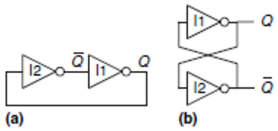
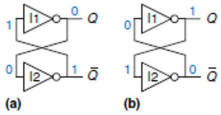
Sequential Logic DDCA 3.1-3.3, 3.5, 4.1-4.5, 4.7-4.8, 5.4

Sunday, January 17, 2021 11:17 AM

3.2 Latches and Flip-Flops



- Bistable element, 2 inverters connected in a loop, cross coupled
- cyclic circuit: Q depends on ~Q and ~Q depends on Q
- If Q=0, ~Q=1, Q=0
- If Q=1, ~Q=0, Q=1



- **SR Latch**
  - two cross coupled NOR gates, inputs S and R, outputs Q and ~Q
  - state controlled through S and R inputs, set and reset output Q
  - use a truth table

Case	S	R	Q	$\bar{Q}$
IV	0	0	Q_{prev}	$\bar{Q}_{prev}$
I	0	1	0	1
II	1	0	1	0
III	1	1	0	0

Figure 3.5 SR latch truth table

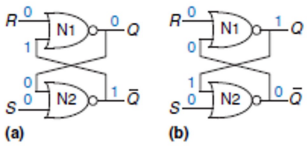
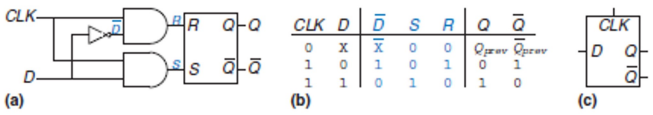


Figure 3.6 SR latch symbol

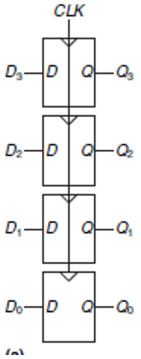
  - bistable element, 1 bit of state stored in Q
  - when R asserted, state reset to 0, when S asserted, state set to 1
  - cons:
    - behaves strangely when S and R both asserted
    - conflate issues of what and when, can't tell when it should change
- **D Latch**
  - 2 inputs, data D controls what the next state should be, clock input CLK controls when the state should change
  - CLK controls when data flows through the latch
    - when CLK=1, latch is transparant, D flows to Q as if latch is a buffer
    - when CLF=0, latch is opaque
  - use a truth table

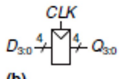


CLK	D	$\bar{D}$	S	R	Q	$\bar{Q}$
0	X	$\bar{X}$	0	0	Q_{prev}	$\bar{Q}_{prev}$
1	0	1	0	1	0	1
1	1	0	1	0	1	0

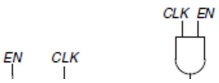
Figure 3.7 D latch: (a) schematic, (b) truth table, (c) symbol

- **D Flip-Flop**
  - 2 back to back D latches controlled by complementary clocks
  - Latch L1 is the master, latch L2 is called the slave, node etween is N1
    - when CLK=0, master latch transparent, slave is opaque
    - D goes through N1
    - when CLK=1, master is opaque, slave is transparant, N1 goes thoruhg Q
    - N1 is cut off from D, value at D copied to Q imediately after clock rises, else Q returns its old value
  - D flip flop copies D to Q on rising edge of the clock and remembers its state at all other times
- **Register**
  - an N-bit register is a bank of N flipflops that share a common CLK, all bits of register are updated at the same time

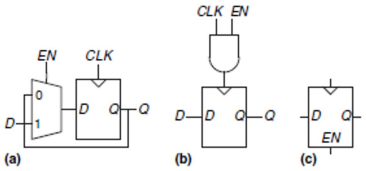


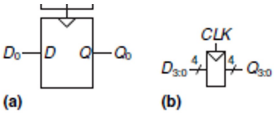


- **Enabled Flip Flop**
  - adds another input called EN or ENABLE to determine whether data is loaded on clk edge
  - when EN TRUE, enabled flip flop behaves like ordinary D flip flop
  - when EN FALSE enabled flip flop ignores clk and retains its state



