DDCA 3.4, 4.6, 5.1-5.3, Appendix A

Monday, January 25, 2021 10:20 AM

3.4 Finite State Machines

- Synchronous sequential circuits can be drawn in FSM form

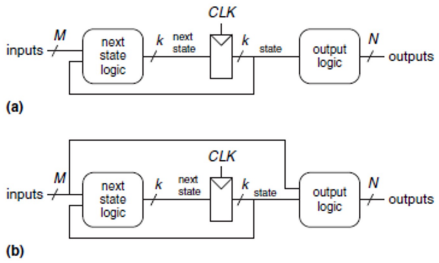
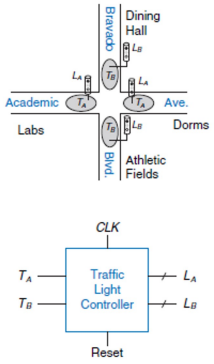
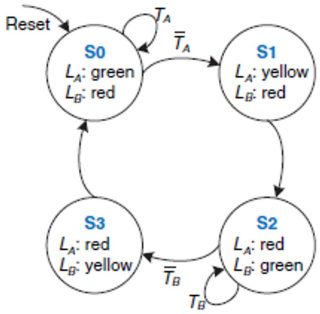


Figure 3.22 Finite state machines: (a) Moore machine, (b) Mealy machine

- a circuit with k registers can be in one of a finite number 2^k of unique states
  - has M inputs, N outputs, k bits of state
  - receives a clock and sometimes reset signal
  - 2 blocks of combinational logic, next state logic and output logic and a register that stores the state
  - on each clk edge, the FSM advances to the next state
  - Moore machines: outputs depends only on current state of machine
  - Mealy machines: outputs depend on both current state and current inputs
- **FSM Design**





  - **State transition diagram:** indicate all possible states of system and transitions
    - when system reset, lights are green on Aca Ave, red on Bravado Blvd.
    - every 5 seconds, controller examines traffic pattern
    - If traffic on Academic Ave, lights do not change
    - When no traffic, lights become yellow for 5 sec, turns red, Bravado Blvd lights turn green
    - Similarly light remains green on Bravado Blvd if traffic present
  - **State transition table**
    - indicates for each state and input what next state S' should be
    - usde don't care X whenver next state does not depend on input

Table 3.1 State transition table

Current State S	Inputs T_A	Inputs T_B	Next State S'
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0

Table 3.2 State encoding

State	Encoding $S_{1:0}$
S0	00
S1	01
S2	10
S3	11

Table 3.3 Output encoding

Output	Encoding $L_{1:0}$
green	00
yellow	01
red	10

$$S_1' = \overline{S}_1 S_0 + S_1 \overline{S}_0 \overline{T}_B + S_1 \overline{S}_0 T_B$$

$$S_0' = \overline{S}_1 \overline{S}_0 \overline{T}_A + S_1 \overline{S}_0 \overline{T}_B$$

  - **Sum of Products**
  - Simplify using Karnaugh maps or inspection

Current State S_1	Current State S_0	Inputs T_A	Inputs T_B	Next State S_1'	Next State S_0'
0	0	0	X	0	1
0	0	1	X	0	0

Current State S_1	Current State S_0	Outputs L_{A1}	Outputs L_{A0}	Outputs L_{B1}	Outputs L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0

