CPE/EE 422/522
Advanced Logic Design
L04

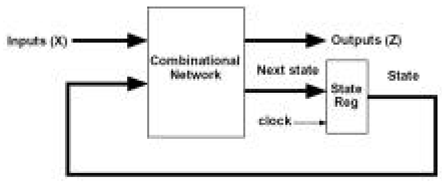
Electrical and Computer Engineering
University of Alabama in Huntsville

Outline

- What we know
  - Combinational Networks
    - Analysis, Synthesis, Simplification, Hazards, Building Blocks, PALs, PLAs, ROMs
  - Sequential Networks: Basic Building Blocks
  - Design: Mealy
  - Setup and hold times, Max clock frequency
- What we do not know
  - Design: Moore
  - Equivalent States
  - State Table Reduction
  - Intro to VHDL

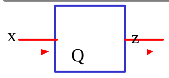
Review: Mealy Sequential Networks

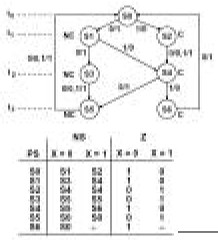
General model of Mealy Sequential Network



(1) X inputs are changed to a new value
(2) After a delay, the Z outputs and next state appear at the output of CM
(3) The next state is clocked into the state register and the state changes

Review: 8421 BCD to Excess3 BCD Code Converter

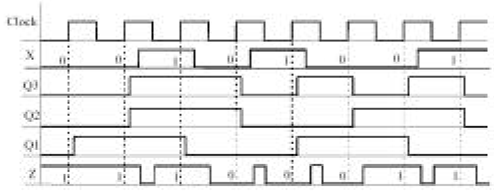




X (inputs)				Z (outputs)			
t3	t2	t1	t0	t3	t2	t1	t0
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

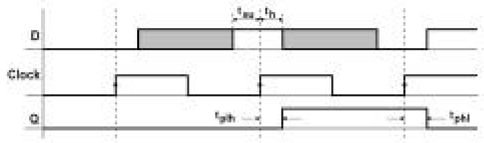
Sequential Network Timing (cont'd)

Timing diagram assuming a propagation delay
of 10 ns for each flip-flop and gate
(State has been replaced with the state of three flip-flops)



Setup and Hold Times

- For a real D-FF
  - D input must be stable for a certain amount of time before the active edge of clock cycle => *Setup time*
  - D input must be stable for a certain amount of time after the active edge of the clock => *Hold time*
- Propagation time: from the time the clock changes to the time the output changes



Manufacturers provide minimum tsu, th, and maximum tplh, tphl

Maximum Clock Frequency

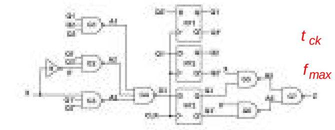
$t_{c\ max}$ - Max propagation delay through the combinational network

$t_{p\ max}$ - Max propagation delay from the time the clock changes to the flip-flop output changes {= max(tplh, tphl)}

t_{ck} - Clock period

$$t_{c\ max} + t_{p\ max} \le t_{ck} - t_{su}$$

$$t_{ck} \ge t_{c\ max} + t_{p\ max} + t_{su}$$



Example:

$t_{p\ max} = 15\ ns$, $t_{su} = 5\ ns$,

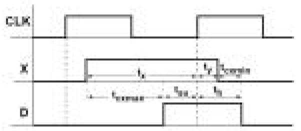
$t_{gate} = 15\ ns$

$t_{ck} = 2 * 15 + 15 + 5 = 50\ ns$

$f_{max} = \dfrac{1}{50\ ns} = 20\ MHz$

Hold Time Violation

- Occur if the change in Q fed back through the combinational network and cause D to change too soon after the clock edge



Hold time is satisfied if:

$t_{p\ min} + t_{c\ min} \ge t_h$

What about X?

Make sure that input changes propagate to the flip-flops inputs such that setup time is satisfied.

$t_x \ge t_{cx\ max} + t_{su}$

Make sure that X does not change too soon after the clock.