- when EN FALSE enabled flip flop ignores clk and retains its state





- **Resettable Flip Flop**
  - adds another input called RESET
  - when RESET FALSE resettable flip flop behaves like ordinary D flip flop
  - when RESET TRUE, resettable flip flop ignores D and resets output to 0
  - may be synchronously or asynchronously resettable
- **Summary**
  - D latch is level sensitive
  - D flip flop edge triggered
  - D latch transparent when CLK=1 D flows to Q
  - D flip flop copies D to Q on rising edge of CLK
  - all other times latches and flip flops retain old state
  - register is a bank of several D flip flops that share common CLK signal

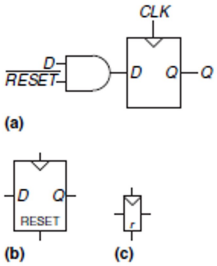


Figure 3.11 Synchronously resettable flip-flop: (a) schematic, (b, c) symbols

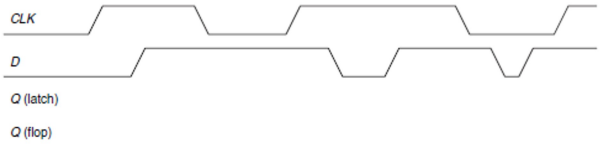


Figure 3.14 Example waveforms

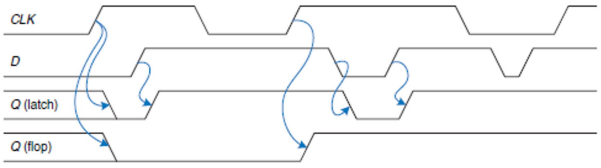


Figure 3.15 Solution waveforms

3.3 Synchronous Logic Design

- sequential circuits include circuits whose output cannot be determined by looking at the current inputs
- **Astable Circuits**
  - circuit has no stable states and is unstable or astable
  - each node oscillates bw 0 and 1 with a period
  - ring oscillator, sequential circuit with 0 inputs and one output changing periodically

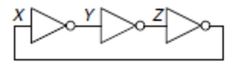
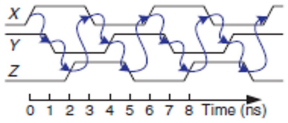
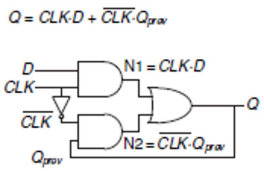


Figure 3.16 Three-inverter loop



- **Race Conditions**
  - circuit fails when certain gates are slower than others
  - delay through inverter from CLK to ~CLK is longer compared to delays of AND and OR gates

CLK	D	Q_{prev}	Q
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



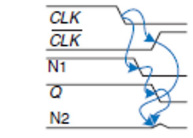
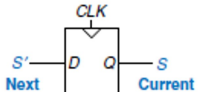


Figure 3.19 Latch waveforms illustrating race condition

- **Synchronous Sequential Circuits**
  - combinational logic has no cyclic paths and no races unlike sequential logic
  - the registers contain state of system changing at clk edge, state is synchronized to the clock
  - if clock is slow enough s o that inputs to all registers settle before next clock edge, all races are eliminated
  - circuit is a synchronous sequential circuit if it consists of interconnected circuit elements such that:
    - **Rules of synchronous sequential circuit composition**
      - every circuit element is either a register or combinational circuit
      - at least one circuit element is a register
      - all registers receive the same clock signal
      - every cyclic path contains at least one register
  - current state variable S and next state variable S'



- □ at least one circuit element is a register
- □ all registers receive the same clock signal
- □ every cyclic path contains at least one register

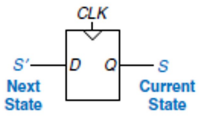


Figure 3.20 Flip-flop current state and next state

- current state variable S and next state variable S'
- other common types of synchronous sequential circuits are finite state machines and pipelines