$$L_{A1} = S_1$$

$$L_{A0} = \overline{S}_1 S_0$$

$$L_{B1} = \overline{S}_1$$

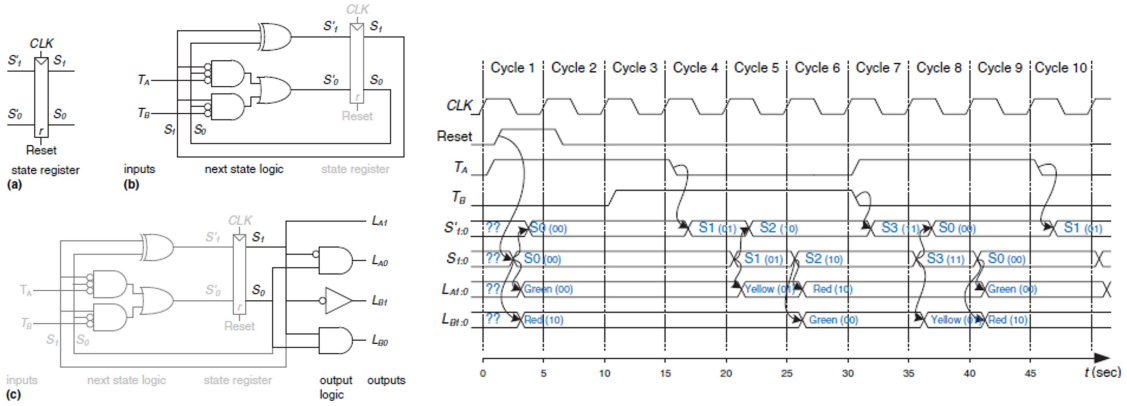
Current State S_1	Current State S_0	Inputs T_A	Inputs T_B	Next State S'_1	Next State S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

Current State S_1	Current State S_0	Outputs L_{A1}	Outputs L_{A0}	Outputs L_{B1}	Outputs L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

$$L_{A1} = S_1$$
$$L_{A0} = \overline{S}_1 S_0$$
$$L_{B1} = \overline{S}_1$$
$$L_{B0} = S_1 S_0$$

$$S'_1 = S_1 \oplus S_0$$
$$S'_0 = \overline{S}_1 \overline{S}_0 \overline{T}_A + S_1 \overline{S}_0 \overline{T}_B$$

- Sketch Moore FSM
  - 2 bit state register
  - each clk edge, state register copies next state
  - receives sync or async reset to initialize FSM at startup
  - compute next state from current state and inputs



- **State Encodings**
  - state and output encodings were selected arbitrarily
  - finding he best encoding by inspection
    - binary encoding vs one-hot encoding
  - **Binary encoding**
    - each state represented as a binary nubmer, K binary numbers represented by log_2(K)
    - uses two bits of state, no inputs, next state depends only on current state

Table 3.9 State transition table with binary encoding

Current State S_1	Current State S_0	Next State S'_1	Next State S'_0
0	0	0	1
0	1	1	0
1	0	0	0

$$S'_1 = \overline{S}_1 S_0$$
$$S'_0 = \overline{S}_1 \overline{S}_0$$
$$Y = \overline{S}_1 \overline{S}_0$$

  - **One-hot encoding**
    - a separate bit of state used for each state
    - one bit is TRUE (hot) at any time
    - ex. one hot encoded FSM with 3 states would have state encodings of 001, 010, and 100
    - each bit of state is stored in flip flop (requires more flip flops than binary), but fewer gates required

Table 3.10 State transition table with one-hot encoding

Current State S_2	Current State S_1	Current State S_0	Next State S'_2	Next State S'_1	Next State S'_0
0	0	1	0	1	0
0	1	0	1	0	0
1	0	0	0	0	1

$$S'_2 = S_1$$
$$S'_1 = S_0$$
$$S'_0 = S_2$$
$$Y = S_0$$

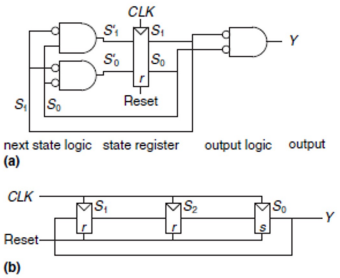


Figure 3.29 Divide-by-3 circuits for (a) binary and (b) one-hot encodings

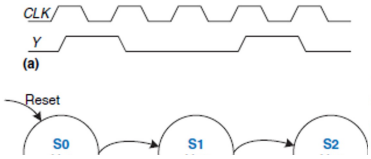


Table 3.6 Divide-by-3 counter state transition table

Current State	Next State

Table 3.7 Divide-by-3 counter output table

Current State	Output

(a)

