BEE 271 Digital circuits and systems

Spring 2017

Lecture 13: State machines

Nicole Hamilton

https://faculty.washington.edu/kd1uj

Today's topics

1. A BCD counter
2. State machines
3. State equivalence and optimization
4. Analysis of existing circuits

A 6-digit BCD counter for the DE1-SoC board

```verilog
module SevenSegment(
      input [ 3:0 ] hexDigit,
      input enable,
      output [ 6:0 ] segments );

   // Seven segment decoder as a case statement.
   reg [ 6:0 ] s;
   always @( * )
      case ( hexDigit )
         0: s = 'b011_1111;
         1: s = 'b000_0110;
         2: s = 'b101_1011;
         3: s = 'b100_1111;
         4: s = 'b110_0110;
         5: s = 'b110_1101;
         6: s = 'b111_1101;
         7: s = 'b000_0111;
         8: s = 'b111_1111;
         9: s = 'b110_1111;
       'hA: s = 'b111_0111;
       'hB: s = 'b111_1100;
       'hC: s = 'b011_1001;
       'hD: s = 'b101_1110;
       'hE: s = 'b111_1001;
       'hF: s = 'b111_0001;
      endcase

   // Invert the output for active low on
   // the DE1-SoC board.
   assign segments = ~( enable ? s : 0 );

endmodule
```

```
module Digit1( input clock, reset, enable,
      output reg [ 3:0 ] digit = 0,
      output rollover );

   parameter limit = 9;
   assign rollover = ( digit == limit ) & enable;
   always @( posedge clock )
      if ( reset )
         digit <= 0;
      else
         if ( enable )
            digit <= rollover ? 0 : digit + 1;

endmodule
```

```
module Digit2( input clock, reset, enable,
      output reg [ 3:0 ] digit = 0,
      output rollover );

   wire isNine = digit == 9;
   assign rollover = isNine & enable;

   always @( posedge clock )
      casex ( { reset, enable, isNine } )
         'b1xx: digit <= 0;
         'b00x: digit <= digit;
         'b010: digit <= digit + 1;
         'b011: digit <= 0;
      endcase

endmodule
```

```
module ClockDivider1( input clock, reset,
      output reg Q );

   // This divider toggles at the desired freq and
   // could be used as a clock but will cause skew.

   reg [ 31:0 ] counter;
   parameter desiredFrequency = 10,
      clockDivisor = 50_000_000 / desiredFrequency;

   always @( posedge clock )
      if ( counter == 0 )
         begin
         counter <= clockDivisor;
         Q <= ~Q;
         end
      else
         counter <= counter - 1;

endmodule
```

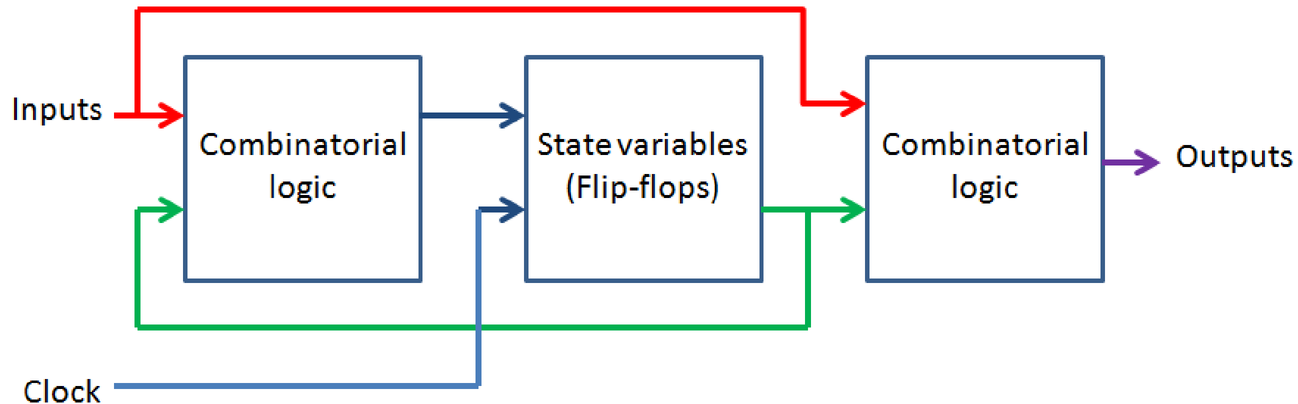
```
module ClockDivider2( input clock, reset,
      output enable );

   // Better:  This divider gives a one-tick enable
   // at the desired frequency to avoid clock skew.

   parameter clockFreq = 50_000_000,
      desiredFreq = 10,
      divisor = clockFreq / desiredFreq;

   reg [ 31:0 ] counter;
   assign enable = counter == 0;

   always @( posedge clock )
      counter <= reset | ( counter == 0 ) ?
         divisor : counter - 1;

endmodule
```

```
module BCDCounter1(
      input CLOCK_50,
      output [ 6:0 ] HEX0, HEX1, HEX2,
                     HEX3, HEX4, HEX5,
      input  [ 3:0 ] KEY,
      output [ 9:0 ] LEDR,
      input  [ 9:0 ] SW );

   wire [ 23:0 ] BCD;

   wire reset, stop, start, clear, countEnable;
   assign { reset, stop, start, clear } = ~KEY;
   reg run;

   always @( posedge CLOCK_50 )
      casex ( { reset, stop, start } )
         3'b1xx: run <= 1;
         3'b01x: run <= 0;
         3'b001: run <= 1;
      endcase

   ClockDivider2 #( .desiredFreq( 100 ) ) divider
       ( CLOCK_50, reset, countEnable );

   SixDigits1 digits( CLOCK_50, reset | clear,
      countEnable & run, BCD );

   // Each digit is enabled if it or a higher-order
   // digit is non-zero.

   SevenSegment s0( BCD[  3:0  ], 1,                HEX0 );
   SevenSegment s1( BCD[  7:4  ], |BCD[ 23:4  ], HEX1 );
   SevenSegment s2( BCD[ 11:8  ], |BCD[ 23:8  ], HEX2 );
   SevenSegment s3( BCD[ 15:12 ], |BCD[ 23:12 ], HEX3 );
   SevenSegment s4( BCD[ 19:16 ], |BCD[ 23:16 ], HEX4 );
   SevenSegment s5( BCD[ 23:20 ], |BCD[ 23:20 ], HEX5 );

endmodule
```

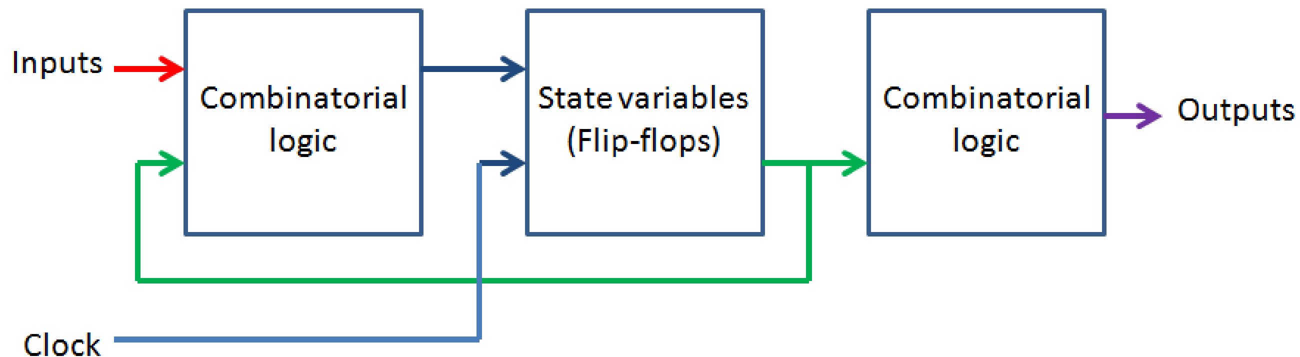
```
module BCDCounter2(
      input CLOCK_50,
      output [ 6:0 ] HEX0, HEX1, HEX2,
                     HEX3, HEX4, HEX5,
      input  [ 3:0 ] KEY,
      output [ 9:0 ] LEDR,
      input  [ 9:0 ] SW );

   wire [ 3:0 ] BCD[ 5:0 ];
   wire [ 5:0 ] enable;

   wire reset, stop, start, clear, countEnable;
   assign { reset, stop, start, clear } = ~KEY;

   reg run;

   always @( posedge CLOCK_50 )
      casex ( { reset, stop, start } )
         3'b1xx: run <= 1;
         3'b01x: run <= 0;
         3'b001: run <= 1;
      endcase

   ClockDivider2 #( .desiredFreq( 100 ) ) divider
       ( CLOCK_50, reset, countEnable );

   SixDigits2 digits( CLOCK_50, reset | clear,
      countEnable & run, BCD );

   // Each digit is enabled if it or a higher-order
   // digit is non-zero.

   assign enable = { |BCD[ 5 ],
                     enable[ 5 ] | |BCD[ 4 ],
                     enable[ 4 ] | |BCD[ 3 ],
                     enable[ 3 ] | |BCD[ 2 ],
                     enable[ 2 ] | |BCD[ 1 ],
                     1 };

   SevenSegment s0( BCD[ 0 ], enable[ 0 ], HEX0 );
   SevenSegment s1( BCD[ 1 ], enable[ 1 ], HEX1 );
   SevenSegment s2( BCD[ 2 ], enable[ 2 ], HEX2 );
   SevenSegment s3( BCD[ 3 ], enable[ 3 ], HEX3 );
   SevenSegment s4( BCD[ 4 ], enable[ 4 ], HEX4 );
   SevenSegment s5( BCD[ 5 ], enable[ 5 ], HEX5 );
endmodule
```

```
module BCDCounter3(
      input CLOCK_50,
      output [ 6:0 ] HEX0, HEX1, HEX2,
                     HEX3, HEX4, HEX5,
      input  [ 3:0 ] KEY,
      output [ 9:0 ] LEDR,
      input  [ 9:0 ] SW );

   wire [ 3:0 ] BCD[ 5:0 ];
   reg  [ 5:0 ] enable;

   wire reset, stop, start, clear, countEnable;
   assign { reset, stop, start, clear } = ~KEY;

   reg run;

   always @( posedge CLOCK_50 )
      casex ( { reset, stop, start } )
         3'b1xx: run <= 1;
         3'b01x: run <= 0;
         3'b001: run <= 1;
      endcase

   ClockDivider2 #( .desiredFreq( 100 ) ) divider
       ( CLOCK_50, reset, countEnable );

   SixDigits2 digits( CLOCK_50, reset | clear,
      countEnable & run, BCD );

   // Each digit is enabled if it or a higher-order
   // digit is non-zero.

   always @( * )
      begin
      integer i;
      enable[ 5 ] = |BCD[ 5 ];
      for ( i = 4; i; i-- )
         enable[ i ] = enable[ i + 1 ] | ( |BCD[ i ] );
      enable[ 0 ] = 1;
      end

   SevenSegment s0( BCD[ 0 ], enable[ 0 ], HEX0 );
   SevenSegment s1( BCD[ 1 ], enable[ 1 ], HEX1 );
   SevenSegment s2( BCD[ 2 ], enable[ 2 ], HEX2 );
   SevenSegment s3( BCD[ 3 ], enable[ 3 ], HEX3 );
   SevenSegment s4( BCD[ 4 ], enable[ 4 ], HEX4 );
   SevenSegment s5( BCD[ 5 ], enable[ 5 ], HEX5 );

endmodule
```

State machines again



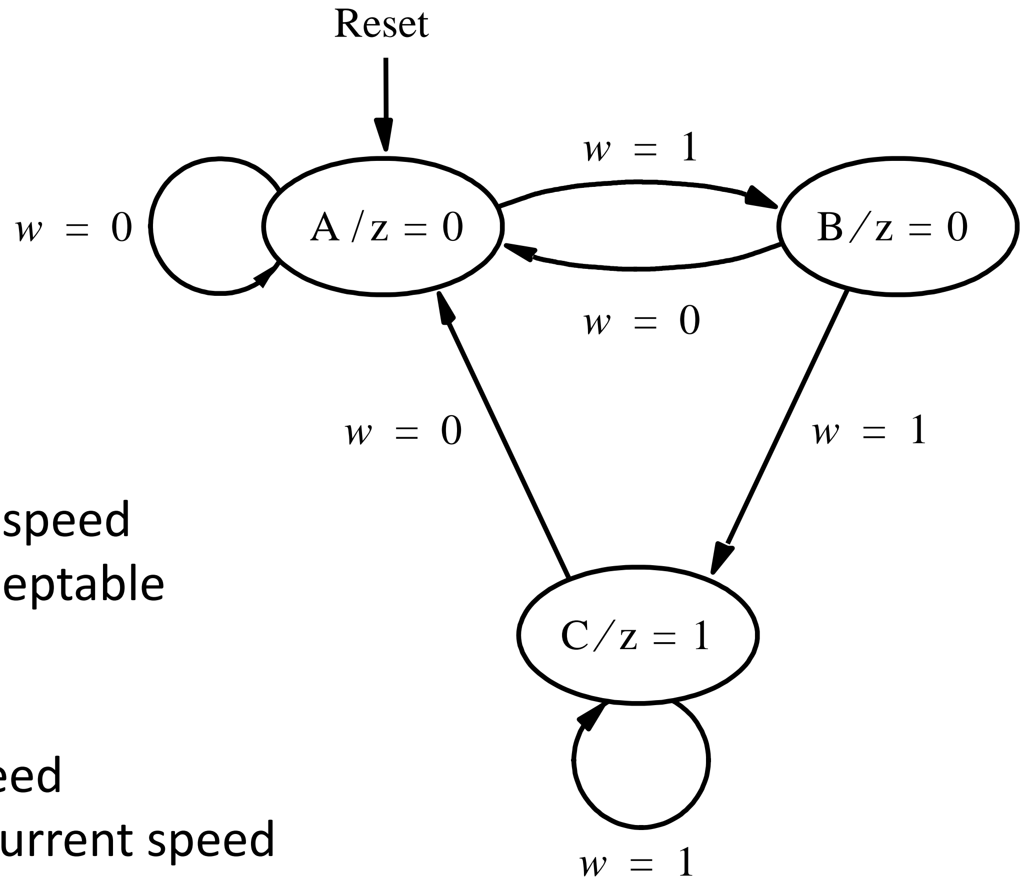
Mealy machines require fewer states but if the inputs change asynchronously, the outputs can change asynchronously as well.



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

Example: A simple speed control

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



Input:

w == 1 → excessive speed

w == 0 → speed acceptable

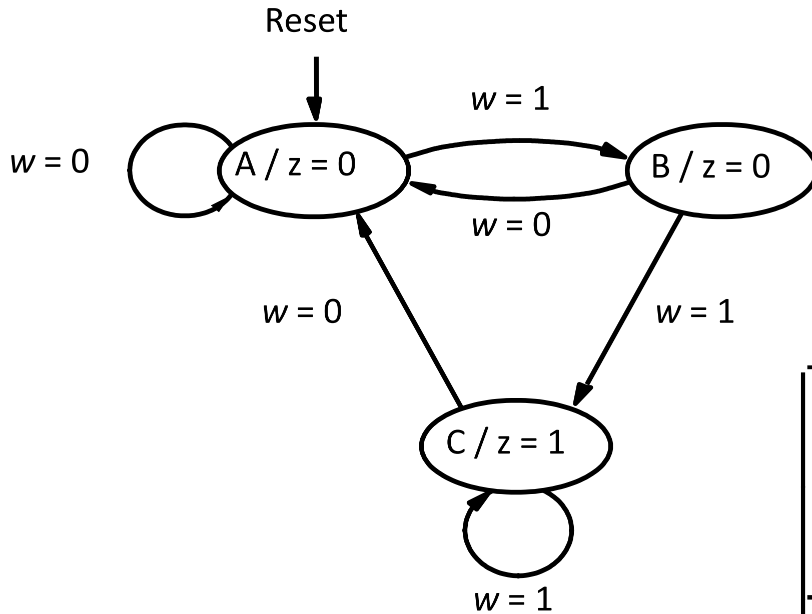
Output:

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.

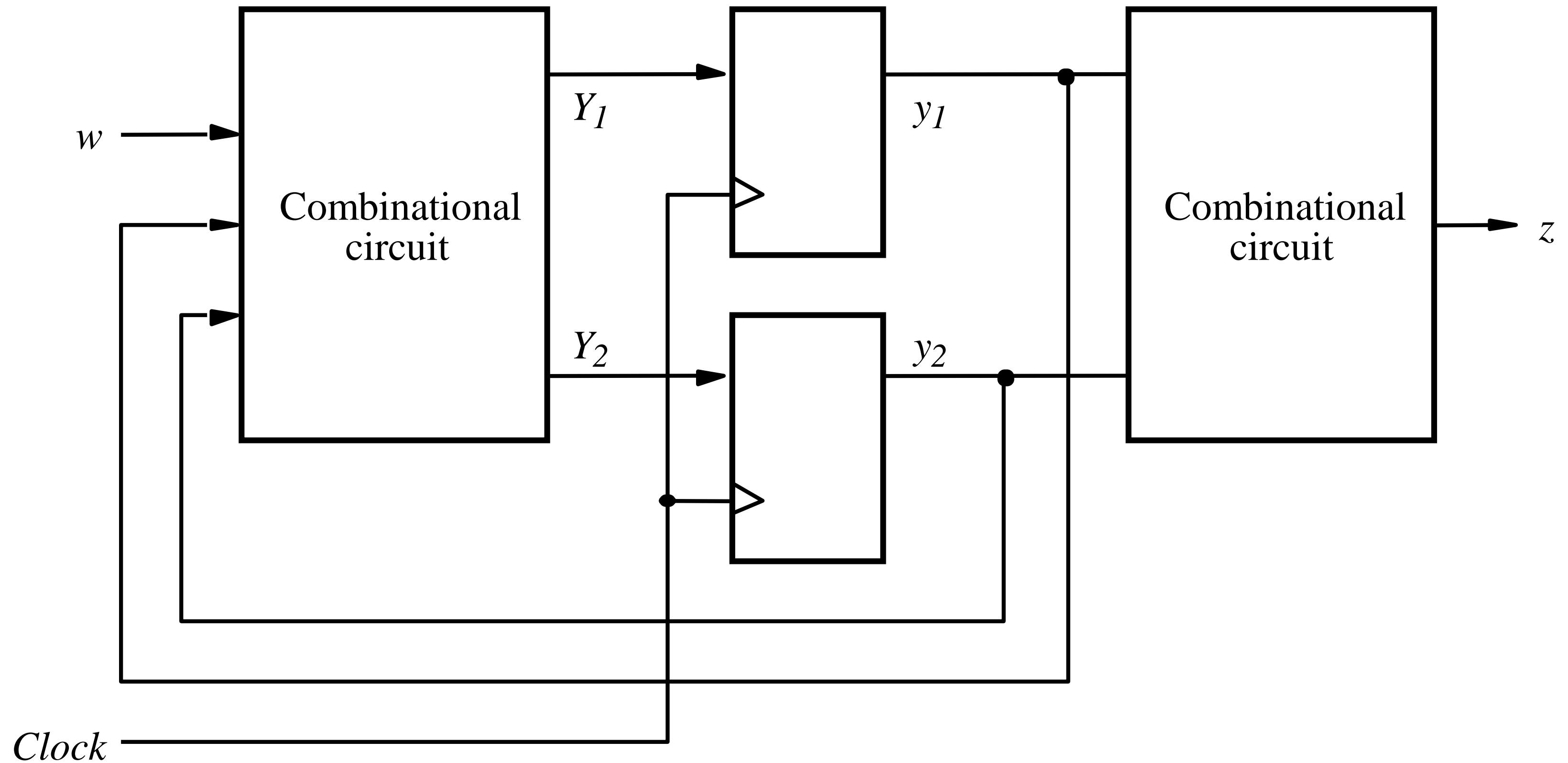
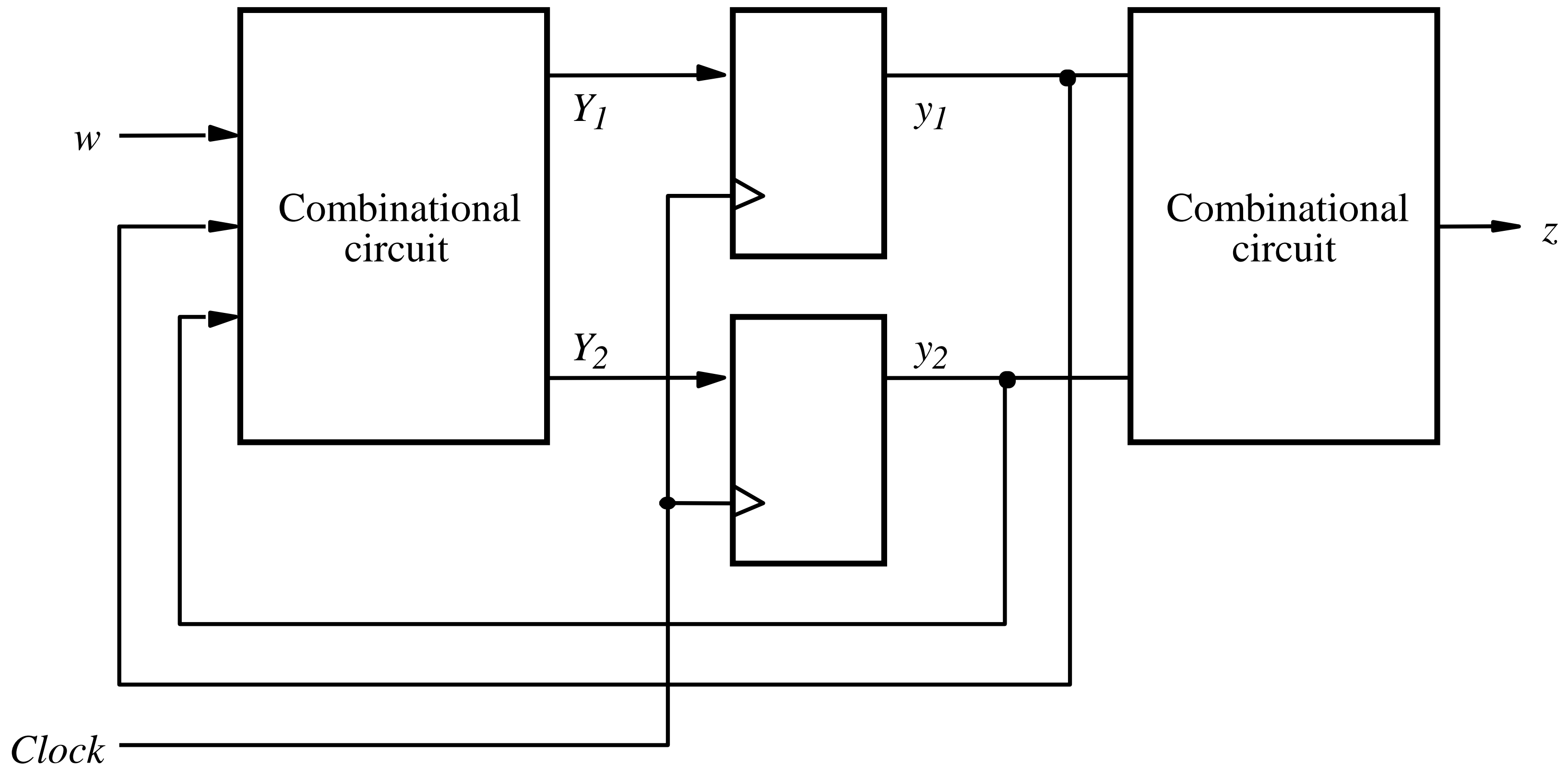


Figure 6.5. A generalized solution to the speed controller, with input *w*, output *z*, and two flip-flops for the three states.



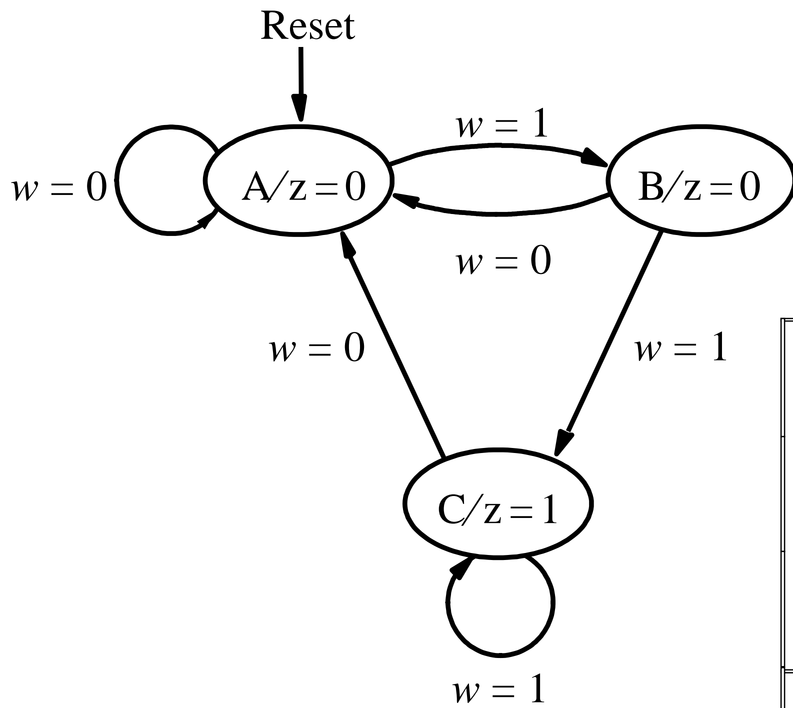
The present state variables, y_1 and y_2, determine the present state of the circuit.

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	Next state		Output
		$w = 0$	$w = 1$	z
	$y_2 y_1$	$Y_2 Y_1$	$Y_2 Y_1$	
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.

	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*

Y1 = w y1' y2'

Y2 = w (y1 + y2)

z = y2

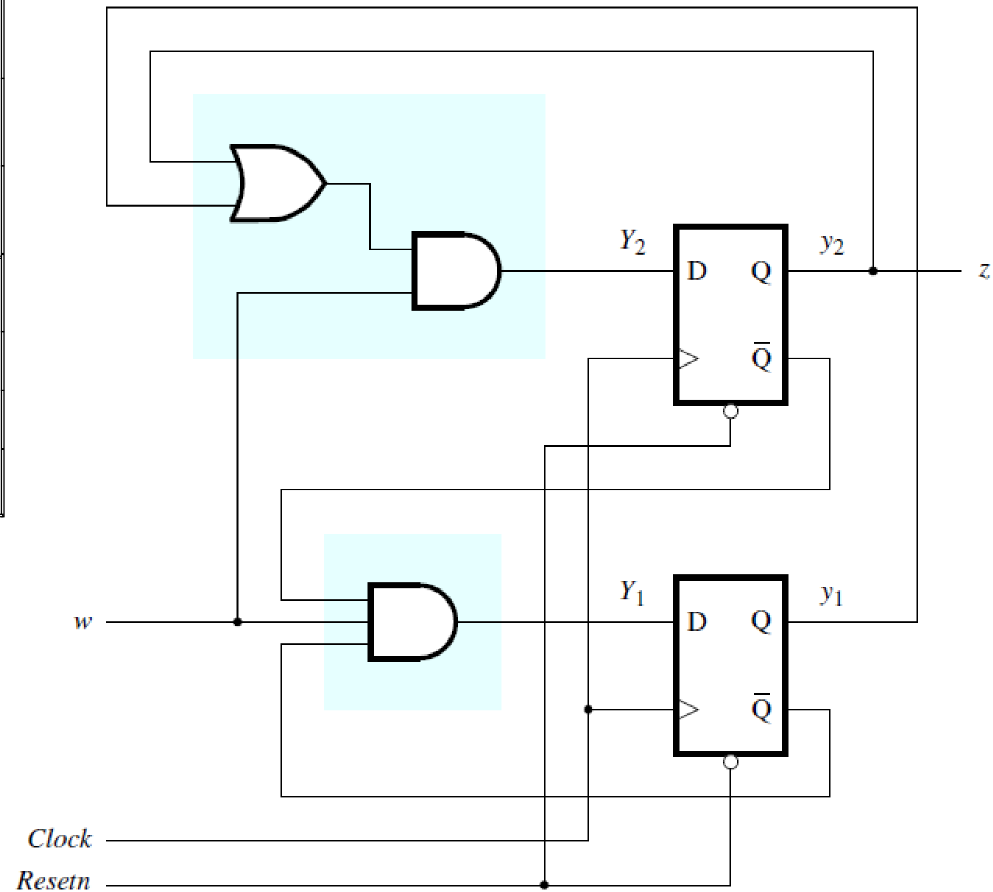
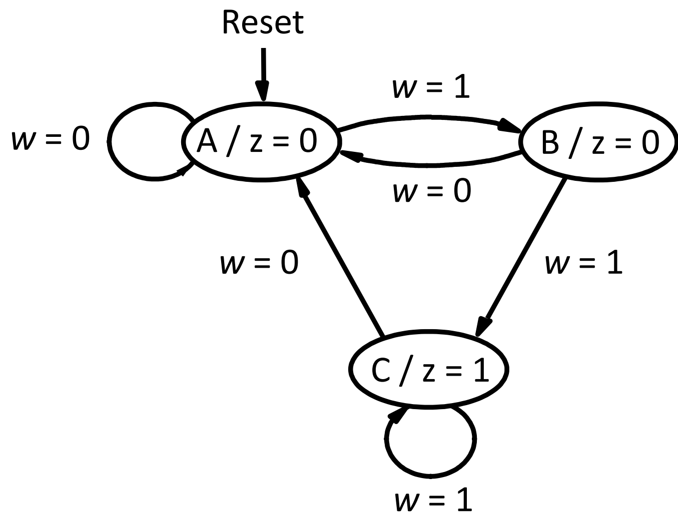


Figure 6.8. Final implementation of the speed controller using the don't cares.



	Present state $y_2 y_1$	Next state $w = 0$ $Y_2 Y_1$	Next state $w = 1$ $Y_2 Y_1$	Output 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
		$w = 0$	$w = 1$	
	$y_2 y_1$	$Y_2 Y_1$	$Y_2 Y_1$	z
A	00	00	01	0
B	01	00	11	0
C	11	00	11	1
	10	*dd*	*dd*	*d*

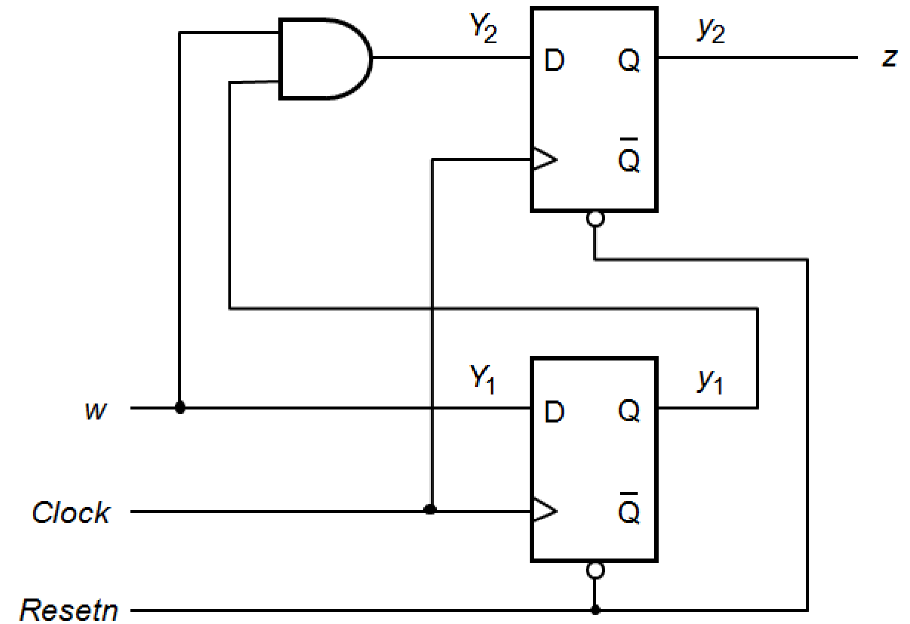
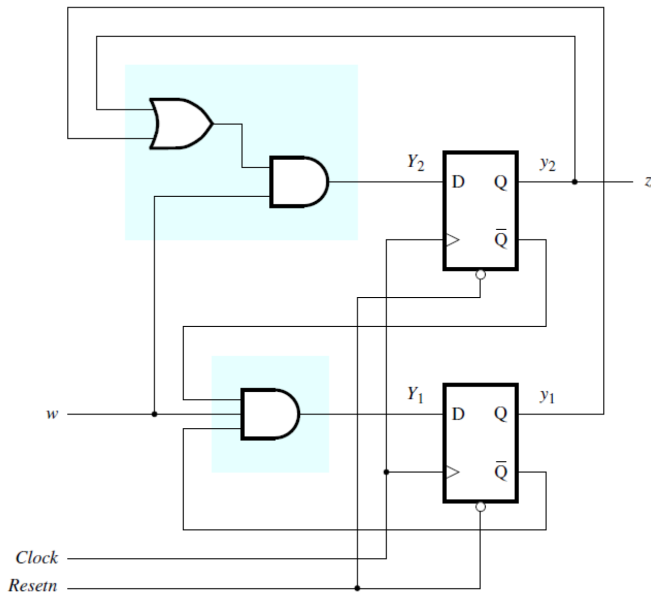
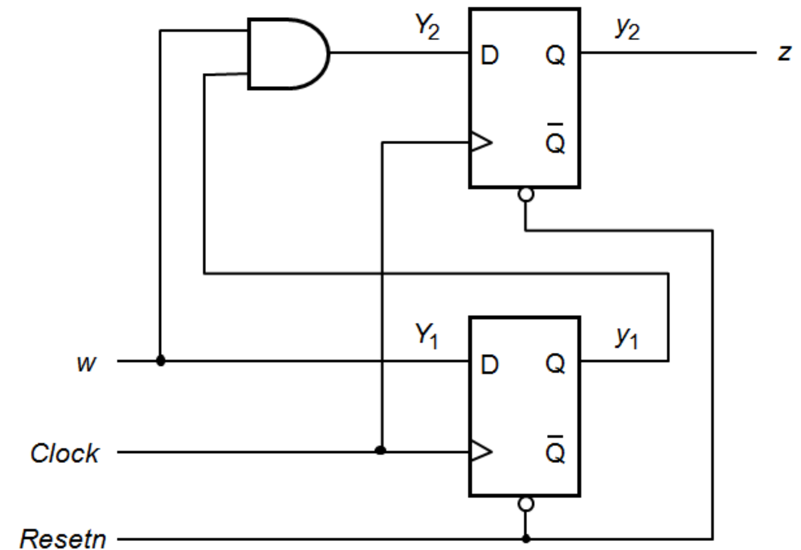


Figure 6.17. Final circuit for the improved state assignment for the speed controller.