- **Synchronous and Asynchronous Circuits**
  - asynchronous is more general than synchronous
  - timing not limited by clocked registers
  - asynchronous circutis can use any kind of feedback

3.5 Timing of Sequential Logic

- flip flop copies input D to output Q on rising edge of clock, sampling D
- If D is changing at same time as clock rises, what happens?
  - aperture time during which obect must remain still and stable for flip flop to produce well-defined output
  - setup time and hold time before and after clock edge
  - dynamic discipline limits us to using signals that change outside the aperture time
- **The Dynamic Discipline**
  - the inputs of a synchronous sequential circuit must be stable during the setupt and hold aperture time around the clock edge
  - guarantee that the flip flops sample signals while they are not changing
- **System Timing**
  - clock period/cycle time T is the time bw rising edges of a repetitive clock signal

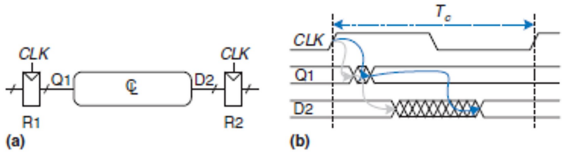
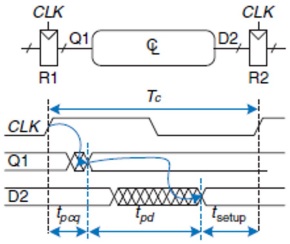


Figure 3.38 Path between registers and timing diagram

  - **Setup time constraint**
    - to satisfy the setup time of R2, D2 must settle no later than the setup time before the next clock edge

      $T_c \geq t_{pcq} + t_{pd} + t_{\text{setup}}$
    - solve for max propagation delay through combinational logic, under control of individual designer

      $t_{pd} \leq T_c - (t_{pcq} + t_{\text{setup}})$
    - t_pcq+t_setup = sequencing over-head

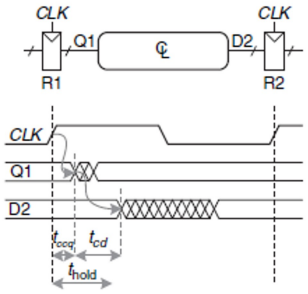


  - **Hold time constraint**
    - input D2 must not change until t_hold after rising edge of clock
    - D2 might change as soon as t_ccq+t_cd

      $t_{ccq} + t_{cd} \geq t_{\text{hold}}$
    - solve for min contamination delay through combinational logic

      $t_{cd} \geq t_{\text{hold}} - t_{ccq}$
    - A reliable flip flop must have a hold time shorter than its contamination delay

      $t_{\text{hold}} \leq t_{ccq}$



- **Metastability**
  - not always possible to guarantee that input to sequential circuit is stable during aperture time
  - **Metastable State**
    - when flip flop samples input that is changing during its aperture, output Q may take on voltage bw 0-V_DD
    - Metastable state, eventually flip flop will resolve output to stable state of 0 or 1
    - resolution time required to reach stable state is unboundedd

- **Resolution time**
  - if flip flop input changes at random time during clock cycle, resolution time t_res required to resolve to a stable state
  - if input changes outside aperture time, t_res=t_pcq, else can take longer

$$P(t_{res} > t) = \frac{T_0}{T_c} e^{-\frac{t}{\tau}}$$

- **Synchronizers**
  - to guarantee good logic levels, all asynchronous inputs should be passed through synchronizers
  - device that receives an async input D and clock CLK, produces output Q within bounded amount of time, output has valid logic level with extremely high probability
  - if D stable during aperture, Q=D
  - if D changes during aperture, Q takes either HIGH or LOW value but cannot be metastable

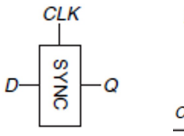
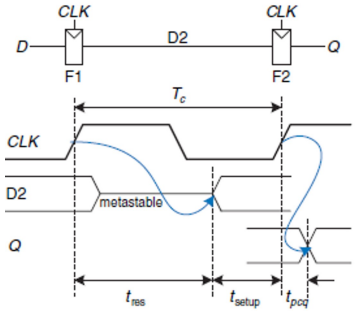


