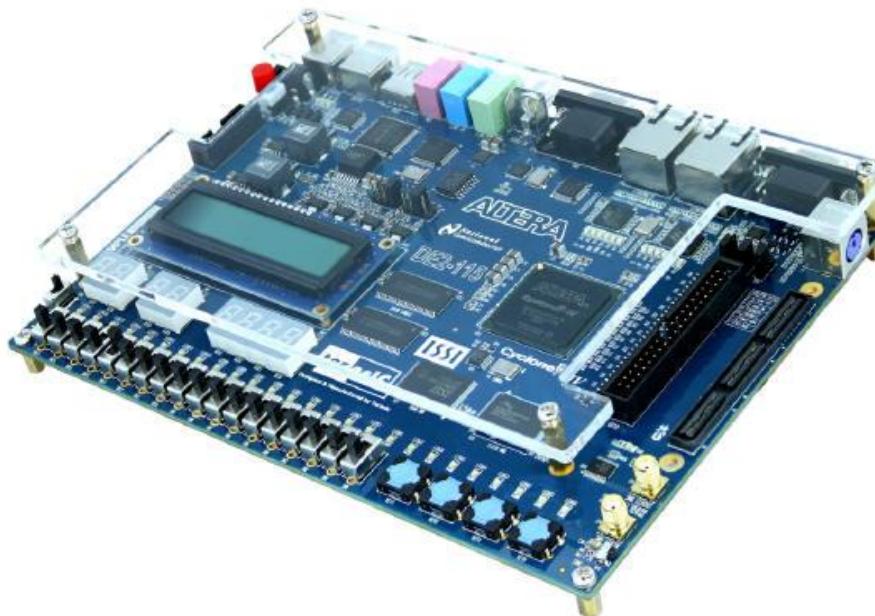


CPE 322

Digital Hardware Design Fundamentals

Electrical and Computer Engineering
UAH

Extended State Graph Finite State Machine
Representations

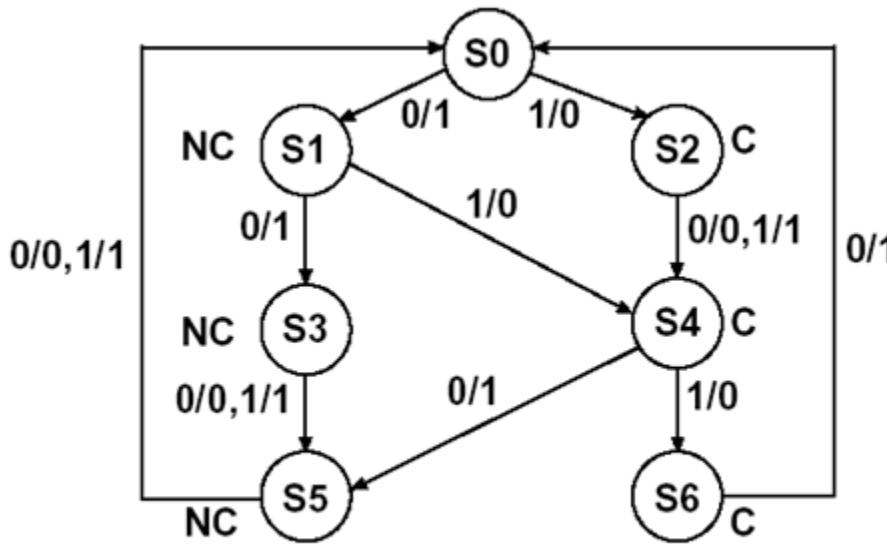


Extended State Graph

- Similarities to basic State (Transition) Graph
 - Nodes represent states
 - Arcs (or edges) represent transition between states
 - Labelling on arcs represent Inputs/Outputs on Mealy Machines
 - Labelling on arcs represent Inputs and labelling on nodes represent Outputs on Moore Machines
- Differences with basic State (Transition) Graph
 - To reduce Clutter
 - Only Inputs that impact a transition from one state to another are present on the graph – others are ignored.
 - Only Outputs that are TRUE (logic high) for a given transition are listed.

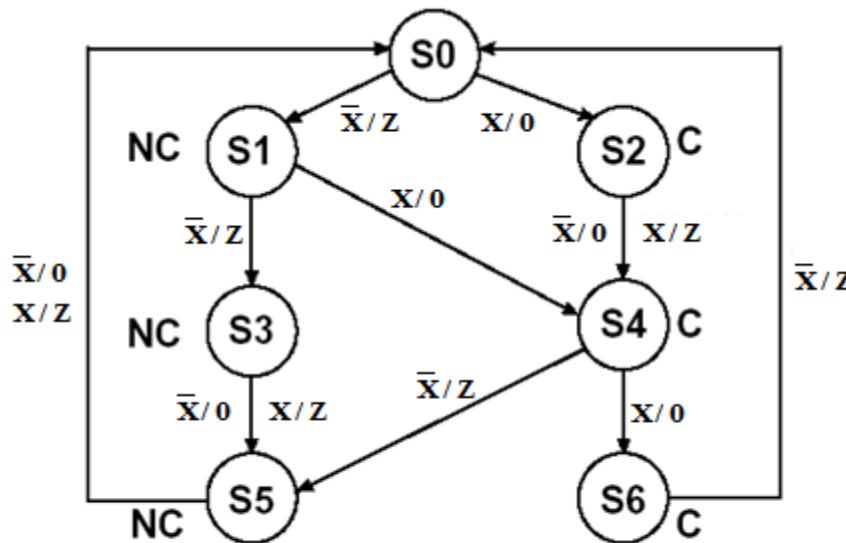
BCD to Excess 3 Mealy FSM Example

X/Z



Standard State Graph Notation

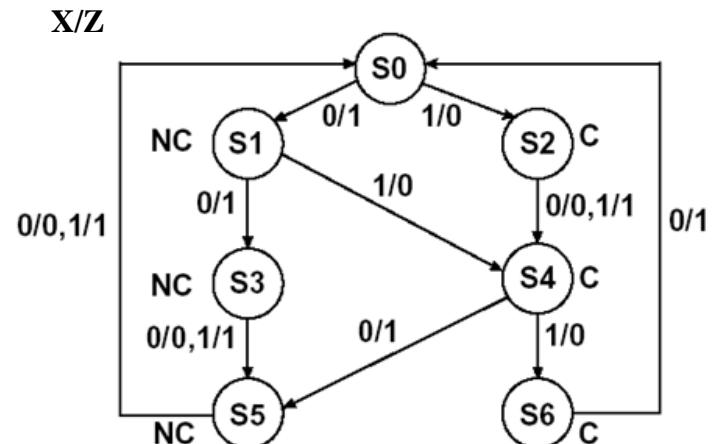
BCD to Excess 3 Mealy FSM Example



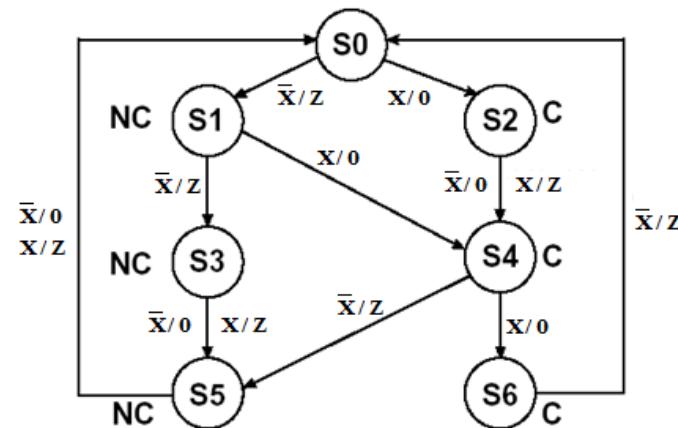
Extended State Graph Notation

- Only Inputs that impact a transition from one state to another are present on the graph – others are ignored.
- Only Outputs that are TRUE (logic high) for a given transition are listed.