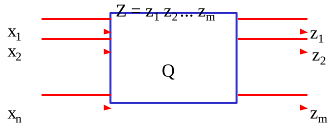
If X changes at time ty after the active edge, hold time is satisfied if

$t_y \ge t_h - t_{cx\ min}$

Moore Sequential Networks

Outputs depend only on present state!

X = $x_1\ x_2 ...\ x_n$
Q = $Q_1\ Q_2 ...\ Q_k$
Z = $z_1\ z_2 ...\ z_m$

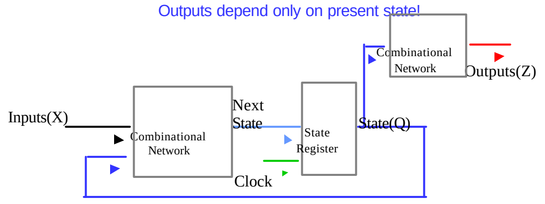


$$Z(t) = F(Q(t))$$

$$Q(t^+) = G(X(t), Q(t))$$

General Model of Moore Sequential Machine

Outputs depend only on present state!

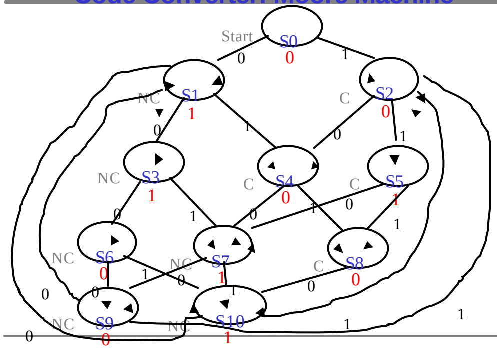


X = $x_1\ x_2 ...\ x_n$
Q = $Q_1\ Q_2 ...\ Q_k$
Z = $z_1\ z_2 ...\ z_m$

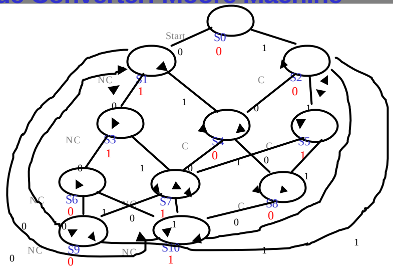
$$Q(t^+) = G(X(t), Q(t))$$

$$Z(t) = F(Q(t))$$

Code Converter: Moore Machine



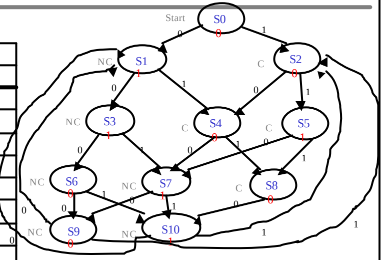
Code Converter: Moore Machine



Do we need state S0?
How many states does Moore machine have?
How many states does Mealy machine have?

Moore Machine: State Table

PS	NS		Z
	X=0	X=1	
S0	S1	S2	0
S1	S3	S4	1
S2	S4	S5	0
S3	S6	S7	1
S4	S7	S8	0
S5	S7	S8	1
S6	S9	S10	0
S7	S9	S10	1
S8	S10	-	0
S9	S1	S2	0
S10	S1	S2	1

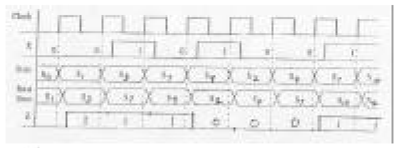


Note: state S0 could be eliminated (S0 == S9), if S9 was start state!

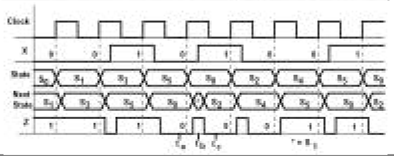
Moore Machine Timing

- X = 0010_1001 => Z = 1110_0011

Moore



Mealy



State Assignments

Guidelines to reduce the amount of combinational logic

I. States which have the same next state (NS) for a given input should be given adjacent assignments (look at the columns of the state table).