Figure 3.52 Synchronizer symbol



  - a synchronizer fails if Q output is metastable
    - D2 has not resolved to a valid level by time it must setup at F2

$$P(\text{failure}) = \frac{T_0}{T_c} e^{-\frac{T_c - t_{setup}}{\tau}}$$

    - If D changes N times per second, probability of failure per second is N times great:

$$P(\text{failure})/\text{sec} = N \frac{T_0}{T_c} e^{-\frac{T_c - t_{setup}}{\tau}}$$

    - Mean time between failures (MTBF)

$$MTBF = \frac{1}{P(\text{failure})/\text{sec}} = \frac{T_c e^{\frac{T_c - t_{setup}}{\tau}}}{N T_0}$$

4.1 Hardware Description Languages

- computer aided design (CAD) tool to produce optimized gates
- specifications given in hardware description language (HDL)
- **Modules**
  - block with inputs and outputs
  - Behavioural models describe what a module does
  - Structural models describe how a module is built from simpler pieces, an application of hierarchy
- **Simulation and synthesis**
  - HDL purpose for logic simulation and synthesis
  - **Simulation**
    - inputs are applied to module, outputs are checked to verify that the module operates correctly
    - testing in lab is time consuming, logic simulation is essential to test a system before it is built
  - **Synthesis**
    - Textual description of a module is transformed into logic gates
    - Logic synthesis transforms HDL code into a netlist describing the hardware (logic gates and wires connecting them)
    - perform optimizations to reduce amount of hardware required

- testbench contains code to apply inputs to a module, check whether the output results are correct, print descrepancies between expected and actual outputs, cannot be synthesized

4.2 Combinational Logic

- outputs of combinational logic depend only on current inputs
- **Bitwise Operators**
  - act on single-bit signals or multi-bit busses
  - ex. inv describes 4 inverters connected to 4-bit busses
  - endianness of a bus is arbitrary

```
module inv(input  logic [3:0] a,
           output logic [3:0] y);

  assign y = ~a;
endmodule
```

- **Reduction Operators**
  - imply multiple input gate acting on single bus
  - ex. eight-input AND gate with inputs a7-a0
  - OR, XOR, NAND, NOR, XNOR

```
module and8(input  logic [7:0] a,
            output logic       y);

  assign y = &a;

  // &a is much easier to write than
  // assign y = a[7] & a[6] & a[5] & a[4] &
  //            a[3] & a[2] & a[1] & a[0];
endmodule
```

- **Conditional Assignment**
  - select the output from among alternatives based on a condition input
  - ex. 2:1 multiplexer and 4:1 multiplexer

```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output logic [3:0] y);

  assign y = s ? d1 : d0;
endmodule
```

```
module mux4(input  logic [3:0] d0, d1, d2, d3,
            input  logic [1:0] s,
            output logic [3:0] y);

  assign y = s[1] ? (s[0] ? d3 : d2)
                  : (s[0] ? d1 : d0);
endmodule
```

- **Internal Variables**
  - break complex func into steps
  - ex. full adder

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);

  logic p, g;

  assign p = a ^ b;
  assign g = a & b;