BCD to Excess 3 Mealy FSM Example

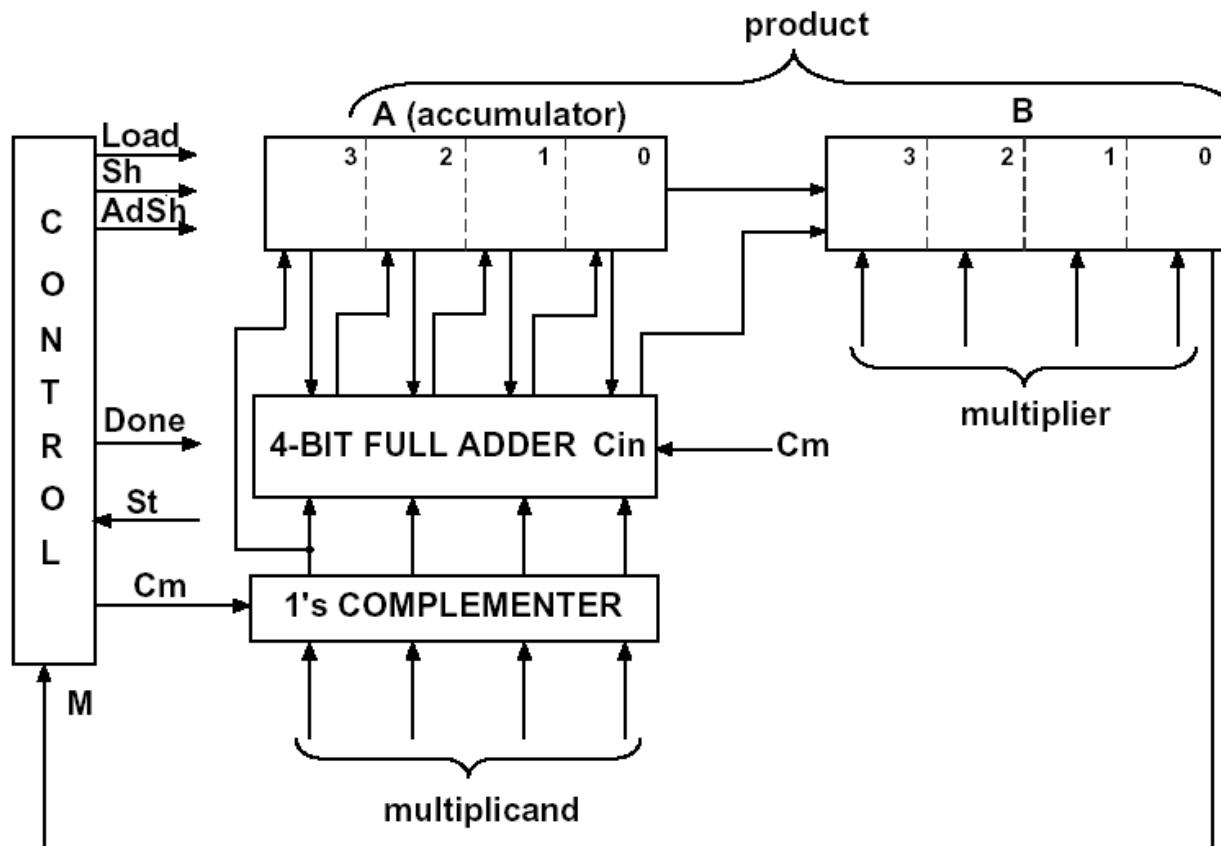


Standard State Graph Notation



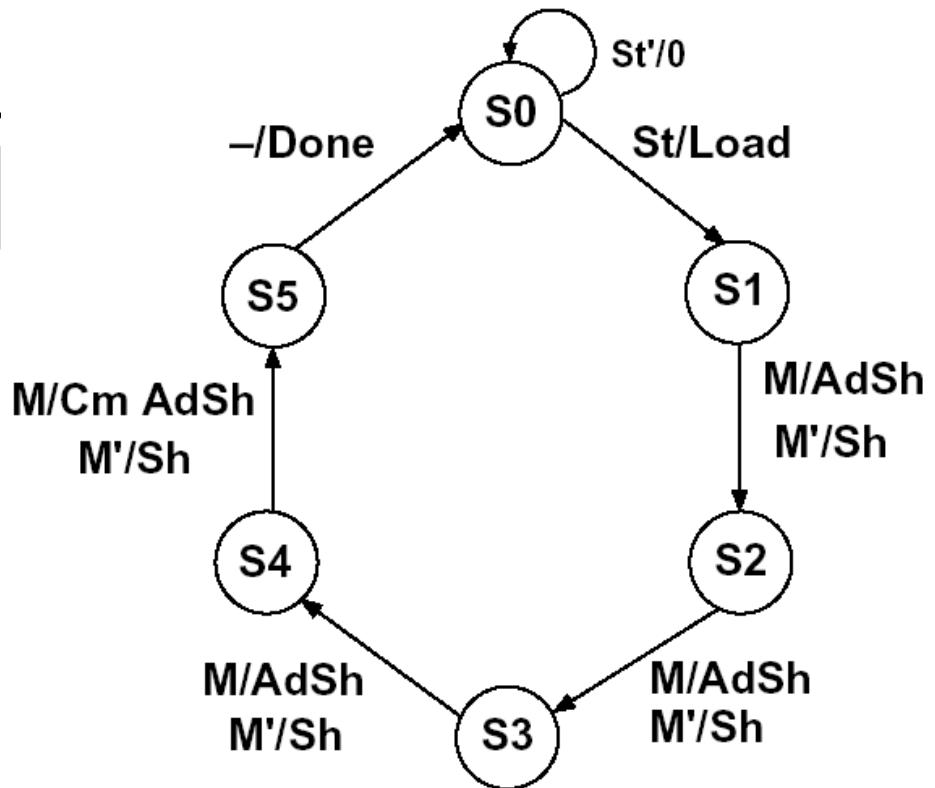
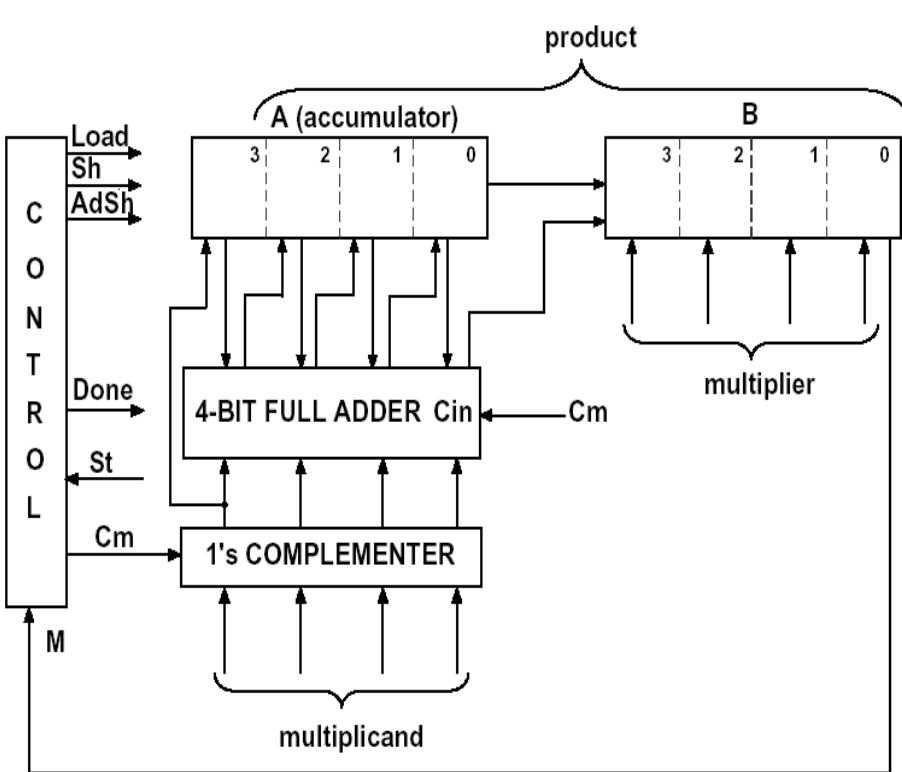
Extended State Graph Notation

Example: Faster Multiplier



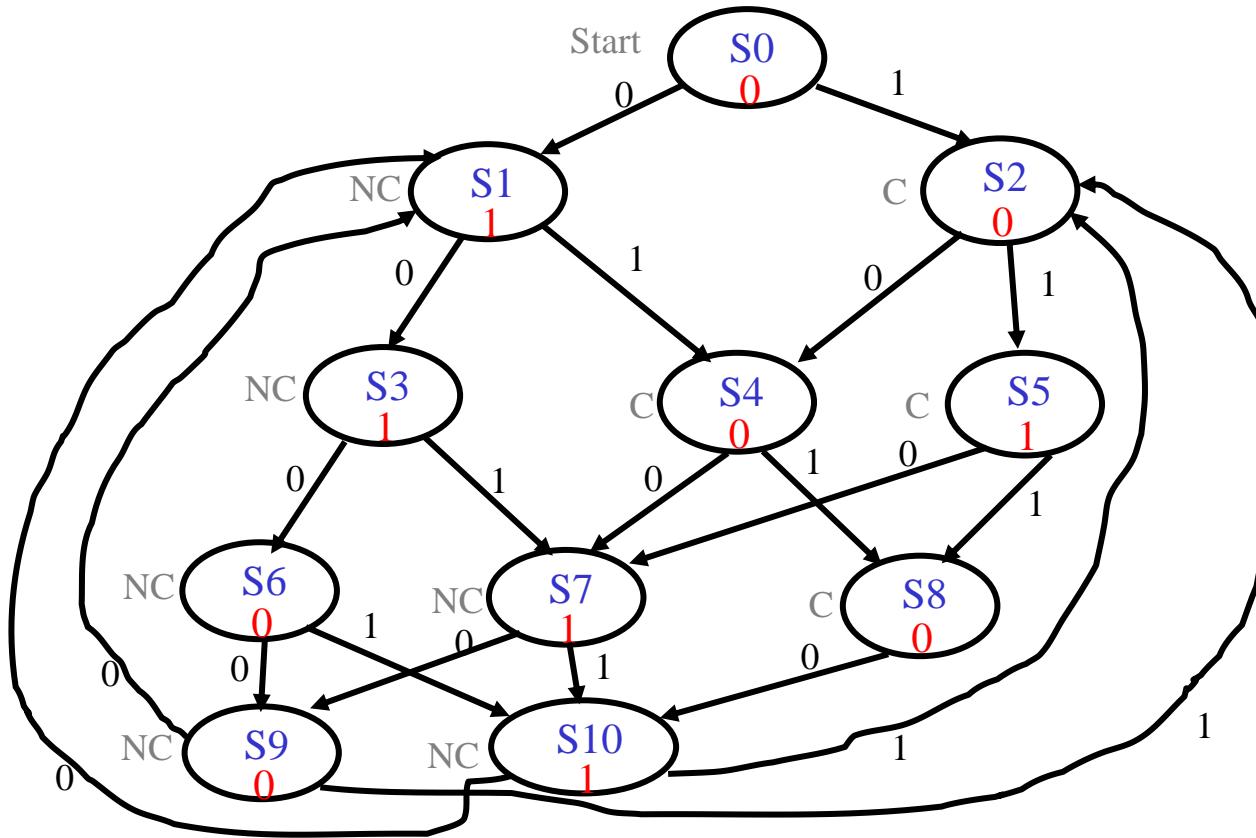
- Move wires from the adder outputs one position to the right => add and shift can occur at the same clock cycle