	Present state	Next state		Output
		$w = 0$	$w = 1$	z
	$y_2 y_1$	$Y_2 Y_1$	$Y_2 Y_1$	
A	00	00	01	0
B	01	00	10	0
C	10	00	10	1
	11	dd	dd	d

	Present state	Next state		Output
		$w = 0$	$w = 1$	z
	$y_2 y_1$	$Y_2 Y_1$	$Y_2 Y_1$	
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.

```verilog
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
```

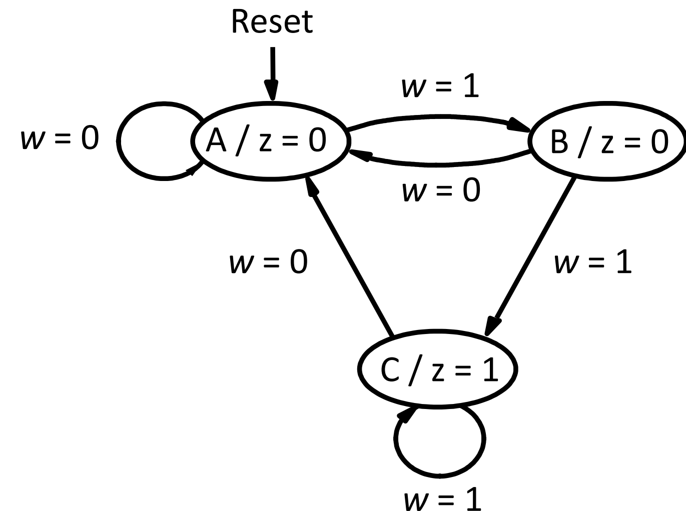
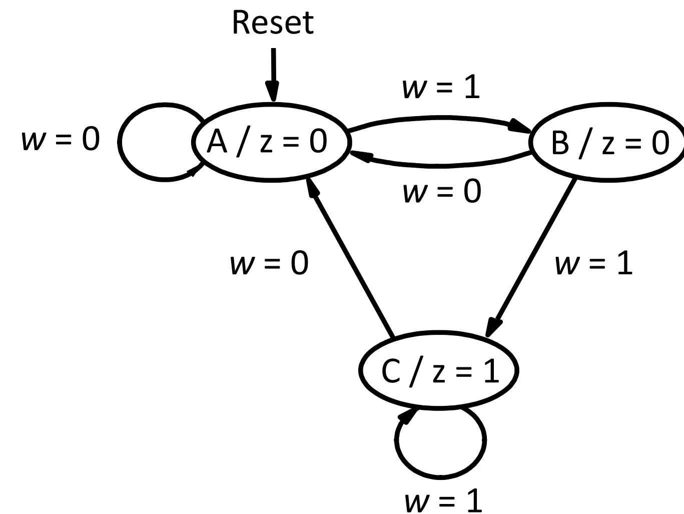


Figure 6.29. Verilog code for the speed controller.

```
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

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

   reg [ 2:1 ] 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.

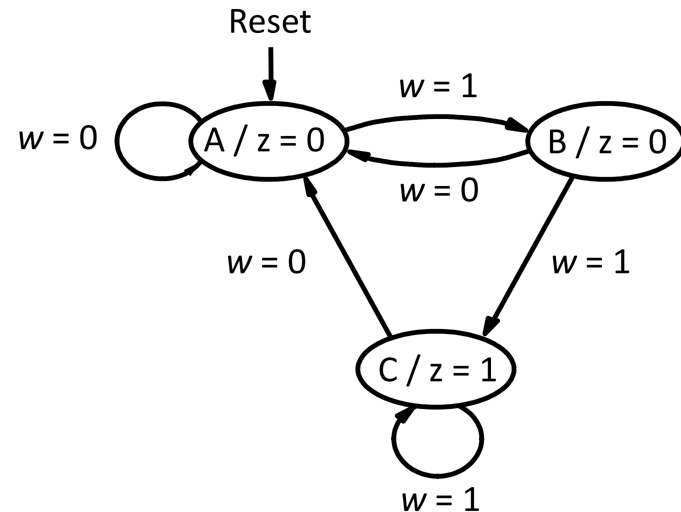
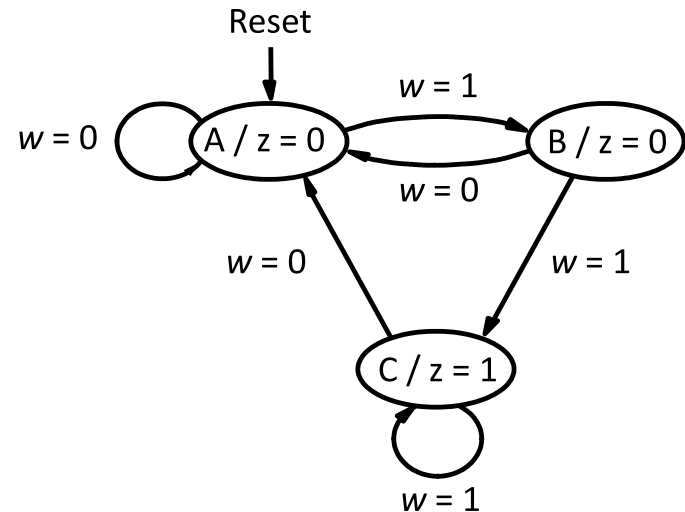
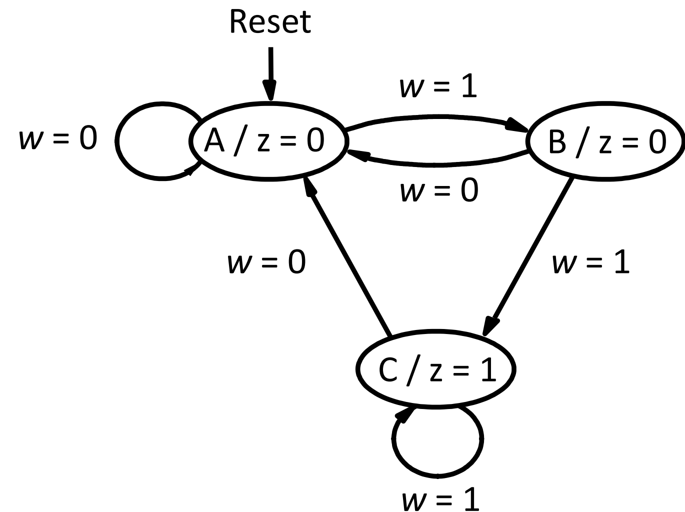


Figure 6.34

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

   reg [ 2:1 ] 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 } )
         4'b1xxx: y <= A;
         4'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;
   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
```



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

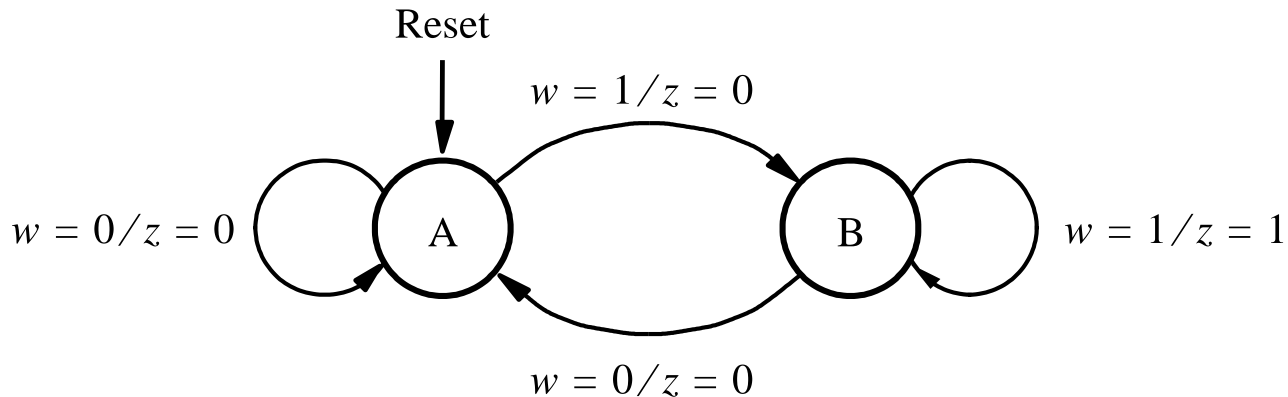
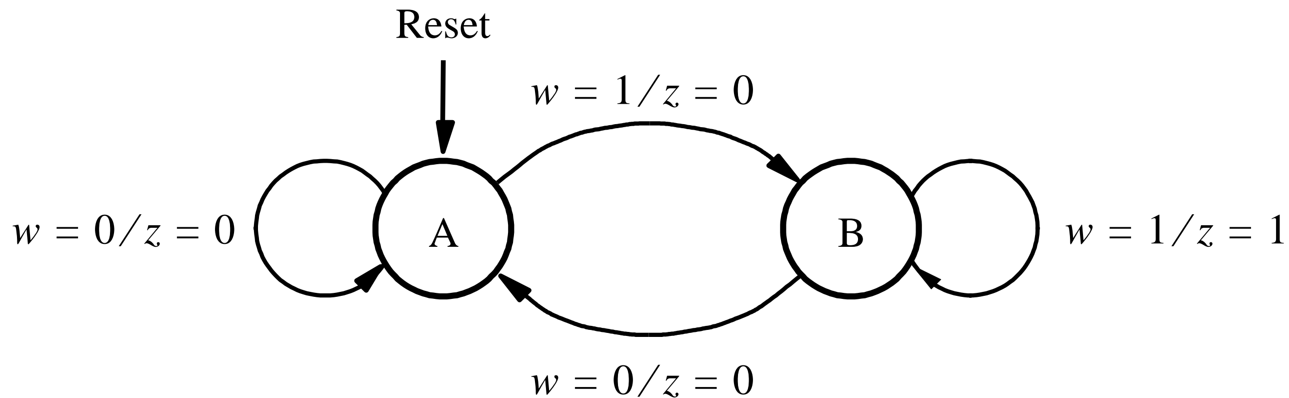


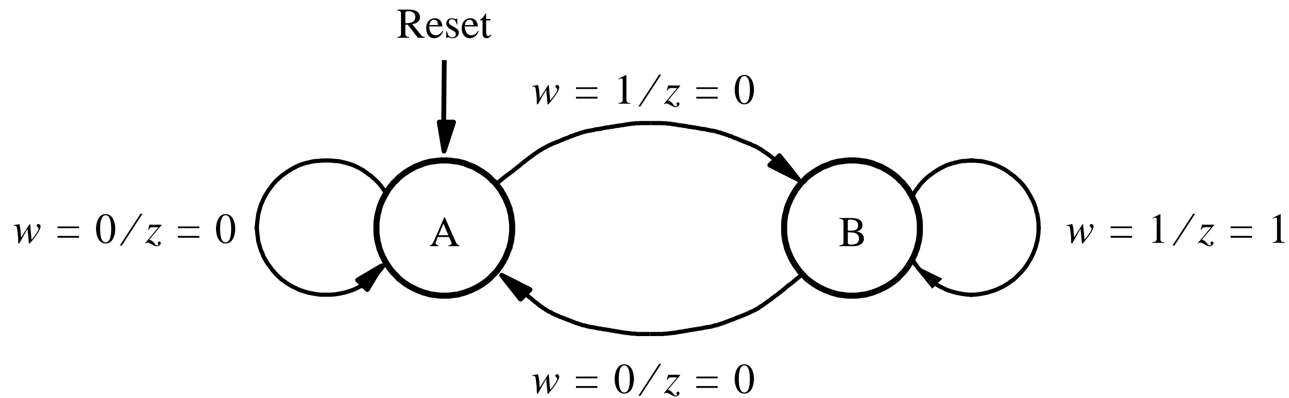
Figure 6.23. State diagram of the alternate speed controller.

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



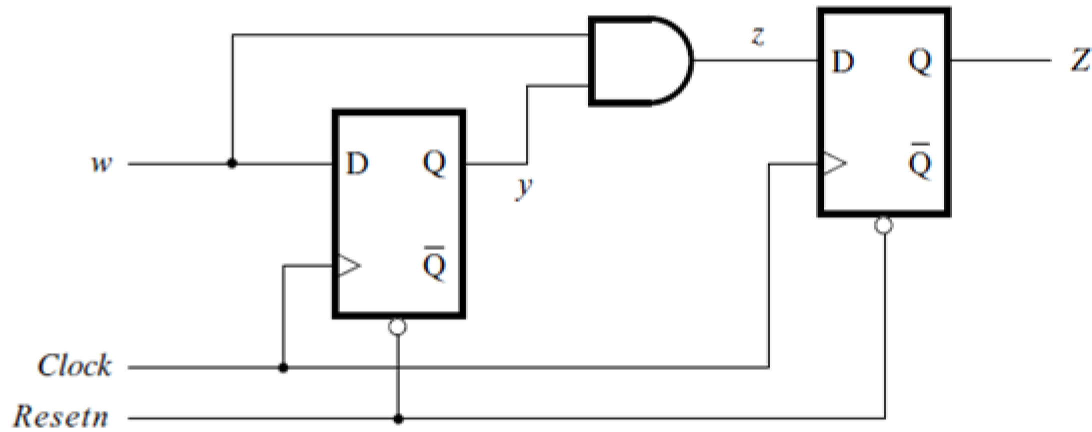
Present state	Next state		Output z	
	w = 0	w = 1	w = 0	w = 1
A	A	B	0	0
B	A	B	0	1

Figure 6.24. State table for the alternate speed controller.

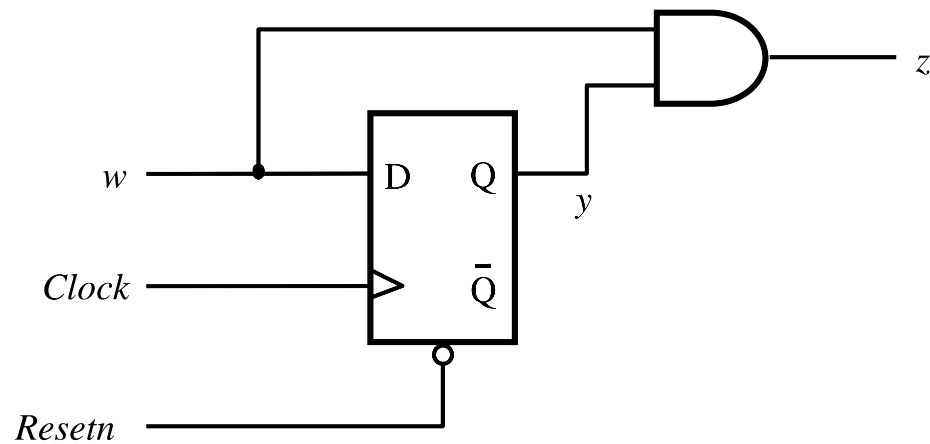


	Present state	Next state		Output	
		$w = 0$	$w = 1$	$w = 0$	$w = 1$
	y	Y	Y	z	z
A	0	0	1	0	0
B	1	0	1	0	1

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



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
```

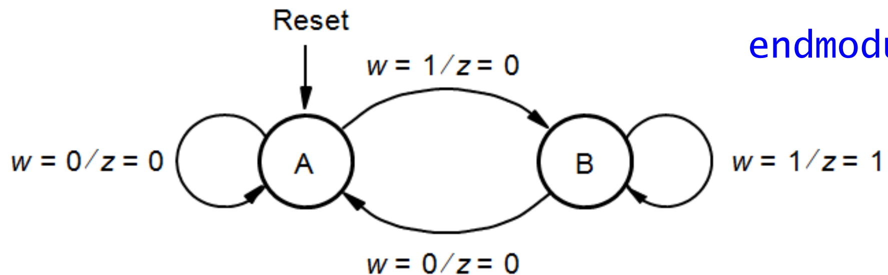
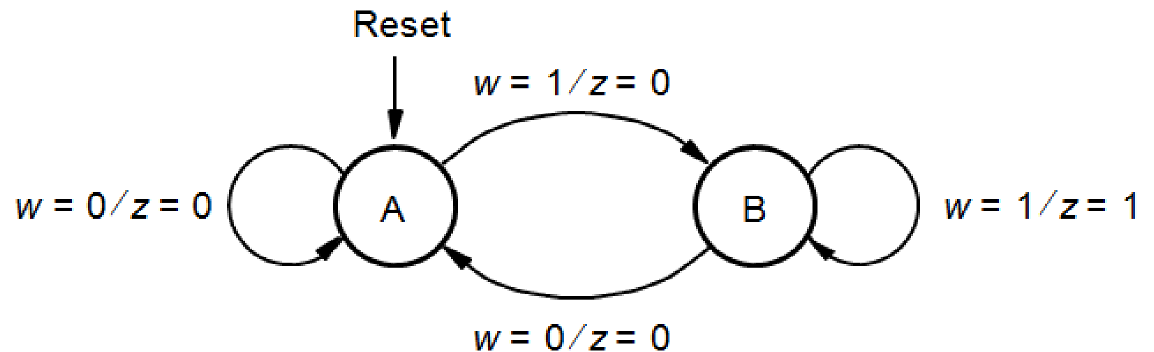
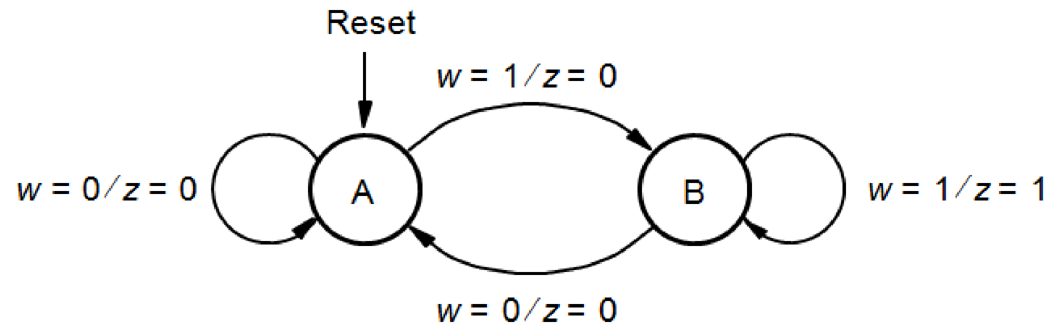


Figure 6.36

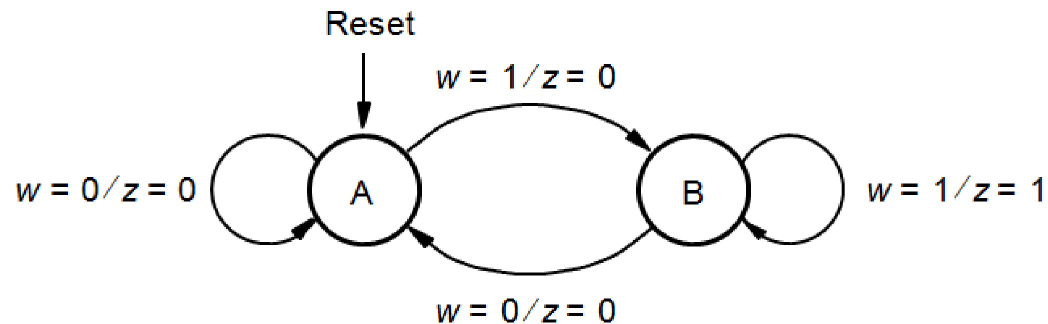
```
module mealy2( input clock, reset, w,
    output reg z );

  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
```



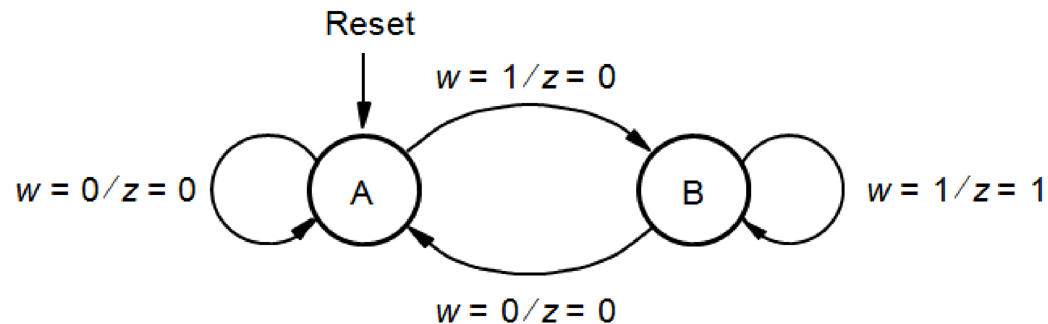
```
module mealy3( input clock, reset, w,
      output z );

   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
```



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

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

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

   assign z = state == B && w;

endmodule
```



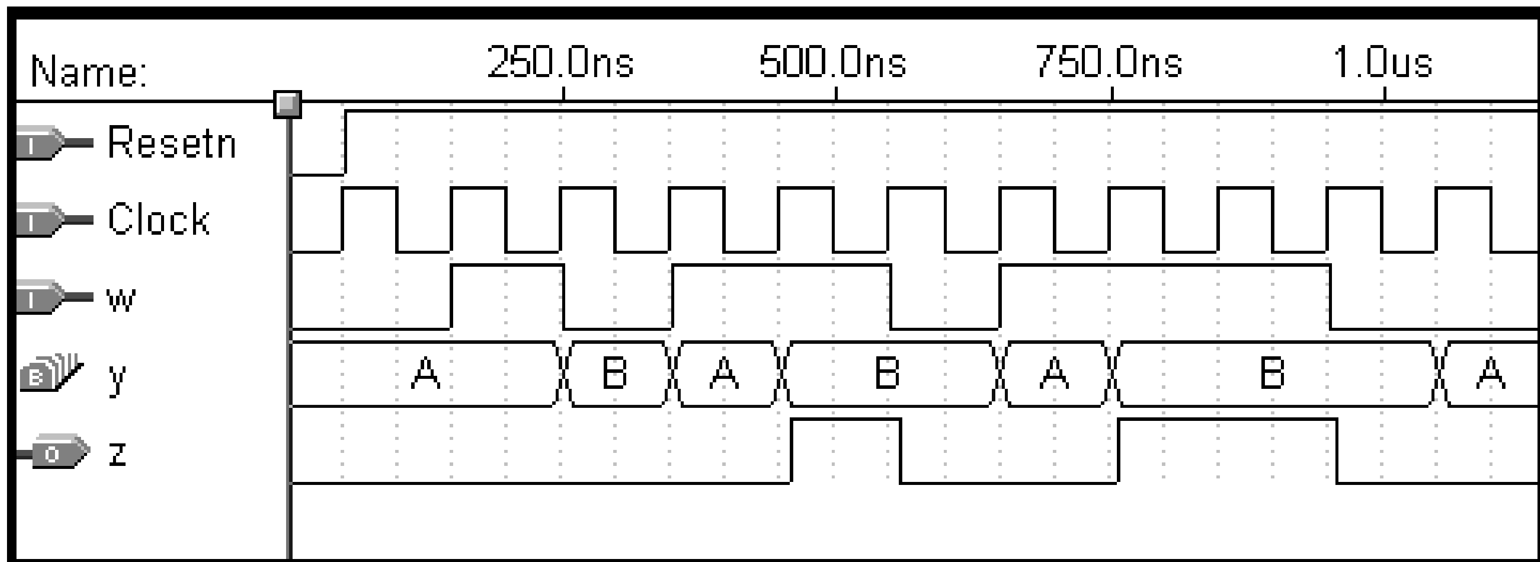
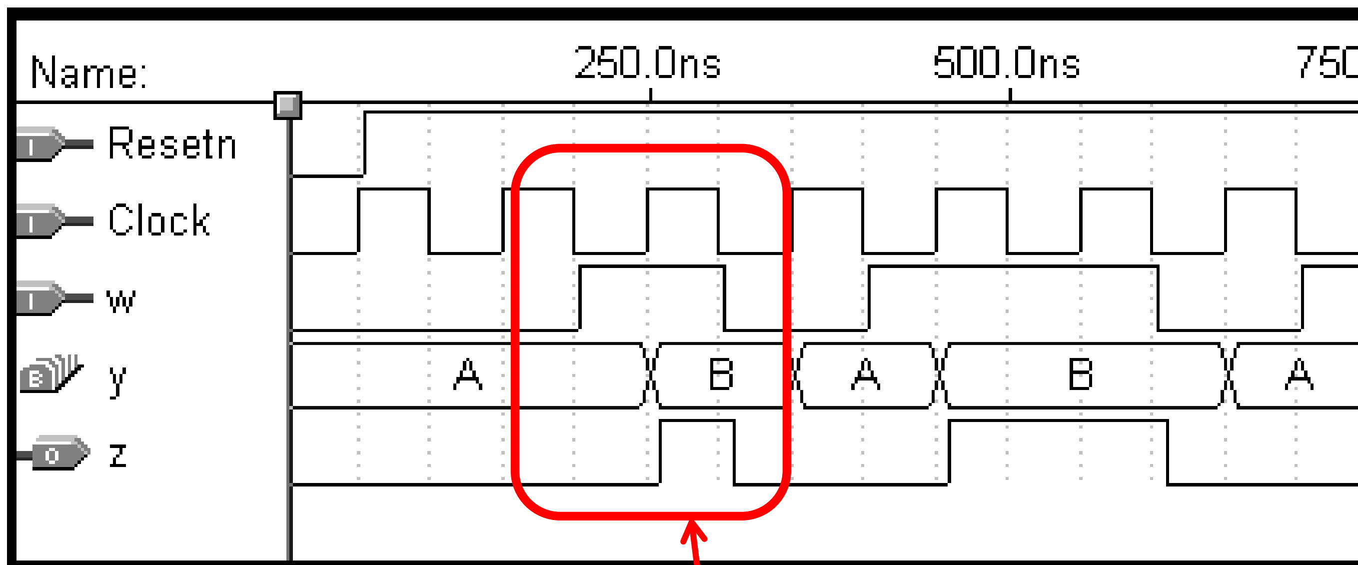


Figure 6.37. Simulation results for the Mealy machine.



The input, w, changes on the falling edge of the clock but the outputs change on the rising edge causing an incorrect output.

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

Example: Exchange registers

Meant to resemble exchanging register contents, perhaps in a CPU.

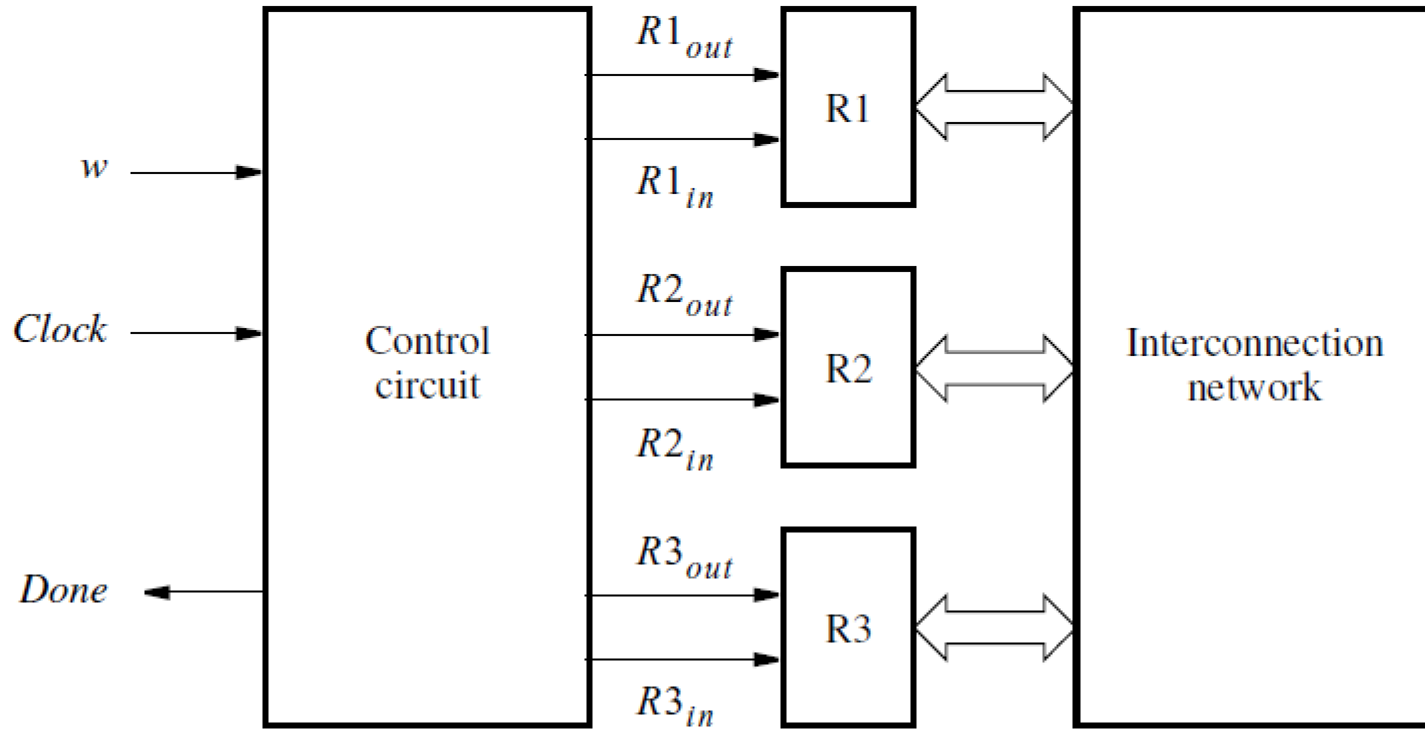


Figure 6.10. System for register exchange controller.