  assign s = p ^ cin;
  assign cout = g | (p & cin);
endmodule
```

- **Precedence**
  - operator precedence from highest to lowest including arithmetic, shift, and comparison operators

	Op	Meaning
Highest	~	NOT
	*, /, %	MUL, DIV, MOD
	+, -	PLUS, MINUS
	<<, >>	Logical Left/Right Shift
	<<<, >>>	Arithmetic Left/Right Shift
	<, <=, >, >=	Relative Comparison
	==, !=	Equality Comparison
	&, ~&	AND, NAND
	^, ~^	XOR, XNOR
Lowest	|, ~|	OR, NOR
	?:	Conditional

Numbers	Bits	Base	Val	Stored
3'b101	3	2	5	101
'b11	?	2	3	000 ... 0011
8'b11	8	2	3	00000011
8'b1010_1011	8	2	171	10101011
3'd6	3	10	6	110
6'o42	6	8	34	100010
8'hAB	8	16	171	10101011
42	?	10	42	00 ... 0101010

- **Numbers**
  - can be specified in binary, octal, decimal, or hexadecimal
  - number of bits may be optionally given, leading zeros are insterted to reach this size
- **Z's and X's**
  - z floating value, describing a tristate bugger, whose output floats when enable is 0
  - if buffer enabled, output same as input, else output assigned floating value z
  - x invalid logic level, if bus is driven to 0 and 1 simultaneously by two enabled tristate buggers, result is x, contention

```
module tristate(input  logic [3:0] a,
                input  logic       en,
                output tri   [3:0] y);

  assign y = en ? a : 4'bz;
endmodule
```

Table 4.5 SystemVerilog AND gate truth table with z and x

&		A			
		0	1	z	x
B	0	0	0	0	0
	1	0	1	x	x
	z	0	x	x	x
	x	0	x	x	x

- **Bit Swizzling**

- operate on subset of a bus or concatenate signals to form busses

```
assign y = {c[2:1], {3{d[0]}}, c[0], 3'b101};
```

- **Delays**
  - predict how fast circuit will work, and understand cause and effect of bad output
  - ignored during sythesis

```
`timescale 1ns/1ps

module example(input  logic a, b, c,
               output logic y);

  logic ab, bb, cb, n1, n2, n3;

  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```

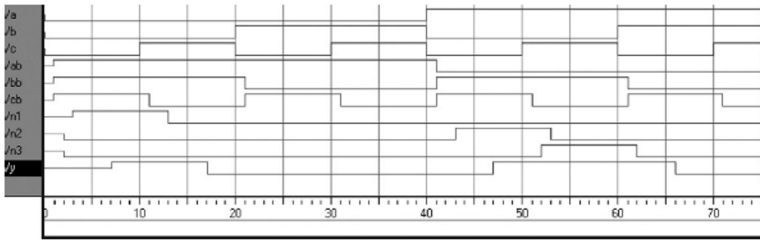


Figure 4.10 Example simulation waveforms with delays (from the ModelSim simulator)

4.3 Structural Modeling

- describing module in terms of how it is composed of simpler modules
  - ex. assembling 4:1 multiplexer from 3 2:1 multiplexers
  - each multiplexer called an instance
- multiple isntances of same modules distinguished by distinct names
  - ex. 2:1 multiplexer strucural model using tristate buffers

```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output tri   [3:0] y);

  tristate t0(d0, ~s, y);
  tristate t1(d1, s, y);
endmodule
```

```
module mux4(input  logic [3:0] d0, d1, d2, d3,
            input  logic [1:0] s,
            output logic [3:0] y);

  logic [3:0] low, high;

  mux2 lowmux(d0, d1, s[0], low);
  mux2 highmux(d2, d3, s[0], high);
  mux2 finalmux(low, high, s[1], y);
endmodule
```

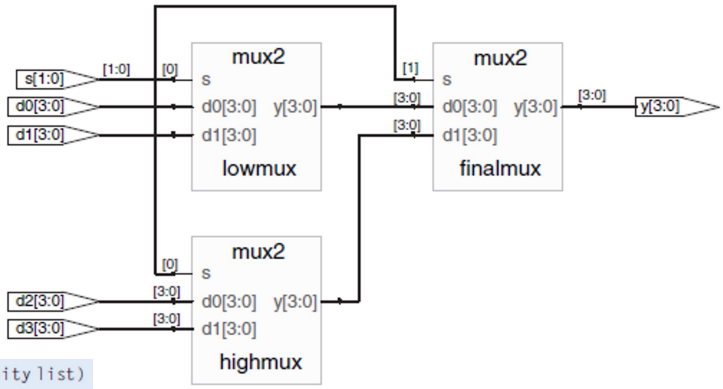


Figure 4.11 mux4 synthesized circuit

4.4 Sequential Logic

- **Registers**

```
module flop(input  logic       clk,
            input  logic [3:0] d,
            output logic [3:0] q);

  always_ff @(posedge clk)
    q <= d;
endmodule
```