Extended State Graph for Fast Multiplier

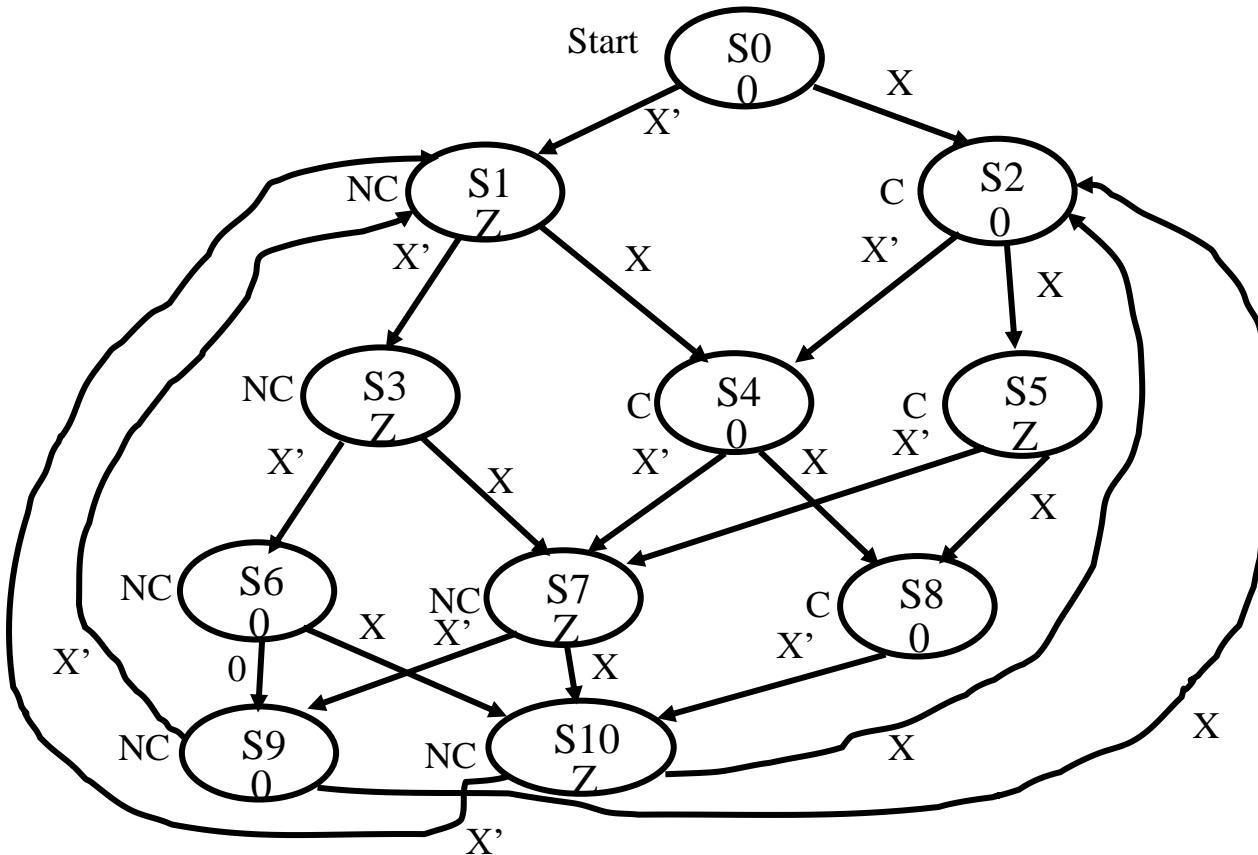


- Note: '-' on the input means no inputs are monitored and transition occurs to the next state no matter what the inputs are

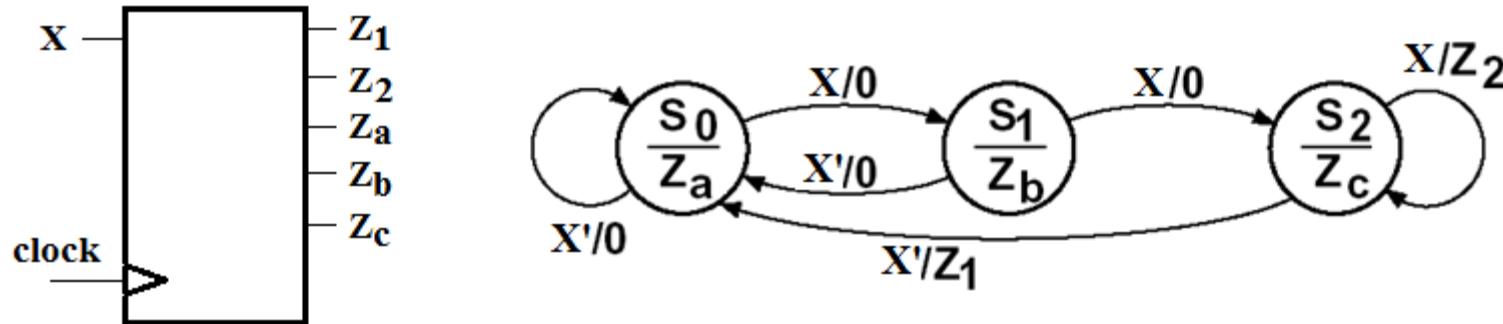
BCD to Excess 3 Moore FSM Example (Standard State Graph Representation)



BCD to Excess 3 Moore FSM Example (Extended State Graph Representation)



Combined Mealy/Moore FSM (Extended State Graph Representation)

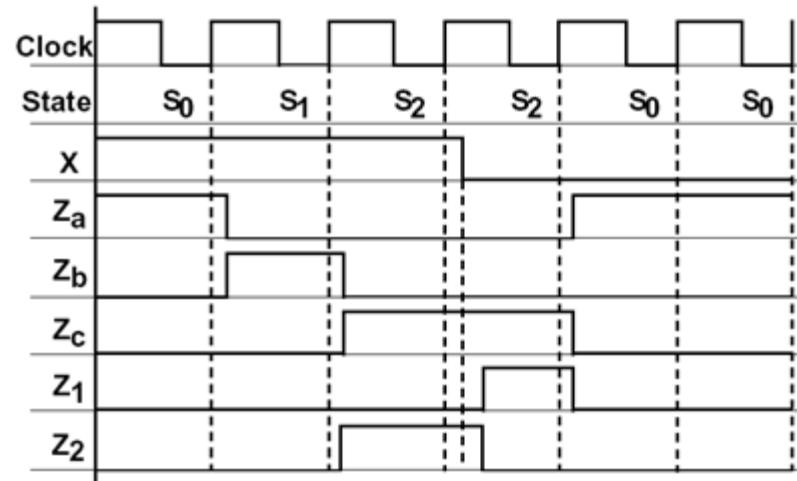


Input:

X

Outputs:

Z_a, Z_b, Z_c { Moore Outputs
 Z_1, Z_2 { Mealy Outputs

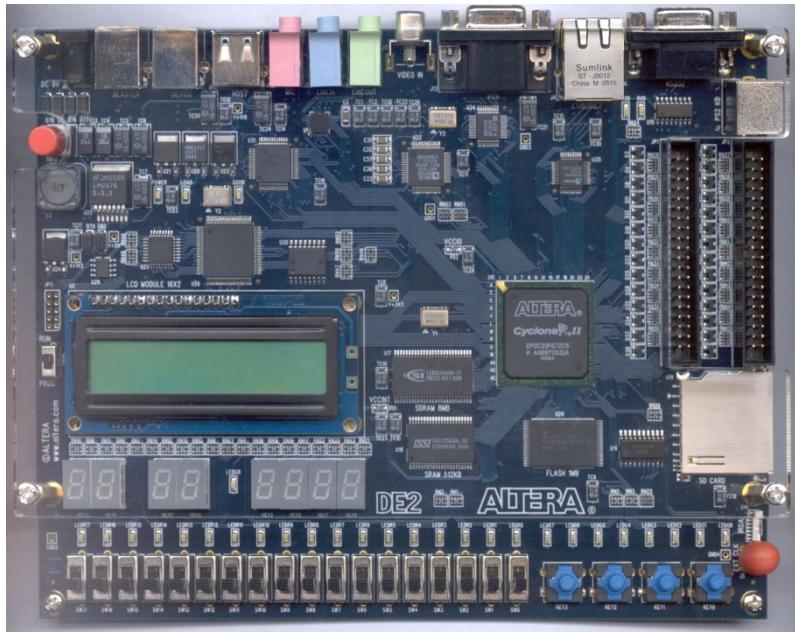


CPE 322

Digital Hardware Design Fundamentals

Electrical and Computer Engineering
UAH

Designing With FPGAs Architectural Issues and Features



Designing with FPGAs

Topics to be covered:

- Review of Generic FPGA Features
- Tradeoffs arising from the structure of the basic FPGA building blocks – some hand mapped examples
- Large Boolean Function Decomposition – Shannon's Expansion
- FPGA Support for commonplace operations – Carry and Cascade Chains
- Dedicated Memory and Multipliers in FPGAs

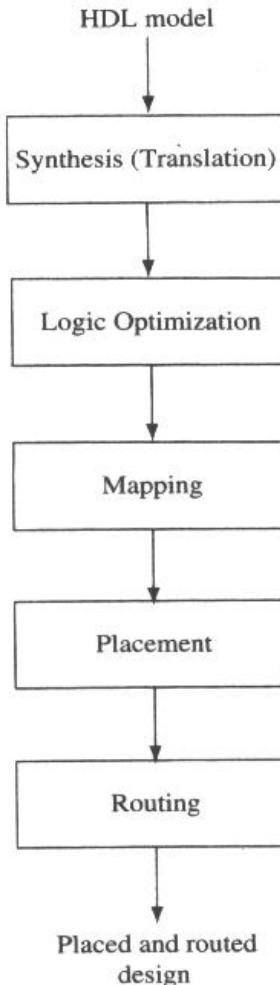
Chapter 6: Designing with FPGAs

Topics to be covered:

- Example Real-world FPGA Logic Blocks
- Cost of Reconfigurability
- FPGAs and One-Hot State Assignment
- Capacity Metrics; Maximum Gates versus Usable Gates
- FPGA Design Flow – modeling, synthesis, optimization, mapping, place and route