II. States which are the next states of the same state should be given adjacent assignments (look at the rows).

III. States which have the same output for a given input should be given adjacent assignments.

Rule I: (S0, S9, S10), (S4, S5), (S6, S7)

Rule II: (S1, S2), (S3, S4), (S4, S5), (S6, S7), (S7, S8), (S9, S10)

Rule III: (S0, S2, S4, S6, S8, S9)

(S1, S3, S5, S7, S10)

S0 – 0010

S1 - 0111

….

S10 - 0100

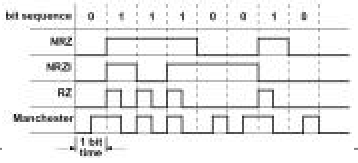
Q3Q4 \ Q1Q2	00	01	11	10
00	S9	s10	S8	
01				S5
11		S1	S3	S4
10	S0	S2	S7	S6

PS	NS		Z
	X=0	X=1	
S0	S1	S2	0
S1	S3	S4	1
S2	S4	S5	0
S3	S6	S7	1
S4	S7	S8	0
S5	S7	S8	1
S6	S9	S10	0
S7	S9	S10	1
S8	S10	-	0
S9	S1	S2	0
S10	S1	S2	1

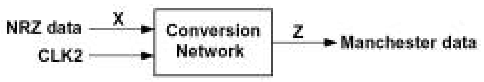
Moore Machine: Another Example

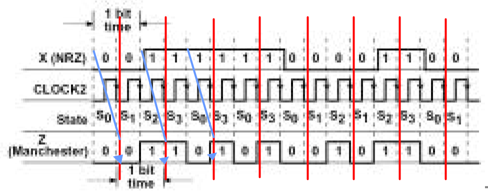
A Converter for Serial Data Transmission: NRZ-to-Manchester

- Coding schemes for serial data transmission
  - NRZ: nonreturn-to-zero
  - NRZI: nonreturn-to-zero-inverted
    - 0 in input sequence – the bit transmitted is the same as the previous bit;
    - 1 in input sequence – transmit the complement of the previous bit
  - RZ: return-to-zero
    - 0 – 0 for full bit time; 1 – 1 for the first half, 0 for the second half
  - Manchester

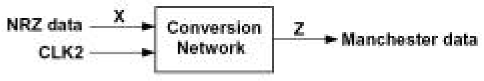


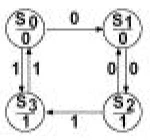
Moore Network for NRZ-to-Manchester





Moore Network for NRZ-to-Manchester

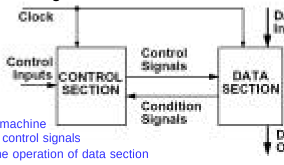




Present State	Next State X = 0	Next State X = 1	Present Output (Z)
S_0	S_1	S_3	0
S_1	S_2	–	0
S_2	S_1	S_3	1
S_3	–	S_0	1

Synchronous Design

- Use a clock to synchronize the operation of all flip-flops, registers, and counters in the system
  - all changes occur immediately following the active clock edge
  - clock period must be long enough so that all changes flip-flops, registers, counters will have time to stabilize before the next active clock edge
- Typical design: Control section + Data Section



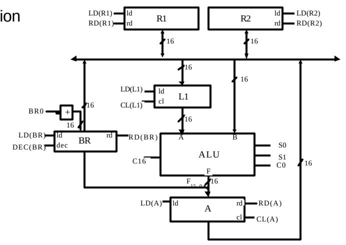
Data registers
Arithmetic Units
Counters
Buses, Muxes, …

Sequential machine
to generate control signals
to control the operation of data section

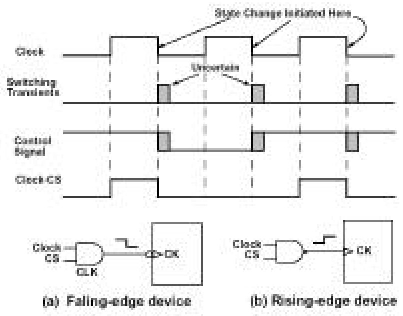
An Example

- Data section // s= n*(n+a) // R1=n, R2=a // R1=s
- Design flowchart for SMUL operation
- Design Control section
- S0 S1 F