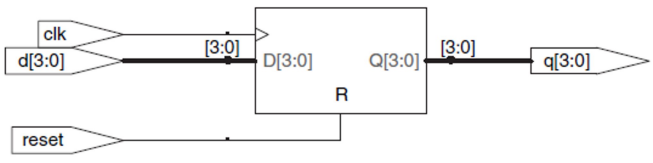
  - always statement written in form:

```
always @(sensitivity list)
  statement;
```

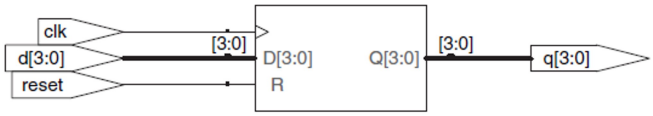
  - statement executed only when event specified in sensitivity list occurs
    - ex. q<=d, flip flop copies d to 1 on posedge else remembers old q
    - <= is nonblocking assignment, used instead of assign inside always statement
- **Resetable Registers**
  - when simulation begins or power first applied to circuit, output of flop register unknown, indicated with x
  - asynchronous reset occurs immediately, synchronous reset clears output only on next rising edge

```
module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // asynchronous reset
  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else       q <= d;
endmodule
```

```
module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // synchronous reset
  always_ff @(posedge clk)
    if (reset)  q <= 4'b0;
    else        q <= d;
endmodule
```



(a)



(b)

Figure 4.15 flopr synthesized circuit (a) asynchronous reset, (b) synchronous reset

- **Enabled Registers**
  - respond to clk only when enabled asserted

```
module flopenr(input  logic       clk,
               input  logic       reset,
               input  logic       en,
               input  logic [3:0] d,
               output logic [3:0] q);

  // asynchronous reset
  always_ff @(posedge clk, posedge reset)
    if      (reset) q <= 4'b0;
    else if (en)    q <= d;
endmodule
```

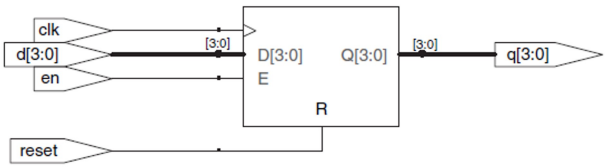
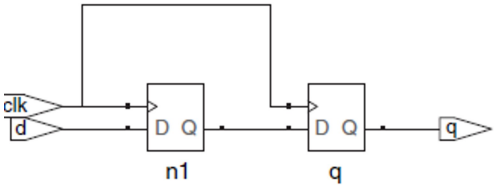


Figure 4.16 `flopenr` **synthesized circuit**

- **Multiple Registers**
  - always/process can be used to describe multiple pieces of hardware
  - ex. synchronizer made of 2 flipflops

```
module sync(input  logic clk,
            input  logic d,
            output logic q);

  logic n1;

  always_ff @(posedge clk)
    begin
      n1 <= d; // nonblocking
      q <= n1; // nonblocking
    end
endmodule
```



- **Latches**
  - D latch transparent when clk HIGH, data flows from input to output
  - avoid and use edge-triggered flipflops instead
  - if latch unexpected, inferred latches

4.5 More Combinational Logic

- **D Latch**

```
module latch(input  logic       clk,
             input  logic [3:0] d,
             output logic [3:0] q);

  always_latch
    if (clk) q <= d;
endmodule
```