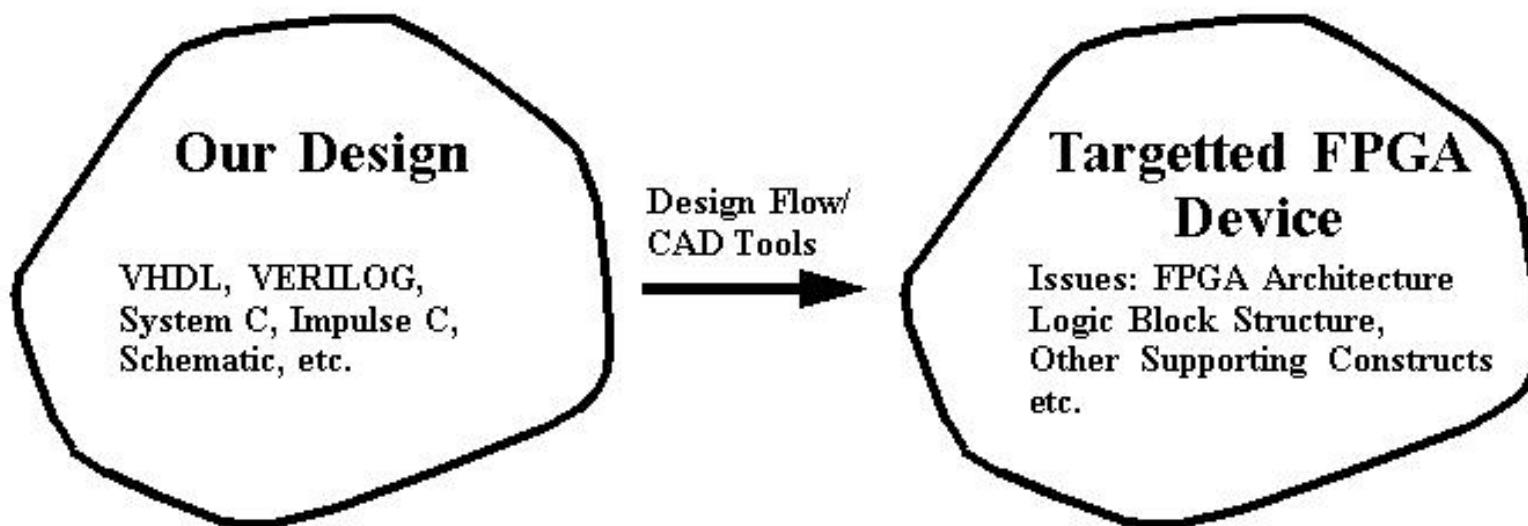
Implementing Functions in FPGAs

Partitioning a Design in an FPGA



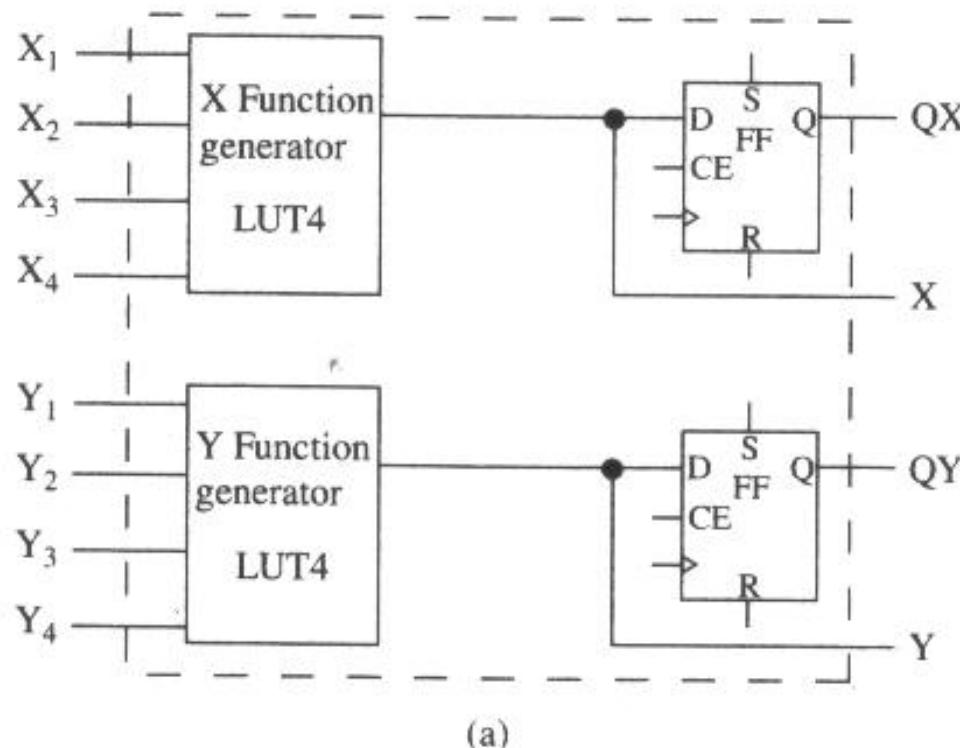
Implementing Functions in FPGAs

Partitioning a Design in an FPGA

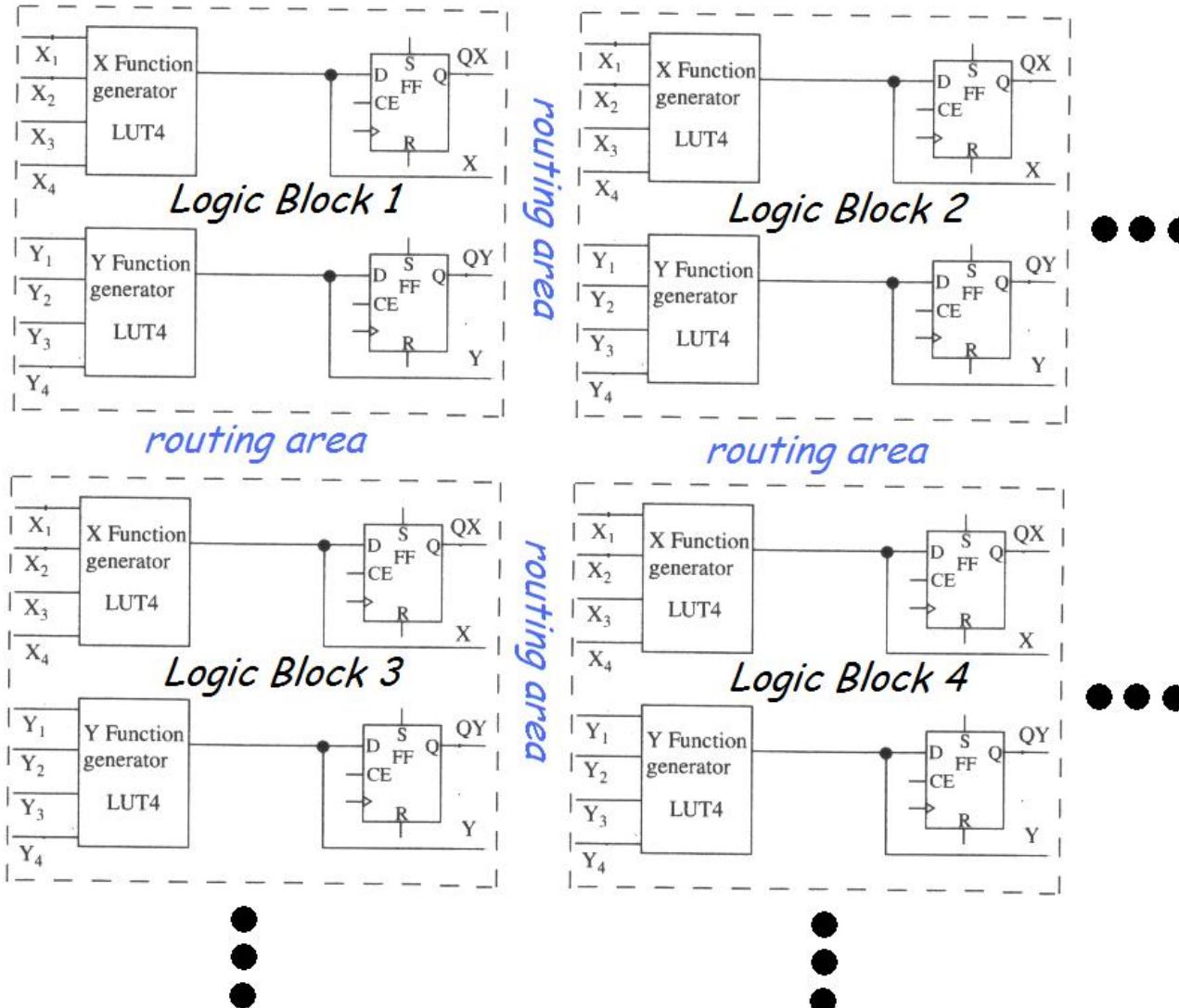


Implementing a Desired Function in an FPGA – Hand-mapped 4-to-1 MUX example

- Desired Function:
 - 4-to-1 Multiplexer
- Targeted FPGA Logic Block



Implementing a Desired Function in an FPGA – Hand-mapped 4-to-1 MUX example



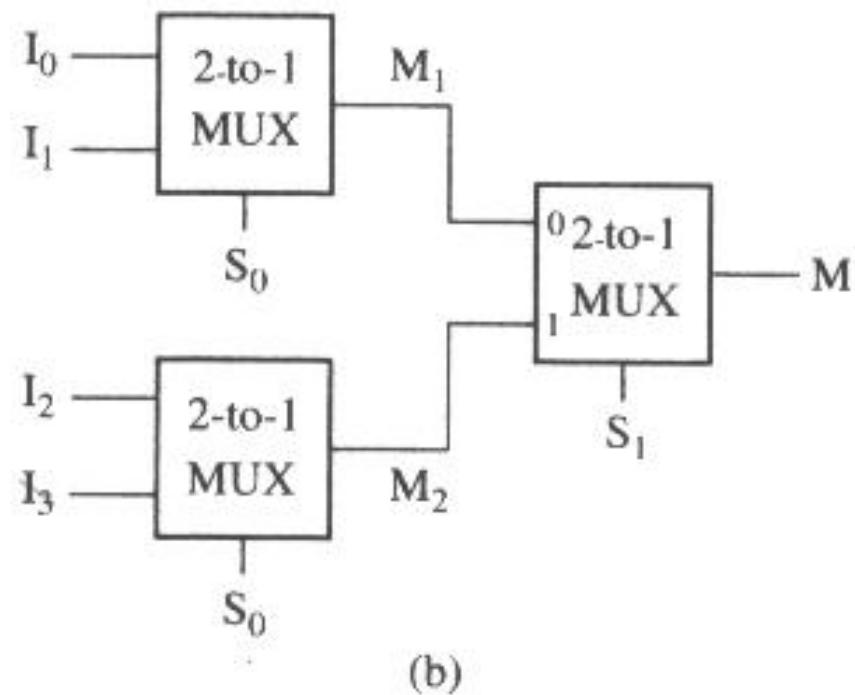
Implementing a Desired Function in an FPGA – Hand-mapped 4-to-1 MUX example

- 4-to-1 Multiplexer
 - $M = S'_1S'_0I_0 + S'_1S_0I_1 + S_1S'_0I_2 + S_1S_0I_3$ (**6 variables – will not fit in LUT4**)
- 4-to-1 MUX can be decomposed into 3 2-to-1 MUXes