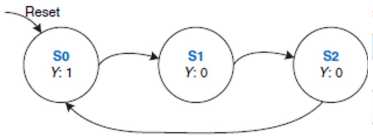


Table 3.6 Divide-by-3 counter state transition table

Current State	Next State
S0	S1
S1	S2
S2	S0

Table 3.7 Divide-by-3 counter output table

Current State	Output
S0	1
S1	0
S2	0

- **Moore and Mealy Machines**
  - Moore: outout depends on state of system
  - state transition diagrams for Moore machines labeled in the circles
  - Mealy machines outputs can depend on inputs as well as current state, outputs labeled on the arcs instead of in circles

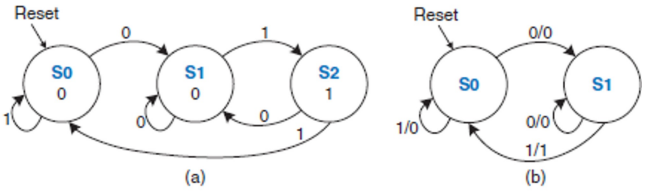


Figure 3.30 FSM state transition diagrams: (a) Moore machine, (b) Mealy machine

Table 3.11 Moore state transition table

Current State S	Input A	Next State S'
S0	0	S1
S0	1	S0
S1	0	S1
S1	1	S2
S2	0	S1
S2	1	S0

Table 3.12 Moore output table

Current State S	Output Y
S0	0
S1	0
S2	1

Table 3.15 Mealy state transition and output table

Current State S	Input A	Next State S'	Output Y
S0	0	S1	0
S0	1	S0	0
S1	0	S1	0
S1	1	S0	1

Table 3.13 Moore state transition table with state encodings

Current State S_1	Current State S_0	Input A	Next State S_1'	Next State S_0'
0	0	0	0	1
0	0	1	0	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0

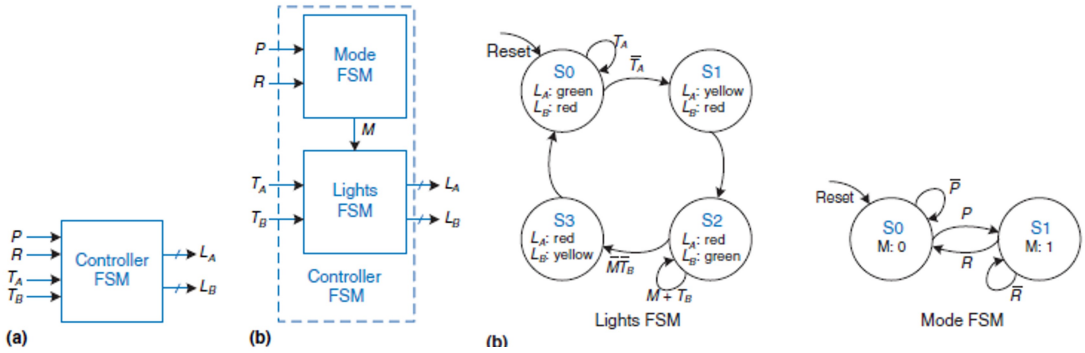
Table 3.14 Moore output table with state encodings

Current State S_1	Current State S_0	Output Y
0	0	0
0	1	0
1	0	1

Table 3.16 Mealy state transition and output table with state encodings

Current State S_0	Input A	Next State S_0'	Output Y
0	0	1	0
0	1	0	0
1	0	1	0
1	1	0	1

- **Factoring State Machines**
  - application of hierarchy and modularity = factoring of state machines
  - single vs factored designs for modified traffic light controller