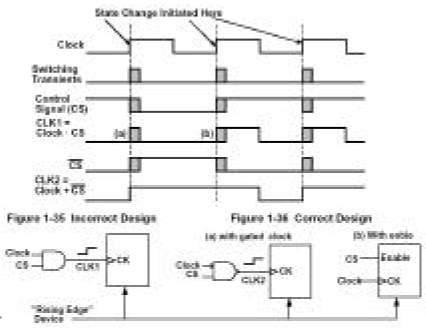
S0	S1	F
0	0	B
0	1	B – C0
1	0	B + C0
1	1	A + B



Timing Chart for System with Falling-edge Devices



Timing Chart for System with Rising-edge Devices



Principles of Synchronous Design

- Method
  - All clock inputs to flip-flops, registers, counters, etc., are driven directly from the system clock or from the clock ANDed with a control signal
- Result
  - All state changes occur immediately following the active edge of the clock signal
- Advantage
  - All switching transients, switching noise, etc., occur between the clock pulses and have no effect on system performance

Asynchronous Design

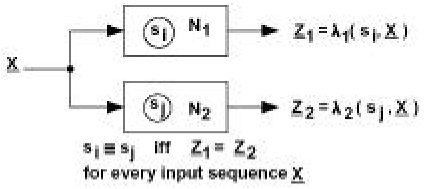
- Disadvantage - More difficult
  - Problems
    - Race conditions: final state depends on the order in which variables change
    - Hazards
  - Special design techniques are needed to cope with races and hazards
- Advantages = Disadvantages of Synchronous Design
  - In high-speed synchronous design propagation delay in wiring is significant => clock signal must be carefully routed so that it reaches all devices at essentially same time
  - Inputs are not synchronous with the clock – need for synchronizers
  - Clock cycle is determined by the worst-case delay

To Do

- Read
  - Textbook chapters 1.6, 1.7, 1.8, 1.10, 1.11, 1.12

Equivalent States

- Two state are equivalent if we cannot tell them apart by observing input and output sequences


X
s_i N_1
$Z_1 = \lambda_1(s_i, \underline{X})$
s_j N_2
$Z_2 = \lambda_2(s_j, \underline{X})$
$s_i \equiv s_j$ iff $Z_1 = Z_2$
for every input sequence $\underline{X}$

Definition: Two states are equivalent si==sj only and only if, for every input sequence X, the output sequences Z1 and Z2 are the same.

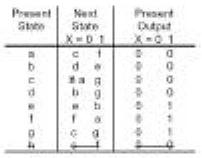
Not practical => try all sequences (what is the length of sequence?)

Equivalent States

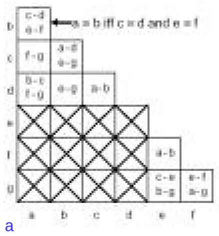
State Equivalence Theorem

- Two state are equivalent Si == Sj if and only if for every single input X, the outputs are the same and the next states are equivalent

State Table Reduction

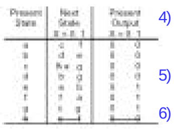


Present State	Next State X = 0	1	Present Output X = 0	1
a	c	f	0	0
b	d	e	0	0
c	~~h~~ a	g	0	0
d	b	g	0	0
e	e	b	0	1
f	f	a	0	1
g	c	g	0	1
~~h~~	~~c~~	~~f~~	~~0~~	~~0~~



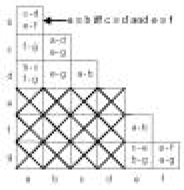
1) States a and h have the same next states and outputs (when X=0 and X=1)
2) Eliminate h from the table and replace with a
3) States a and b have the same output => they are same iff c==d and f==e.
   We say c-d and e-f are implied pairs for a-b.
   To keep track of the implied pairs we make an implication chart.