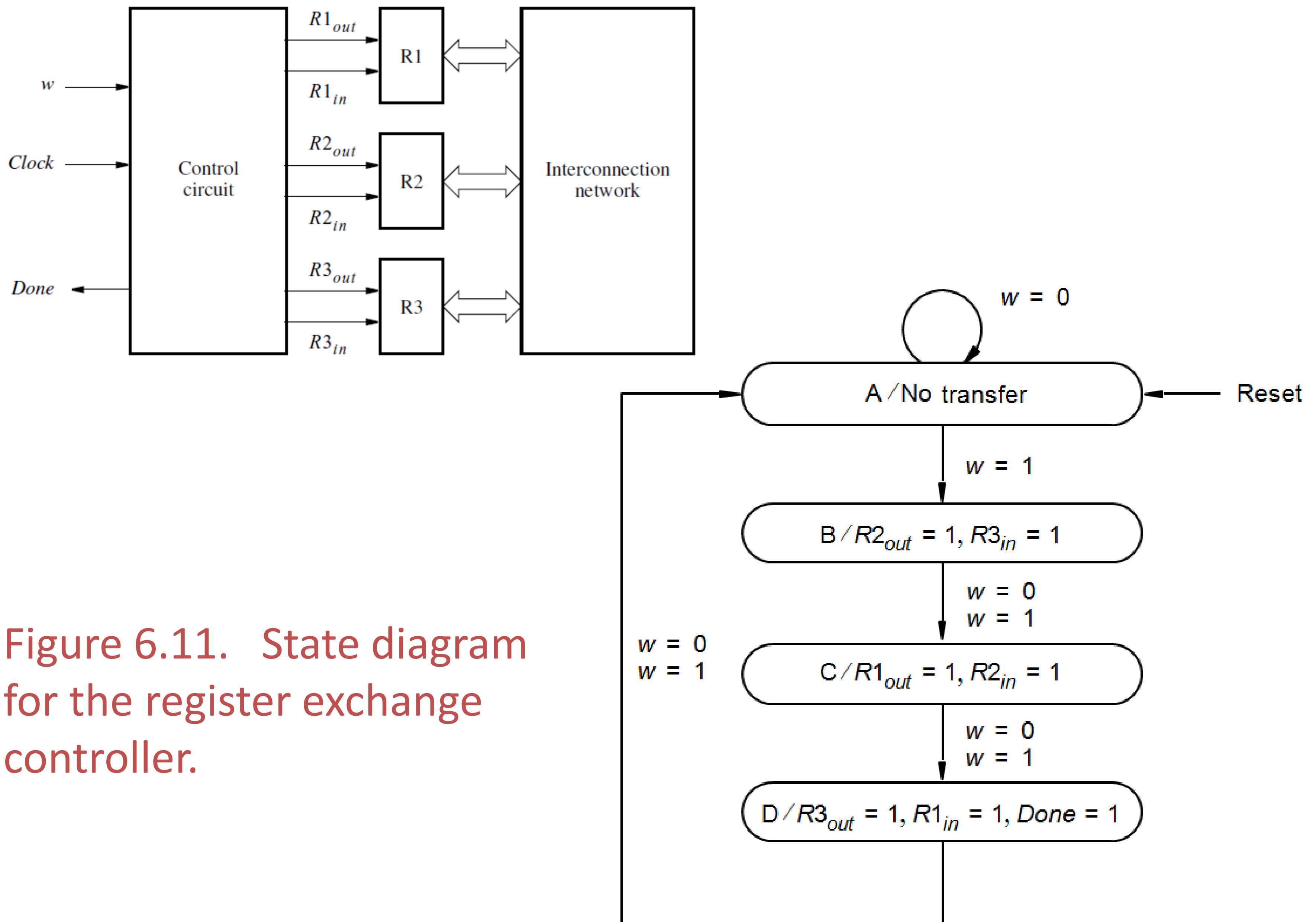
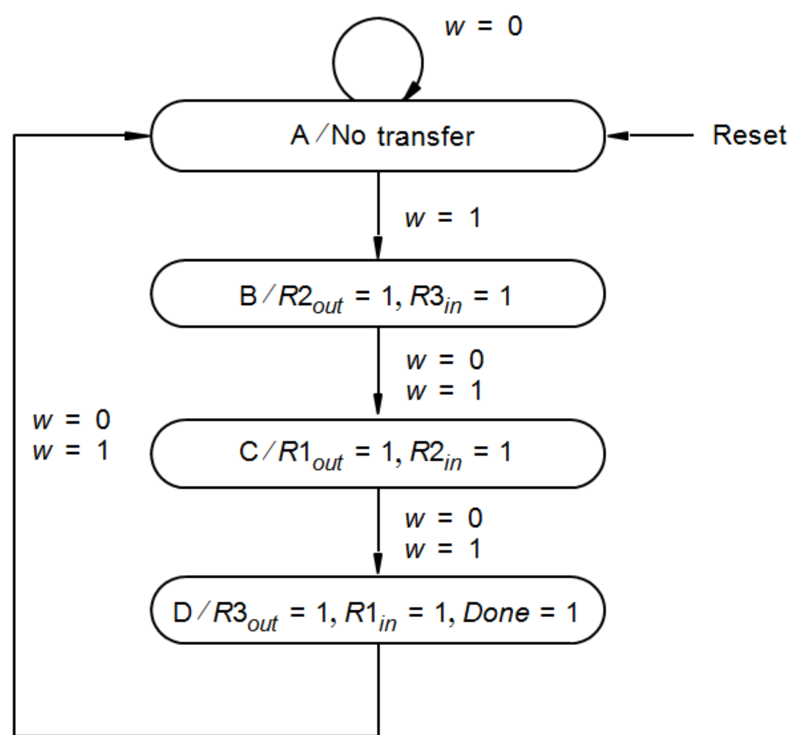
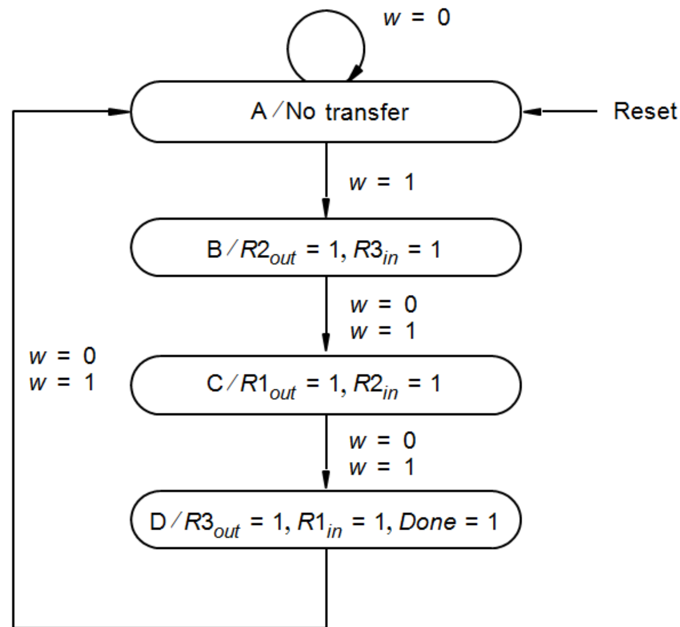


Figure 6.11. State diagram for the register exchange controller.



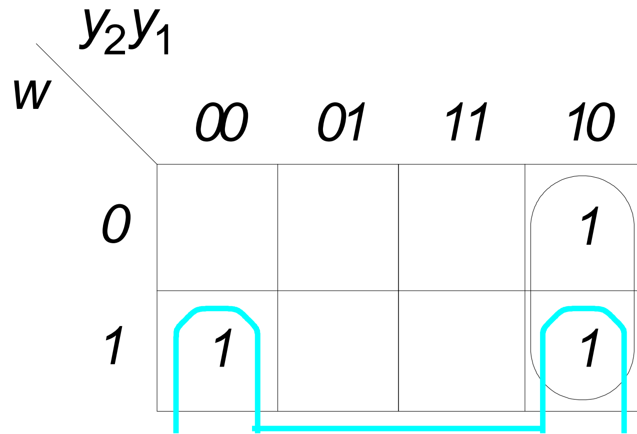
Present state	Next state		Outputs						
	$w=0$	$w=1$	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	A	B	0	0	0	0	0	0	0
B	C	C	0	0	1	0	0	1	0
C	D	D	1	0	0	1	0	0	0
D	A	A	0	1	0	0	1	0	1

Figure 6.12. State table for the register exchange.

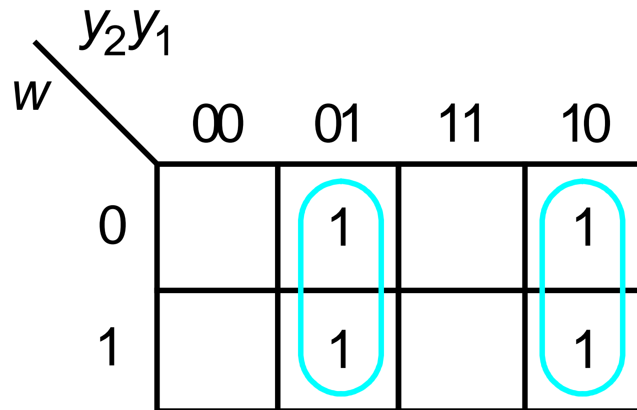


	Present state	Next state		Outputs						
		$w = 0$	$w = 1$							
	y_2y_1	Y_2Y_1	Y_2Y_1	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	00	00	0 1	0	0	0	0	0	0	0
B	01	10	1 0	0	0	1	0	0	1	0
C	10	11	1 1	1	0	0	1	0	0	0
D	11	00	0 0	0	1	0	0	1	0	1

Figure 6.13. State-assigned table for the register exchange.



$$Y_1 = w\bar{y}_1 + \bar{y}_1 y_2$$



$$Y_2 = y_1\bar{y}_2 + \bar{y}_1 y_2$$

Figure 6.14. Derivation of next-state expressions for the for the register exchange sequence.

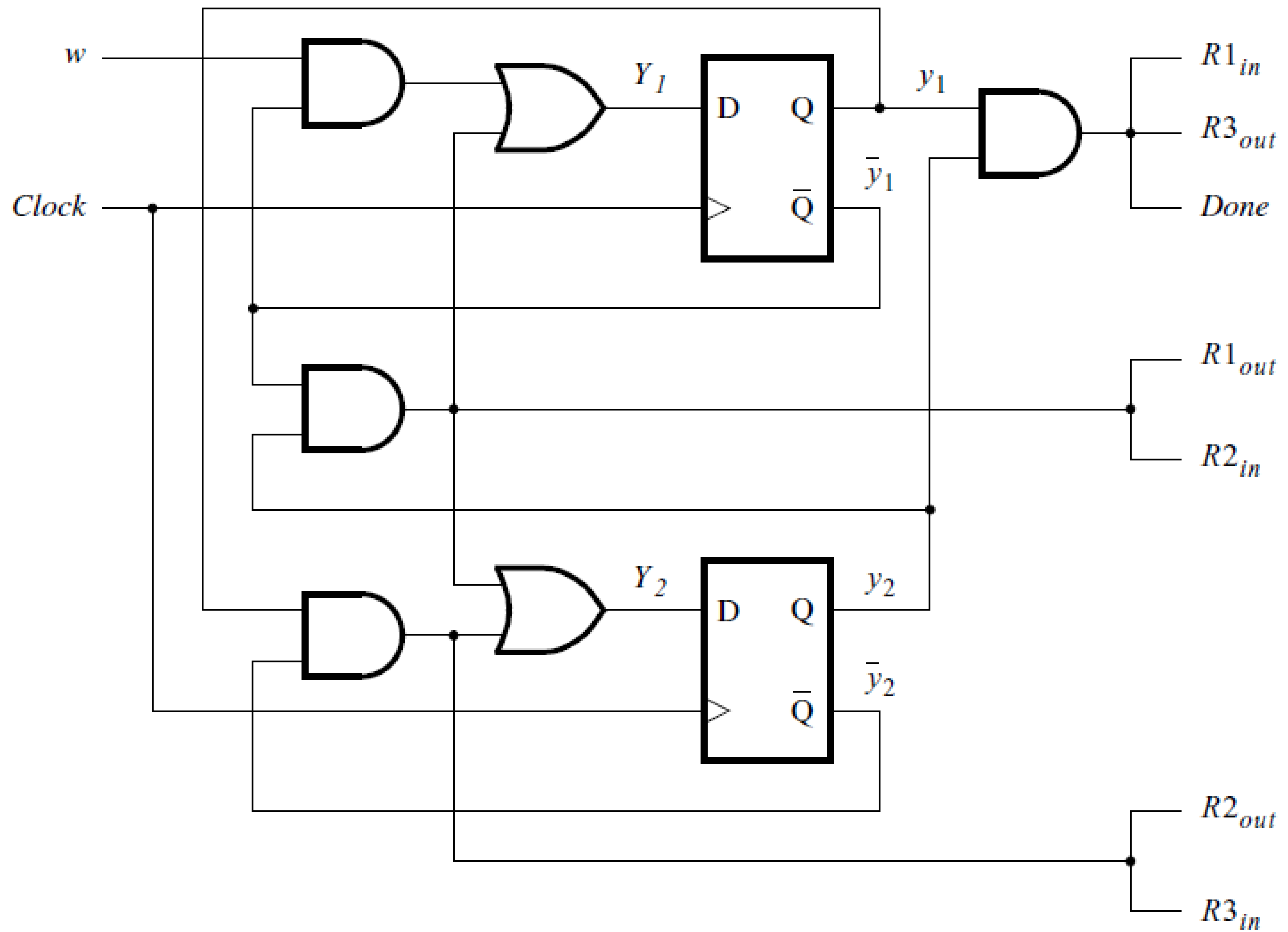


Figure 6.15. Final implementation of for the register exchange controller.

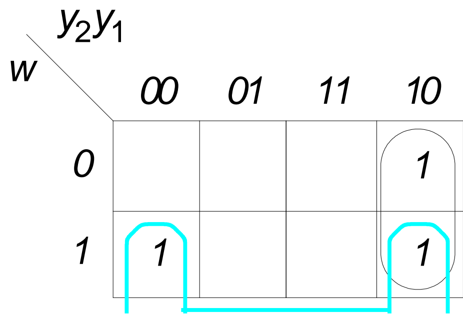
Original

	Present state	Next state		Outputs						
		$w=0$	$w=1$							
	y_2y_1	Y_2Y_1	Y_2Y_1	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	00	00	0 1	0	0	0	0	0	0	0
B	01	10	1 0	0	0	1	0	0	1	0
C	10	11	1 1	1	0	0	1	0	0	0
D	11	00	0 0	0	1	0	0	1	0	1

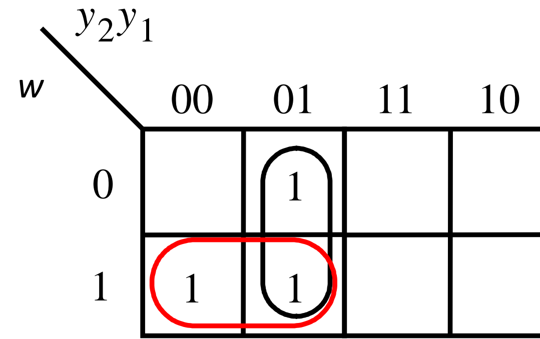
Improved

	Present state	Nextstate		Outputs						
		$w=0$	$w=1$							
	y_2y_1	Y_2Y_1	Y_2Y_1	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	00	0 0	01	0	0	0	0	0	0	0
B	01	1 1	11	0	0	1	0	0	1	0
C	11	1 0	10	1	0	0	1	0	0	0
D	10	0 0	00	0	1	0	0	1	0	1

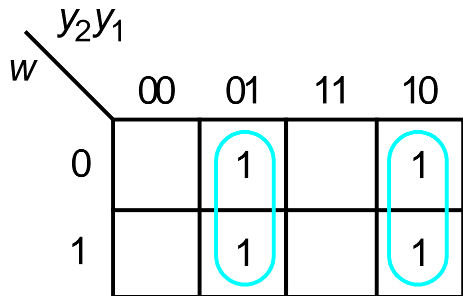
Figure 6.18. Original vs improved state assignment for the register transfer controller.



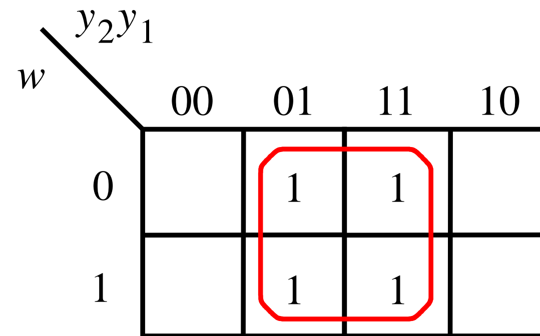
$$Y_1 = w\bar{y}_1 + \bar{y}_1 y_2$$



$$Y_1 = w\bar{y}_2 + y_1\bar{y}_2$$



$$Y_2 = y_1\bar{y}_2 + \bar{y}_1 y_2$$



$$Y_2 = y_1$$

Figure 6.19. Derivation of next-state expressions for the improved register transfer controller.

	Present state	Nextstate		Outputs						
		$w = 0$	$w = 1$							
	$y_4 y_3 y_2 y_1$	$Y_4 Y_3 Y_2 Y_1$	$Y_4 Y_3 Y_2 Y_1$	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	0 001	0001	0010	0	0	0	0	0	0	0
B	0 010	0100	0100	0	0	1	0	0	1	0
C	0 100	1000	1000	1	0	0	1	0	0	0
D	1 000	0001	0001	0	1	0	0	1	0	1

Y1 = w'

Y2 = w y1

Y3 = w Y1'

z = y3

Figure 6.21. One-hot state assignment for the register transfer controller.

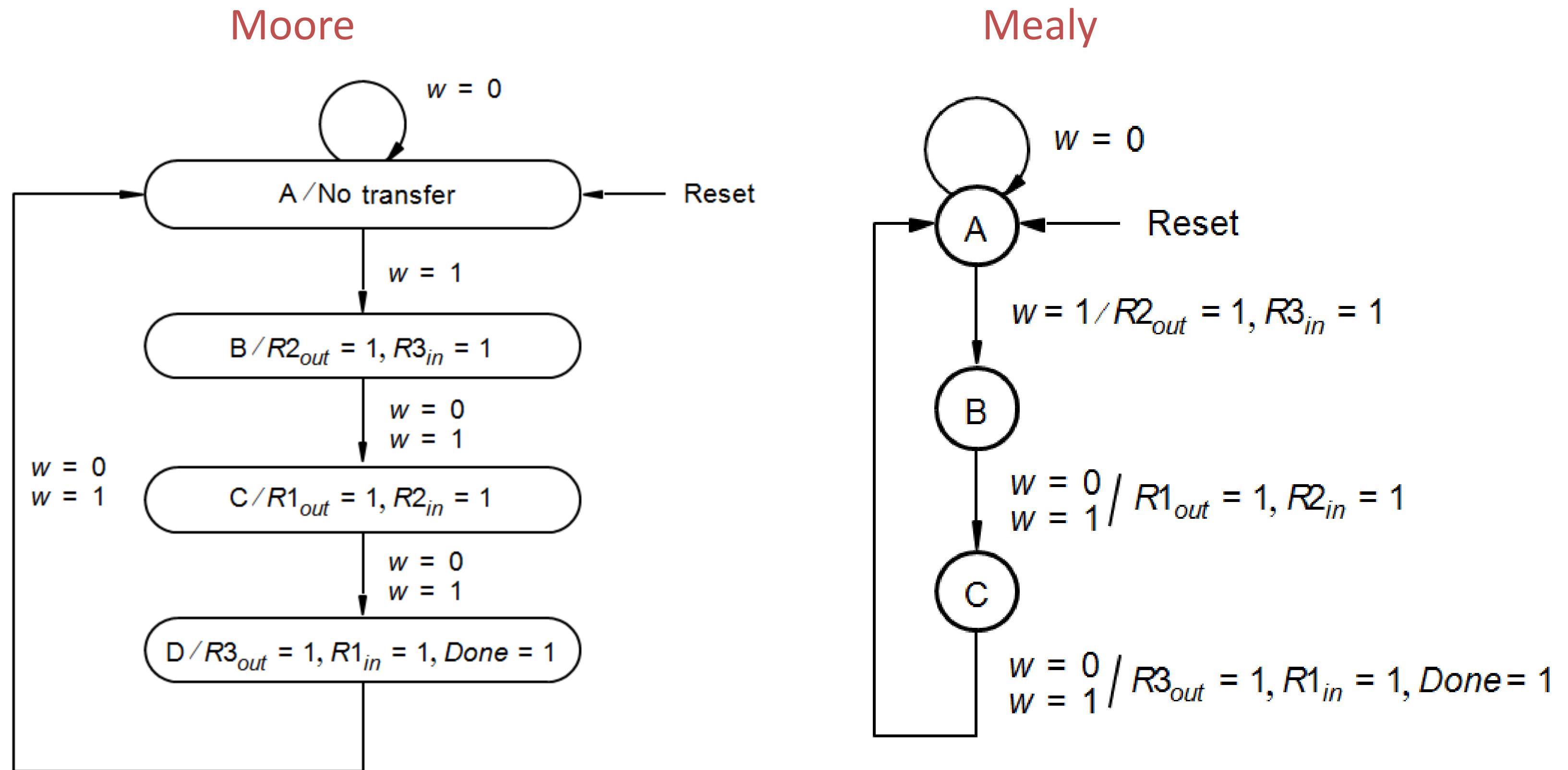


Figure 6.28. Moore vs Mealy designs for the register transfer controller.

```verilog
module Control1( input clock, reset, w,
      output R1in, R1out, R2in, R2out,
         R3in, R3out, Done);

   reg [ 2:1 ] y, Y;
   parameter [ 2:1 ] A = 2'b00, B = 2'b01,
      C = 2'b10, D = 2'b11;

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

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

   // Define outputs
   assign R2out = y == B;
   assign R3in  = y == B;
   assign R1out = y == C;
   assign R2in  = y == C;
   assign R3out = y == D;
   assign R1in  = y == D;
   assign Done  = y == D;

endmodule
```

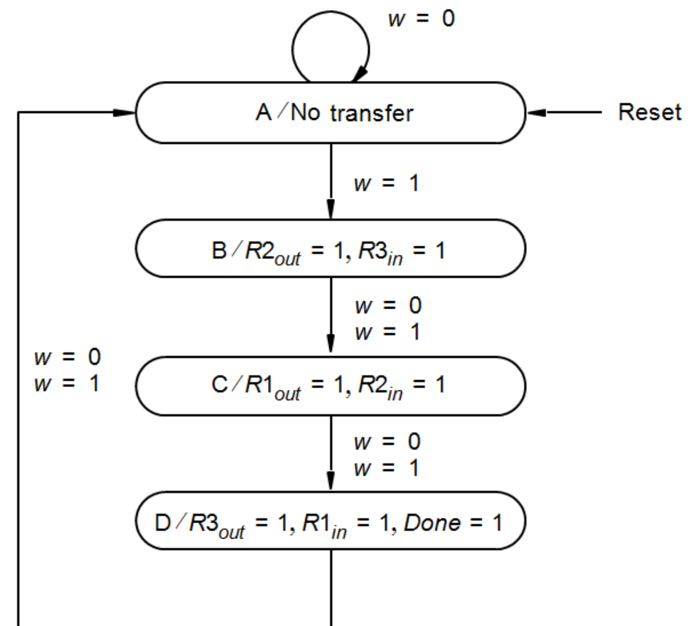
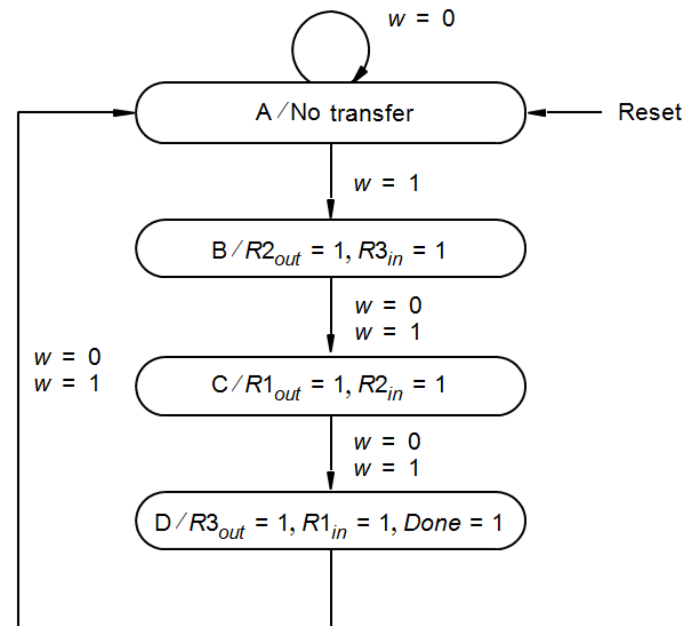
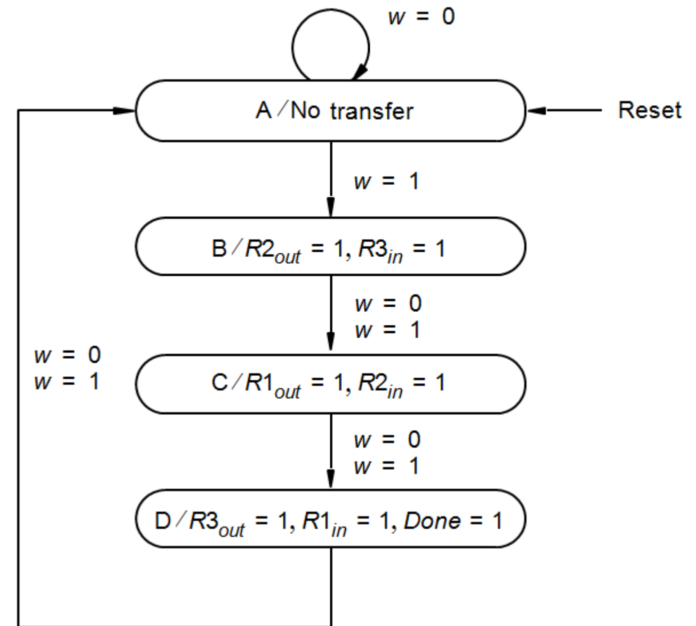


Figure 6.35

```
module Control2( input clock, reset, w,
      output R1in, R1out, R2in, R2out,
         R3in, R3out, Done);

   reg [ 2:1 ] y;
   parameter [ 2:1 ] A = 2'b00, B = 2'b01,
      C = 2'b10, D = 2'b11;

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

   // Define outputs
     assign R2out = y == B;
     assign R3in  = y == B;
     assign R1out = y == C;
     assign R2in  = y == C;
     assign R3out = y == D;
     assign R1in  = y == D;
     assign Done  = y == D;

endmodule
```



```
module Control3( input clock, reset, w,
      output R1in, R1out, R2in, R2out,
         R3in, R3out, Done);

   enum { A, B, C, D } y;

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

// Define outputs
   assign R2out = y == B;
   assign R3in  = y == B;
   assign R1out = y == C;
   assign R2in  = y == C;
   assign R3out = y == D;
   assign R1in  = y == D;
   assign Done  = y == D;

endmodule
```



Example: Serial adder

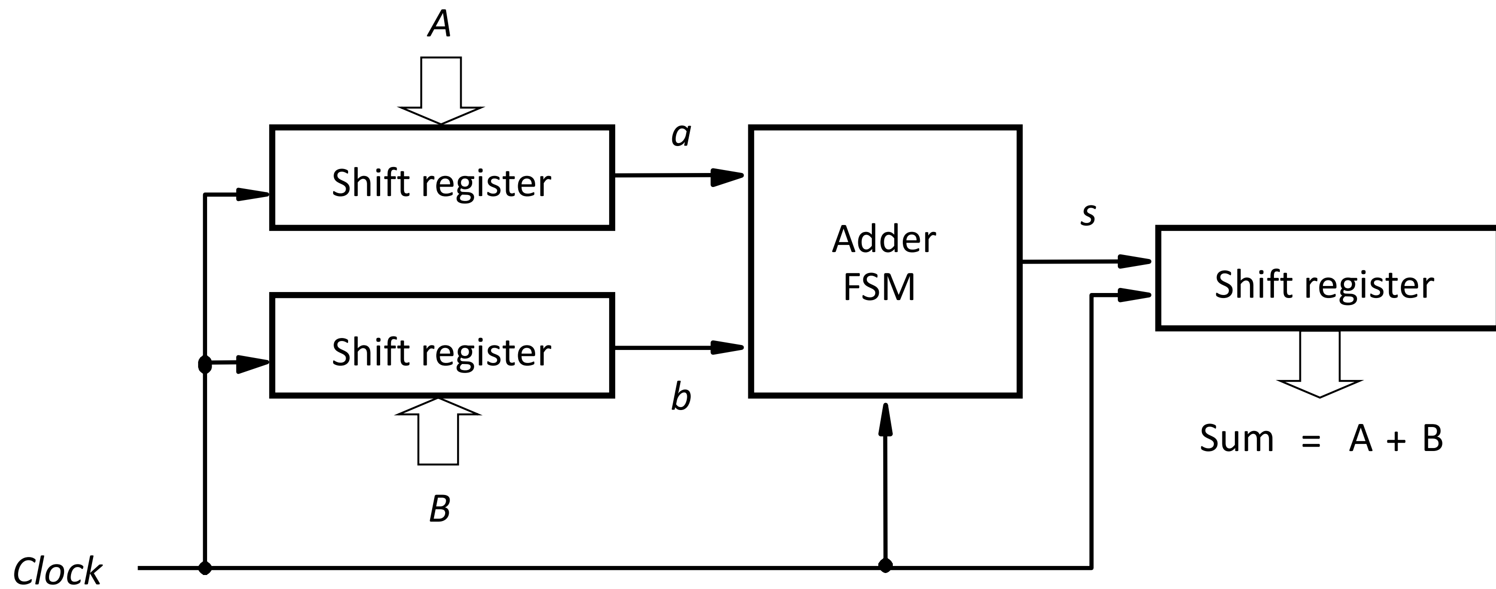


Figure 6.39. Block diagram for the serial adder.

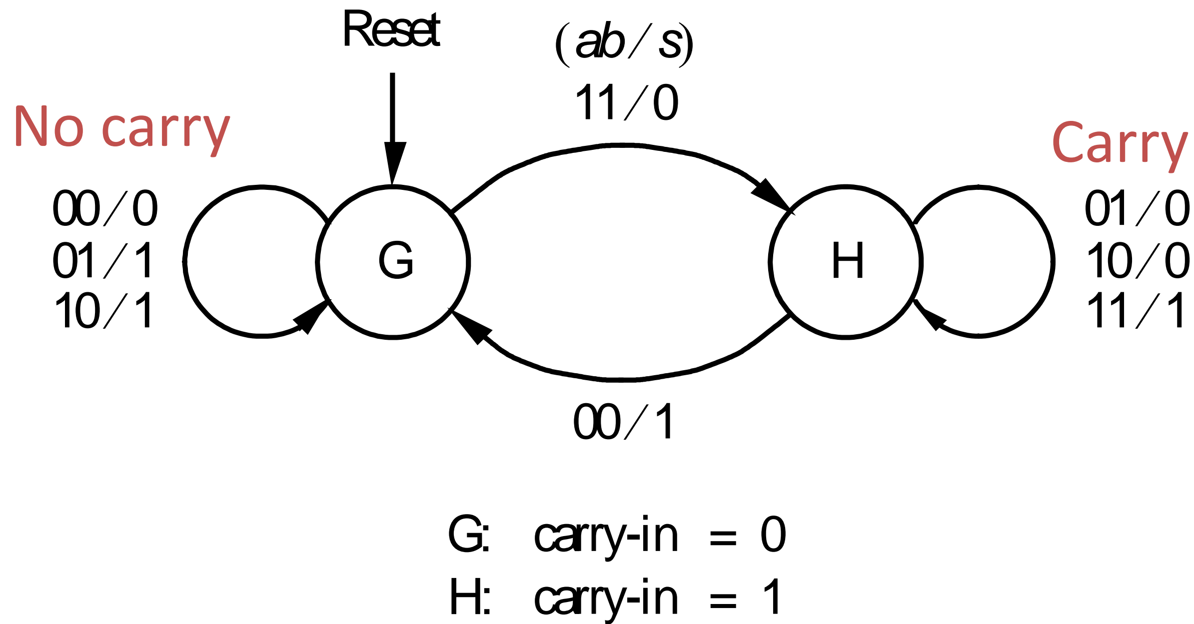
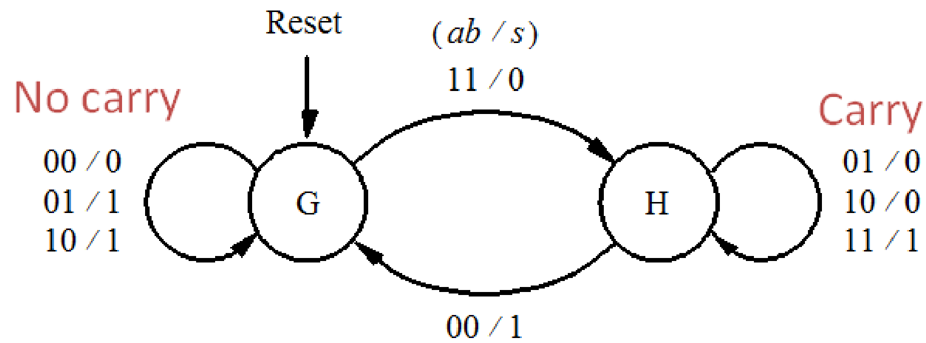


Figure 6.40. State diagram for the serial adder FSM.



Present state	Next state				Output s			
	ab =00	01	10	11	00	01	10	11
G	G	G	G	H	0	1	1	0
H	G	H	H	H	1	0	0	1

Figure 6.41. State table for the serial adder FSM.

Present state	Next state				Outputs s			
	ab =00	01	10	11	00	01	10	11
G	G	G	G	H	0	1	1	0
H	G	H	H	H	1	0	0	1

Present state	Next state				Output			
	ab =00	01	10	11	00	01	10	11
y	Y				s			
0	0	0	0	1	0	1	1	0
1	0	1	1	1	1	0	0	1

Figure 6.42. State-assigned table for the serial adder.