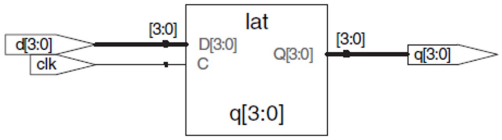


Figure 4.19 `latch` **synthesized circuit**

- **Full Adder using always/process**

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);
  logic p, g;

  always_comb
    begin
      p = a ^ b;          // blocking
      g = a & b;          // blocking

      s = p ^ cin;        // blocking
      cout = g | (p & cin);  // blocking
    end
endmodule
```

- **Case statements**
  - ex. seven-segment display decode takes advantage of the case statement that must appear inside always/process statement

```
module sevenseg(input  logic [3:0] data,
                output logic [6:0] segments);
  always_comb
    case(data)
      //                  abc_defg
      0:       segments = 7'b111_1110;
      1:       segments = 7'b011_0000;
      2:       segments = 7'b110_1101;
      3:       segments = 7'b111_1001;
      4:       segments = 7'b011_0011;
      5:       segments = 7'b101_1011;
      6:       segments = 7'b101_1111;
      7:       segments = 7'b111_0000;
      8:       segments = 7'b111_1111;
      9:       segments = 7'b111_0011;
      default: segments = 7'b000_0000;
    endcase
endmodule
```

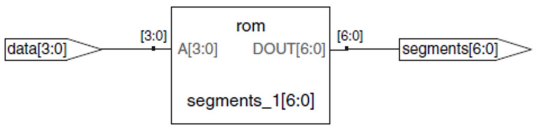


Figure 4.20 `sevenseg` **synthesized circuit**

- **If statements**
  - ex. 3:8 decoder, priority circuit

```
module decoder3_8(input  logic [2:0] a,
```

```
endmodule
```

- **If statements**
  - ex. 3:8 decoder, priority circuit

```
module decoder3_8(input  logic [2:0] a,
                  output logic [7:0] y);
  always_comb
    case(a)
      3'b000:  y=8'b00000001;
      3'b001:  y=8'b00000010;
      3'b010:  y=8'b00000100;
      3'b011:  y=8'b00001000;
      3'b100:  y=8'b00010000;
      3'b101:  y=8'b00100000;
      3'b110:  y=8'b01000000;
      3'b111:  y=8'b10000000;
      default: y=8'bxxxxxxxx;
    endcase
endmodule
```

```
module priorityckt(input  logic [3:0] a,
                   output logic [3:0] y);

  always_comb
    if      (a[3]) y<=4'b1000;
    else if (a[2]) y<=4'b0100;
    else if (a[1]) y<=4'b0010;
    else if (a[0]) y<=4'b0001;
    else           y<=4'b0000;
endmodule
```

  - N- input priority circuit sets output TRUE that corresponds to the most significant input that is TRUE
- **Truth Tables with Don't Cares**
  - truth tables may include don't care's to simplify logic

```
module priority_casez(input  logic [3:0] a,
                      output logic [3:0] y);
  always_comb
    casez(a)
      4'b1???: y<=4'b1000;
      4'b01??: y<=4'b0100;
      4'b001?: y<=4'b0010;
      4'b0001: y<=4'b0001;
      default: y<=4'b0000;
    endcase
endmodule
```

- **Blocking and Nonblocking Assignments**
  - To model synchronouse sequential logic use always_ff @(posedge clk) and nonblocking assignments

```
always_ff @(posedge clk)
  begin
    n1 <= d; // nonblocking
    q <= n1; // nonblocking
  end
```

  - to model simpole combinational logic use continuous assignments

```
assign y=s ? d1 : d0;
```

  - to model more complicated combinational logic use always_comb

```
always_comb
  begin
    p=a ^ b; // blocking
    g=a & b; // blocking
    s=p ^ cin;
    cout=g | (p & cin);
  end
```

  - do not make assignments to the same signal in more than one always statement

4.7 Data Types

- if signal appears on left side of <= or = in an always block, must be declared as reg, else declared as wire
- **Unsigned and signed multiplier**