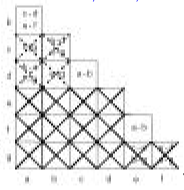
State Table Reduction



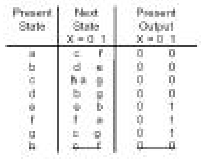
Present State	Next State X = 0	1	Present Output X = 0	1
a	c	f	0	0
b	d	e	0	0
c	~~h~~ a	g	0	0
d	b	g	0	0
e	e	b	0	1
f	f	a	0	1
g	c	g	0	1
~~h~~	~~c~~	~~f~~	~~0~~	~~0~~

4) Make another pass through the chart.
   E-g cell contains c-e and b-g;
   since c-e cell contains x, c!=e => e!=g (put X).
5) Repeat the step 4 until no additional squares are X-ed. {Put X in f-g, a-c, a-d, b-c, b-d squares}.
6) The remaining squares indicate equivalent state pairs => a==b, c==d, e==f.

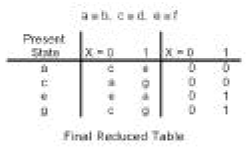




State Table Reduction



Present State	Next State X = 0	1	Present Output X = 0	1
a	c	f	0	0
b	d	e	0	0
c	~~h~~ a	g	0	0
d	b	g	0	0
e	e	b	0	1
f	f	a	0	1
g	c	g	0	1
~~h~~	~~c~~	~~f~~	~~0~~	~~0~~



a ≡ b, c ≡ d, e ≡ f

Present State	X = 0	1	X = 0	1
a	c	a	0	0
c	a	g	0	0
e	e	a	0	1
g	c	g	0	1

Final Reduced Table

Implication Table Method

- 1. Construct a chart that contains a square for each pair of states.
- 2. Compare each pair in the state table. If the outputs associated with states i and j are different, place an X in square i-j to indicate that i!=j.
  If outputs are the same, place the implied pairs in square i-j.
  If outputs and next states are the same (or i-j implies only itself), i==j.
- 3. Go through the implication table square by square.
  If square i-j contains the implied pair m-n, and square m-n contains X, then i!=j, and place X in square i-j.
- 4. If any Xs were added in step 3, repeat step 3 until no more Xs are added.
- 5. For each square i-j that does not contain an X, i==j.

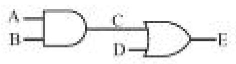
Intro to VHDL

- Technology trends
  – 1 billion transistor chip running at 20 GHz in 2007
- Need for Hardware Description Languages
  – Systems become more complex
  – Design at the gate and flip-flop level becomes very tedious and time consuming
- HDLs allow
  – Design and debugging at a higher level before conversion to the gate and flip-flop level
  – Tools for synthesis do the conversion
- VHDL, Verilog
- VHDL – VHSIC Hardware Description Language

Intro to VHDL

- Developed originally by DARPA
  – for specifying digital systems
- International IEEE standard (IEEE 1076-1993)
- Hardware Description, Simulation, Synthesis
- Provides a mechanism for digital design and reusable design documentation
- Support different description levels
  – Structural (specifying interconnections of the gates),
  – Dataflow (specifying logic equations), and
  – Behavioral (specifying behavior)
- Top-down, Technology Dependent

VHDL Description of Combinational Networks



Concurrent Statements

```
C <= A and B  after 5 ns;
E <= C or D   after 5 ns;
```

If delay is not specified, "delta" delay is assumed

```
C <= A and B;
E <= C or D;
```

Order of concurrent statements is not important

```
E <= C or D;
C <= A and B;
```