Present state	Next state				Output			
	ab =00	01	10	11	00	01	10	11
y	*Y*				*s*			
0	0	0	0	1	0	1	1	0
1	0	1	1	1	1	0	0	1

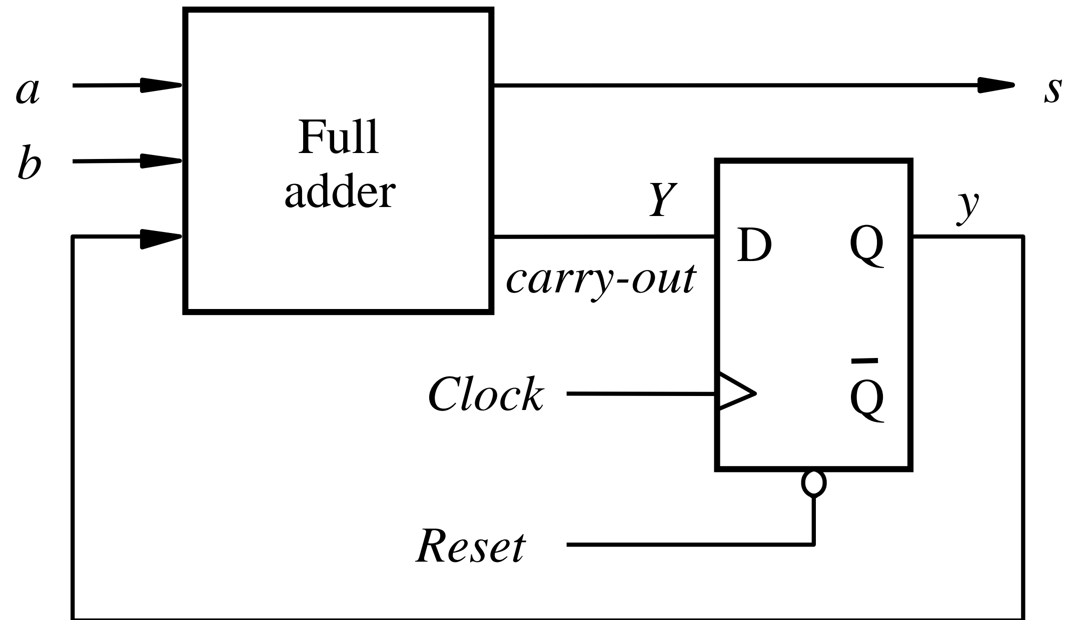


Figure 6.43. Circuit for the serial adder.

```
module ShiftReg1( input [ n - 1:0 ] R,
      input L, E, w, clock,
      output reg [ n - 1:0 ] Q );
   parameter n = 8;
   integer k;

   always @( posedge clock )
      if ( L )
         Q <= R;
      else
         if ( E )
            begin
            for ( k = n - 1; k > 0; k = k - 1 )
               Q[ k - 1 ] <= Q [ k ];
            Q[ n - 1 ] <= w;
            end
endmodule
```

Figure 6.48

```
module ShiftReg2( input clock, reset, enable, D,
      input [ n - 1:0 ] resetValue,
      output reg [ n - 1 : 0 ] Q );

   parameter n = 8;
   integer k;

   always @( posedge clock )
      if ( reset )
         Q <= resetValue;
      else
         if ( enable )
            Q <= { D, Q[ n-1:1 ] };

endmodule
```

```
module SerialAdder1( input [ 7:0 ] A, B,
      input reset, clock,
      output wire [ 7:0 ] C );

   parameter G = 0, H = 1;
   reg  [ 3:0 ] count;
   reg  s, y, Y;
   wire [ 7:0 ] QA, QB;
   wire run;

   ShiftReg1 sA( A, reset, 1,   0, clock, QA );
   ShiftReg1 sB( B, reset, 1,   0, clock, QB );
   ShiftReg1 sC( 0, reset, Run, s, clock, C  );

   // Adder FSM
   always @( * )
      case ( y )
         G: begin
            s = QA[ 0 ] ^ QB[ 0 ];
            if ( QA[ 0 ] & QB[ 0 ] )
               Y = H;
            else
               Y = G;
            end
         H: begin
            s = QA[ 0 ] ~^ QB[ 0 ];
            if ( ~QA[ 0 ] & ~QB[ 0 ] )
               Y = G;
            else
               Y = H;
            end
         default: Y = G;
      endcase

   // Sequential block
   always @( posedge clock )
      if (reset)
         y <= G;
      else
         y <= Y;

   // Control the shifting process
   always @( posedge clock )
      if ( reset )
         count <= 8;
      else
         if ( run )
            count <= count - 1;

   assign run = |count;

endmodule
```

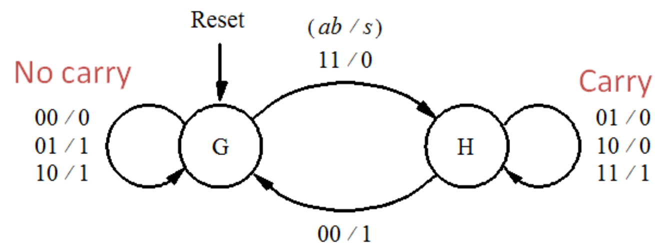
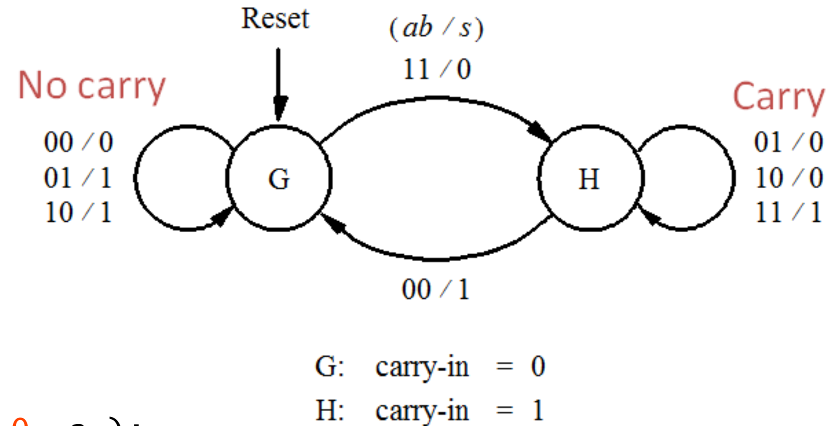


Figure 6.49

```
module SerialAdder2( input clock, reset,
    input [ 7:0 ] A, B, output [ 7:0 ] C );

  enum { NoCarry, Carry } s;
  wire [ 1:0 ] sum;
  wire [ 7:0 ] a, b;
  wire run, cin;

  assign sum = a[ 0 ] + b[ 0 ];
  assign cin = sum[ 1 ];
  assign run = |{ a, b };

  ShiftReg2 sA( clock, reset, run, A, 0, a );
  ShiftReg2 sB( clock, reset, run, B, 0, b );
  ShiftReg2 sC( clock, reset, run, 0, s, C );

  // Adder FSM
  always @( posedge clock )
    if ( reset )
      s <= NoCarry;
    else
      case ( s )
        NoCarry: s <= cin ? Carry   : NoCarry;
        Carry  : s <= cin ? NoCarry : Carry;
      endcase

endmodule
```



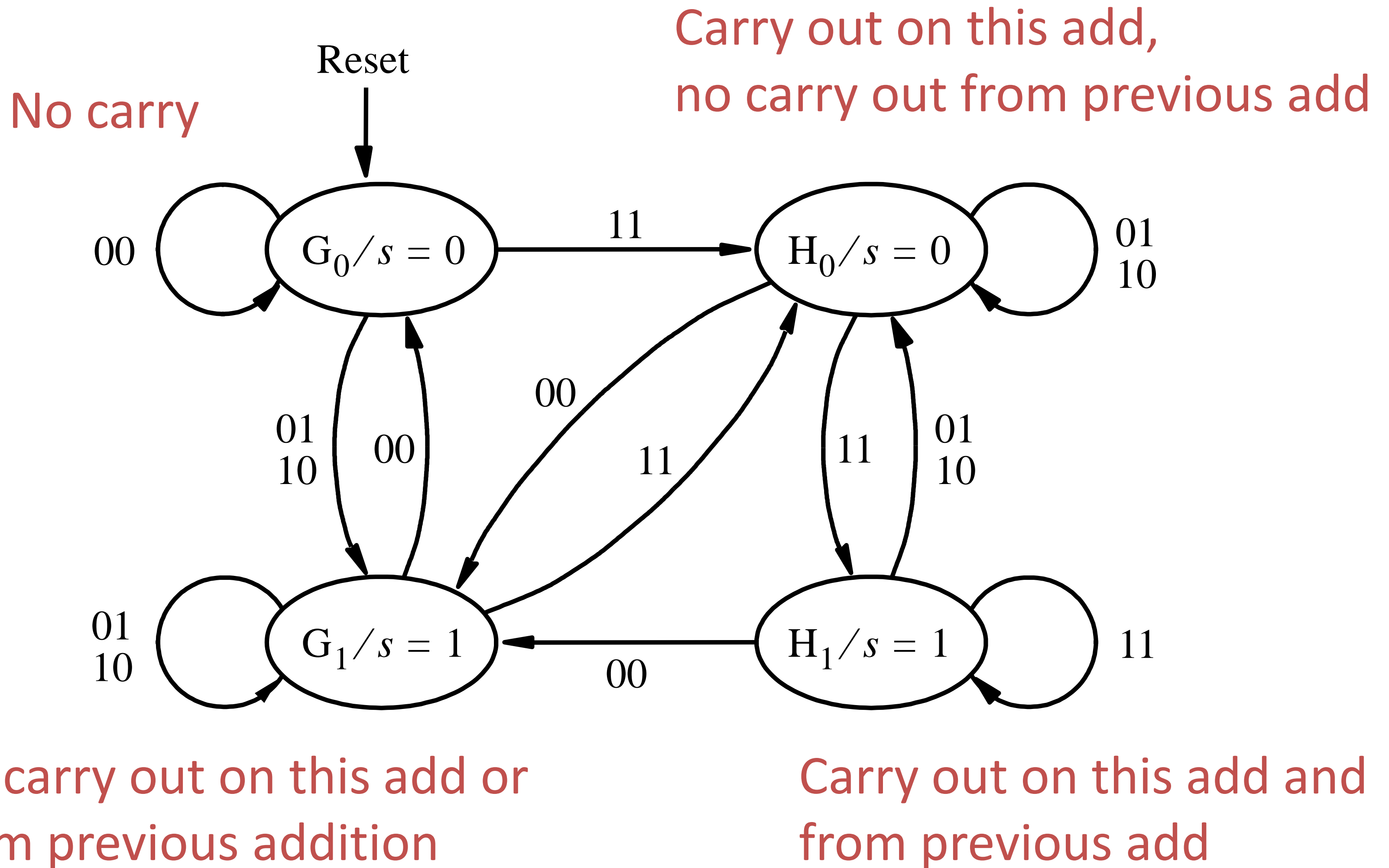
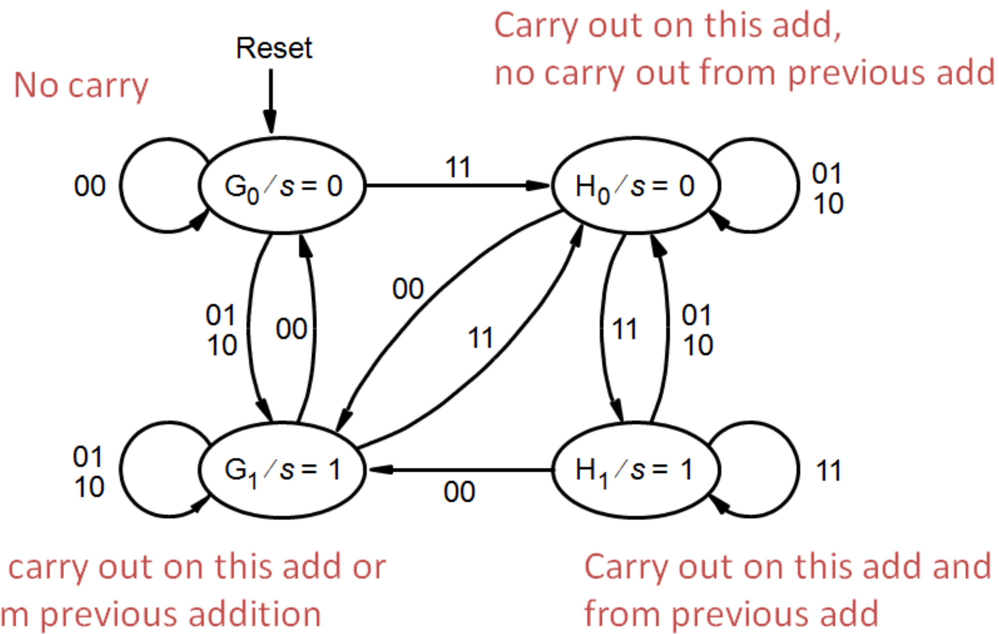


Figure 6.44. State diagram for the Moore-type serial adder FSM.



Present state	Next state				Output
	ab =00	01	10	11	*s*
G_0	G_0	G_1	G_1	H_0	0
G_1	G_0	G_1	G_1	H_0	1
H_0	G_1	H_0	H_0	H_1	0
H_1	G_1	H_0	H_0	H_1	1

Figure 6.45. State table for the Moore-type serial adder FSM.

Present state	Next state				Output
	ab =00	01	10	11	s
G_0	G_0	G_1	G_1	H_0	0
G_1	G_0	G_1	G_1	H_0	1
H_0	G_1	H_0	H_0	H_1	0
H_1	G_1	H_0	H_0	H_1	1

Present state y_2y_1	Next state				Output s
	ab =00	01	10	11	
	Y_2Y_1				
00	0 0	01	0 1	10	0
01	0 0	01	0 1	10	1
10	0 1	10	1 0	11	0
11	0 1	10	1 0	11	1

Figure 6.46. State-assigned table for Figure 6.45.

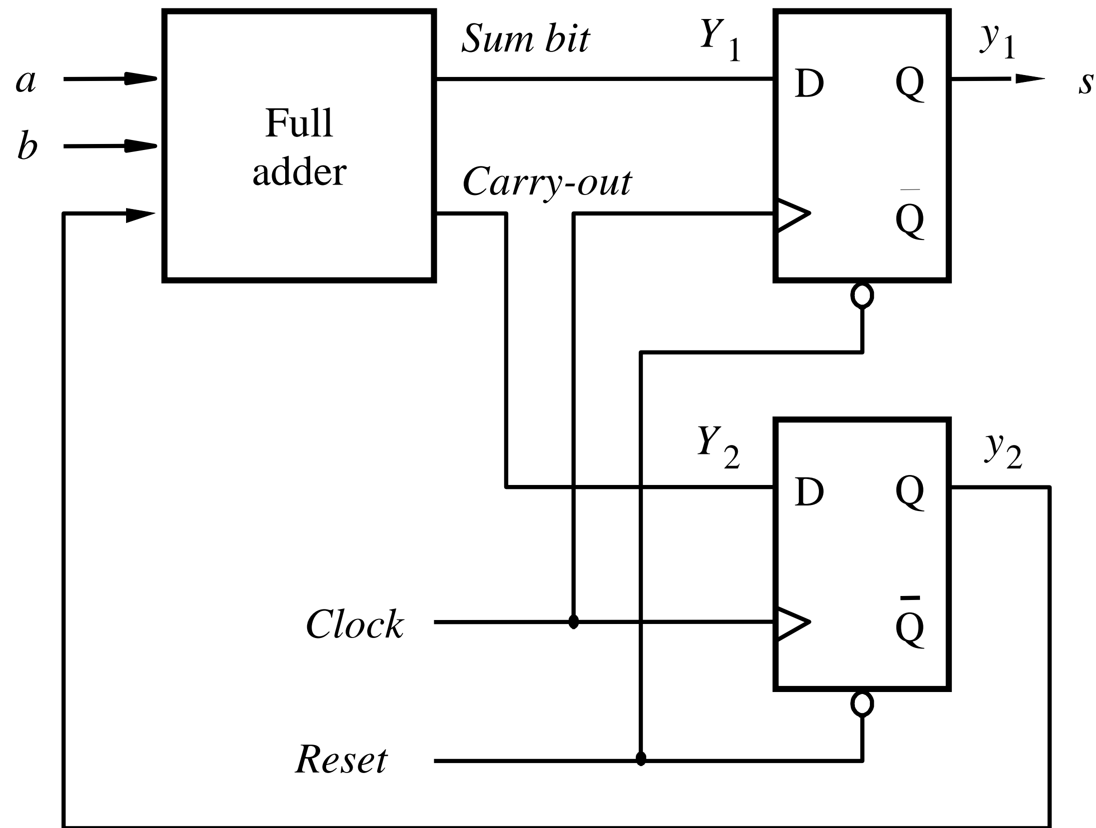
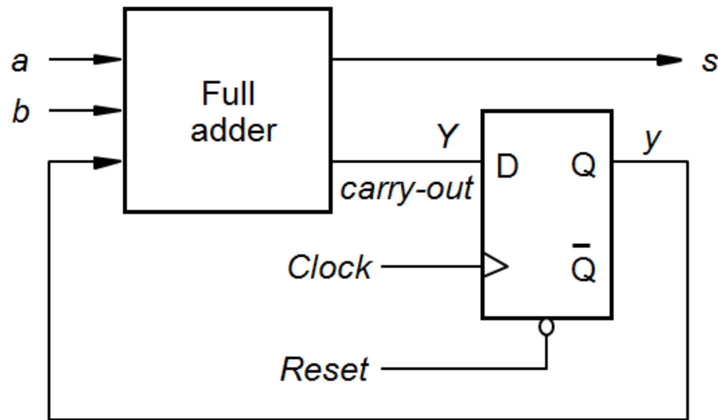
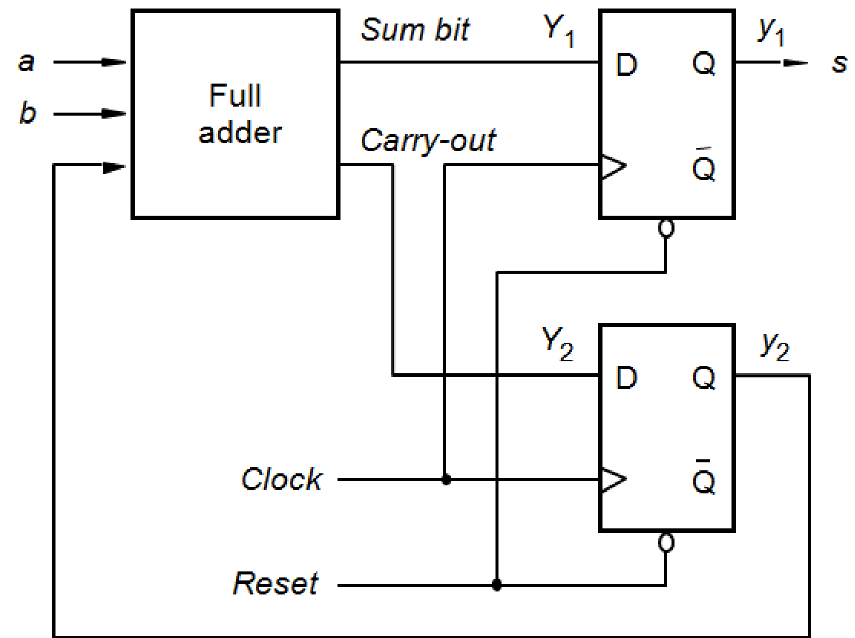


Figure 6.47. Circuit for the Moore-type serial adder FSM.

Mealy

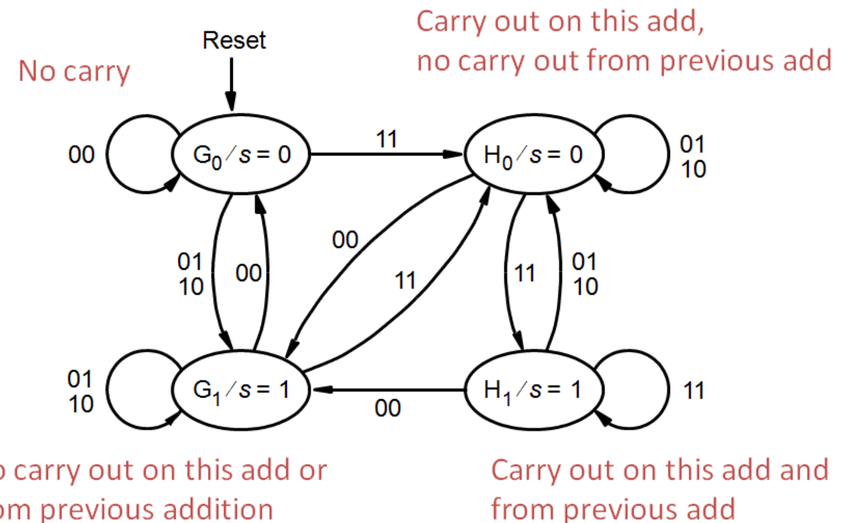


Moore



Mealy and Moore versions of the serial adder, side-by-side.

```
module MooreAdder( input clock, reset, a, b,
      output s );

   enum { G0, G1, H0, H1 } state;
   assign s = ( state == G1 || state == H1 );

   always @( posedge clock )
      if ( reset )
         state <= G0;
      else
         case ( { state, a, b } )
            { G0, 2'b00 }: state <= G0;
            { G0, 2'b01 }: state <= G1;
            { G0, 2'b10 }: state <= G1;
            { G0, 2'b11 }: state <= H0;

            { G1, 2'b00 }: state <= G0;
            { G1, 2'b01 }: state <= G1;
            { G1, 2'b10 }: state <= G1;
            { G1, 2'b11 }: state <= H0;

            { H0, 2'b00 }: state <= G1;
            { H0, 2'b01 }: state <= H1;
            { H0, 2'b10 }: state <= H1;
            { H0, 2'b11 }: state <= H1;

            { H1, 2'b00 }: state <= G1;
            { H1, 2'b01 }: state <= H0;
            { H1, 2'b10 }: state <= H0;
            { H1, 2'b11 }: state <= H1;
         endcase

endmodule
```



Equivalence of states

Two states S_i and S_j are said to be equivalent if and only if for every possible input sequence, the same output sequence will be produced regardless of whether S_i or S_j is the initial state.

Present state	Next state		Output z
	$w = 0$	$w = 1$	
A	B	C	1
B	D	F	1
C	F	E	0
D	B	G	1
E	F	C	0
F	E	D	0
G	F	G	0

Figure 6.51. Non-optimized state table for the speed controller that detects if speed is too high on 2+ successive samples.

Partition states first by output

Present state	Next state		Output z
	$w = 0$	$w = 1$	
A	B	C	1
B	D	F	1
C	F	E	0
D	B	G	1
E	F	C	0
F	E	D	0
G	F	G	0

Output z = 1

(A, B, D)

Output z = 0

(C, E, F, G)

If two states are in the same partition, it means it's still possible they're equivalent.

Two states with different outputs cannot be equivalent.

Next partition by 0-successors and 1-successors

Present state	Next state		Output
	$w = 0$	$w = 1$	z
A	B	C	1
B	D	F	1
C	F	E	0
D	B	G	1
E	F	C	0
F	E	D	0
G	F	G	0

Initial partition this iteration:
(A, B, D) (C, E, F, G)

New partition:
(A, B, D) (C, E, G) (F)

For (A, B, D),

0-successors are (B, D, B), all in same partition.

1-successors are (C, F, G), all also in the same partition.

→ (A, B, D) remain in same partition.

For (C, E, F, G),

0-successors are (F, F, E, F), all in same partition.

1-successors are (E, C, D, G), NOT in the same partition.

→ (C, E, G) are in one partition, (F) is in another.

Continue partitioning by 0-successors and 1-successors

Present state	Next state		Output
	$w = 0$	$w = 1$	z
A	B	C	1
B	D	F	1
C	F	E	0
D	B	G	1
E	F	C	0
F	E	D	0
G	F	G	0

Initial partition this iteration:
(A, B, D) (C, E, G) (F)

New partition:
(A, D) (B) (C, E, G) (F)

For (A, B, D),

0-successors are (B, D, B), all in same partition.

1-successors are (C, F, G) are different partitions

→ (A, B, D) split into (A, D)(B).

For (C, E, G),

0-successors are (F, F, F), all in same partition.

1-successors are (E, C, G), all in the same partition.

→ (C, E, G) remain in same partition

No more opportunities to split partitions.

Collapse the equivalent states:
(A, D) (B) (C, E, G) (F)

Present state	Next state $w = 0$	Next state $w = 1$	Output z
A	B	C	1
B	D	F	1
C	F	E	0
D	B	G	1
E	F	C	0
F	E	D	0
G	F	G	0

Present state	Next state $w = 0$	Next state $w = 1$	Output z
A	B	C	1
B	A ~~D~~	F	1
C	F	C ~~E~~	0
A ~~D~~	B	C ~~G~~	1
C ~~E~~	F	C	0
F	C ~~E~~	A ~~D~~	0
C ~~G~~	F	C ~~G~~	0

Replace D with A and both E & G with C in the original table and eliminate duplicate rows.

Original state table

Present state	Next state		Output z
	$w = 0$	$w = 1$	
A	B	C	1
B	A ~~D~~	F	1
C	F	C ~~E~~	0
A ~~D~~	B	C ~~G~~	1
C ~~E~~	F	C	0
F	C ~~E~~	A ~~D~~	0
C ~~G~~	F	C ~~G~~	0

Reduced state table after partitioning:
(A, D) (B) (C, E, G) (F)

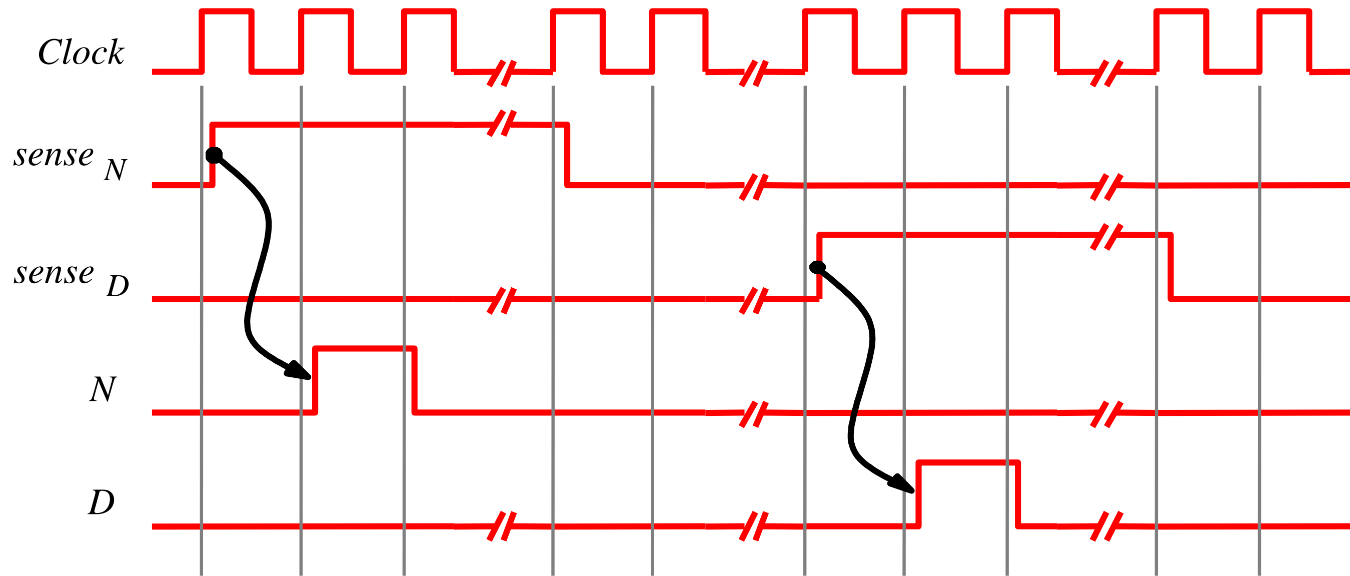
Present state	Nextstate		Output z
	w = 0	w = 1	
A	B	C	1
B	A	F	1
C	F	C	0
F	C	A	0

Figure 6.52. Minimized state table for the speed controller.

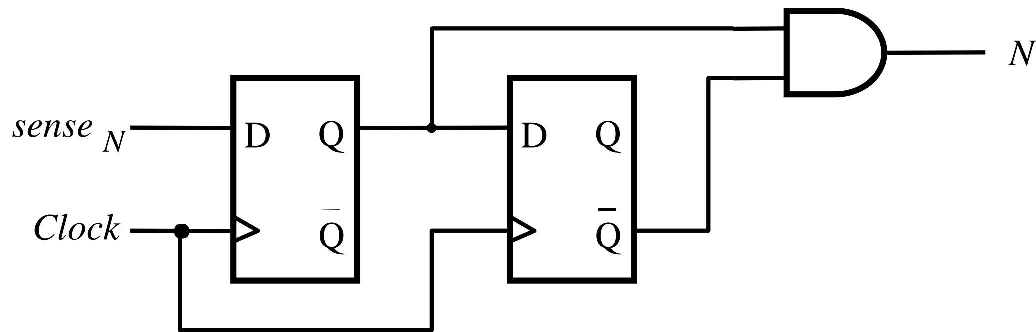
Example: A candy machine

Dispense candy for 15 cents.

If 20 cents deposited, dispense candy and credit 5 cents toward next purchase.



(a) Timing diagram



(b) Circuit that generates N

Figure 6.53. Candy machine coin sensors produce a 1-clock pulse.

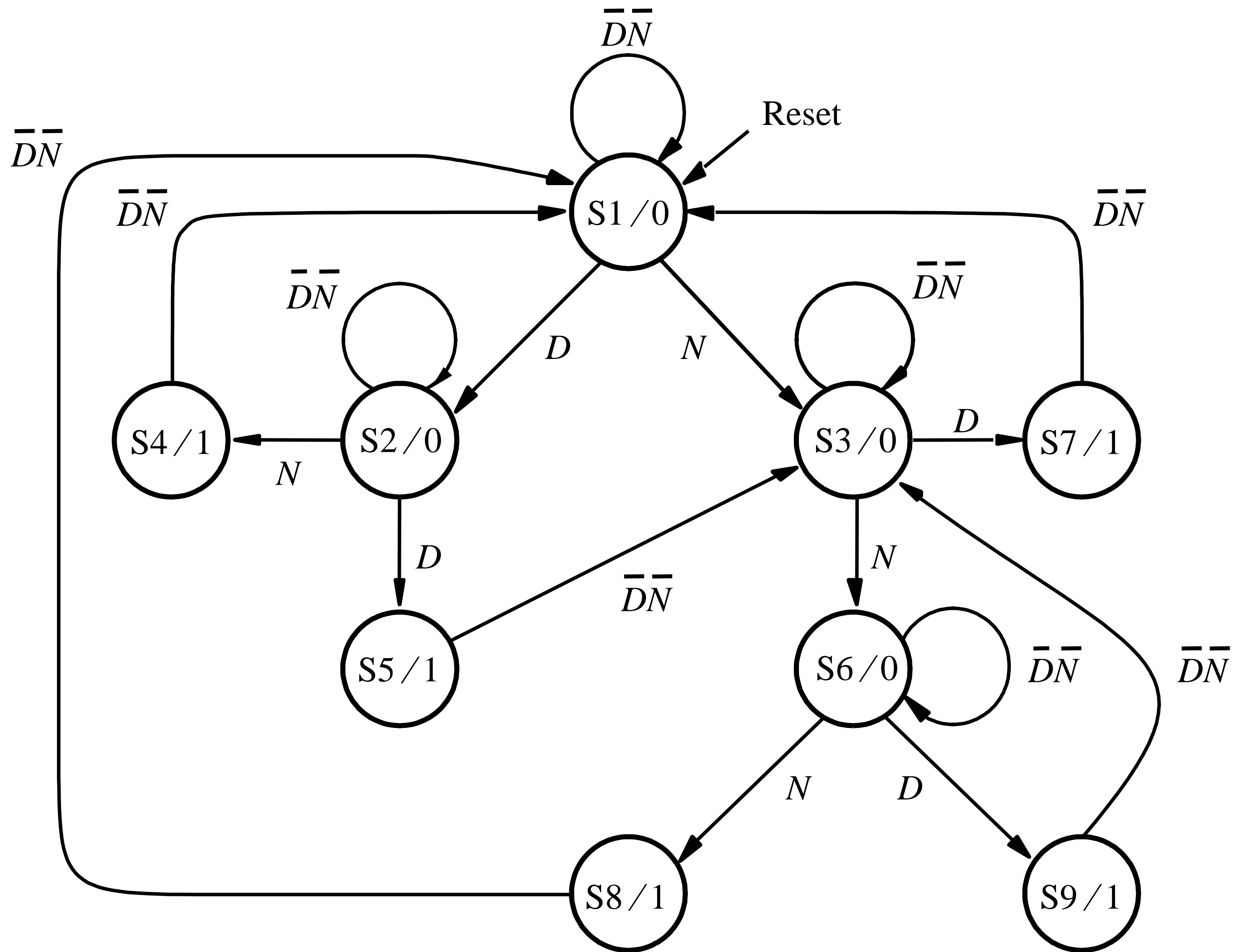


Figure 6.54. State diagram for the candy machine.