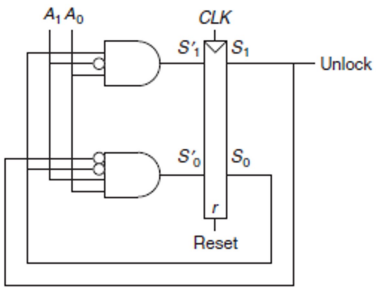


- **Deriving an FSM from a Schematic**
  - Examine circuit, stating inputs, outputs, state bits
  - write next state and output equations
  - create next state and output tables
  - reduce next state table to eliminate unreachable states
  - assign each valid state bit combination a name
  - rewrite next state and output tables with state names
  - draw state transition diagram
  - state in words what the FSM does

Table 3.17 Next state table derived from circuit in Figure 3.35

Current State	Input	Next State

- rewrite next state and output tables with state names
- draw state transition diagram
- state in words what the FSM does



$$S'_1 = S_0 \overline{A_1} A_0$$
$$S'_0 = \overline{S_1}\,\overline{S_0} A_1 A_0$$
$$Unlock = S_1$$

Figure 3.35 Circuit of found FSM for Example 3.9

Table 3.17 Next state table derived from circuit in Figure 3.35

Current State S_1	Current State S_0	Input A_1	Input A_0	Next State S'_1	Next State S'_0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	1	0
1	1	1	0	0	0
1	1	1	1	0	0

Table 3.18 Output table derived from circuit in Figure 3.35

Current State S_1	Current State S_0	Output *Unlock*
0	0	0
0	1	0
1	0	1
1	1	1

Table 3.19 Reduced next state table

Current State S_1	Current State S_0	Input A_1	Input A_0	Next State S'_1	Next State S'_0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	X	X	0	0

Table 3.20 Reduced output table

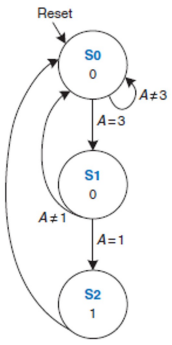
Current State S_1	Current State S_0	Output *Unlock*
0	0	0
0	1	0
1	0	1

Table 3.21 Symbolic next state table

Current State S	Input A	Next State S'
S0	0	S0
S0	1	S0
S0	2	S0
S0	3	S1
S1	0	S0
S1	1	S2
S1	2	S0
S1	3	S0
S2	X	S0

Table 3.22 Symbolic output table

Current State S	Output *Unlock*
S0	0
S1	0
S2	1



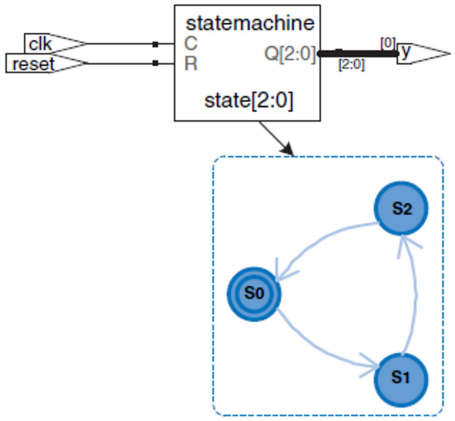
- **FSM Review**
  - Designing FSM:
    - Identify intputs outputs
    - sketch state transition diagram
    - Moore Machine:
      - write state transition table
      - write output table
    - Mealy machine:
      - write combined state transition and output table
    - select state encodings, your selection affects the hardware design
    - write boolean equations for next state and output logic

- sketch circuit schematic

4.6 Finite State Machines

- state register and 2 blocks of combinational logic to compute next state and output given current state and input
- 3 parts: state register, next state logic, output logic
- Divide by 3 FSM:

```
module divideby3FSM(input  logic clk,
                    input  logic reset,
                    output logic y);
  typedef enum logic [1:0] {S0, S1, S2} statetype;
  statetype [1:0] state, nextstate;

  // state register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always_comb
    case (state)
      S0:      nextstate <= S1;
      S1:      nextstate <= S2;
      S2:      nextstate <= S0;
      default: nextstate <= S0;
    endcase

  // output logic
  assign y = (state == S0);
endmodule
```