This statement executes repeatedly

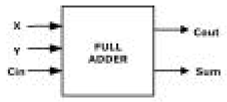
```
CLK <= not CLK after 10 ns;
```

This statement causes a simulation error

```
CLK <= not CLK;
```

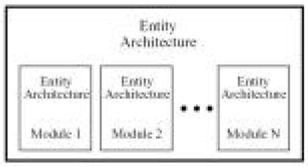
Entity-Architecture Pair

Full Adder Example



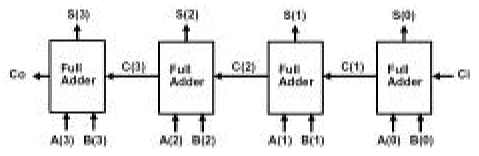
```
entity FullAdder is
  port (X, Y, Cin: in bit;       -- Inputs
    Cout, Sum: out bit);         -- Outputs
end FullAdder;

architecture Equations of FullAdder is
begin                            -- Concurrent Assignments
  Sum  <= X xor Y xor Cin after 10 ns;
  Cout <= (X and Y) or (X and Cin) or (Y and Cin) after 10 ns;
end Equations;
```

VHDL Program Structure



```
entity entity-name is
  [port(interface-signal-declaration);]
end [entity] [entity-name];

architecture architecture-name of entity-name is
  [declarations]
begin
  architecture body
end [architecture] [architecture-name];
```

4-bit Adder



```
entity Adder4 is
  port (A, B: in bit_vector(3 downto 0); Ci: in bit;     -- Inputs
        S: out bit_vector(3 downto 0); Co: out bit);     -- Outputs
end Adder4;
```

4-bit Adder (cont'd)

```
entity Adder4 is
  port (A, B: in bit_vector(3 downto 0); Ci: in bit;     -- Inputs
        S: out bit_vector(3 downto 0); Co: out bit);     -- Outputs
end Adder4;

architecture Structure of Adder4 is
component FullAdder
   port (X, Y, Cin: in bit;          -- Inputs
         Cout, Sum: out bit);        -- Outputs
end component;
signal C: bit_vector(3 downto 1);
begin    --instantiate four copies of the FullAdder
   FA0: FullAdder port map (A(0), B(0), Ci, C(1), S(0));
   FA1: FullAdder port map (A(1), B(1), C(1), C(2), S(1));
   FA2: FullAdder port map (A(2), B(2), C(2), C(3), S(2));
   FA3: FullAdder port map (A(3), B(3), C(3), Co, S(3));
end Structure;
```

4-bit Adder - Simulation

```
list A B Co C Ci S    -- put these signals on the output list
force A 1111          -- set the A inputs to 1111
force B 0001          -- set the B inputs to 0001
force Ci 1            -- set the Ci to 1
run 50                -- run the simulation for 50 ns
force Ci 0
force A 0101
force B 1110
run 50

ns  delta    a     b    co  c   ci   s
 0     +0   0000  0000   0 000   0 0000
 0     +1   1111  0001   0 000   1 0000
10     +0   1111  0001   0 001   1 1111
20     +0   1111  0001   0 011   1 1101
30     +0   1111  0001   0 111   1 1001
40     +0   1111  0001   1 111   1 0001
50     +0   0101  1110   1 111   0 0001
60     +0   0101  1110   1 110   0 0101
70     +0   0101  1110   1 100   0 0111
80     +0   0101  1110   1 100   0 0011
```

Modeling Flip-Flops Using VHDL Processes

General form of process

```
process(sensitivity-list)
  begin
    sequential-statements
  end process;
```

- Whenever one of the signals in the sensitivity list changes, the sequential statements are executed in sequence one time

Concurrent Statements vs. Process

A, B, C, D are integers
A=1, B=2, C=3, D=0
D changes to 4 at time 10

```
A <= B;   -- statement 1
B <= C;   -- statement 2
C <= D;   -- statement 3
```

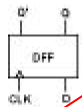
```
process (B, C, D)
begin
   A <= B;   -- statement 1
   B <= C;   -- statement 2
   C <= D;   -- statement 3
end process;
```

Simulation Results

time	delta	A	B	C	D	
0	+0	0	1	2	0	
10	+0	1	2	3	4	(stat. 3 exe.)
10	+1	1	2	4	4	(stat. 2 exe.)
10	+2	1	4	4	4	(stat. 1 exe.)
10	+3	4	4	4	4	(no exec.)

```
time  delta  A B C D
0     +0     1 2 3 0
10    +0     1 2 3 4  (statements 1,2,3 execute; then update A,B,C)
10    +1     2 3 4 4  (statements 1,2,3 execute; then update A,B,C)
10    +2     3 4 4 4  (statements 1,2,3 execute; then update A,B,C)
10    +3     4 4 4 4  (no further execution occurs)
```