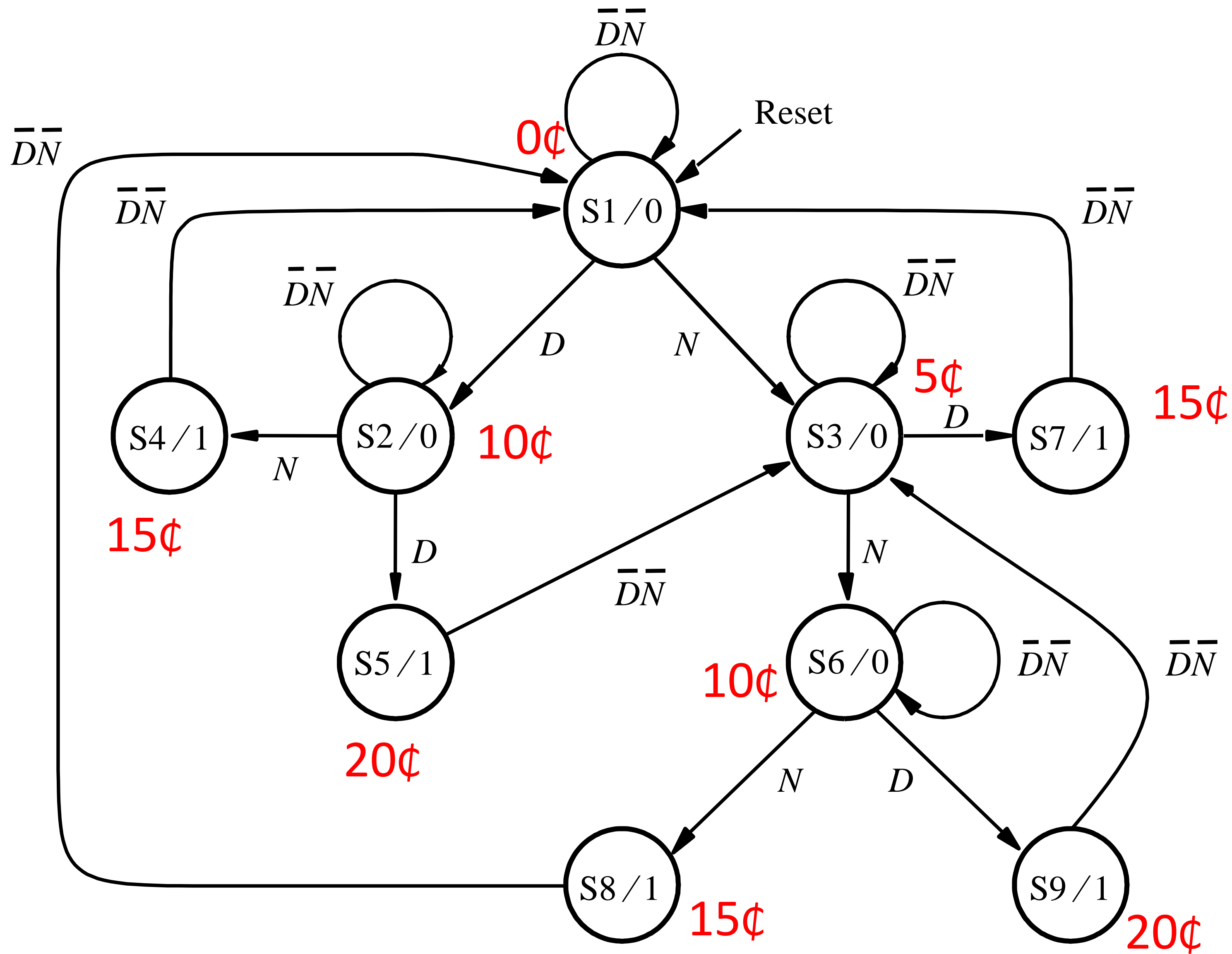
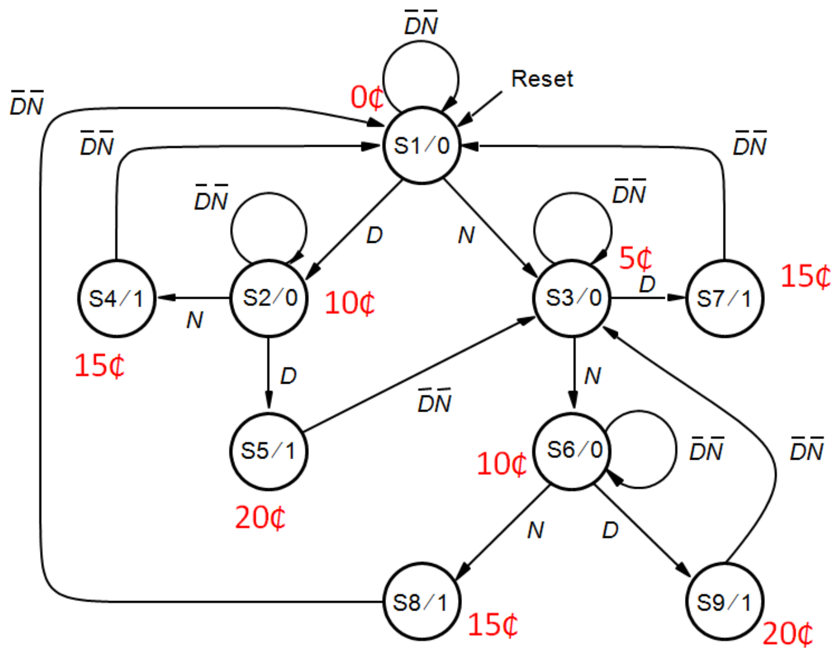


Figure 6.54. State diagram for the candy machine with amounts labeled to show the duplicates.



Present state	Next state DN =00	01	10	11	Output z
S1	S1	S3	S2	–	0
S2	S2	S4	S5	–	0
S3	S3	S6	S7	–	0
S4	S1	–	–	–	1
S5	S3	–	–	–	1
S6	S6	S8	S9	–	0
S7	S1	–	–	–	1
S8	S1	–	–	–	1
S9	S3	–	–	–	1

Don’t cares result from assumption that once enough money is deposited, we’ll go to the next state too fast for another coin can be inserted.

Figure 6.55. State table for the candy machine.

Partition states first by output

Present state	Next state				Output z
	DN =00	01	10	11	
S1	S1	S3	S2	–	0
S2	S2	S4	S5	–	0
S3	S3	S6	S7	–	0
S4	S1	–	–	–	1
S5	S3	–	–	–	1
S6	S6	S8	S9	–	0
S7	S1	–	–	–	1
S8	S1	–	–	–	1
S9	S3	–	–	–	1

Output z = 1

(S4, S5, S7, S8, S9)

Output z = 0

(S1, S2, S3, S6)

Next partition by successors

Present state	Next state				Output
	DN =00	01	10	11	*z*
S1	S1	S3	S2	–	0
S2	S2	S4	S5	–	0
S3	S3	S6	S7	–	0
S4	S1	–	–	–	1
S5	S3	–	–	–	1
S6	S6	S8	S9	–	0
S7	S1	–	–	–	1
S8	S1	–	–	–	1
S9	S3	–	–	–	1

Initial partition this iteration:
(S4, S5, S7, S8, S9) (S1, S2, S3, S6)

New partition:
(S4, S5, S7, S8, S9) (S1) (S2, S6) (S3)

For (S4, S5, S7, S8, S9),

00-successors are (S1, S3, S1, S1, S3).

01, 10 and 11 successors are (-, -, -, -, -).

→ (S4, S5, S7, S8, S9) remain in one partition.

For (S1, S2, S3, S6),

00-successors are (S1, S2, S3, S6), all in same partition.

01-successors are (S3, S4, S6, S8), NOT in the same partition.

10-successors are (S2, S5, S7, S9)

→ split into (S1) (S2, S6) (S3).

Once again by successors

Present state	Next state				Output
	DN =00	01	10	11	*z*
S1	S1	S3	S2	–	0
S2	S2	S4	S5	–	0
S3	S3	S6	S7	–	0
S4	S1	–	–	–	1
S5	S3	–	–	–	1
S6	S6	S8	S9	–	0
S7	S1	–	–	–	1
S8	S1	–	–	–	1
S9	S3	–	–	–	1

For (S4, S5, S7, S8, S9),

00-successors are (S1, S3, S1, S1, S3).

01, 10 and 11 successors are *don't care*.

→ split into (S4, S7, S8) (S5, S9)

For (S1) (S2, S6) (S3),

No further splits possible.

Initial partition this iteration:
(S4, S5, S7, S8, S9) (S1) (S2, S6) (S3)

New partition:
(S4, S7, S8) (S5, S9) (S1) (S2, S6) (S3)

Collapse the equivalent states:

(S4, S7, S8) (S5, S9) (S1) (S2, S6) (S3)

Present state	Next state				Output
	DN =00	01	10	11	*z*
S1	S1	S3	S2	–	0
S2	S2	S4	S5	–	0
S3	S3	S2 ~~S6~~	S4 ~~S7~~	–	0
S4	S1	–	–	–	1
S5	S3	–	–	–	1
S2 ~~S6~~	S6	S4 ~~S8~~	S5 ~~S9~~	–	0
S4 ~~S7~~	S1	–	–	–	1
S4 ~~S8~~	S1	–	–	–	1
S5 ~~S9~~	S3	–	–	–	1

Present state	Next state				Output
	DN =00	01	10	11	*z*
S1	S1	S3	S2	–	0
S2	S2	S4	S5	–	0
S3	S3	S2	S4	–	0
S4	S1	–	–	–	1
S5	S3	–	–	–	1

Figure 6.56. Minimized state table for the candy machine.

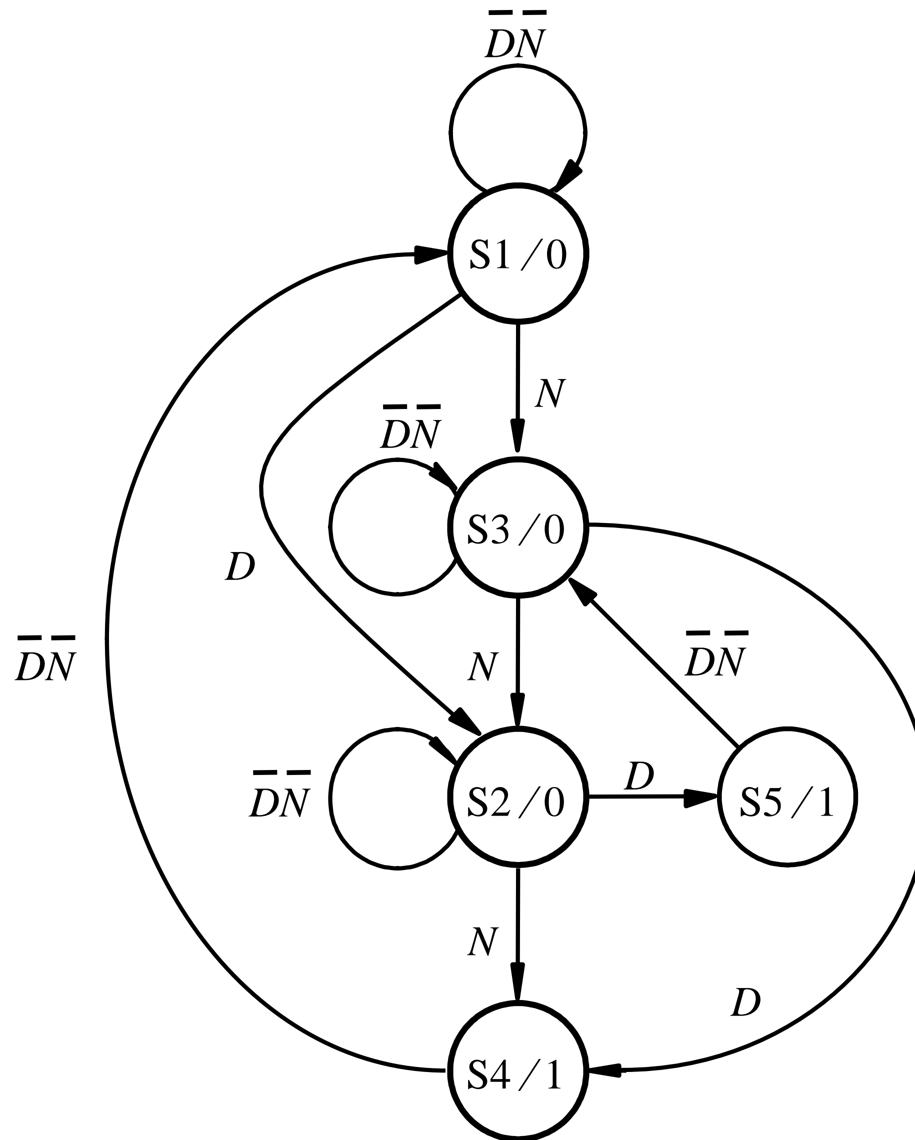


Figure 6.57. Minimized state diagram for the candy machine.

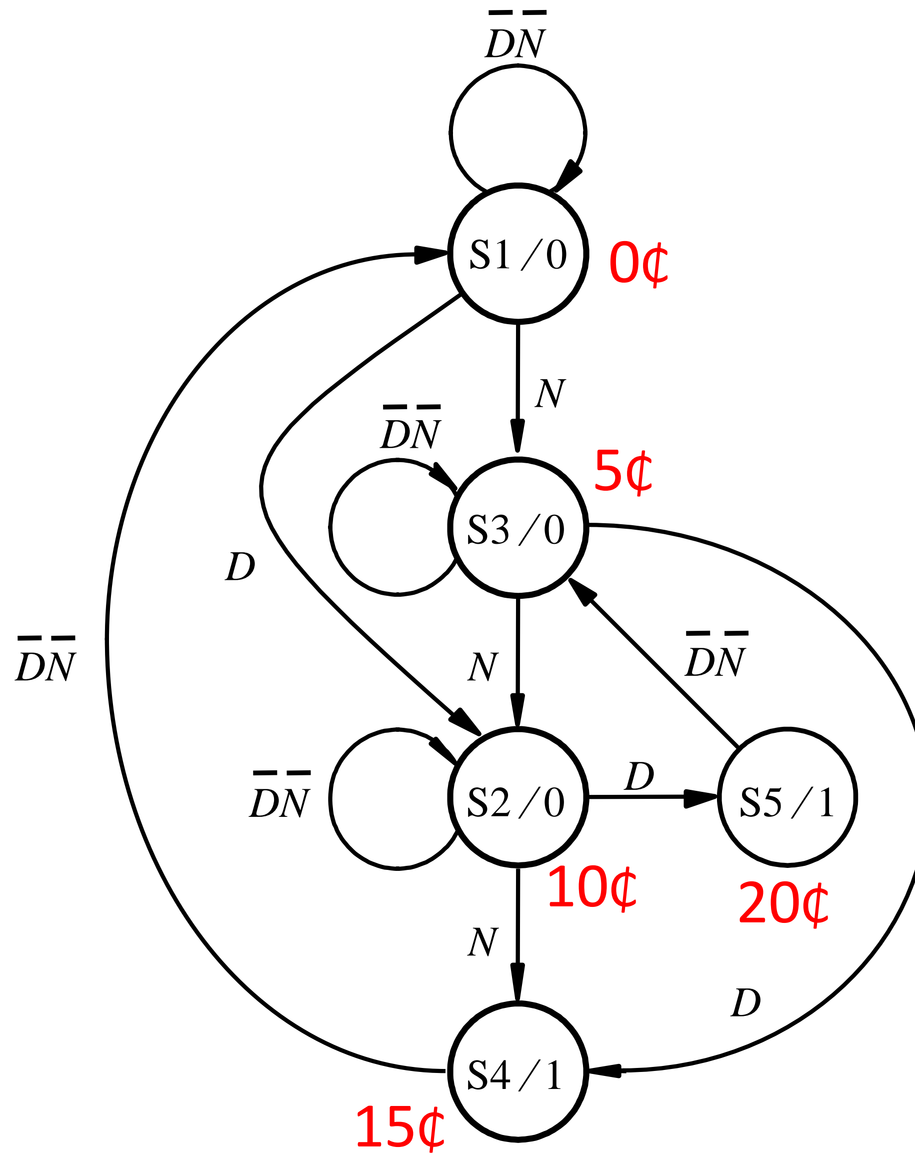


Figure 6.57. Minimized state diagram for the candy machine with amounts labeled.

Moore

$\overline{D}\,\overline{N}$

S1 / 0 0¢

N

$\overline{D}\,\overline{N}$

S3 / 0 5¢

D

$\overline{D}\,\overline{N}$

N

$\overline{D}\,\overline{N}$

$\overline{D}\,\overline{N}$

S2 / 0 D S5 / 1

10¢ 20¢

N

D

S4 / 1

15¢

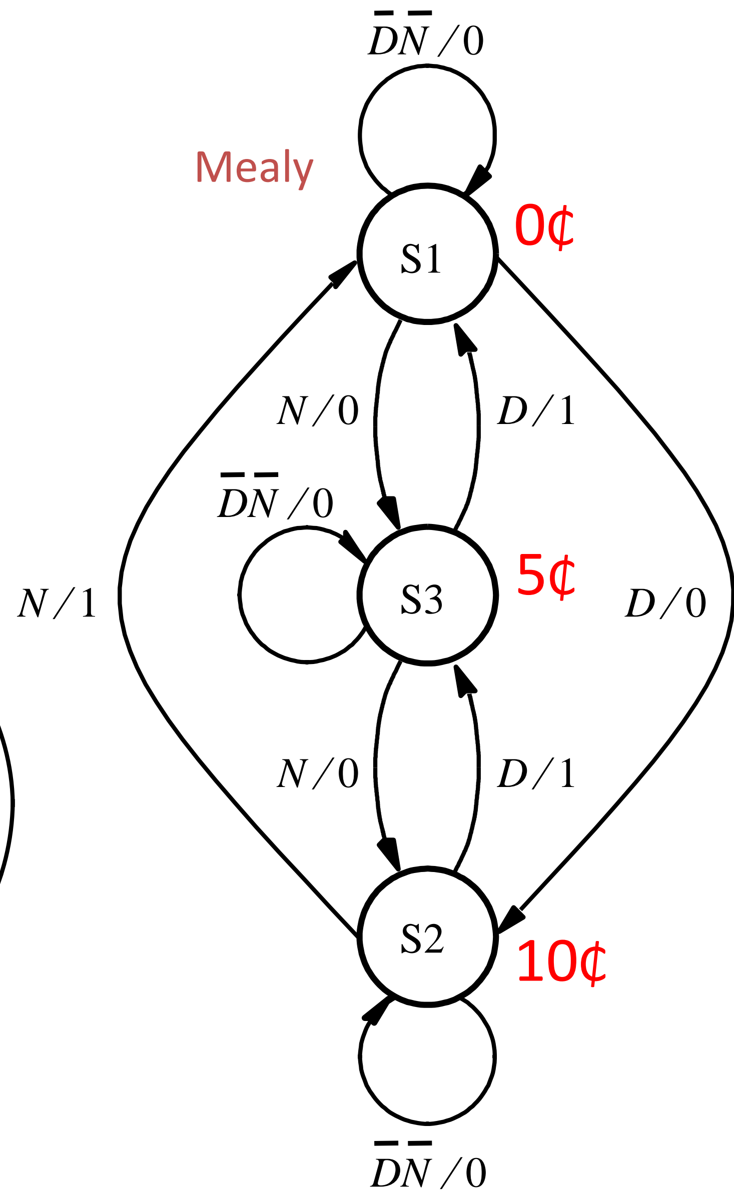


Figure 6.58. Mealy design for the candy machine eliminates states S4 and S5.

Incompletely specified FSMs

Partitioning works well when all the entries in the state table are *completely specified*.

But what if the table is *incompletely specified* with don’t care conditions?

Present state	Next state		Output z	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
A	B	C	0	0
B	D	–	0	–
C	F	E	0	1
D	B	G	0	0
E	F	C	0	1
F	E	D	0	1
G	F	–	0	–

Must guess whether the don't cares should be 1s or 0s.

Figure 6.59. Incompletely specified state table for the speed controller.

Present state	Next state		Output z	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
A	B	C	0	0
B	D	–	0	–
C	F	E	0	1
D	B	G	0	0
E	F	C	0	1
F	E	D	0	1
G	F	–	0	–

Assume the don't cares are 0s

Partition first by output:
(A, B, D, G) (C, E, F)

Then by successors:

Successors of (A, B, D, G):
0-successors are (B, D, B, F)
1-successors are (C, -, G, -)
→ Split into (A, B) (D) (G)

Successors of (C, E, F):
0-successors are (F, F, E)
1-successors are (E, C, D)
→Split into (C, E) (F)

Partitioning this iteration:
(A, B) (D) (G) (C, E) (F)

Present state	Next state		Output z	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
A	B	C	0	0
B	D	–	0	–
C	F	E	0	1
D	B	G	0	0
E	F	C	0	1
F	E	D	0	1
G	F	–	0	–

Initial partitioning this iteration:
(A, B) (D) (G) (C, E) (F)

Assume the don't cares are 0s

Partitioning by successors:

Successors of (A, B):
0-successors are (B, D)
1-successors are (C, -)
→ Split into (A) (B)

Successors of (C, E):
0-successors are (F, F)
1-successors are (E, C)
→Not split

Final partitioning:
(A) (B) (D) (G) (C, E) (F)

Present state	Next state		Output z	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
A	B	C	0	0
B	D	–	0	–
C	F	E	0	1
D	B	G	0	0
E	F	C	0	1
F	E	D	0	1
G	F	–	0	–

Assume the don't cares are 1s

Partition first by output:
(A, D) (B, C, E, F, G)

Then by successors:

Successors of (A, D):
0-successors are (B, B)
1-successors are (C, G)
→ Not split

Successors of (B, C, E, F, G):
0-successors are (D, F, F, E, F)
1-successors are (-, E, C, D, -)
→Split into (B) (C, E, G) (F)

Final partitioning:
(A, D) (B) (C, E, G) (F)

Present state	Next state		Output z	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
A	B	C	0	0
B	D	–	0	–
C	F	E	0	1
D	B	G	0	0
E	F	C	0	1
F	E	D	0	1
G	F	–	0	–

Final partitioning

Assuming the don't cares are 0s:
(A) (B) (D) (G) (C, E) (F) = 6 states

Assuming the don't cares are 1s:
(A, D) (B) (C, E, G) (F) = 4 states

Could continue by assuming the don't cares differ.

It obviously matters what we assume.

But reducing the number of states may not be as important as choosing good state assignments.

Example: Modulo-8 counter

It also matters what type of flip-flops you choose.

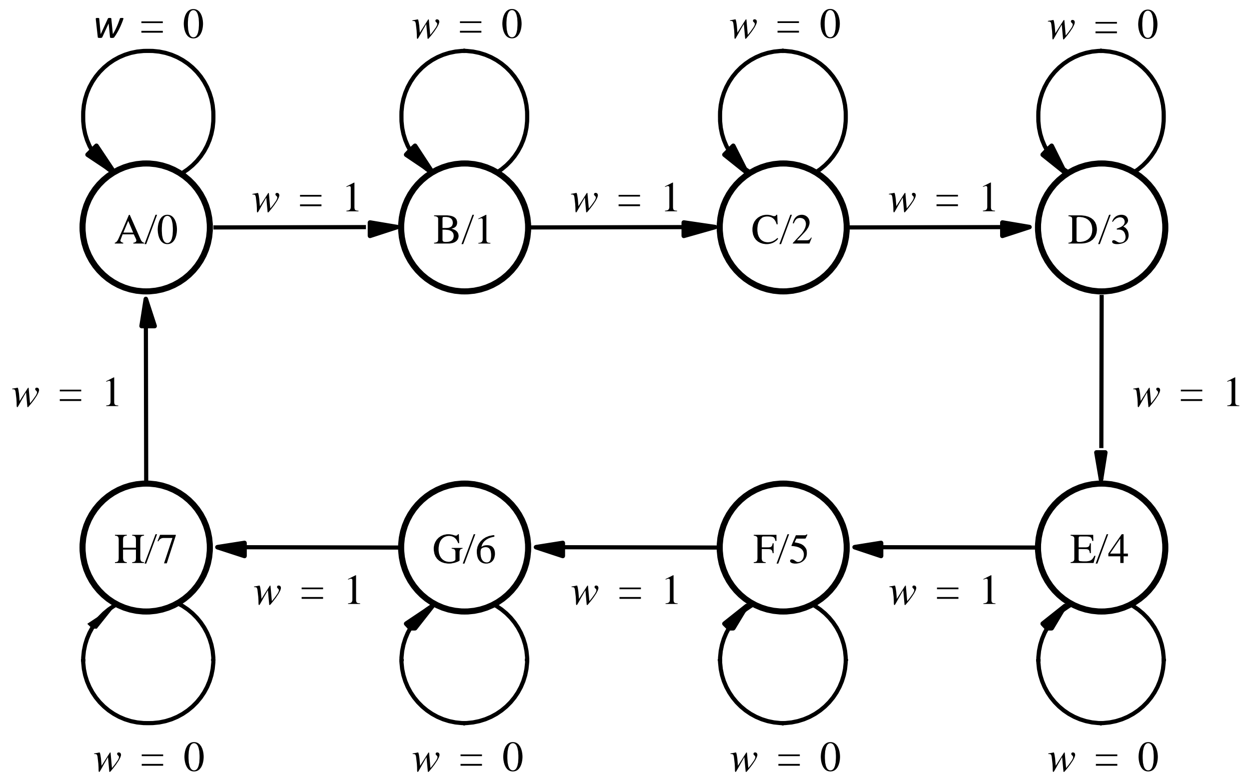
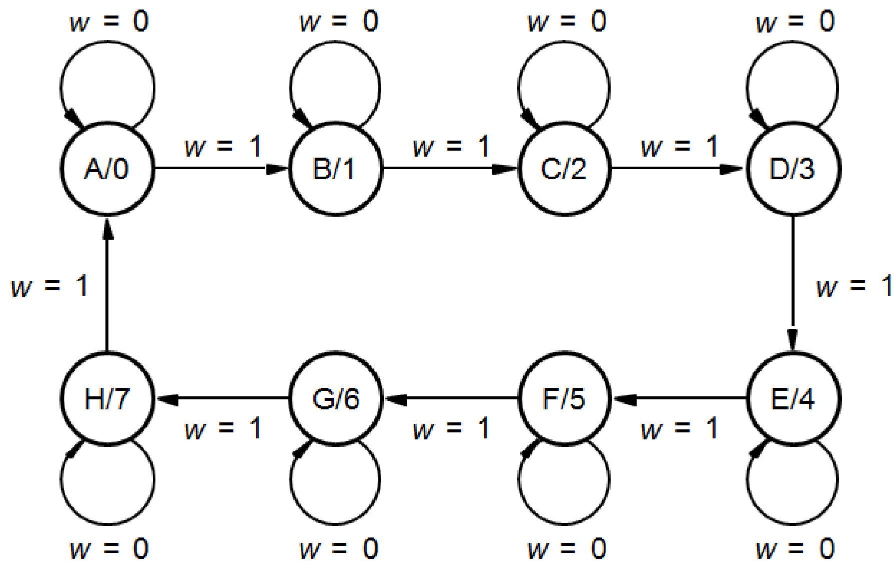
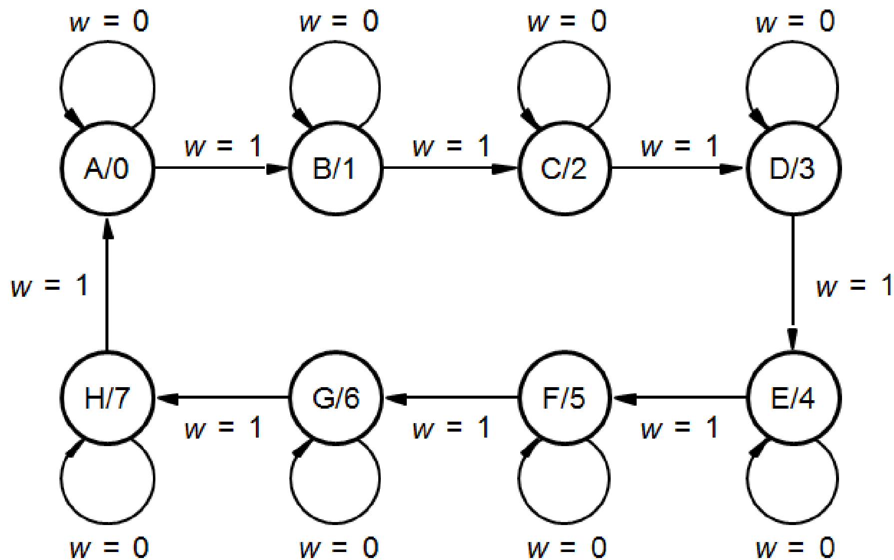


Figure 6.60. State diagram for a modulo 8 counter.



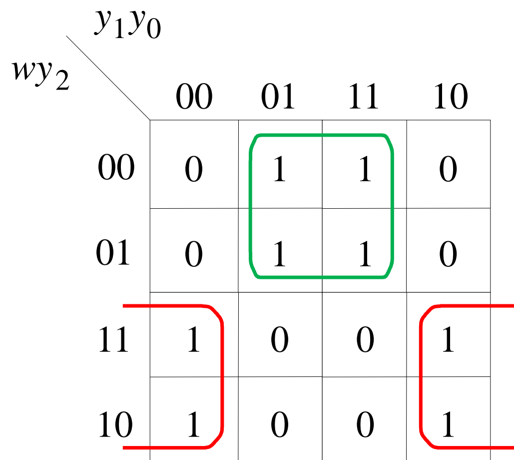
Present state	Next state		Output
	$w = 0$	$w = 1$	
A	A	B	0
B	B	C	1
C	C	D	2
D	D	E	3
E	E	F	4
F	F	G	5
G	G	H	6
H	H	A	7

Figure 6.61. State table for the counter.

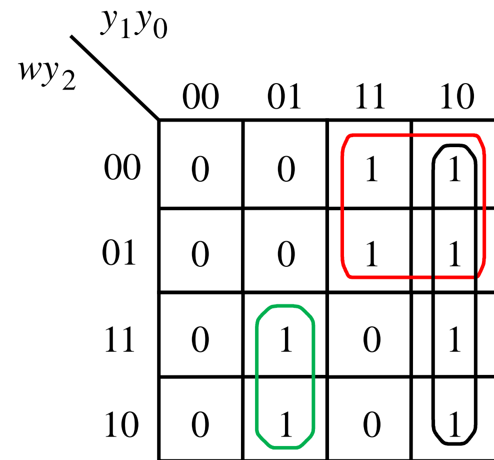


	Present state $y_2y_1y_0$	Next state $w = 0$ $Y_2Y_1Y_0$	Next state $w = 1$ $Y_2Y_1Y_0$	Count $z_2z_1z_0$
A	000	000	001	000
B	001	001	010	001
C	010	010	011	010
D	011	011	100	011
E	100	100	101	100
F	101	101	110	101
G	110	110	111	110
H	111	111	000	111

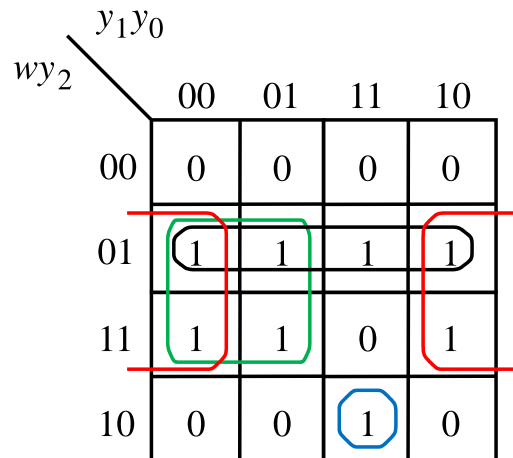
Figure 6.62. State-assigned table for the counter.



$$Y_0 = \bar{w} y_0 + w \bar{y}_0$$



$$Y_1 = \bar{w} y_1 + y_1 \bar{y}_0 + w y_0 \bar{y}_1$$



$$Y_2 = \bar{w} y_2 + \bar{y}_0 y_2 + \bar{y}_1 y_2 + w y_0 y_1 \bar{y}_2$$

Figure 6.63. Karnaugh maps for D flip-flops for the counter.

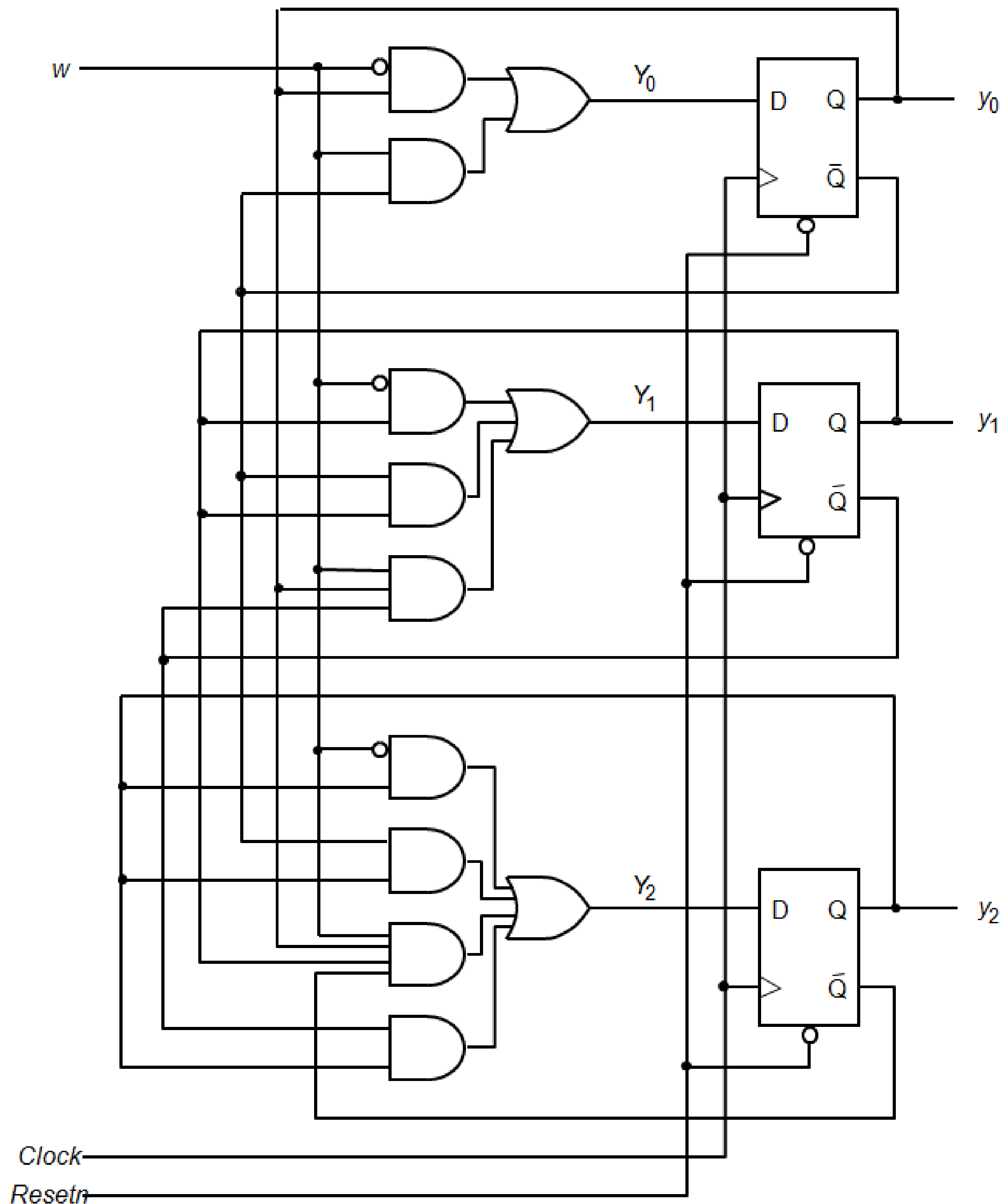
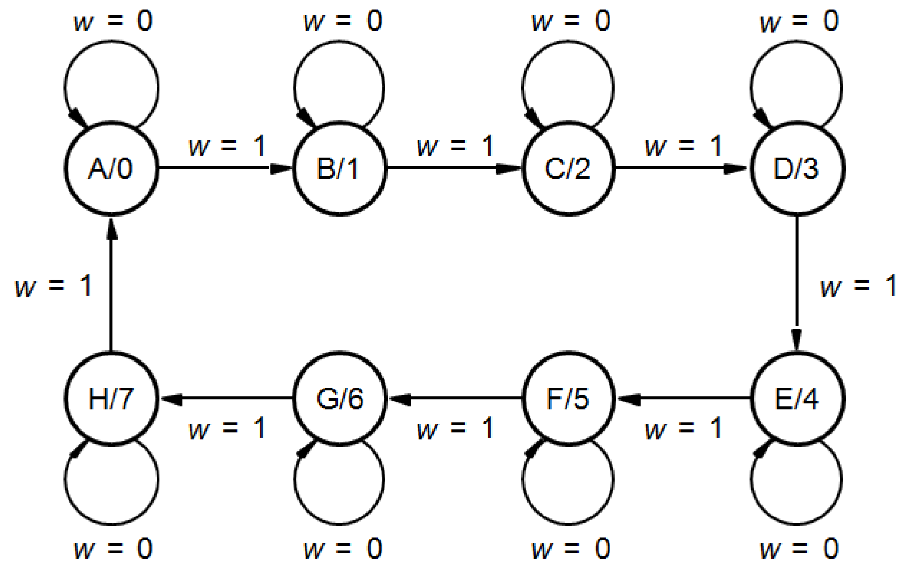


Figure 6.64. Circuit diagram for the counter implemented with D flip-flops.