- typedef = statetype to be 2bit logic with S0, S1, or S2, state and nextstate are statetype signals
- encodings can be set by user
- S0 is reset state
- if wanted output to be HIGH in states S0 and S1

```
// output logic
assign y = (state == S0 | state == S1);
```

- Pattern Recognizer Moore FSM
  - use case and if statements to handle next state and output logic that depends on inputs and current state

```
module patternMoore(input  logic clk,
                    input  logic reset,
                    input  logic a,
                    output logic y);

  typedef enum logic [1:0] {S0, S1, S2} statetype;
  statetype state, nextstate;

  // state register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always_comb
    case (state)
      S0: if (a) nextstate = S0;
          else   nextstate = S1;
      S1: if (a) nextstate = S2;
          else   nextstate = S1;
      S2: if (a) nextstate = S0;
          else   nextstate = S1;
      default:   nextstate = S0;
    endcase

  // output logic
  assign y = (state == S2);
endmodule
```

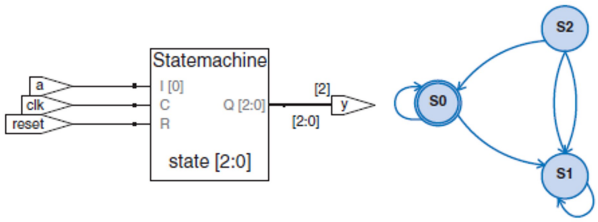


Figure 4.26 `patternMoore` **synthesized circuit**

- Pattern Recognizer Mealy FSM

```
module patternMealy(input  logic clk,
                    input  logic reset,
                    input  logic a,
                    output logic y);

  typedef enum logic {S0, S1} statetype;
  statetype state, nextstate;

  // state register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always_comb
    case (state)
      S0: if (a) nextstate = S0;
          else  nextstate = S1;
      S1: if (a) nextstate = S0;
          else  nextstate = S1;
      default:  nextstate = S0;
    endcase

  // output logic
  assign y = (a & state == S1);
endmodule
```

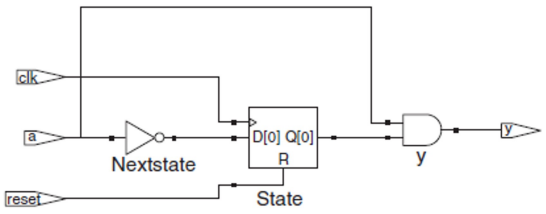
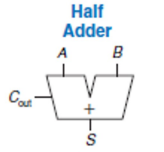


Figure 4.27 `patternMealy` **synthesized circuit**

5.2 Arithmeitc Circuits

- **Addition**
  - **Half Adder**
    - 1bit half adder
    - 2 inputs A and B, two outputs S and Cout
    - S = sum of A+B, if A and B are both 1, S is 2, cant be represented with single binary
    - indicated with carry out Cout
    - can be built from XOR and AND gate
    - multi-bit adder, Cout is added or carried in to the next most significant bit
    - half adder lacks a Cin input to accept Cout of previous column



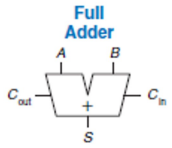
A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S = A \oplus B$
$C_{out} = AB$

  - **Full Adder**
    - accepts carry in Cin input unlike half adder

```
  1
 0001
+0101
 0110
```

Carry bit



C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$S = A \oplus B \oplus C_{in}$
$C_{out} = AB + AC_{in} + BC_{in}$

  - **Carry Propagate Adder (CPA)**
    - N-bit adder sums two N-bit inputs A and B and carry in Cin to produce an N bit result S and carry out Cout
    - carry out of 1 bit propagates to next bit
    - A B and S are busses