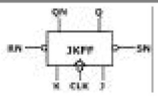
D Flip-flop Model



Bit values are enclosed in single quotes

```
entity DFF is
  port (D, CLK: in bit;
        Q: out bit;  QN: out bit := '1');
-- initialize QN to '1' since bit signals are initialized to '0' by default
end DFF;

architecture SIMPLE of DFF is
begin
  process (CLK)                    -- process is executed when CLK changes
  begin
     if CLK = '1' then             -- rising edge of clock
       Q  <= D after 10 ns;
       QN  <= not D after 10 ns;
     end if;
  end process;
end SIMPLE;
```

JK Flip-Flop Model

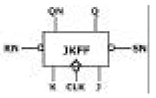


```
entity JKFF is
   port (SN, RN, J, K, CLK: in bit;          -- inputs
         Q: inout bit; QN: out bit := '1');  -- see Note 1
end JKFF;

architecture JKFF1 of JKFF is
begin
  process (SN, RN, CLK)                      -- see Note 2
  begin
    if RN = '0' then  Q<= '0' after 10 ns;                  -- RN=0 will clear the FF
    elsif SN = '0'  then  Q<= '1' after 10 ns;              -- SN=0 will set the FF
    elsif CLK = '0' and CLK'event then                      -- see Note 3
       Q <= (J and not Q) or (not K and Q) after 10 ns;     -- see Note 4
    end if;
  end process;
  QN <= not Q;                               -- see Note 5
end JKFF1;
```

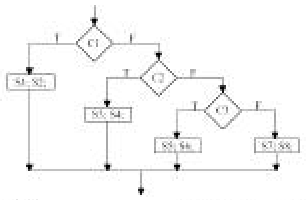
JK Flip-Flop Model

Note 1: Q is declared as inout (rather than out) because it appears on both the left and right sides of an assignment within the architecture.
Note 2: The flip-flop can change state in response to changes in SN, RN, and CLK, so these 3 signals are in the sensitivity list.
Note 3: The condition (CLK = '0' and CLK'event) is TRUE only if CLK has just changed from '1' to '0'.
Note 4: Characteristic equation which describes behavior of J-K flip-flop.
Note 5: Every time Q changes, QN will be updated. If this statement were placed within the process, the old value of Q would be used instead
of the new value.

```
entity JKFF is
   port (SN, RN, J, K, CLK: in bit;          -- inputs
         Q: inout bit; QN: out bit := '1');  -- see Note 1
end JKFF;

architecture JKFF1 of JKFF is
begin
  process (SN, RN, CLK)                      -- see Note 2
  begin
    if RN = '0' then  Q<= '0' after 10 ns;                  -- RN=0 will clear the FF
    elsif SN = '0'  then  Q<= '1' after 10 ns;              -- SN=0 will set the FF
    elsif CLK = '0' and CLK'event then                      -- see Note 3
       Q <= (J and not Q) or (not K and Q) after 10 ns;     -- see Note 4
    end if;
  end process;
  QN <= not Q;                               -- see Note 5
end JKFF1;
```



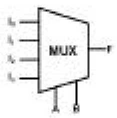
Using Nested IFs and ELSEIFs



```
if (C1) then S1; S2;
   else if (C2) then S3; S4;
         else if (C3) then S5; S6;
               else S7; S8;
         end if;
   end if;
end if;
```

```
if (C1) then S1; S2;
   elsif (C2) then S3; S4;
   elsif (C3) then S5; S6;
   else S7; S8;
end if;
```

VHDL Models for a MUX



```
F <= (not A and not B and I0) or
     (not A and B and I1) or
     (A and not B and I2) or
     (A and B and I3);
```

MUX model using a *conditional signal assignment statement*:

```
F <= I0 when Sel = 0
     else I1 when Sel = 1
     else I2 when Sel = 2
     else I3;
```