	Present state $y_2y_1y_0$	Flip-flop inputs								Count $z_2z_1z_0$
		$w = 0$				$w = 1$				
		$Y_2Y_1Y_0$	J_2K_2	J_1K_1	J_0K_0	$Y_2Y_1Y_0$	J_2K_2	J_1K_1	J_0K_0	
A	000	000	0d	0d	0d	001	0d	0d	1d	000
B	001	001	0d	0d	d0	010	0d	1d	d1	001
C	010	010	0d	d0	0d	011	0d	d0	1d	010
D	011	011	0d	d0	d0	100	1d	d1	d1	011
E	100	100	d0	0d	0d	101	d0	0d	1d	100
F	101	101	d0	0d	d0	110	d0	1d	d1	101
G	110	110	d0	d0	0d	111	d0	d0	1d	110
H	111	111	d0	d0	d0	000	d1	d1	d1	111

Figure 6.65. Excitation table for the counter with JK flip-flops.

wy_2 \ y_1y_0	00	01	11	10
00	0	d	d	0
01	0	d	d	0
11	1	d	d	1
10	1	d	d	1

$J_0 = w$

wy_2 \ y_1y_0	00	01	11	10
00	d	0	0	d
01	d	0	0	d
11	d	1	1	d
10	d	1	1	d

$K_0 = w$

wy_2 \ y_1y_0	00	01	11	10
00	0	0	d	d
01	0	0	d	d
11	0	1	d	d
10	0	1	d	d

$J_1 = wy_0$

wy_2 \ y_1y_0	00	01	11	10
00	d	d	0	0
01	d	d	0	0
11	d	d	1	0
10	d	d	1	0

$K_1 = wy_0$

Figure 6.66. Karnaugh maps for JK flip-flops in the counter.

wy_2 \ y_1y_0	00	01	11	10
00	0	0	0	0
01	d	d	d	d
11	d	d	d	d
10	0	0	1	0

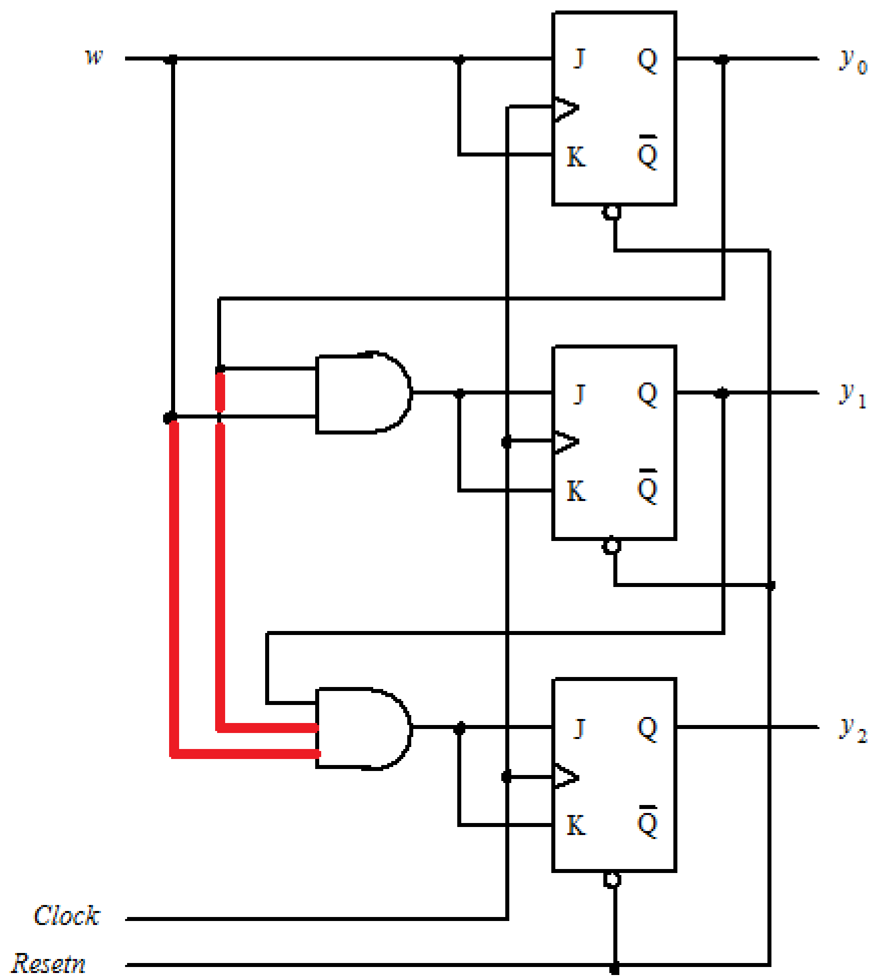
$J_2 = wy_0y_1$

wy_2 \ y_1y_0	00	01	11	10
00	d	d	d	d
01	0	0	0	0
11	0	0	1	0
10	d	d	d	d

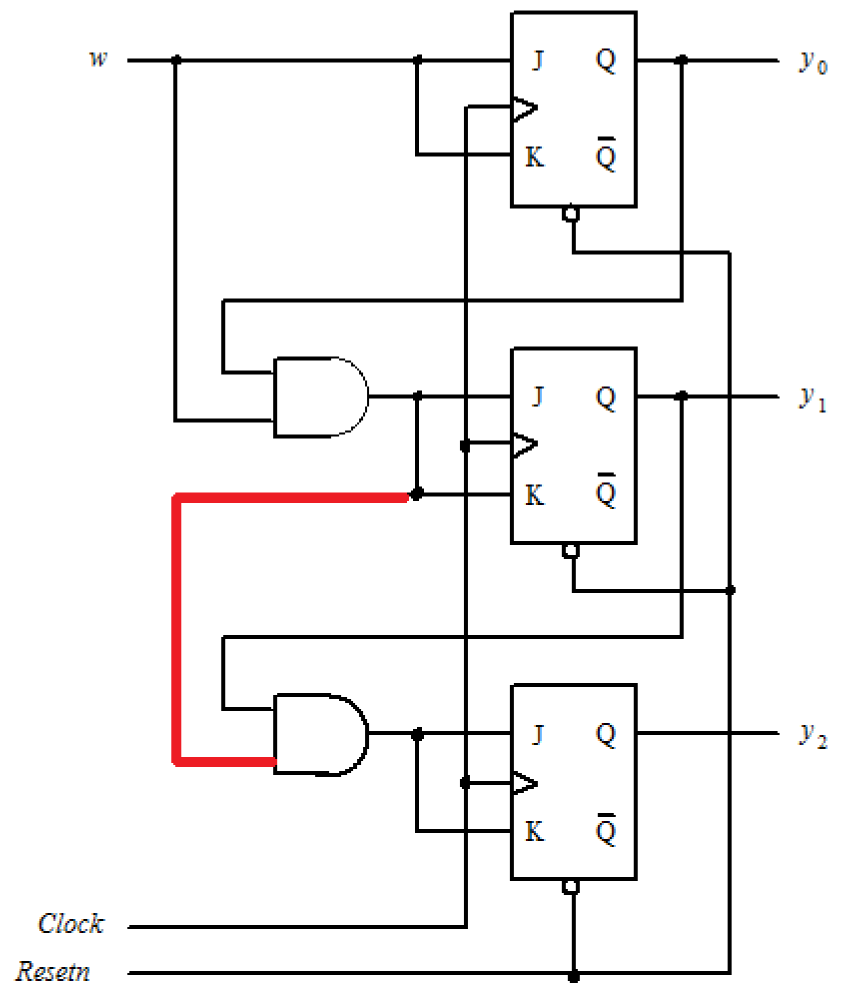
$K_2 = wy_0y_1$

Figure 6.66. Karnaugh maps for JK flip-flops in the counter.

Figure 6.67. Circuit diagram using JK flip-flops.



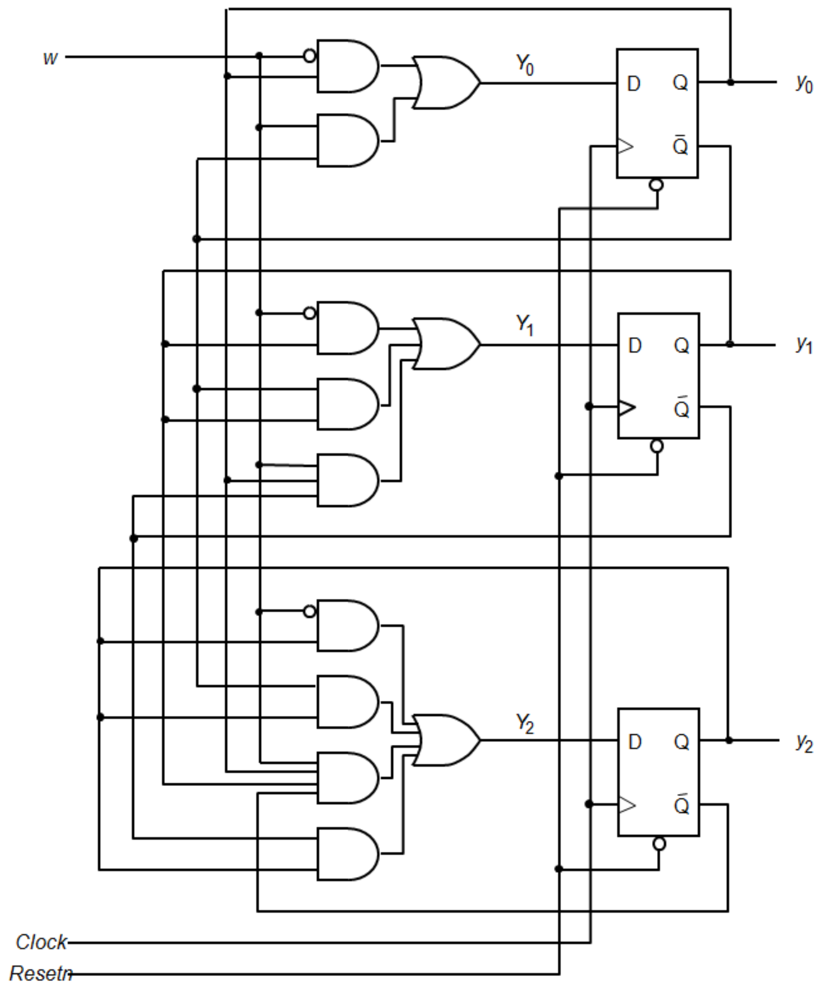
Original

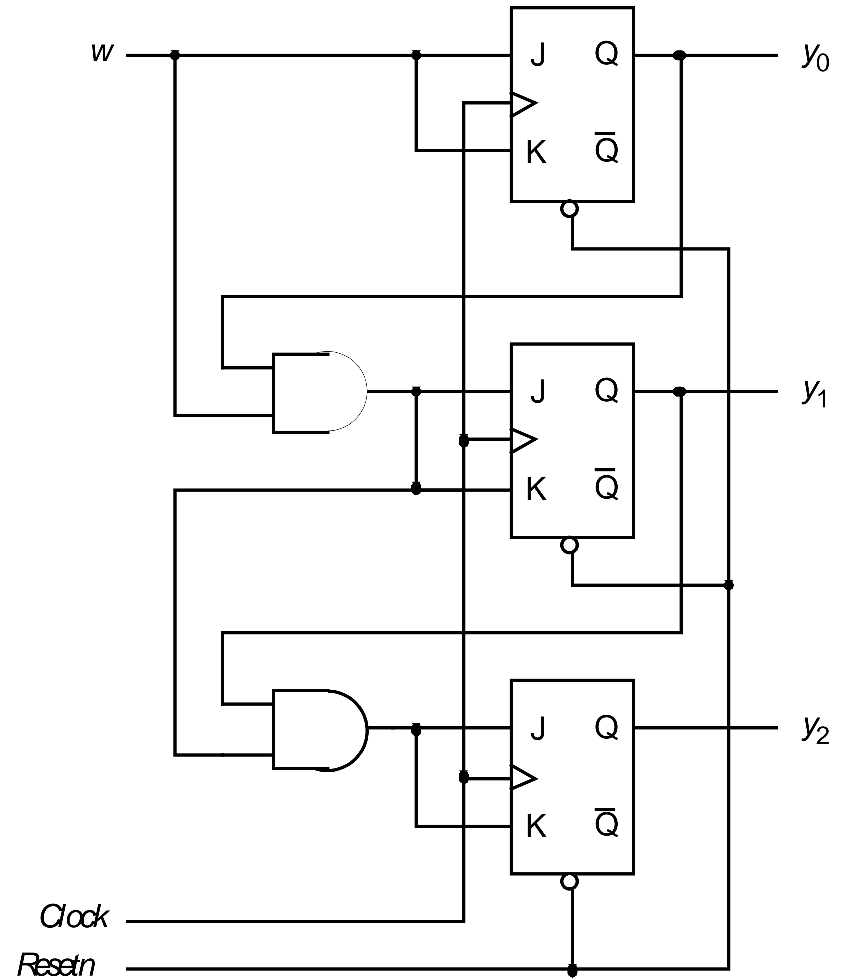


Factored

y1 y0 w = y1 (y0 w)

Figure 6.68. Factored-form implementation of the counter.





D and JK flip-flop solutions side-by-side.

Example: A unusual counter

Instead of 0, 1, 2, 3, 4, 5, 6, 7, 8, …, it counts 0, 4, 2, 6, 1, 5, 3, 7, …

Count 0, 4, 2, 6, 1, 5, 3, 7, …

Present state	Next state	Output $z_2 z_1 z_0$
A	B	000
B	C	100
C	D	010
D	E	110
E	F	001
F	G	101
G	H	011
H	A	111

Do you recognize this sequence?

What if the bits were reversed?

Figure 6.69. State table for an unusual counter.

Present state	Next state	Output $z_2z_1z_0$
A	B	000
B	C	100
C	D	010
D	E	110
E	F	001
F	G	101
G	H	011
H	A	111

Present state $y_2y_1y_0$	Next state $Y_2Y_1Y_0$	Output $z_2z_1z_0$
000	100	000
100	010	100
010	110	010
110	001	110
001	101	001
101	011	101
011	111	011
111	000	111

Figure 6.70. State-assigned table for the counter.

Present state $y_2y_1y_0$	Next state $Y_2Y_1Y_0$	Output $z_2z_1z_0$
000	100	000
100	010	100
010	110	010
110	001	110
001	101	001
101	011	101
011	111	011
111	000	111

D2 = Y2 = y2’

D1 = Y1 = y1 ^ y2

D0 = Y0 = y0 y1’ + y0 y2’ + y0’ y1 y2
= y0 (y1’ + y2’) + y0’ y1 y2
= y0 ^ y1 y2

$D2 = Y2 = y2'$

$D1 = Y1 = y1 \wedge y2$

$D0 = Y0 = y0\ y1' + y0\ y2' + y0'\ y1\ y2$
$= y0\ (y1' + y2') + y0'\ y1\ y2$
$= y0 \wedge y1\ y2$

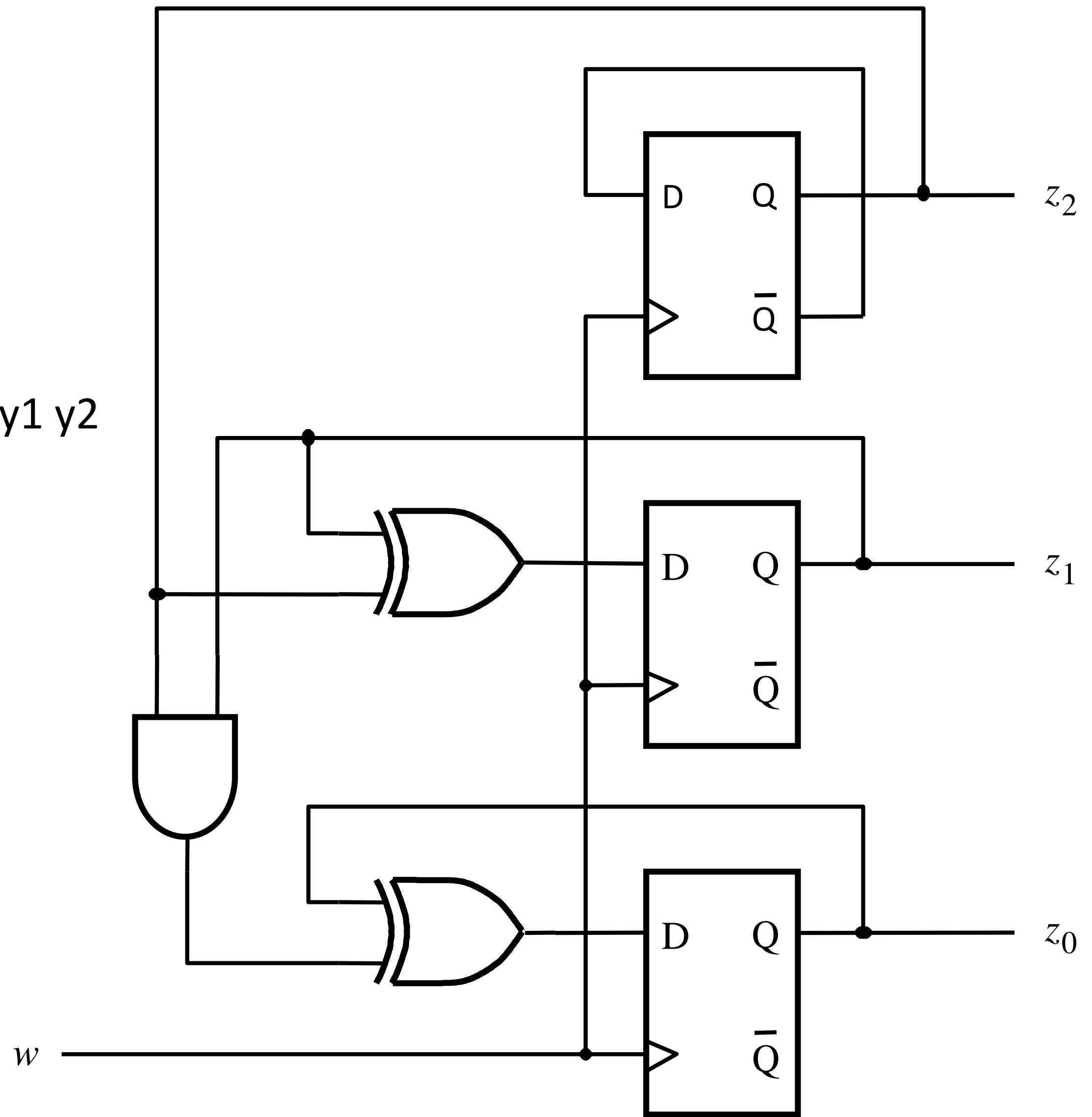
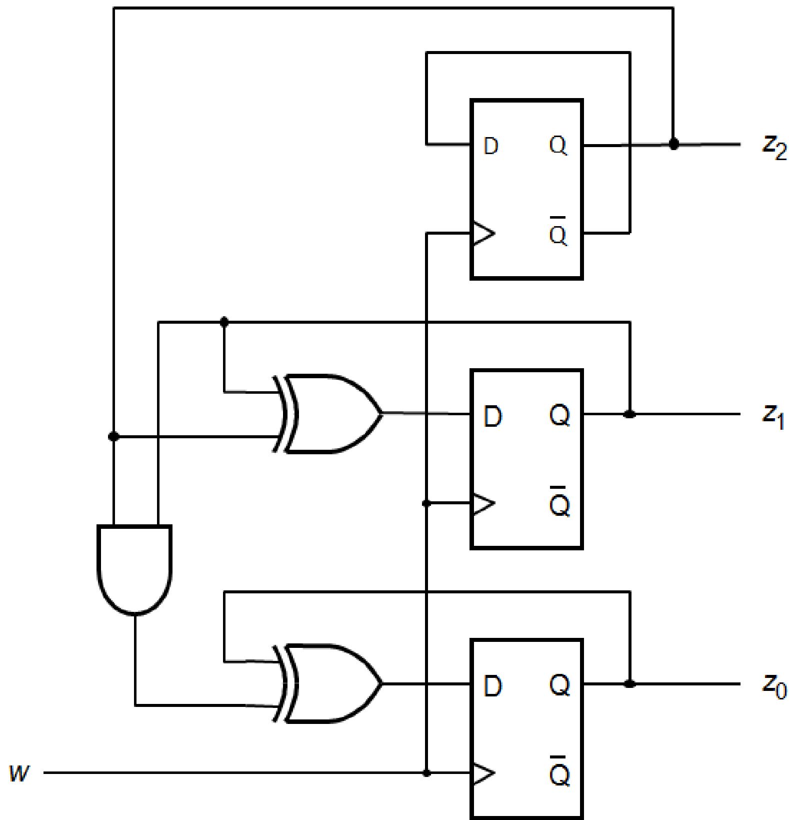
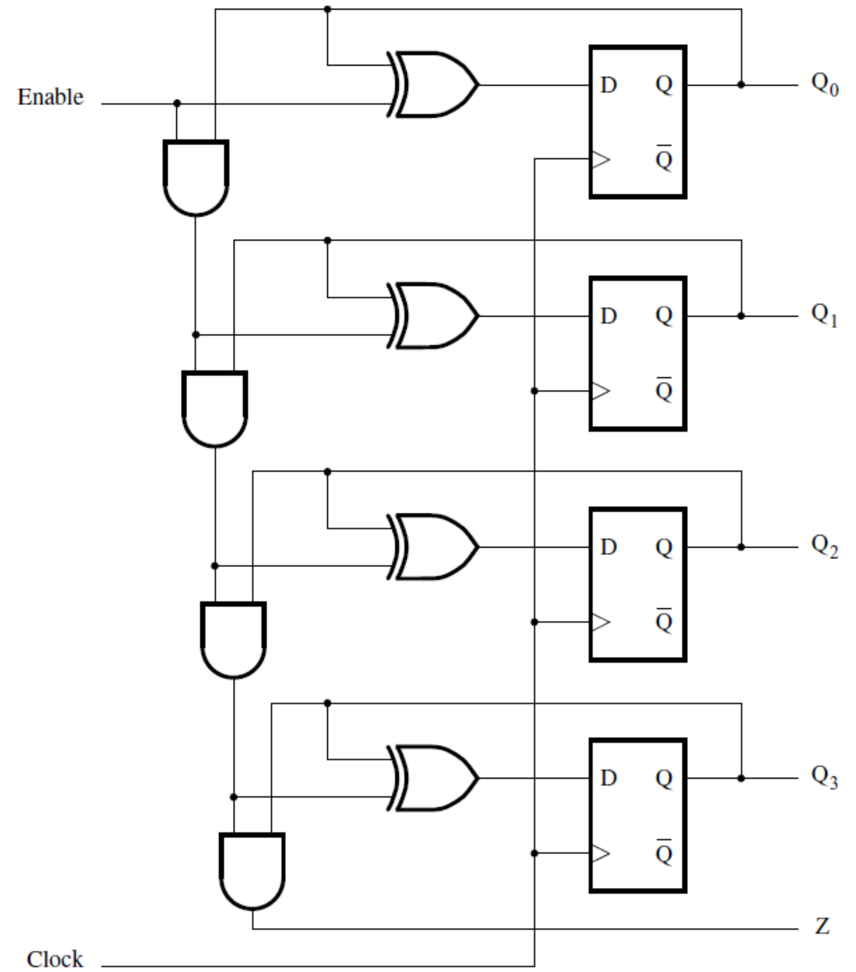


Figure 6.71. Circuit for an unusual counter.





Comparison to an ordinary up-counter.

Example: An arbiter

Grant access to highest priority request.

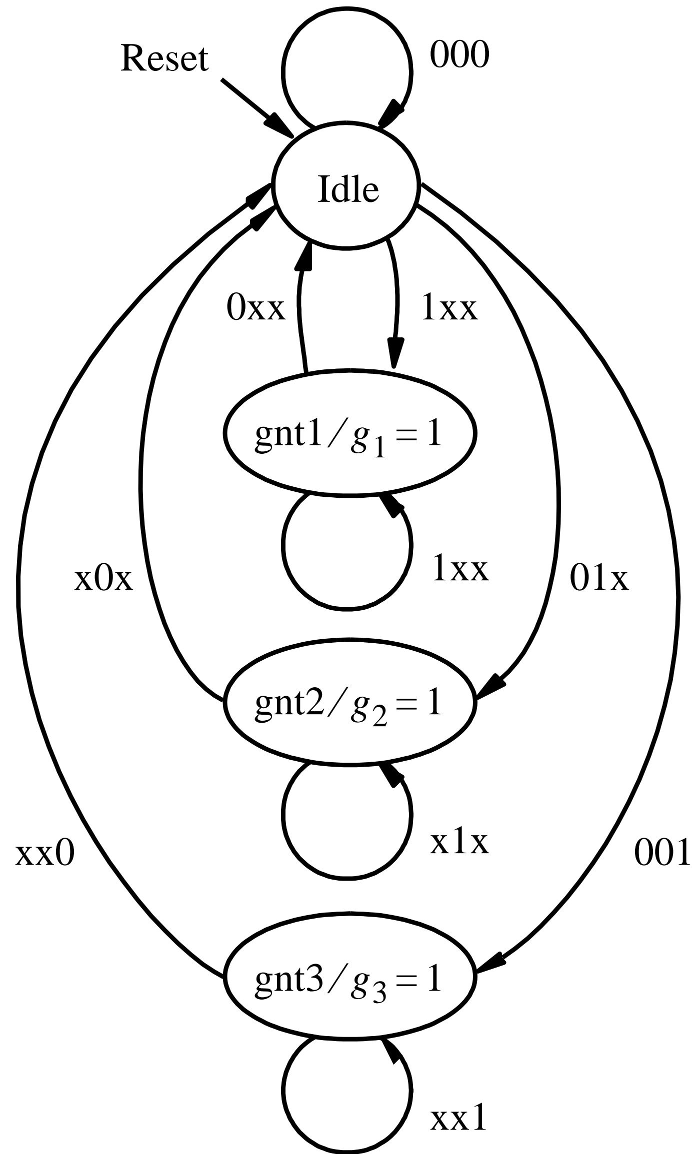


Figure 6.72. State diagram for the arbiter.

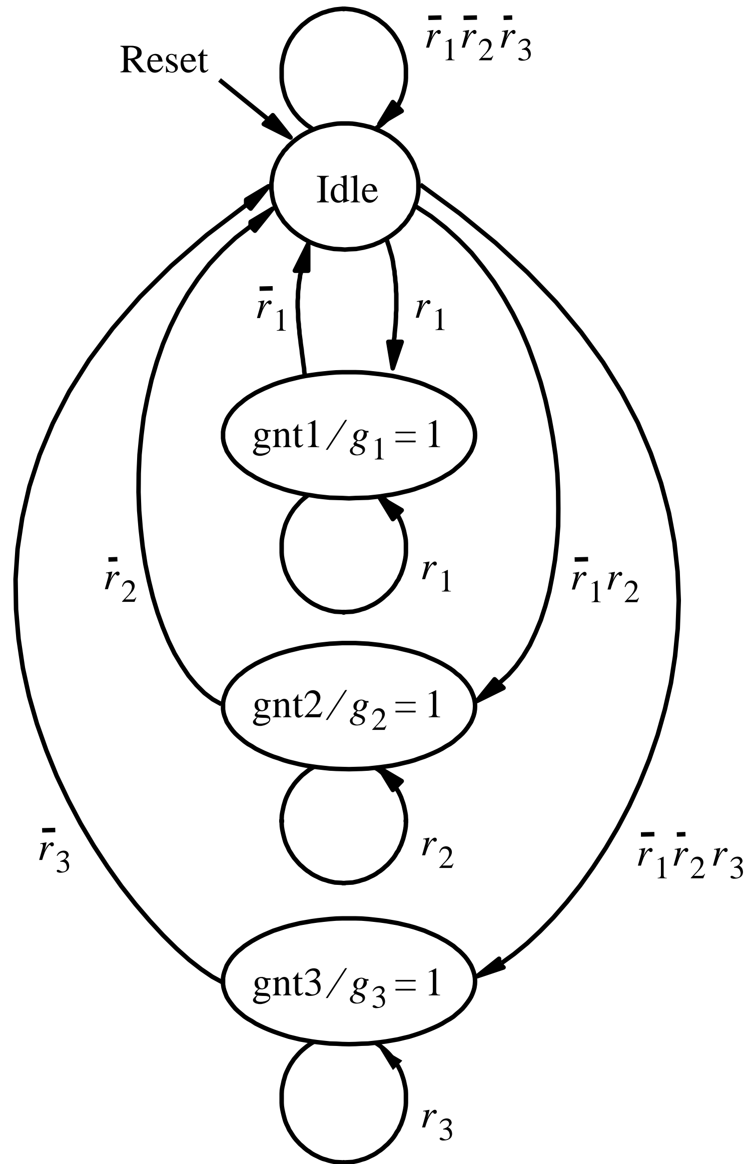
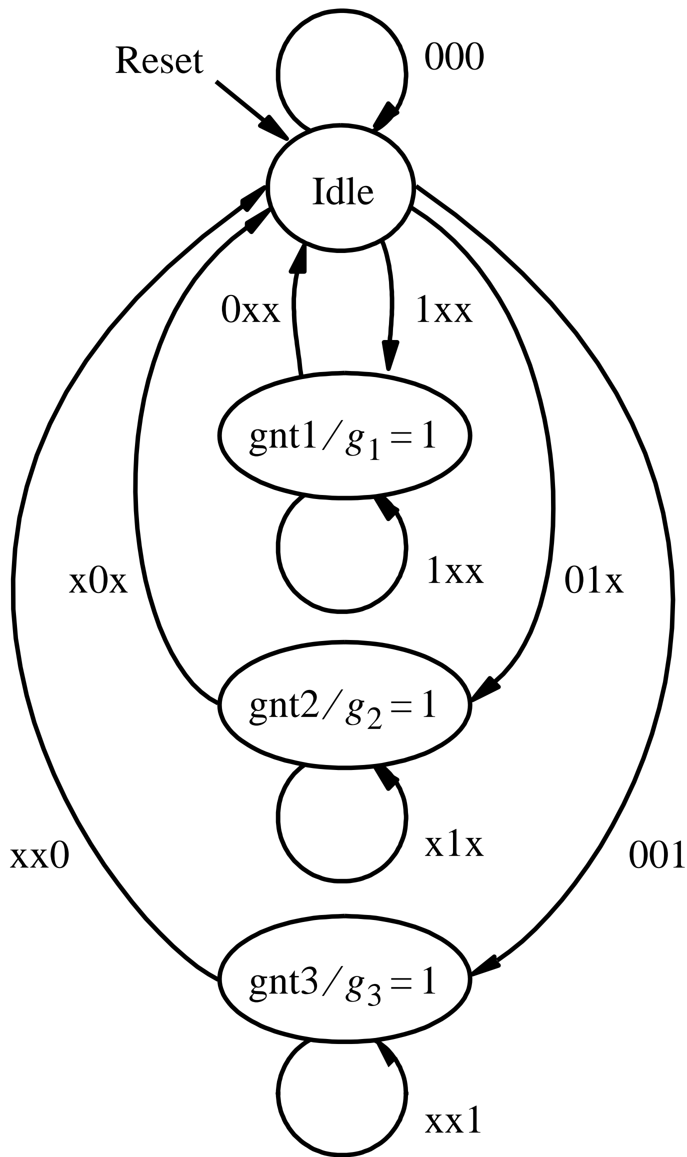


Figure 6.73. Alternative style of state diagram for the arbiter.