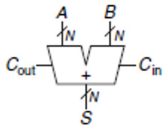


Figure 5.4 Carry propagate adder

  - **Ripple Carry Adder**
    - build N-bit carry propagate adder, chain N full adders
    - Cout of one stage acts as Cin of next stage
    - full adder module reused to form system

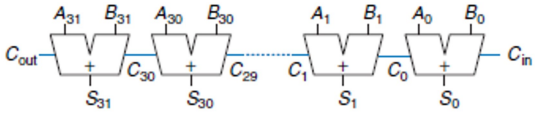
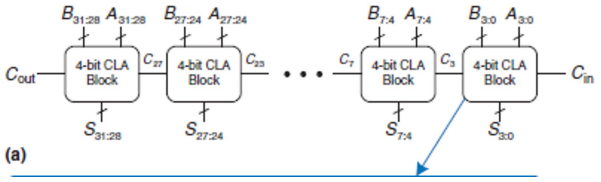


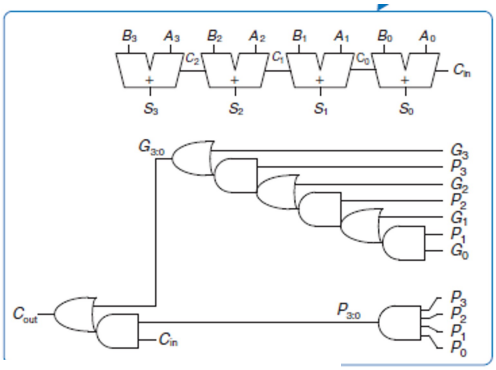
Figure 5.5 32-bit ripple-carry adder

- delay of full Adder: t_ripple = N*t_fa

- **Carry lookahead Adder (CLA)**
  - divide adder into blocks to determine carry out of block as soon as carry in is known
  - look ahead across blocks rather than waiting to ripple through all full adders inside block

$$t_{CLA} = t_{pg} + t_{pg\_block} + \left(\frac{N}{k} - 1\right) t_{AND\_OR} + k t_{FA}$$





```
module adder #(parameter N = 8)
              (input  logic [N-1:0] a, b,
               input  logic         cin,
               output logic [N-1:0] s,
               output logic         cout);

  assign {cout, s} = a + b + cin;
endmodule
```

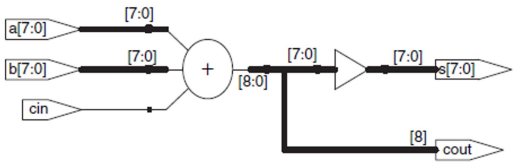
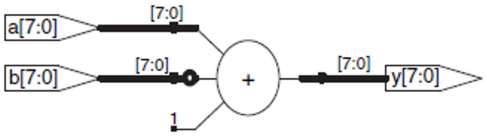


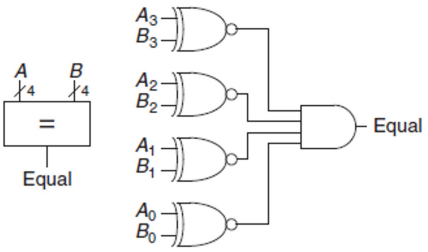
Figure 5.8 Synthesized adder

- **Subtraction**
  - Y = A-B, create two's complement of B: invert bits of B to obtain ~B and add 1 to get -B=~B+1, add to A to get Y=A+~B+1 = A-B
  - can be performed with CPA by adding A+~B with Cin=1

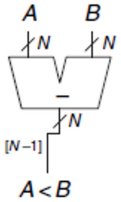
```
module subtractor #(parameter N = 8)
                   (input  logic [N-1:0] a, b,
                    output logic [N-1:0] y);

  assign y = a - b;
endmodule
```