Sel represents the integer equivalent of a 2-bit binary number with bits A and B

If a MUX model is used inside a process, the MUX can be modeled using a CASE statement (cannot use a concurrent statement):

```
case Sel is
    when 0 => F <= I0;
    when 1 => F <= I1;
    when 2 => F <= I2;
    when 3 => F <= I3;
end case;
```

The case statement has the general form:

```
case expression is
    when choice1 => sequential statements1
    when choice2 => sequential statements2
    ...
    [when others => sequential statements]
end case;
```

MUX Models (1)

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity SELECTOR is
 port (
   A   : in  std_logic_vector(15 downto 0);
   SEL : in  std_logic_vector( 3 downto 0);
   Y   : out std_logic);
end SELECTOR;
```

```
architecture RTL1 of SELECTOR is
begin
 p0 : process (A, SEL)
 begin
  if   (SEL = "0000") then    Y <= A(0);
  elsif (SEL = "0001") then   Y <= A(1);
  elsif (SEL = "0010") then   Y <= A(2);
  elsif (SEL = "0011") then   Y <= A(3);
  elsif (SEL = "0100") then   Y <= A(4);
  elsif (SEL = "0101") then   Y <= A(5);
  elsif (SEL = "0110") then   Y <= A(6);
  elsif (SEL = "0111") then   Y <= A(7);
  elsif (SEL = "1000") then   Y <= A(8);
  elsif (SEL = "1001") then   Y <= A(9);
  elsif (SEL = "1010") then   Y <= A(10);
  elsif (SEL = "1011") then   Y <= A(11);
  elsif (SEL = "1100") then   Y <= A(12);
  elsif (SEL = "1101") then   Y <= A(13);
  elsif (SEL = "1110") then   Y <= A(14);
  else     Y <= A(15);
  end if;
 end process;
endRTL1;
```

MUX Models (2)

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity SELECTOR is
 port (
   A   : in  std_logic_vector(15 downto 0);
   SEL : in  std_logic_vector( 3 downto 0);
   Y   : out std_logic);
end SELECTOR;
```

```
architecture RTL3 of SELECTOR is
begin
 with SEL select
   Y <= A(0)  when "0000",
        A(1)  when "0001",
        A(2)  when "0010",
        A(3)  when "0011",
        A(4)  when "0100",
        A(5)  when "0101",
        A(6)  when "0110",
        A(7)  when "0111",
        A(8)  when "1000",
        A(9)  when "1001",
        A(10) when "1010",
        A(11) when "1011",
        A(12) when "1100",
        A(13) when "1101",
        A(14) when "1110",
        A(15) when others;
end RTL3;
```

MUX Models (3)

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity SELECTOR is
 port (
   A   : in  std_logic_vector(15 downto 0);
   SEL : in  std_logic_vector( 3 downto 0);
   Y   : out std_logic);
end SELECTOR;
```

```
architecture RTL2 of SELECTOR is
begin
 p1 : process (A, SEL)
 begin
  case SEL is
    when "0000" => Y <= A(0);
    when "0001" => Y <= A(1);
    when "0010" => Y <= A(2);
    when "0011" => Y <= A(3);
    when "0100" => Y <= A(4);
    when "0101" => Y <= A(5);
    when "0110" => Y <= A(6);
    when "0111" => Y <= A(7);
    when "1000" => Y <= A(8);
    when "1001" => Y <= A(9);
    when "1010" => Y <= A(10);
    when "1011" => Y <= A(11);
    when "1100" => Y <= A(12);
    when "1101" => Y <= A(13);
    when "1110" => Y <= A(14);
    when others => Y <= A(15);
  end case;
 end process;
endRTL2;
```

MUX Models (4)

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity SELECTOR is
 port (
   A   : in  std_logic_vector(15 downto 0);
   SEL : in  std_logic_vector( 3 downto 0);
   Y   : out std_logic);
end SELECTOR;
```

```
architecture RTL4 of SELECTOR is
begin
  Y <= A(conv_integer(SEL));
end RTL4;
```