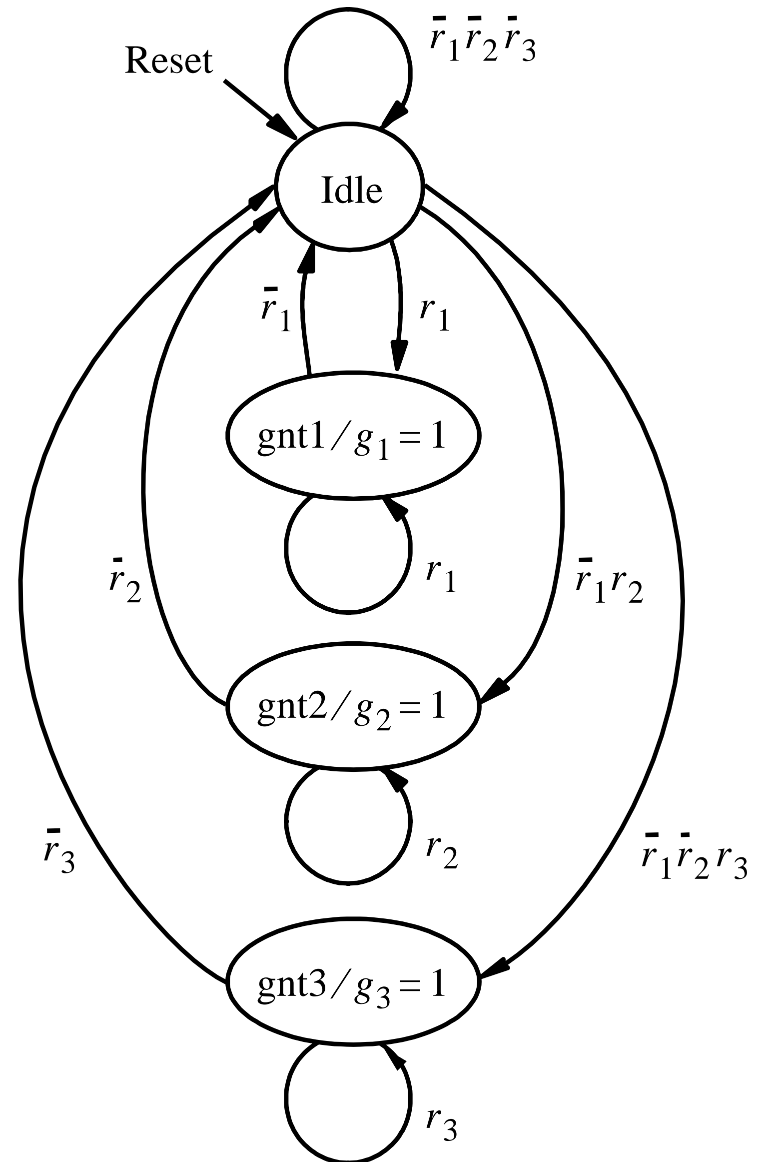
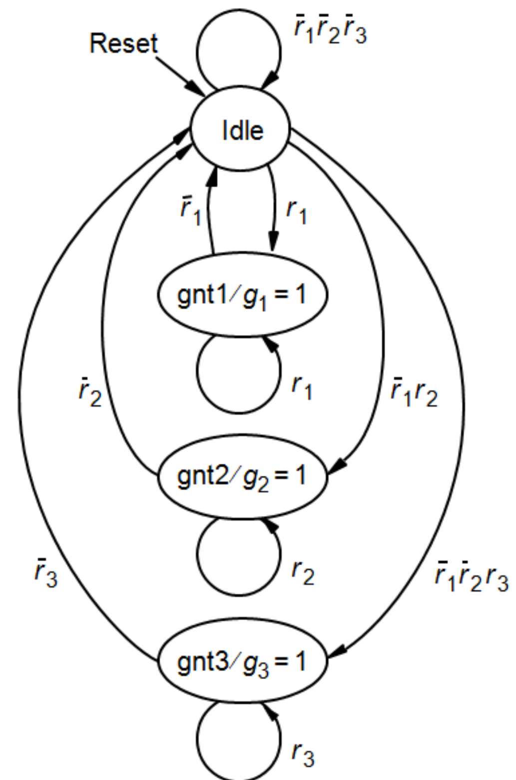


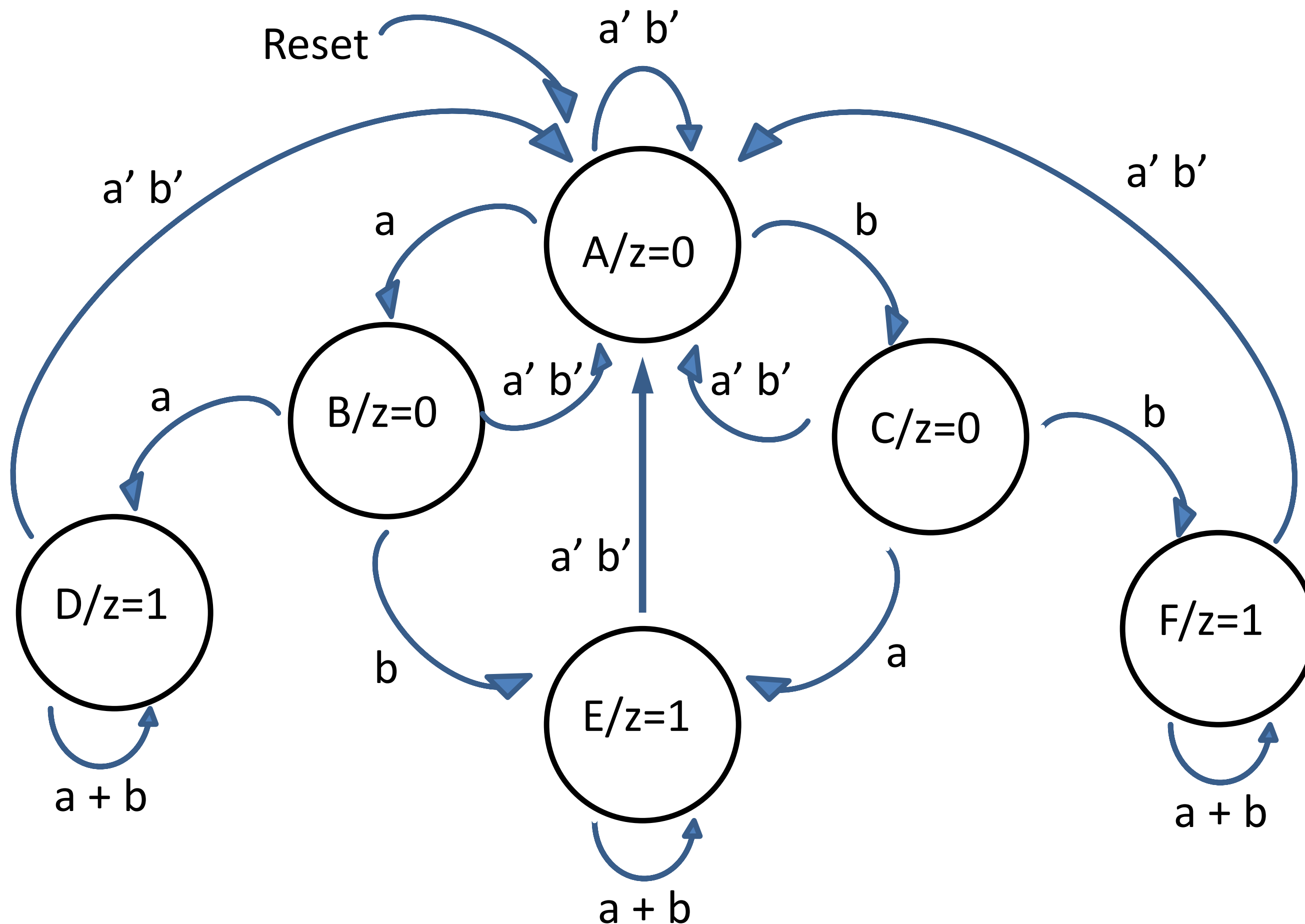
Figure 6.73. Alternative style of state diagram for the arbiter.

```
module Arbiter( input [ 1:3 ] r,
     input reset, clock,
     output wire [ 1:3 ] g );
     reg [ 2:1 ] y, Y;

   parameter Idle = 2'b00, gnt1 = 2'b01,
      gnt2 = 2'b10, gnt3 = 2'b11;

   // Next state combinational circuit
   always @( * )
      case ( y )
         Idle: casex ( r )
                  3'b000: Y = Idle;
                  3'b1xx: Y = gnt1;
                  3'b01x: Y = gnt2;
                  3'b001: Y = gnt3;
                  default: Y = Idle;
               endcase
         gnt1: if ( r[ 1 ] )
                  Y = gnt1;
               else
                  Y = Idle;
         gnt2: if ( r[ 2 ] )
                  Y = gnt2;
               else
                  Y = Idle;
         gnt3: if ( r[ 3 ] )
                  Y = gnt3;
               else
                  Y = Idle;
         default: Y = Idle;
      endcase

   // Sequential block
   always @( posedge clock )
      if ( reset )
         y <= Idle;
      else
         y <= Y;

   // Define output
   assign g[ 1 ] = y == gnt1;
   assign g[ 2 ] = y == gnt2;
   assign g[ 3 ] = y == gnt3;

endmodule
```



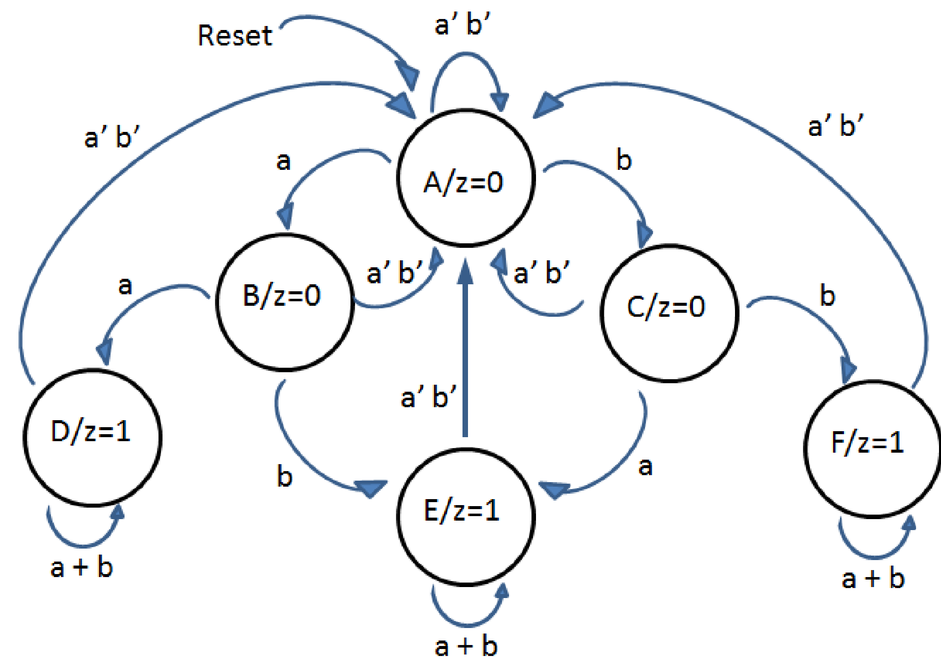
Another example

Consider this state diagram.



Assume the input condition ab is guaranteed never to occur.
Can this diagram be simplified by collapsing equivalent states?

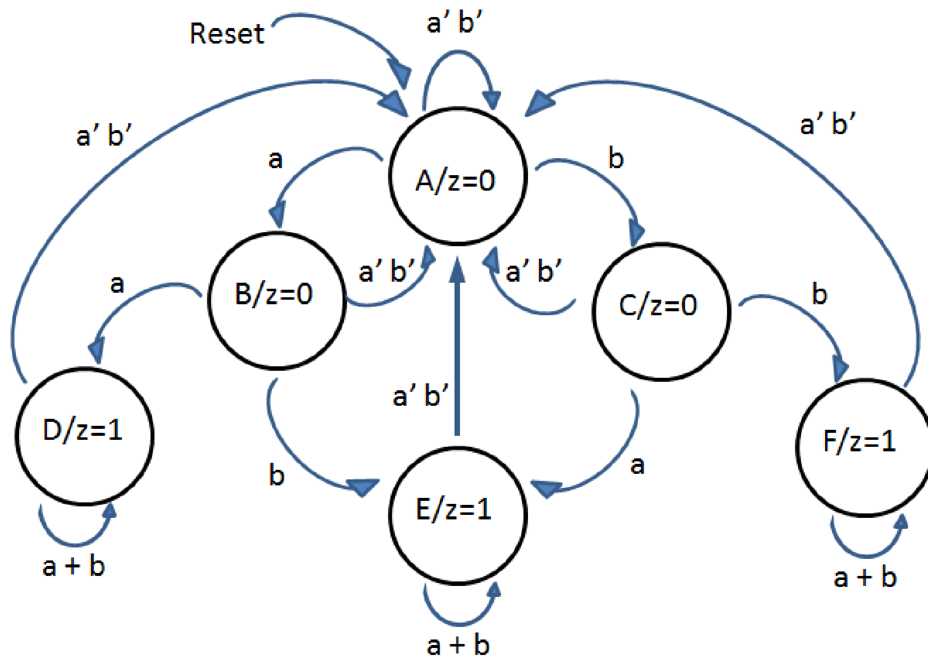
This is a Moore machine because the output only depends on the state.



Since ab is guaranteed never to occur, the entire ab = 11 column is *don't care.*

State table

State	Next state ab = 00	ab = 01	ab = 11	ab = 10	Output Z
A	A	C	-	B	0
B	A	E	-	D	0
C	A	E	-	F	0
D	A	D	-	D	1
E	A	E	-	E	1
F	A	F	-	F	1

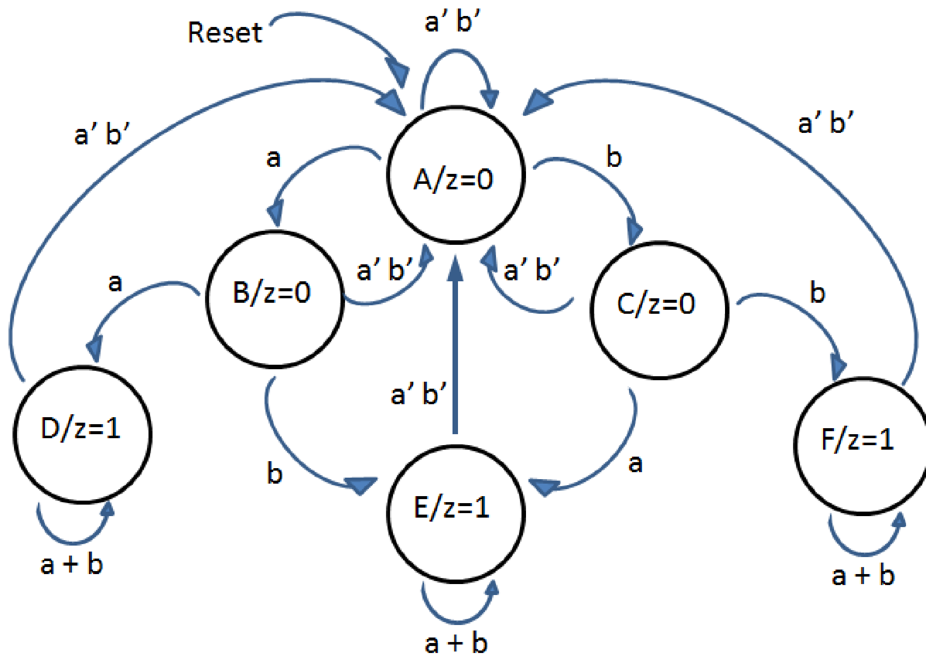


To find equivalent states, partition first by output, then by successors.

If two states have different outputs or successors in different partitions, they can’t be equivalent.

State table

State	Next state ab = 00	ab = 01	ab = 11	ab = 10	Output Z
A	A	C	-	B	0
B	A	E	-	D	0
C	A	E	-	F	0
D	A	D	-	D	1
E	A	E	-	E	1
F	A	F	-	F	1



Partitioning first by output:

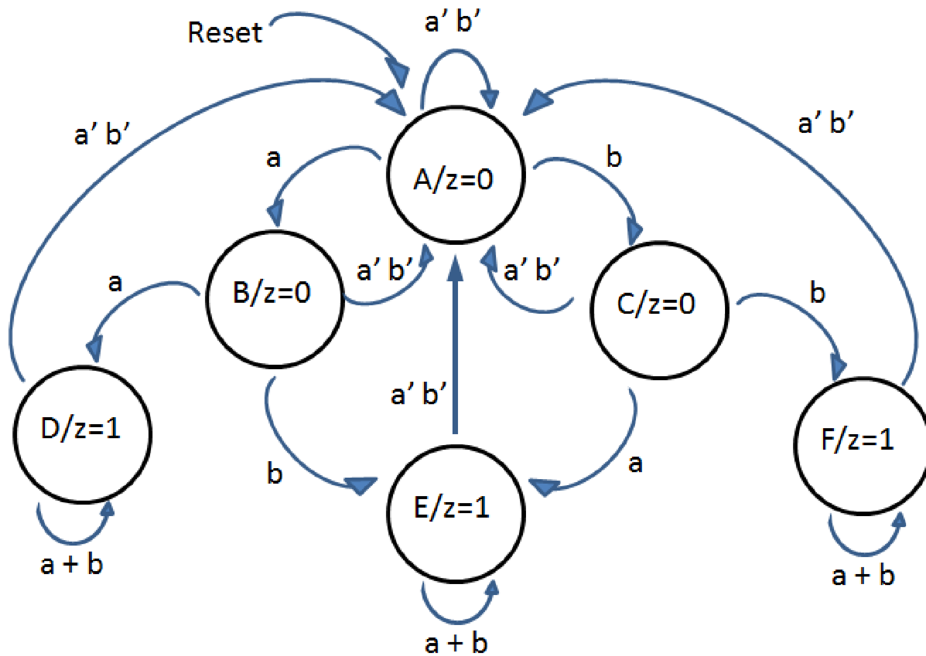
Z = 0 for A, B and C
Z = 1 for D, E and F

Initial partitioning:

(A B C) (D E F)

State table

	Next state				Output
State	ab = 00	ab = 01	ab = 11	ab = 10	Z
A	A	C	-	B	0
B	A	E	-	D	0
C	A	E	-	F	0
D	A	D	-	D	1
E	A	E	-	E	1
F	A	F	-	F	1



State table

State	Next state ab = 00	ab = 01	ab = 11	ab = 10	Output Z
A	A	C	-	B	0
B	A	E	-	D	0
C	A	E	-	F	0
D	A	D	-	D	1
E	A	E	-	E	1
F	A	F	-	F	1

Split so far:

(A B C) (D E F)

Partitioning by successors:

00-successors

All successors = A
No splits

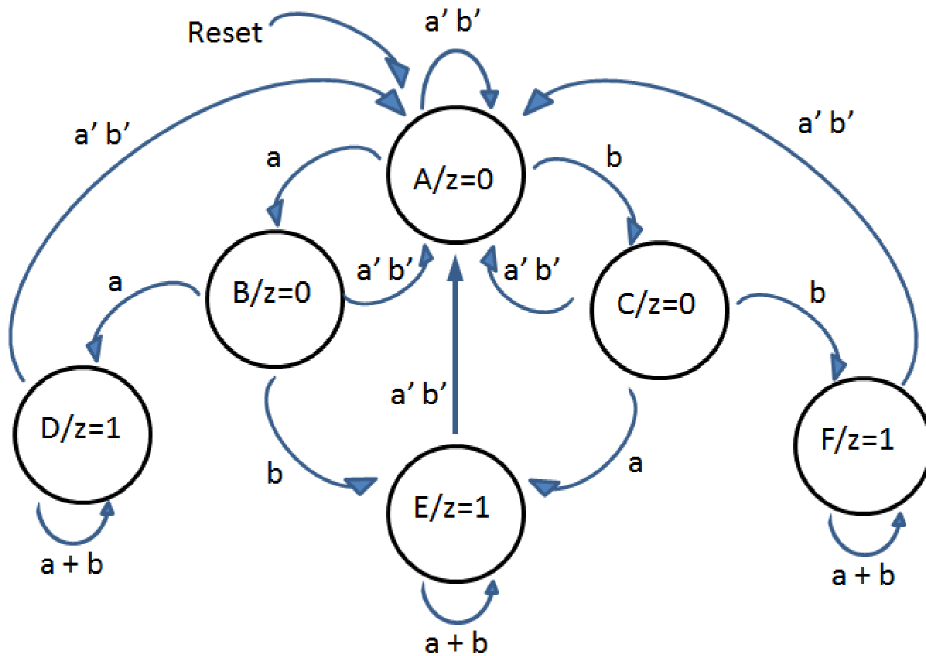
01-successors

(A B C) successors = (C E E)
Split into (A) (B C)

(D E F) successors = (D E F)
No splits

Split so far:

(A) (B C) (D E F)



State table

State	Next state ab = 00	ab = 01	ab = 11	ab = 10	Output Z
A	A	C	-	B	0
B	A	E	-	D	0
C	A	E	-	F	0
D	A	D	-	D	1
E	A	E	-	E	1
F	A	F	-	F	1

Split so far:

(A) (B C) (D E F)

Continuing by successors:

11-successors

All successors = *Don’t care*

No splits

10-successors

(B C) successors = (D F)

No splits

(D E F) successors = (D E F)

No splits

Final partitioning

(A) (B C) (D E F)

Final partitioning

(A) (B C) (D E F)

Rewrite the state table:

C → B
(E F) → D

State table

State	Next state ab = 00	ab = 01	ab = 11	ab = 10	Output Z
A	A	C	-	B	0
B	A	E	-	D	0
C	A	E	-	F	0
D	A	D	-	D	1
E	A	E	-	E	1
F	A	F	-	F	1

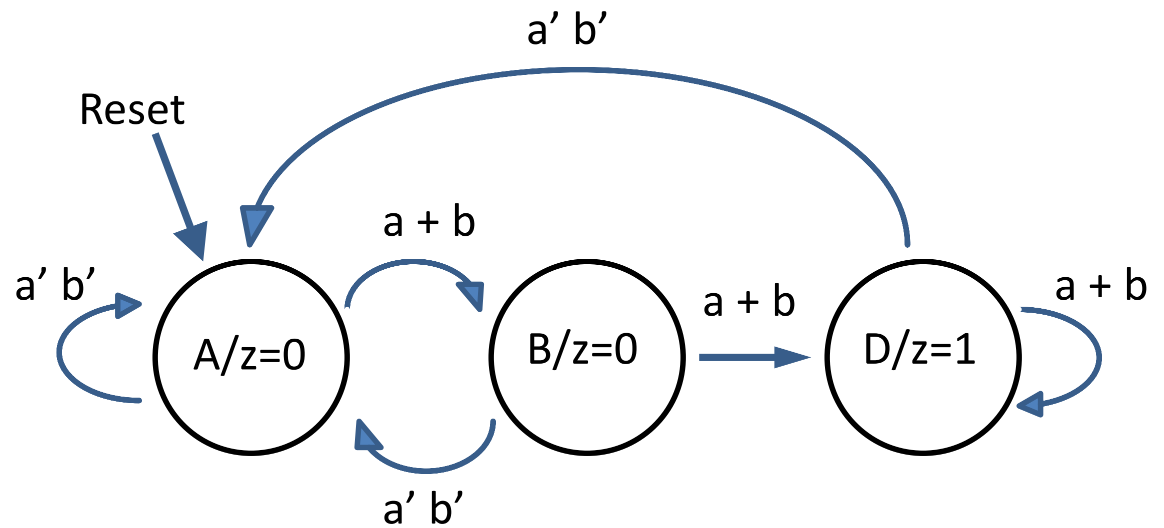
Minimized state table

State	Next state ab = 00	ab = 01	ab = 11	ab = 10	Output Z
A	A	B	-	B	0
B	A	D	-	D	0
D	A	D	-	D	1

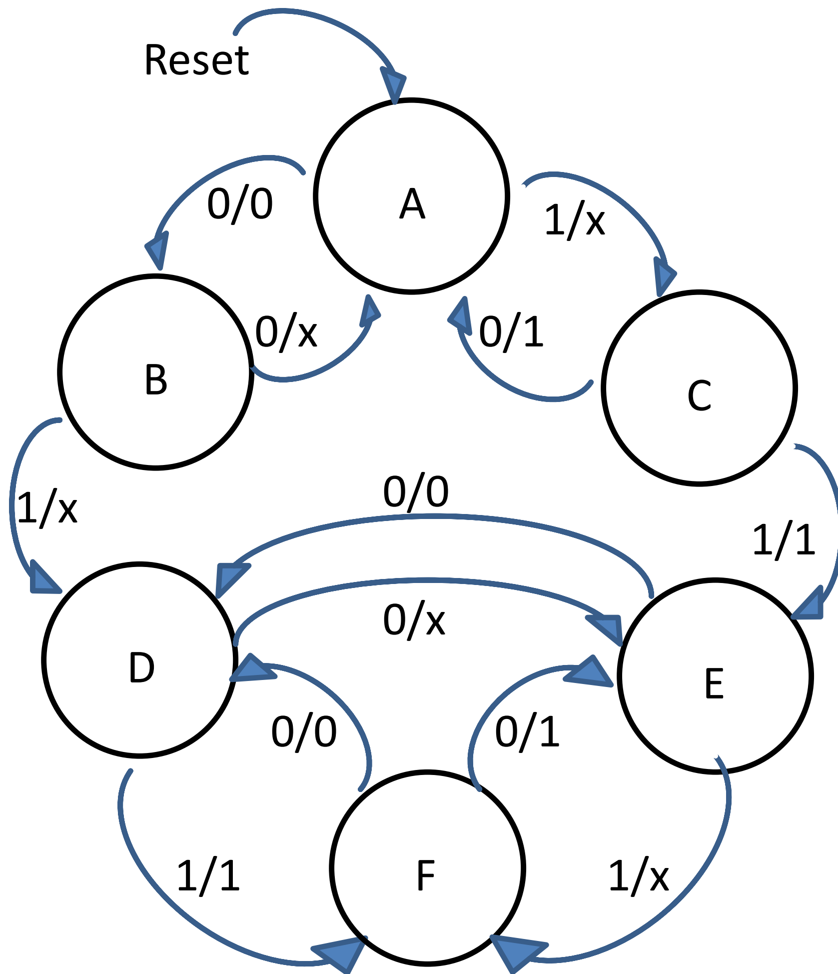
Minimized state table

State	Next state ab = 00	ab = 01	ab = 11	ab = 10	Output Z
A	A	B	-	B	0
B	A	D	-	D	0
D	A	D	-	D	1

Minimized state diagram



Another example

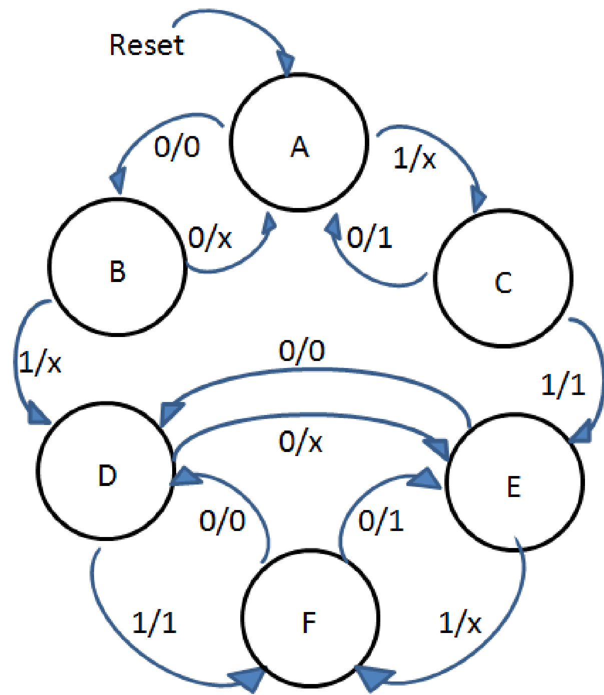


This is a Mealy machine because the output depends on the state and the input.

State table

State	Next state w = 0	Next state w = 1	Output w = 0	Output w = 1
A	B	C	0	x
B	A	D	x	x
C	A	E	1	1
D	E	F	x	1
E	D	F	0	x
F	D	E	0	1

Must guess whether the *don't cares* should be 1's or 0's.



State table, x = 0

State	Next state w = 0	Next state w = 1	Output w = 0	Output w = 1
A	B	C	0	0
B	A	D	0	0
C	A	E	1	1
D	E	F	0	1
E	D	F	0	0
F	D	E	0	1

Assume all the *don't cares* are 0's.

Partition first by output:

(A, B, E) (C) (D, F)

By 0-successor:

(A, B, E) → (B, A, D)

split into (A, B) (E)

(D, F) → (E, D)

split into (D) (F)

By 1-successor:

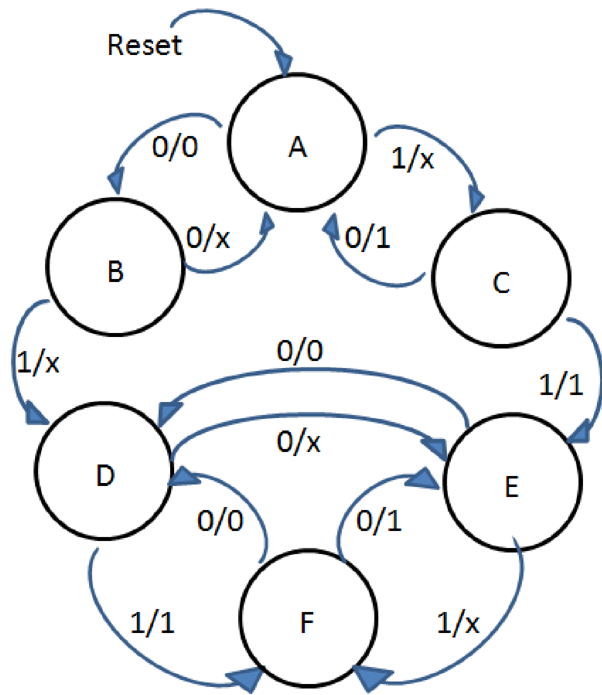
(A, B) → (C, D)

split into (A) (B)

Final partitioning:

(A) (B) (C) (D) (E) (F)

No states eliminated.



State table, x = 1

State	Next state w = 0	Next state w = 1	Output w = 0	Output w = 1
A	B	C	0	1
B	A	D	1	1
C	A	E	1	1
D	E	F	1	1
E	D	F	0	1
F	D	E	0	1

Assume all the *don't cares* are 1's.

Partition first by output:

(A, E, F) (B, C, D)

By 0-successor:

(A, E, F) → (B, D, D)

No splits

(B, C, D) → (A, A, E)

No splits

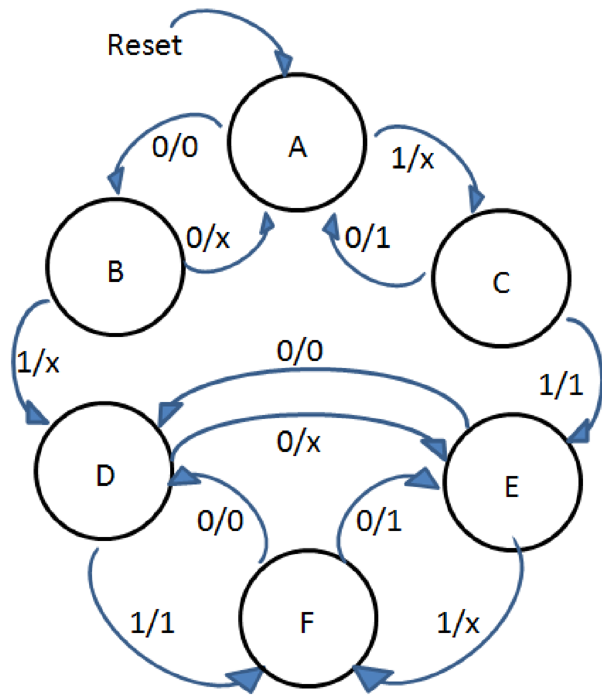
By 1-successor:

(A, E, F) → (C, F, E)

split into (A) (E, F)

(B, C, D) → (D, E, F)

split into (B) (C, D)



State table, x = 1

State	Next state w = 0	Next state w = 1	Output w = 0	Output w = 1
A	B	C	0	1
B	A	D	1	1
C	A	E	1	1
D	E	F	1	1
E	D	F	0	1
F	D	E	0	1

Split so far:

(A) (B) (C, D) (E, F)

Again by 0-successor:

(C, D) → (A, E)

split into (C) (D)

(E, F) → (D, D)

No splits

By 1-successor:

(E, F) → (F, E)

No splits

Final partitioning:

(A) (B) (C) (D) (E, F)

We saved one state.

Final partitioning with all don't cares = 1

(A) (B) (C) (D) (E, F)

To know if this is really the best, must consider every other possible combination for the don't cares. But let's stop here.