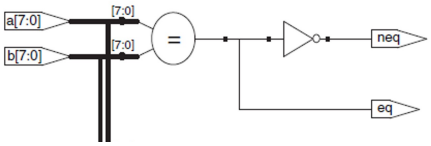
- **Comparator**
  - two binary numbers are equal or if one is greater or less than the other
  - receives two N bit binary numbers A and B
  - **equality comparator** produces 1 output indicating whether A is = to B
    - checks to determine whether corresponding bits in each col of A and B equal using XNOR
    - numbers equal if all of the columns are equal



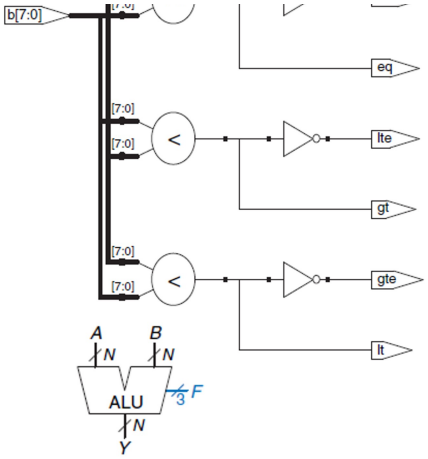
  - **magnitude comparator** produces 1+ outputs indicating relative values of A and B
    - computing A-B and looking at sign (most significant bit)
    - if neg (sign bit 1) A < B, otherwise A >= B



```
module comparator #(parameter N = 8)
                   (input  logic [N-1:0] a, b,
                    output logic eq, neq, lt, lte, gt, gte);

  assign eq  = (a == b);
  assign neq = (a != b);
```



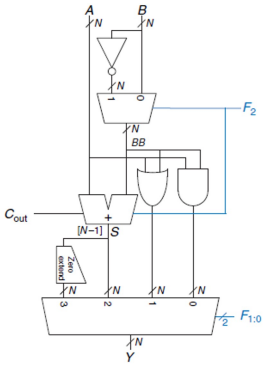
```
                output logic eq, neq, lt, lte, gt, gte);

  assign eq  = (a == b);
  assign neq = (a != b);
  assign lt  = (a < b);
  assign lte = (a <= b);
  assign gt  = (a > b);
  assign gte = (a >= b);
endmodule
```



- **ALU**
  - Arithmetic logical Unit
  - perform addition, subtraction, mag compare AND, and OR
  - signal F specifies which action to perform

Table 5.1 ALU operations

$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A – B
111	SLT



- **Shifters and Rotators**
  - move bits and multiply/divide by powers of 2
  - **shifter**
    - shifts binary number left or right by specified number of positions
    - **Logical shifter:** shift number to left LSL, or right LSR and fills empty spots with 0

      $11001 \text{ LSR } 2 = 00110;\ 11001 \text{ LSL } 2 = 00100$
    - **arithmetic shifter**
      - on right shifts, fills most significant bits with copy of old most significant bit (msb)
      - Arithmetic shift left (ASL) same as logical shift left (LSL)
      - useful to multiply and divide signed numbers

    $11001 \text{ ASR } 2 = 11110;\ 11001 \text{ ASL } 2 = 00100$
  - **rotator**
    - rotates number in circle such that empty spots are filled with bits shifted off other end