```
// 4.33(a): unsigned multiplier
module multiplier(input  logic [3:0] a, b,
                  output logic [7:0] y);
  assign y=a * b;
endmodule

// 4.33(b): signed multiplier
module multiplier(input  logic signed [3:0] a, b,
                  output logic signed [7:0] y);
  assign y=a * b;
endmodule
```

4.8 Parameterized Modules

- unfixed width inputs and outputs
- ex. parameterized 2:1 mutliplexer with default width of 8, used to create 8 12-bit 4:1 mux

```
module mux2
  #(parameter width=8)
   (input  logic [width-1:0] d0, d1,
```

```
module mux4_8(input  logic [7:0] d0, d1, d2, d3,
              input  logic [1:0] s,
              output logic [7:0] y);

 logic [7:0] low, hi;
```

```
module mux4_12(input logic  [11:0] d0, d1, d2, d3,
               input logic  [1:0]  s,
               output logic [11:0] y);

 logic [11:0] low, hi;
```

mux

```
module mux2
  #(parameter width=8)
   (input  logic [width-1:0] d0, d1,
    input  logic             s,
    output logic [width-1:0] y);

   assign y=s ? d1 : d0;
endmodule
```

```
module mux4_8(input  logic [7:0] d0, d1, d2, d3,
              input  logic [1:0] s,
              output logic [7:0] y);

  logic [7:0] low, hi;

  mux2 lowmux(d0, d1, s[0], low);
  mux2 himux(d2, d3, s[0], hi);
  mux2 outmux(low, hi, s[1], y);
endmodule
```

```
module mux4_12(input logic  [11:0] d0, d1, d2, d3,
               input logic  [1:0]  s,
               output logic [11:0] y);

  logic [11:0] low, hi;

  mux2 #(12) lowmux(d0, d1, s[0], low);
  mux2 #(12) himux(d2, d3, s[0], hi);
  mux2 #(12) outmux(low, hi, s[1], y);
endmodule
```

- ex. parameterized N:2^N Decoder

```
module decoder
  #(parameter N=3)
   (input  logic [N-1:0]    a,
    output logic [2**N-1:0] y);

  always_comb
    begin
      y=0;
      y[a]=1;
    end
endmodule
```

5.4 Sequential Building Blocks

- **Counters**
  - N-bit binary counter is a seq arithmetic circuit with clk and reset and N-bit Q output
  - reset initializes Q to 0
  - counter advances through all 2^N possible outputs in binary order incrementing on rising edge of clk

```
module counter #(parameter N=8)
               (input  logic clk,
                input  logic reset,
                output logic [N-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= q+1;
endmodule
```

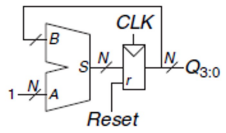
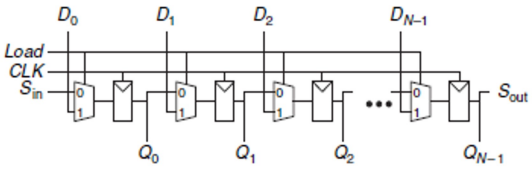


Figure 5.31 *N*-bit counter

- **Shift Register**
  - has a clk, serial input Sin, serial output Sout and N parallel outputs Q_N-1:0
  - on each rising edge of clk, a new bit is shifted in from Sin and all subsequent contents are shifted forward
  - last bit in shift register is available at Sout
  - Shift registers are serial to parallel converters
  - input is provided serially one bit at a time at Sin
  - after N cycles, past N inputs are available in parallel at Q




```
module shiftreg #(parameter N=8)
                 (input  logic         clk,
                  input  logic         reset, load,
                  input  logic         sin,
                  input  logic [N-1:0] d,
                  output logic [N-1:0] q,
                  output logic         sout);

  always_ff @(posedge clk, posedge reset)
    if (reset)       q <= 0;
    else if (load)   q <= d;
    else             q <= {q[N-2:0], sin};

  assign sout = q[N-1];
endmodule
```