset,
         posedge clock )
      y <= reset ? A : Y;

endmodule
```



```
module mealy4( input clock, reset, w,
      output z );

   reg y;
   parameter A = 0, B = 1;

   always @( posedge reset,
         posedge clock )
      y <= ( reset | ~w ) ? A : B;

   assign z = y == B && w;

endmodule
```



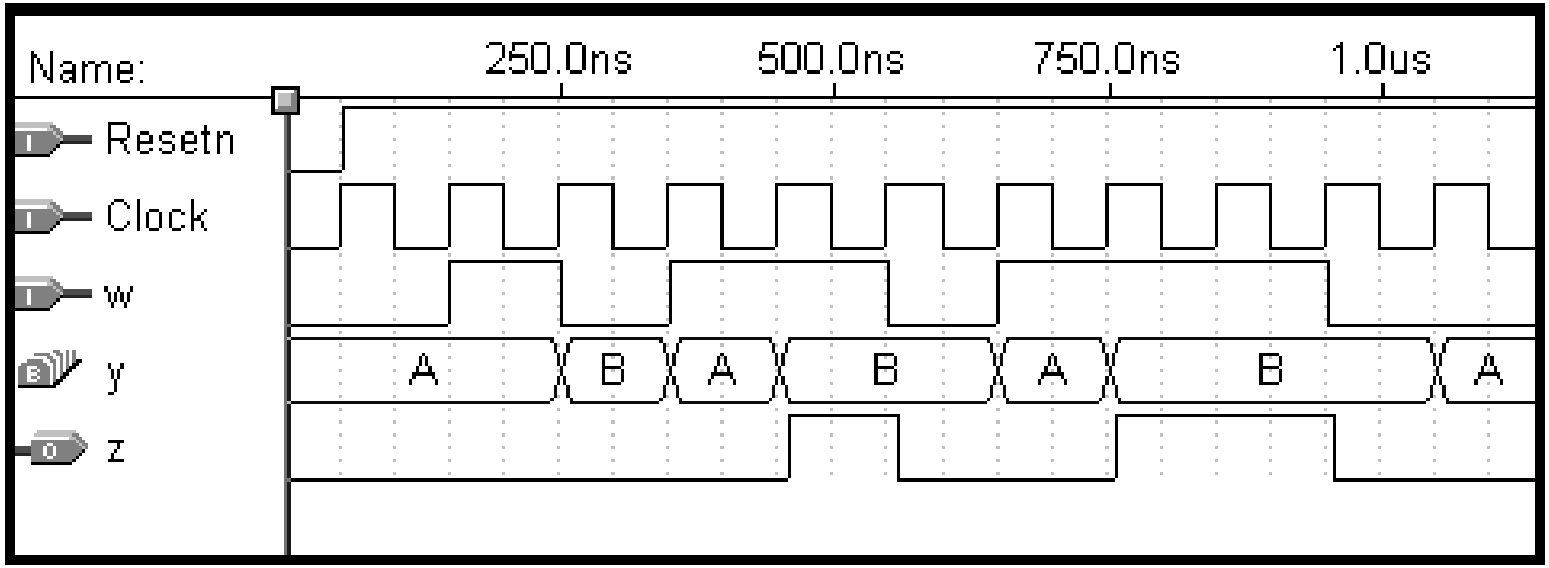


Figure 6.37. Simulation results for the Mealy machine.

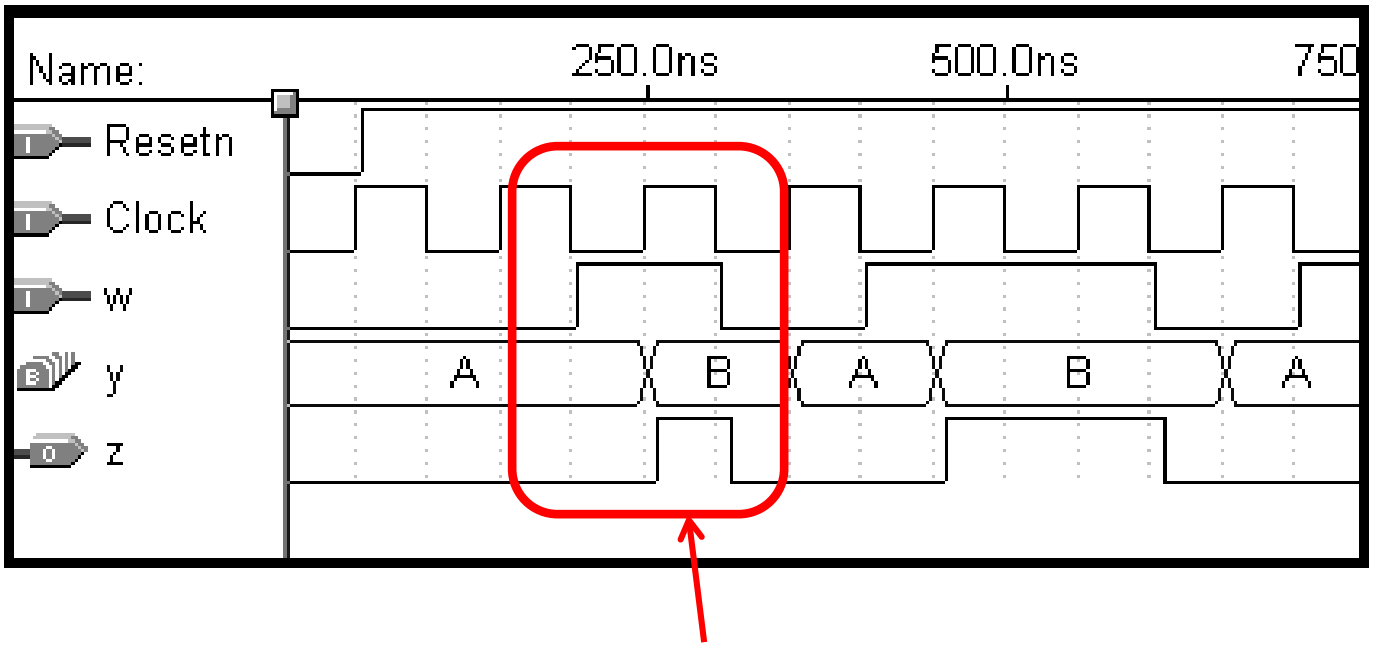


Figure 6.38. Potential problem with asynchronous inputs to the Mealy speed controller.