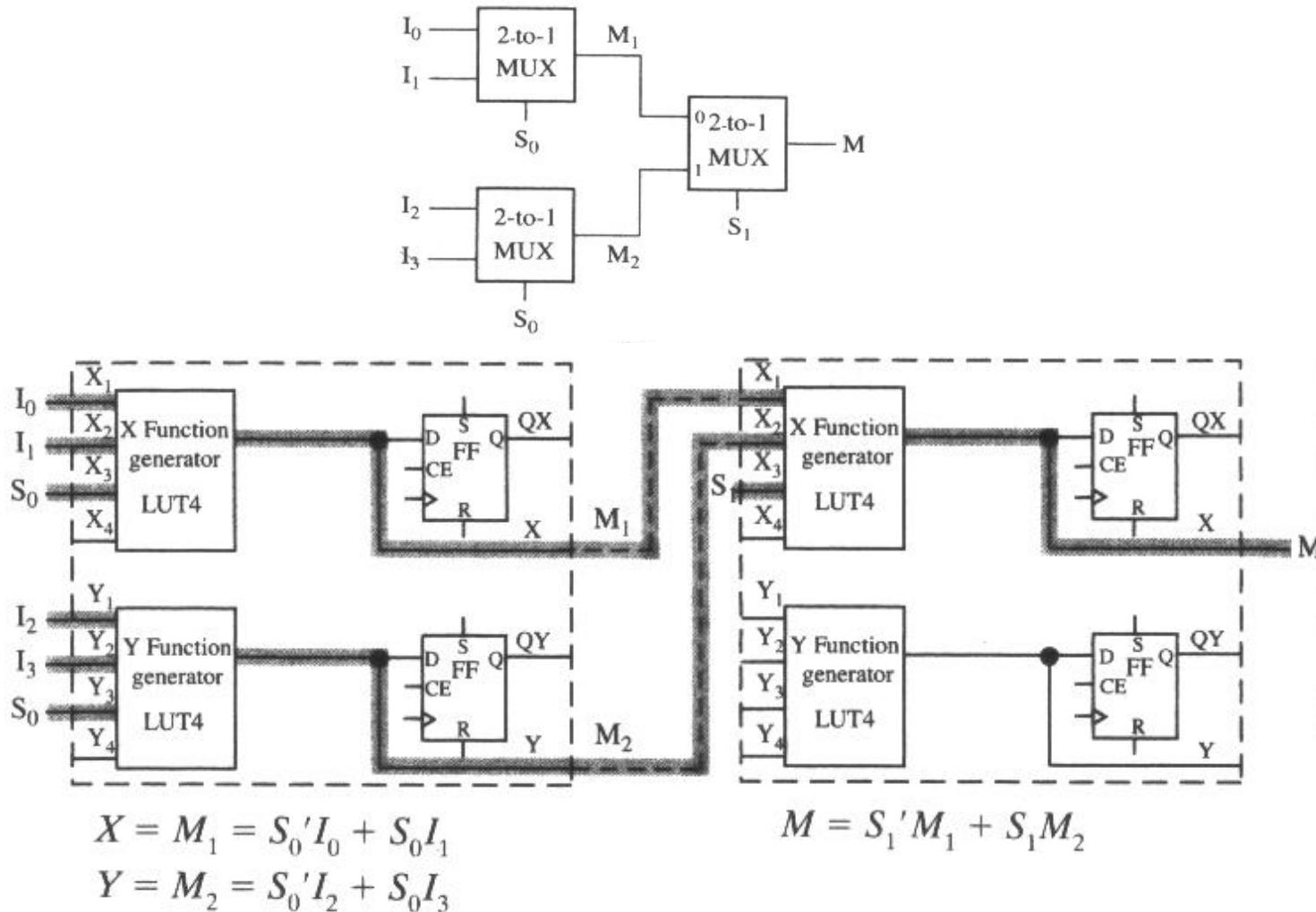
$$M_1 = S'_0I_0 + S_0I_1$$

$$M_2 = S'_0I_2 + S_0I_3$$

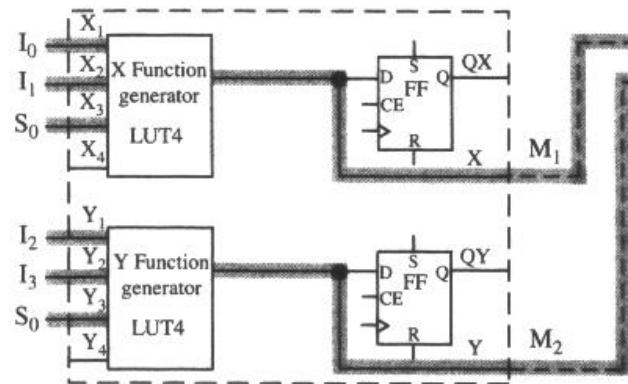
$$M = S'_1M_1 + S_1M_2$$



Implementing a Desired Function in an FPGA – Hand-mapped 4-to-1 MUX example

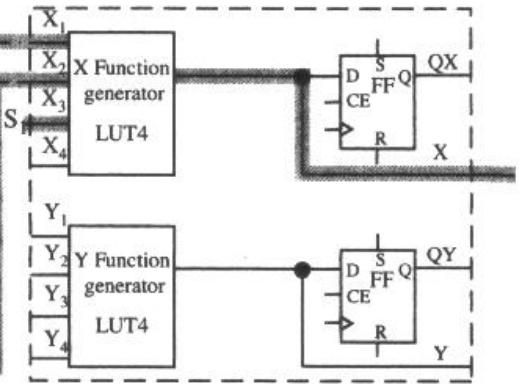


Implementing a Desired Function in an FPGA – Hand-mapped 4-to-1 MUX example

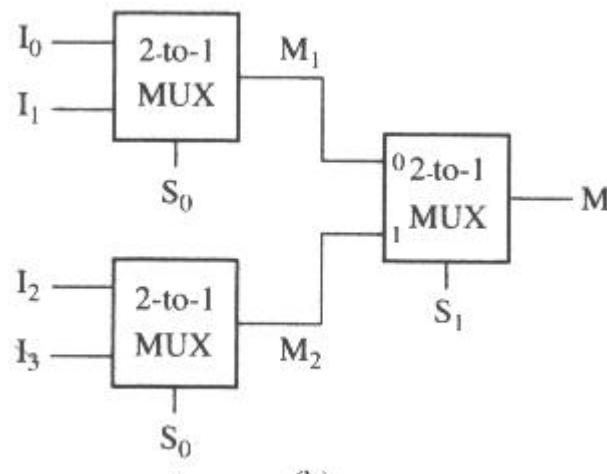


$$X = M_1 = S_0' I_0 + S_0 I_1$$

$$Y = M_2 = S_0' I_2 + S_0 I_3$$



$$M = S_1' M_1 + S_1 M_2$$



(b)

Inputs				Output
X_4	$X_3(S_0)$	$X_2(I_1)$	$X_1(I_0)$	X
x	0	0	0	0
x	0	0	1	1
x	0	1	0	0
x	0	1	1	1
x	1	0	0	0
x	1	0	1	0
x	1	1	0	1
x	1	1	1	1

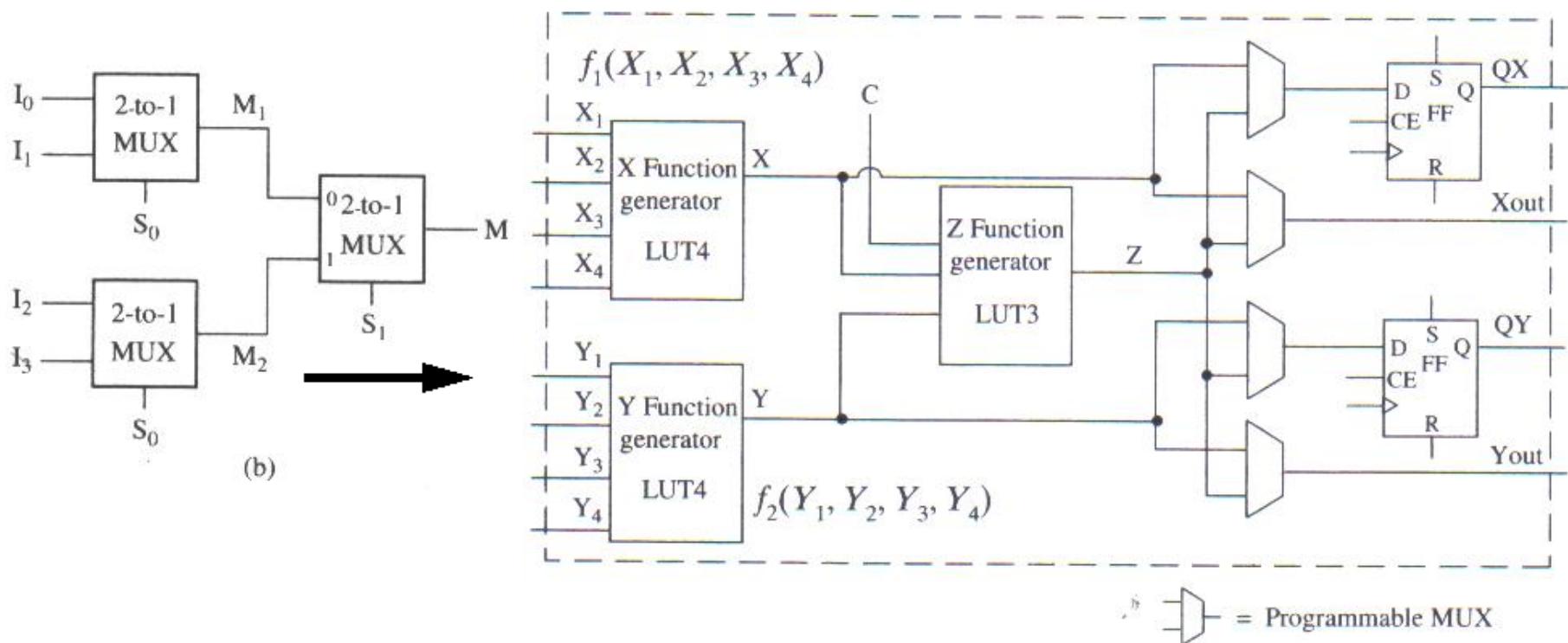
LUT-M1 – 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1

LUT-M2 – 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1

LUT-M – 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1

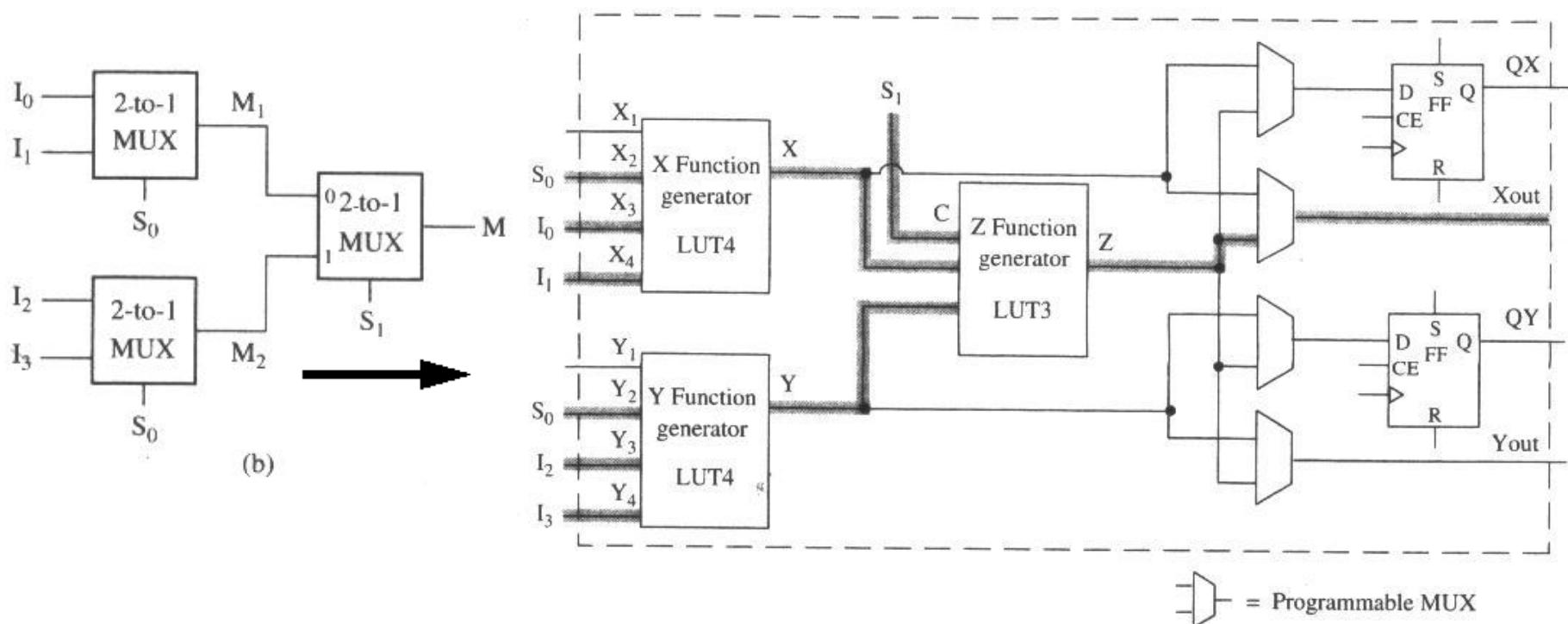
Implementing a Desired Function in an FPGA – Hand-mapped 4-to-1 MUX example

Another FPGA might have a different Logic Block Configuration – for example:



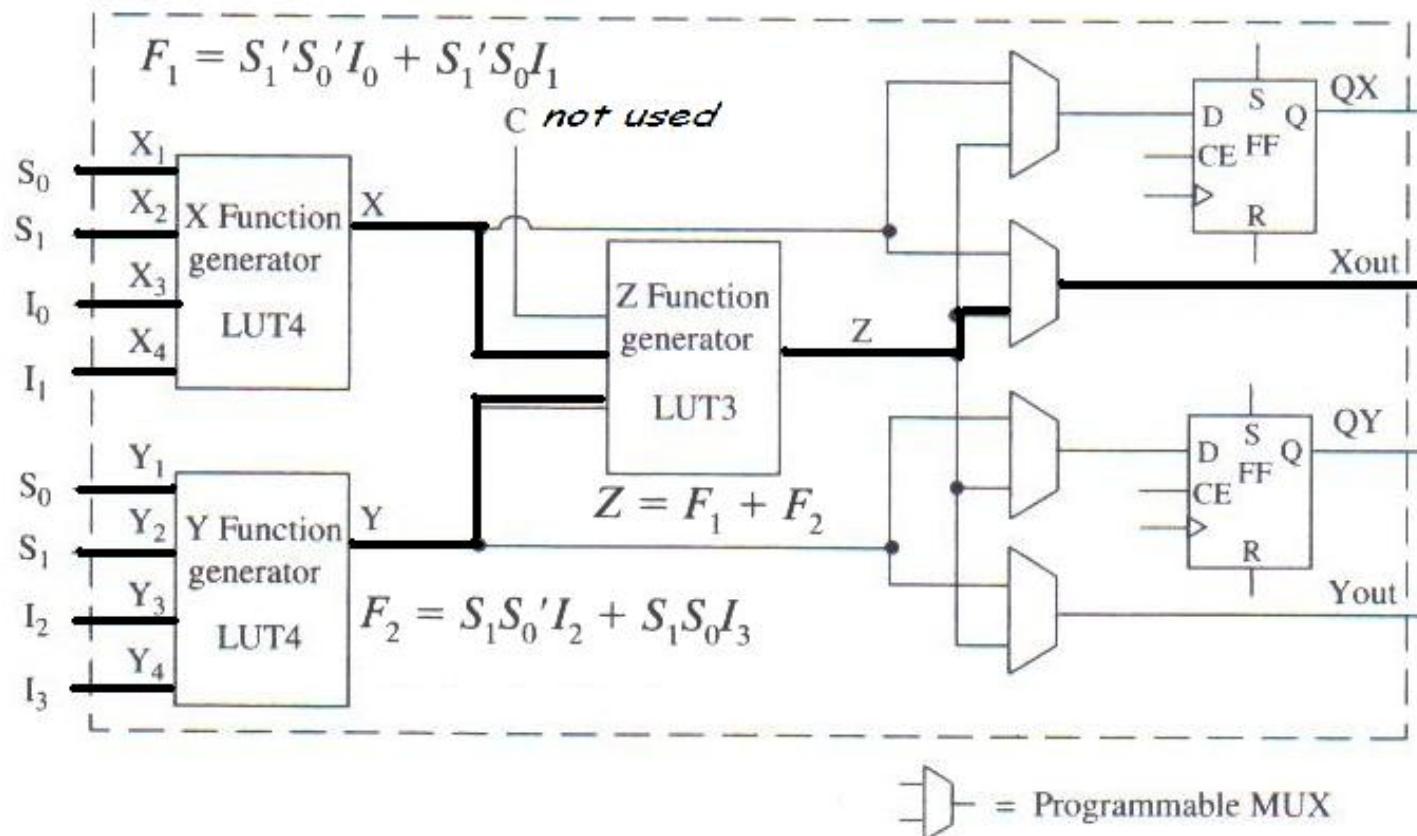
Implementing a Desired Function in an FPGA – Hand-mapped 4-to-1 MUX example

Another FPGA might have a different Logic Block Configuration – for example:

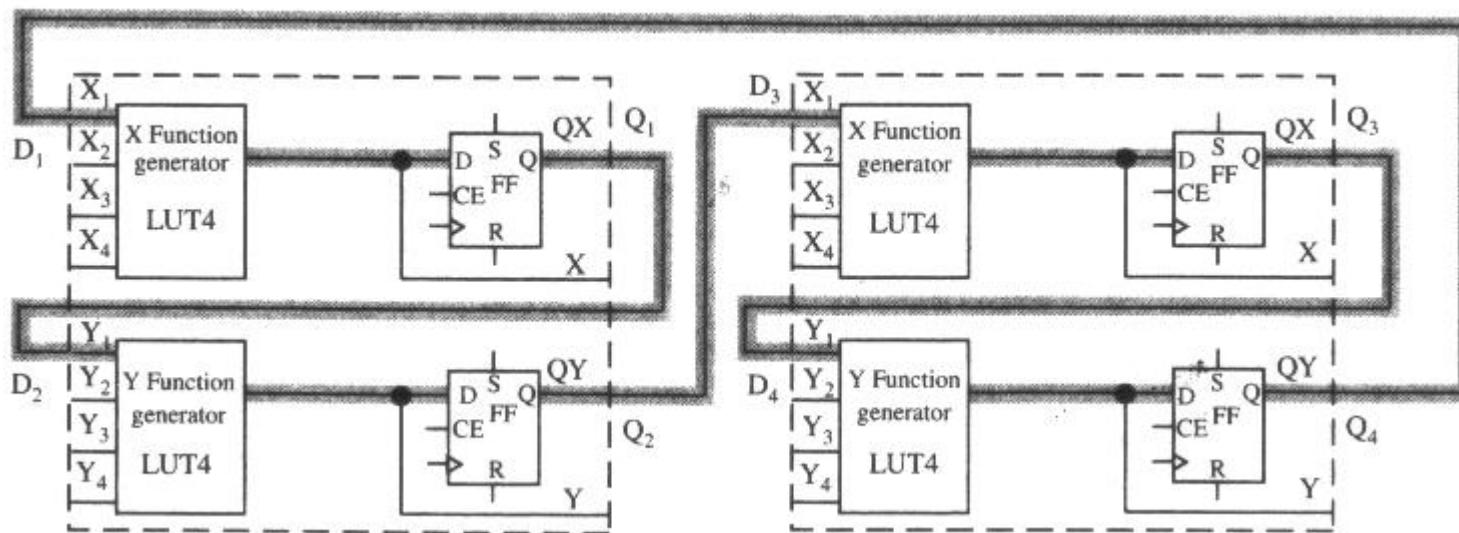
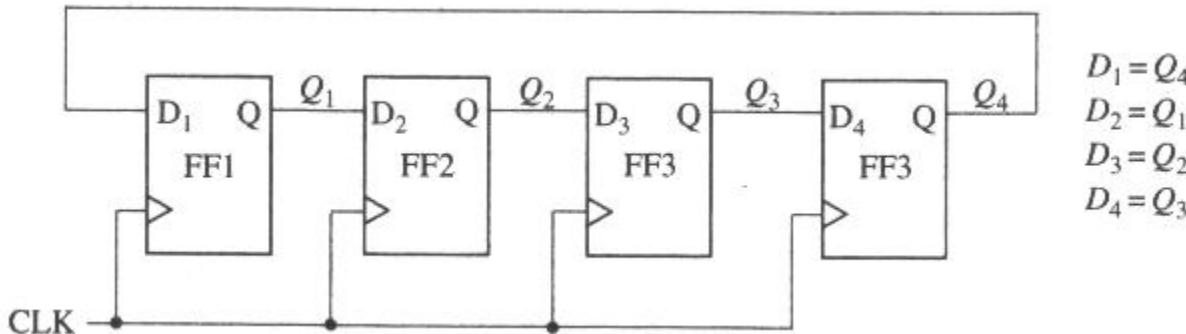


Implementing a Desired Function in an FPGA – Hand-mapped 4-to-1 MUX example

There can also be multiple ways to implement a Function within the same FPGA. Consider this equally valid representation of a 4-to-1 MUX!