    $11001 \text{ ROR } 2 = 01110;\ 11001 \text{ ROL } 2 = 00111$

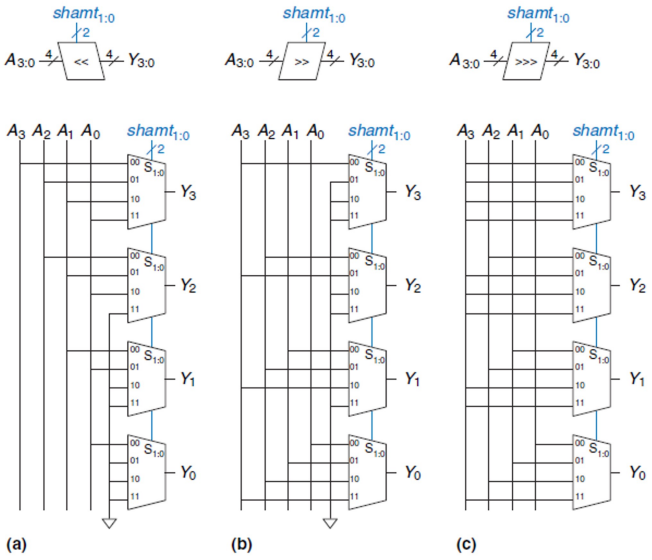
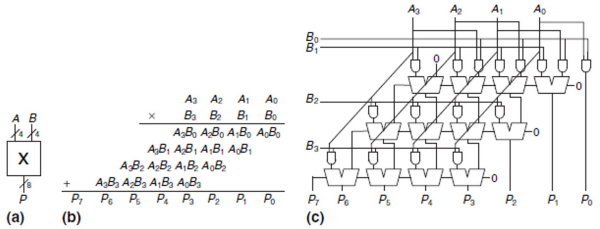


Figure 5.16 4-bit shifters: (a) shift left, (b) logical shift right, (c) arithmetic shift right

- **Multiplication**
  - multiplication of unsigned binary numbers

```
   230    multiplicand        0101
 ×  42    multiplier        × 0111
  460     partial             0101
+ 920     products            0101
 9660                         0101
                            + 0000
           result            0100011
```

2 30×42=9660 5×7=35




```
module multiplier #(parameter N = 8)
                   (input  logic [N-1:0]   a, b,
                    output logic [2*N-1:0] y);

  assign y = a * b;
endmodule
```

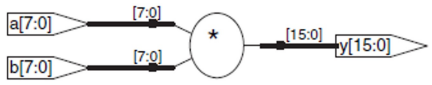


Figure 5.19 Synthesized multiplier

- **Division**

```
R' = 0
for i = N-1 to 0
  R = {R' << 1, A_i}
  D = R - B
  if D < 0 then    Q_i = 0, R' = R    // R < B
  else             Q_i = 1, R' = D    // R ≥ B
R = R'
```

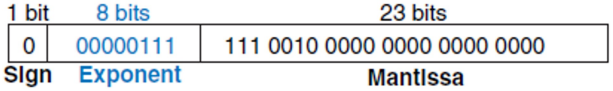
5.3 Number Systems

- **Fixed point number systems**
  - binary point between integer and fraction bits, analogous to decimal point
  - ex. 0.75+(-0.625)
    - convert 0.625 to fixed point binary: 0.625 >=2^-1
    - 1 in 2^-1 col, 0.625-0.5=0.125
    - 0.125<2^-2, 0 in 2^-2 col
    - 0.125>=2^-3, 1 in 2^-3 col, -.125-0.125=0
    - there must be a -0 in 2^-4 col
    - therefore, 0.625_10 = 0000.1010_2

```
 0000.1010   Binary Magnitude
 1111.0101   One's Complement
+       1    Add 1
 1111.0110   Two's Complement
```

```
 0000.1100          0.75
+1111.0110        + (–0.625)
 10000.0010         0.125
    (a)               (b)
```

- **Floating point number systerms**
  - sign, mantissa M, base B, exponent E,



  - ex. 228: convert to binary: 228_10 = 11100100_2 = 1.11001_2 × 2^7
  - 32 bit encoding, sign bit pos 0, 8 exp bits give value 7, remaining 23 bits is the mantissa

(a) 01101100

(b) 0110.1100

(c) $2^2+2^1+2^{-1}+2^{-2}=6.75$

Figure 5.21 Fixed-point notation of 6.75 with four integer bits and four fraction bits

(a) 0010.0110

(b) 1010.0110

(c) 1101.1010

Figure 5.22 Fixed-point representation of −2.375: (a) absolute value, (b) sign and magnitude, (c) two's complement