State table

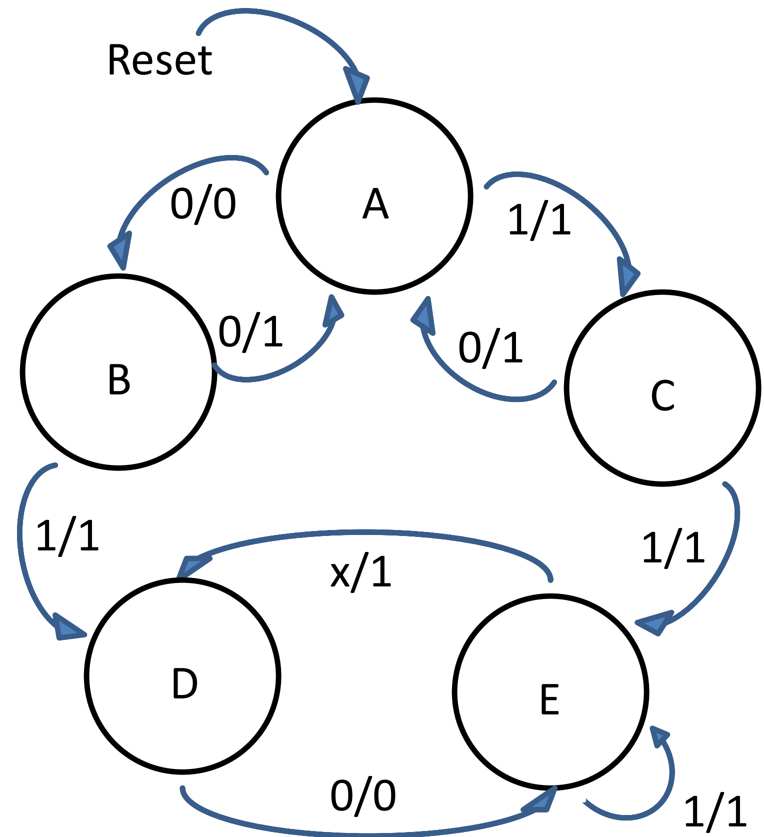
State	Next state w = 0	Next state w = 1	Output w = 0	Output w = 1
A	B	C	0	x
B	A	D	x	x
C	A	E	1	1
D	E	F	x	1
E	D	F	0	x
F	D	E	0	1

Reduced state table

State	Next state w = 0	Next state w = 1	Output w = 0	Output w = 1
A	B	C	0	1
B	A	D	1	1
C	A	E	1	1
D	E	E	1	1
E	D	E	0	1

Reduced state table

State	Next state w = 0	Next state w = 1	Output w = 0	Output w = 1
A	B	C	0	1
B	A	C	1	1
C	A	E	1	1
D	E	E	1	1
E	C	E	0	1



Reduced state diagram

Analysis of an existing circuit

In addition to designing a new sequential circuit, one may need to analyze an existing circuit for a redesign or to compare two implementations.

Solution is to reverse the steps to figure out what it does.

Example: Compare various implementations of the register transfer controller

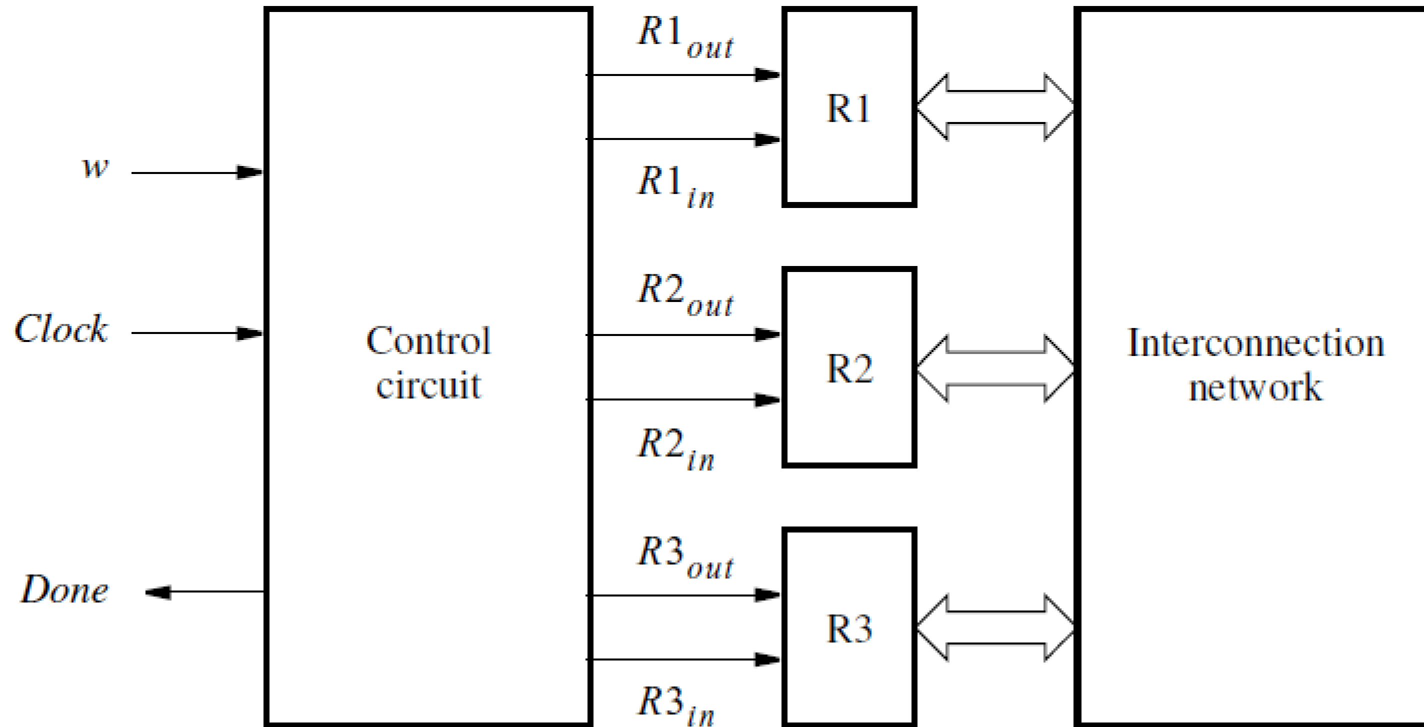


Figure 6.10. From last time: System for register exchange controller.

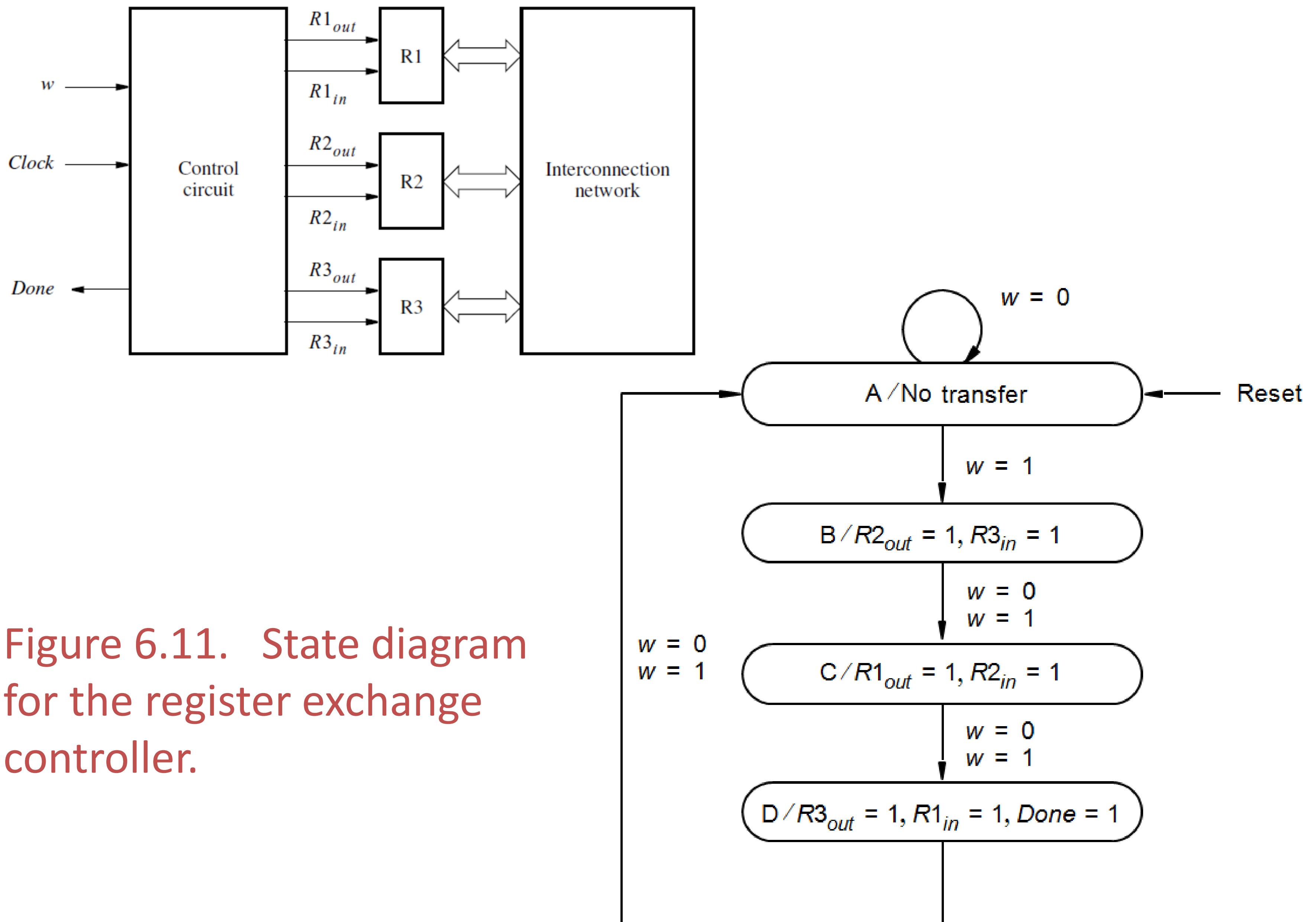
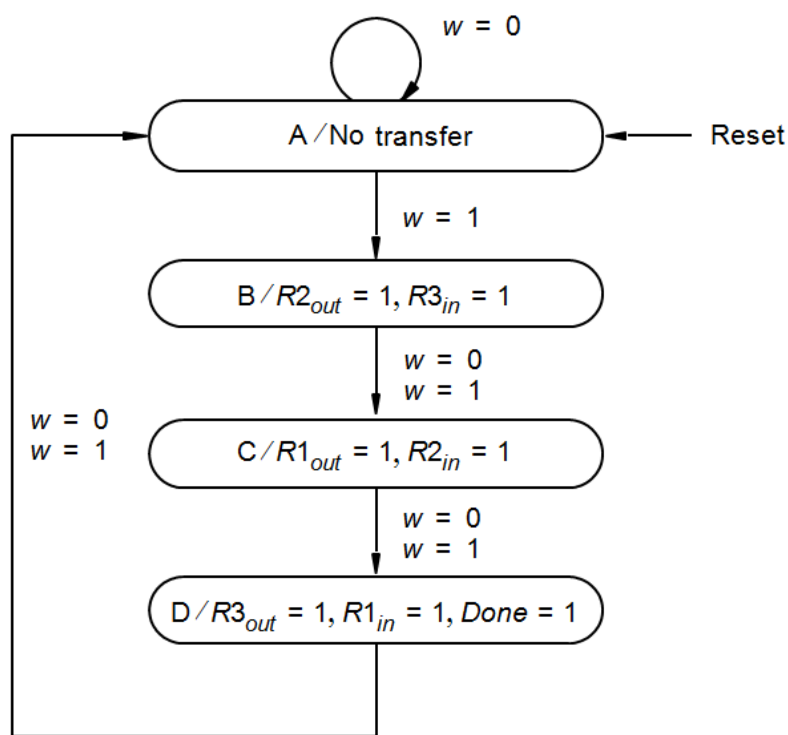
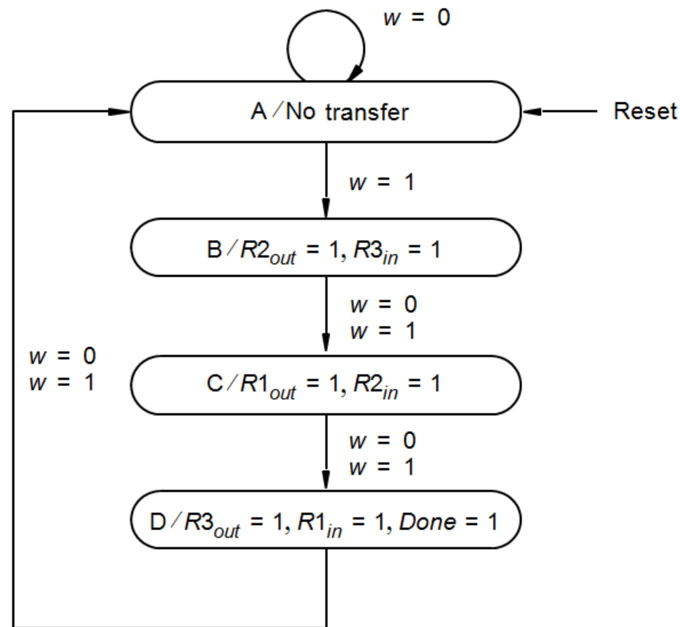


Figure 6.11. State diagram for the register exchange controller.



Present state	Next state		Outputs						
	$w = 0$	$w = 1$	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	A	B	0	0	0	0	0	0	0
B	C	C	0	0	1	0	0	1	0
C	D	D	1	0	0	1	0	0	0
D	A	A	0	1	0	0	1	0	1

Figure 6.12. State table for the register exchange.



	Present state	Next state		Outputs						
		$w = 0$	$w = 1$							
	y_2y_1	Y_2Y_1	Y_2Y_1	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	00	00	01	0	0	0	0	0	0	0
B	01	10	10	0	0	1	0	0	1	0
C	10	11	11	1	0	0	1	0	0	0
D	11	00	00	0	1	0	0	1	0	1

Figure 6.13. State-assigned table for the register exchange.

w \ y_2y_1	00	01	11	10
0				1
1	1			1

$$Y_1 = w\bar{y}_1 + \bar{y}_1 y_2$$

w \ y_2y_1	00	01	11	10
0		1		1
1		1		1

$$Y_2 = y_1\bar{y}_2 + \bar{y}_1 y_2$$

Figure 6.14. Derivation of next-state expressions for the for the register exchange sequence.

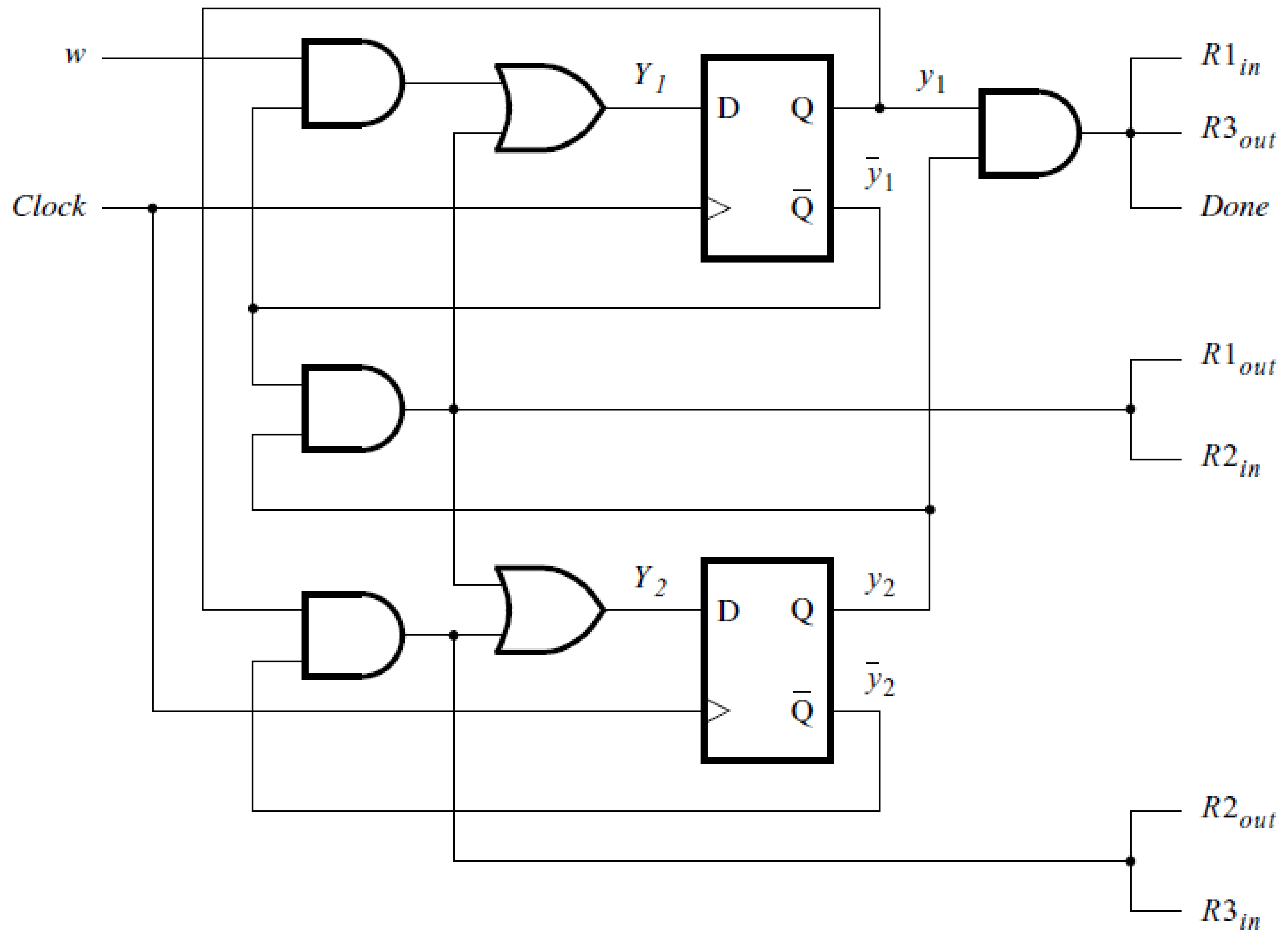
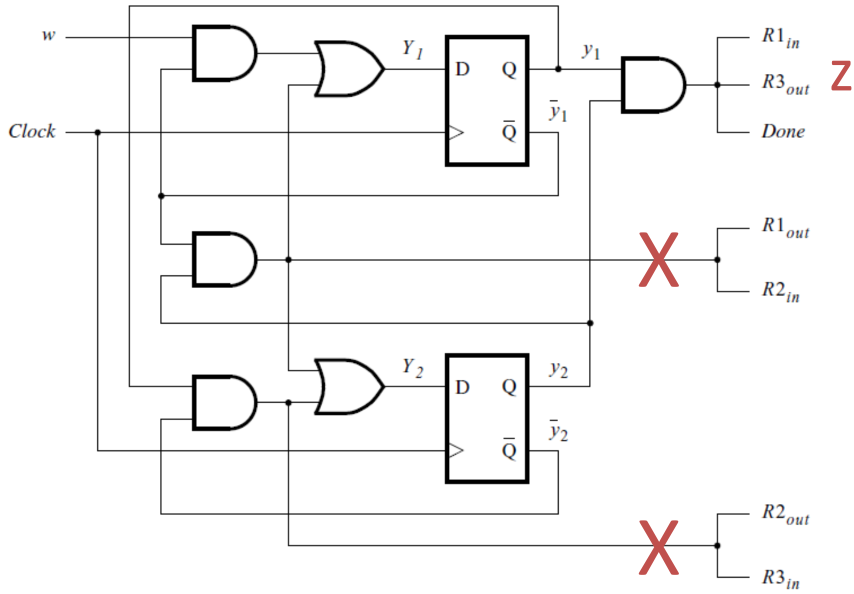
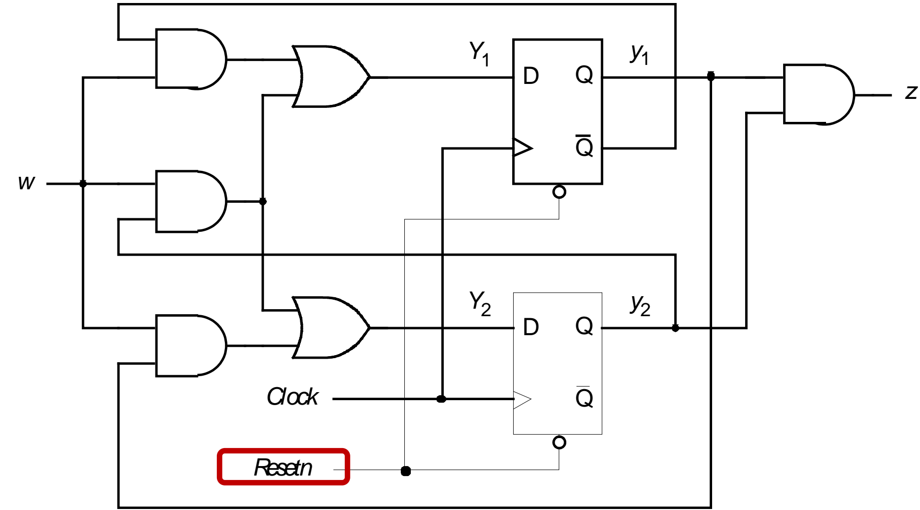


Figure 6.15. The register exchange controller we arrived at using D flip-flops.



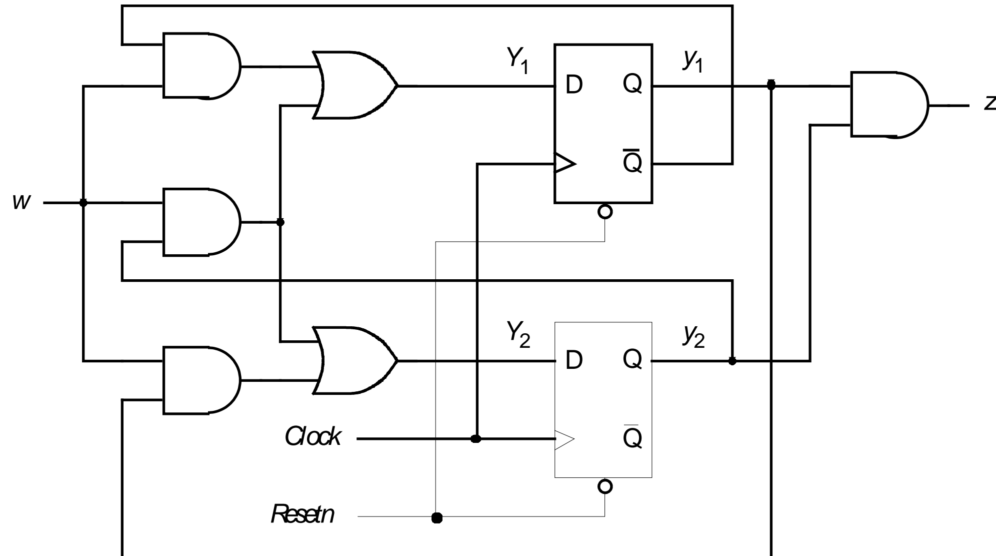
Original



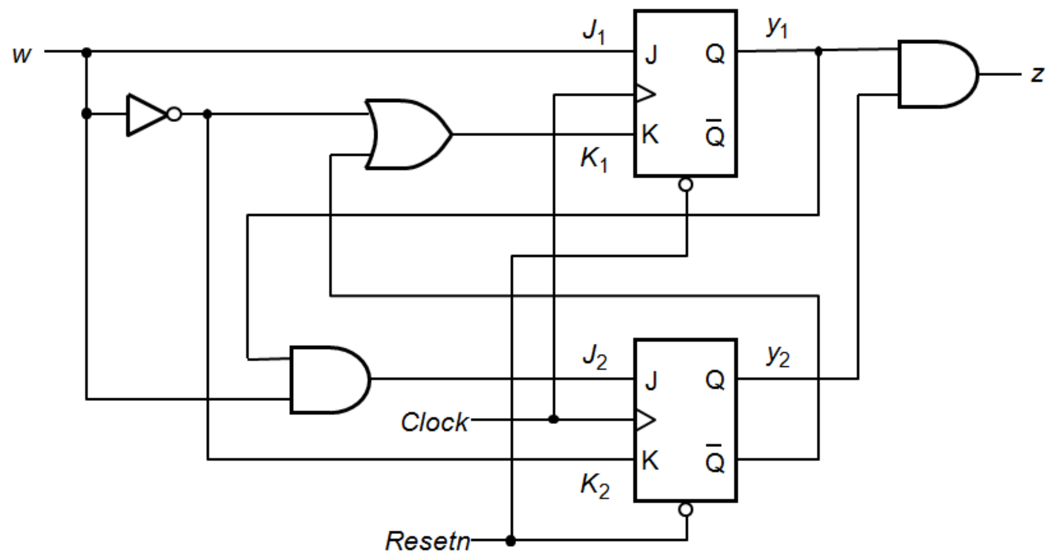
Simplified

Figure 6.75. Let's examine the problem again with slightly simplified version of the register transfer controller.

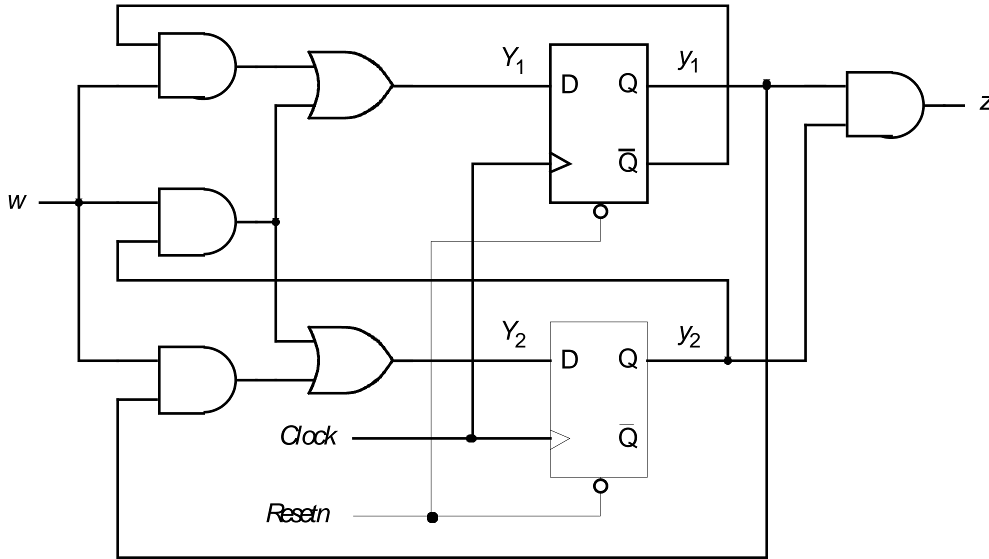
We know what this one does, but let's pretend we don't.



What does this one do?



Do these circuits do the same thing?



Present state y_2y_1	Next State		Output z
	w = 0	w = 1	
	Y_2Y_1	Y_2Y_1	
0 0	0 0	01	0
0 1	0 0	10	0
1 0	0 0	11	0
1 1	0 0	11	1

Derive the equations by inspection:

Y1 = w y1’ + w y2

Y2 = w y1 + w y2

z = y1 y2

Fill in the state-assigned table.

Figure 6.76. Reverse-engineer the circuit using D flip-flops by constructing a state-assigned table for it.

Present state y_2y_1	Next State		Output z
	w = 0	w = 1	
	Y_2Y_1	Y_2Y_1	
0 0	0 0	01	0
0 1	0 0	10	0
1 0	0 0	11	0
1 1	0 0	11	1

(a) State-assigned table

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

(b) State table

Figure 6.76. Then create a symbolic state table for the simplified register transfer.

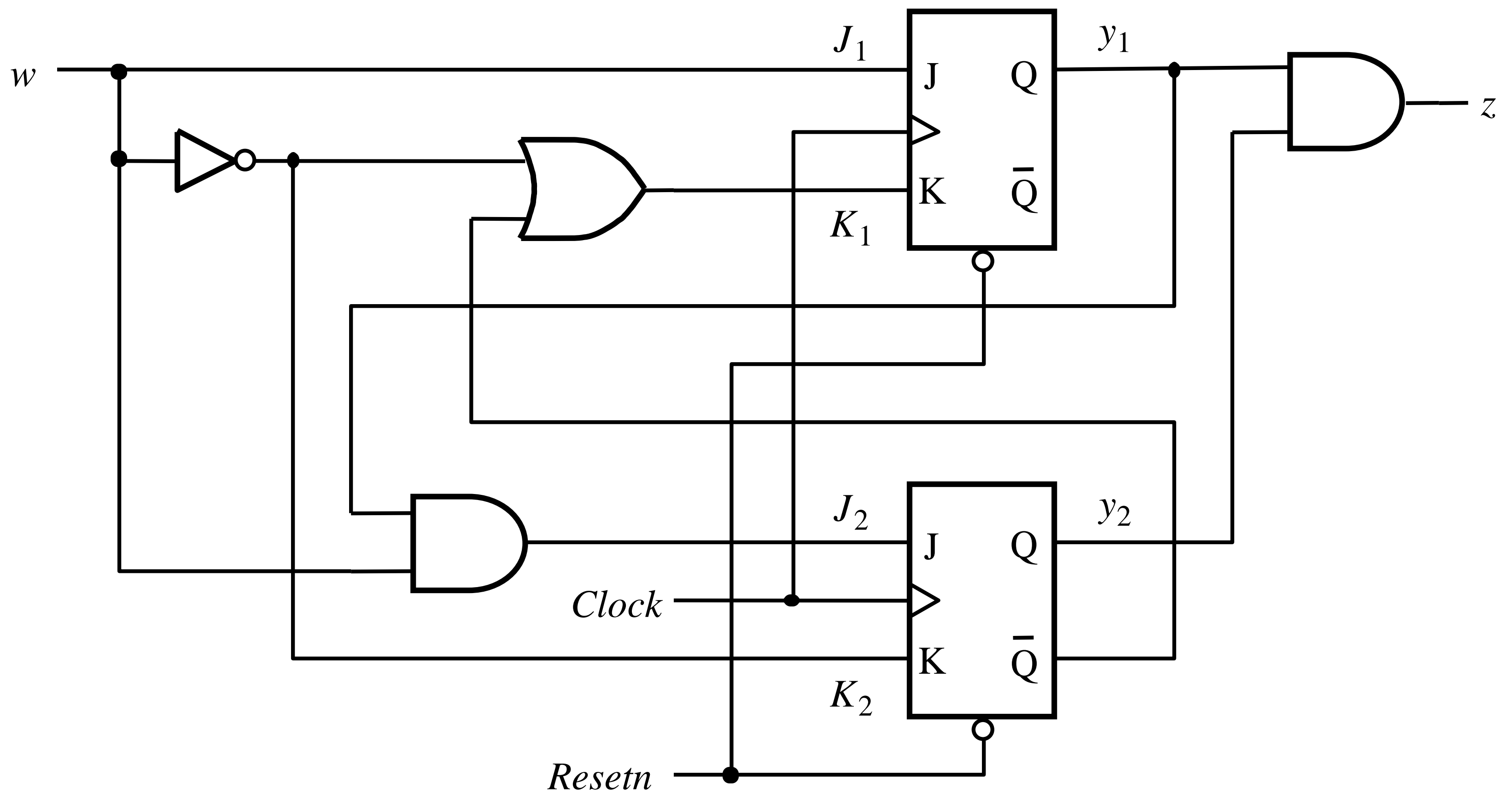
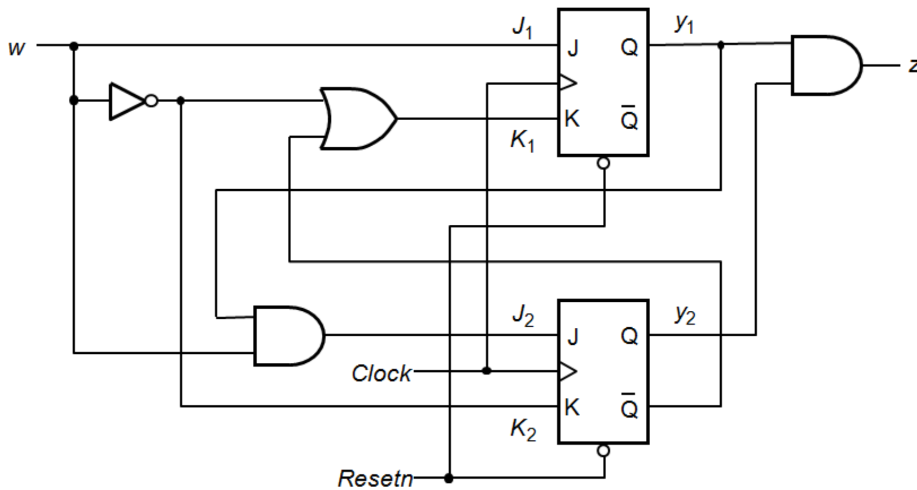


Figure 6.77. The JK flip-flop circuit to be compared.



Present state y_2y_1	Flip-flop inputs				Output z
	$w = 0$		$w = 1$		
	J_2K_2	J_1K_1	J_2K_2	J_1K_1	
00	01	0 1	0 0	1 1	0
01	01	0 1	1 0	1 1	0
10	01	0 1	0 0	1 0	0
11	01	0 1	1 0	1 0	1

Derive the equations by inspection:

J1 = w

K1 = w' + y2'

J2 = w y1

K2 = w'

Fill in the excitation table.

Figure 6.78. Derive the ***excitation table*** for the JK register transfer implementation.

Present state y_2y_1	Flip-flop inputs				Output z
	$w = 0$		$w = 1$		
	J_2K_2	J_1K_1	J_2K_2	J_1K_1	
00	01	0 1	0 0	1 1	0
01	01	0 1	1 0	1 1	0
10	01	0 1	0 0	1 0	0
11	01	0 1	1 0	1 0	1

Present state	**Next state**		**Output z**
	w = 0	w =1	
A (00)	00 (A)	01 (B)	0
B (01)	00 (A)	10 (C)	0
C (10)	00 (A)	11 (D)	0
D (11)	00 (A)	11 (D)	1

Transform the excitation table for the JK register transfer into a state-assigned and then into a state table

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

D state table

Present state	**Next state**		**Output z**
	w = 0	w =1	
A (00)	00 (A)	01 (B)	0
B (01)	00 (A)	10 (C)	0
C (10)	00 (A)	11 (D)	0
D (11)	00 (A)	11 (D)	1

JK state table

Compare JK state table with that of the D state table to verify they behave equivalently.