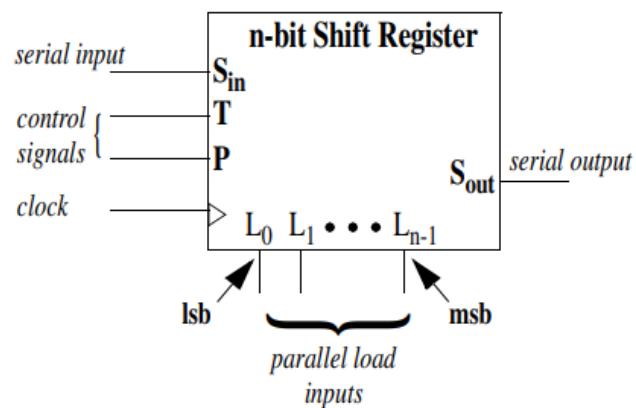
Implementing a Desired Function in an FPGA Hand-mapped ring counter example



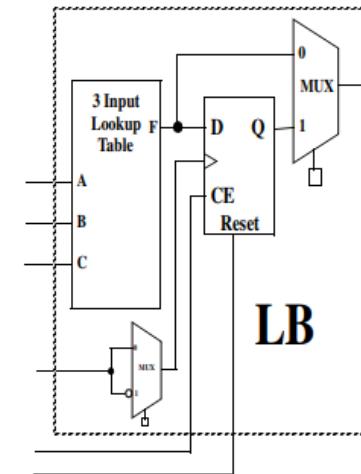
Note: function generator LUT4s largely unused!!
(wastes 64 SRAM cells)

Example Exam 2 Question:

An multiple bit right shift register with parallel load and a single serial output is to be implemented within an FPGA that has a standard island style architecture where the basic logic blocks have the configurations shown below:



**Shift Register Module to be
Designed**

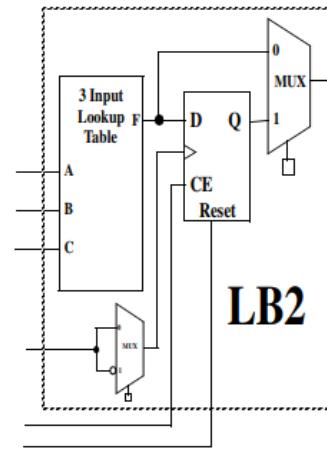
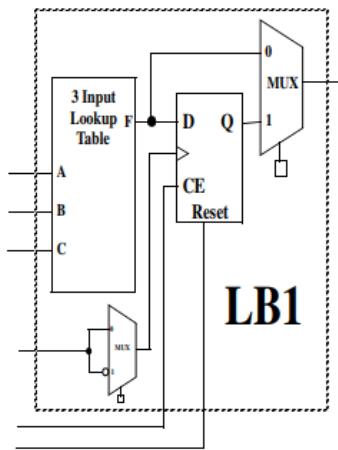


**Internal Logic Block Architecture
of Targeted FPGA**

The Control Signals P and T operate as follows: $P='0'$, do nothing keep current state; $P = '1'$ and $T='1'$ right shift on next rising edge clock; $P='1'$ and $T='0'$ load values from parallel load inputs into shift register on the next rising clock edge.

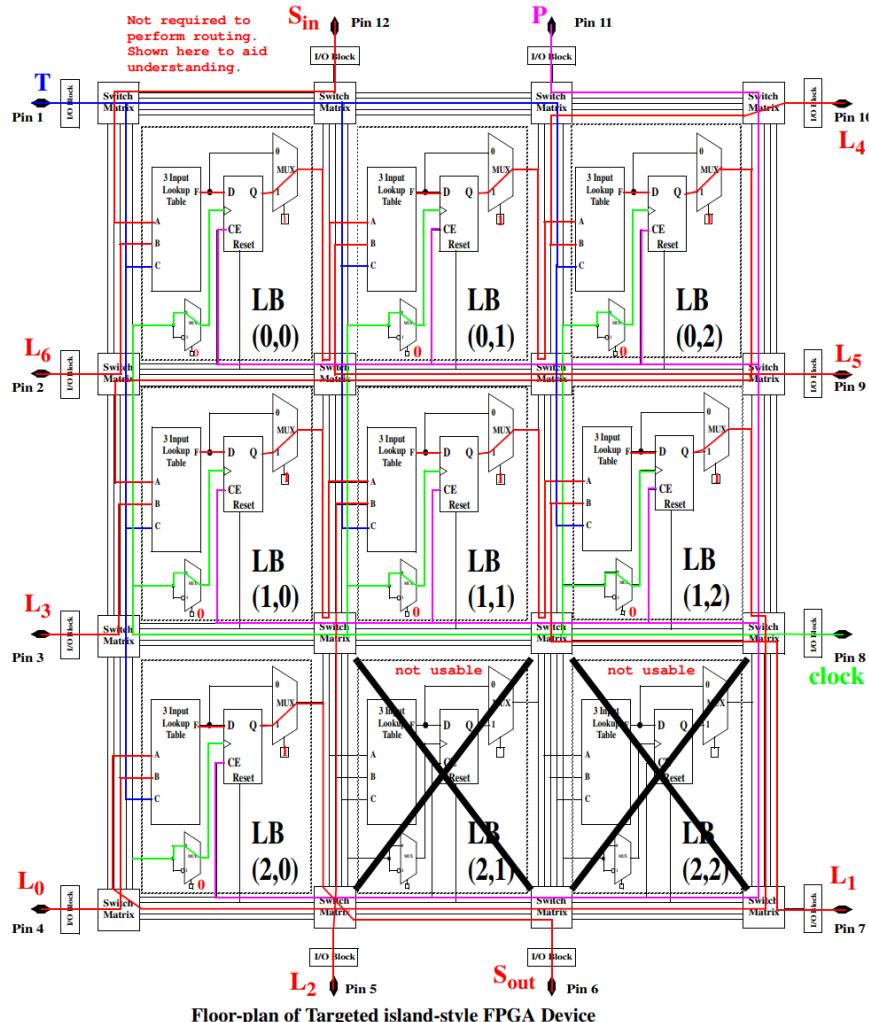
Example Exam 2 Question:

a) On the figure below create a two-bit shift register using these logic blocks as basic building elements of your design. Show the necessary internal connections between subcomponents as well as the logical connections between the logic blocks themselves. Also label signals that enter and leave the logic blocks that must originate or terminate on external FPGA I/O pin locations. Such signals include the serial input, control signals, parallel load inputs and the serial output. Use the names shown on the figure.

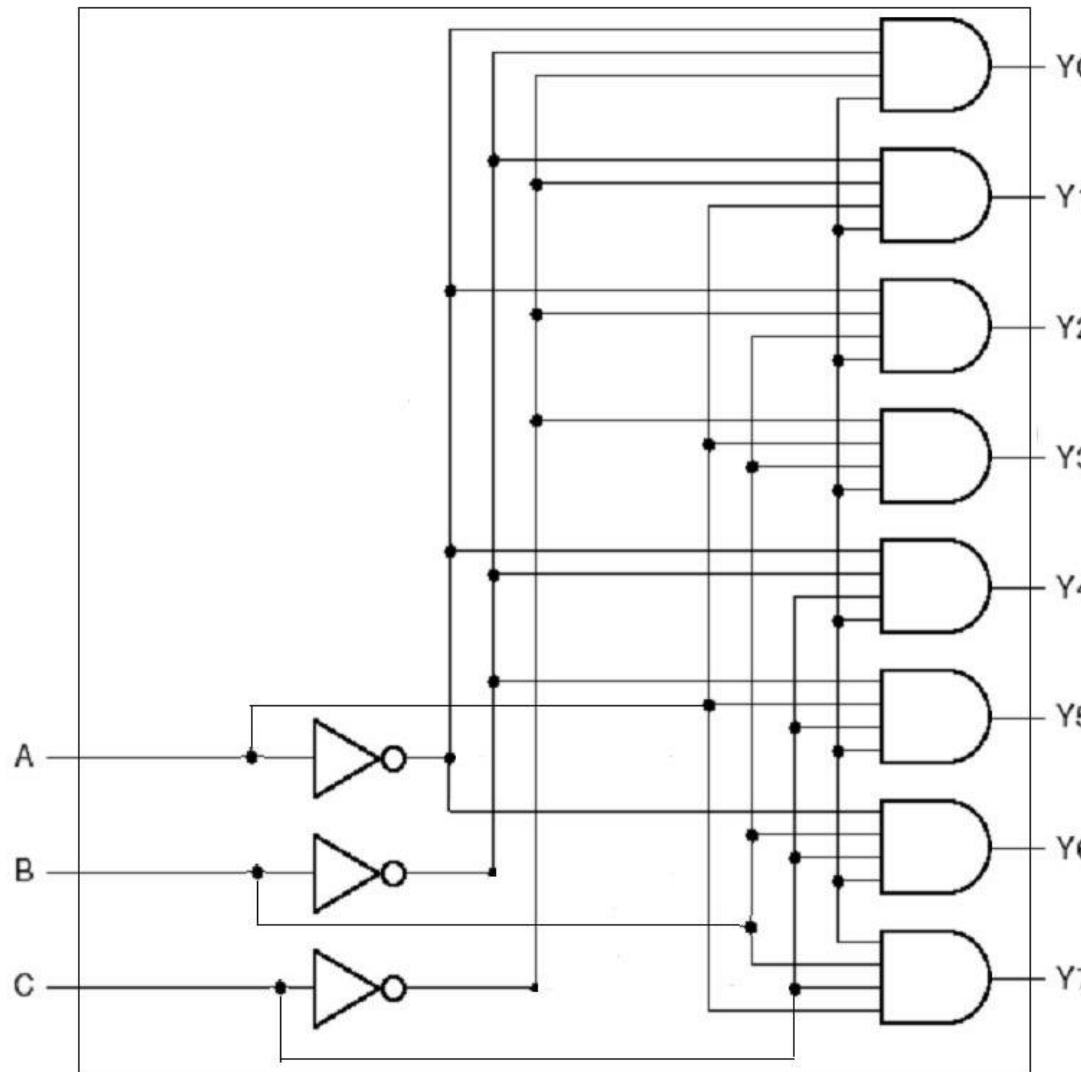


- b) What are the logic equations that should be implemented by the lookup tables in the two logic blocks in part (a) of this problem.
- c) What is the maximum value of n for the n-bit shift register that would fit in the very small FPGA, whose floor plan is shown in the figure on the next page could support? What resource makes this value of n the limiting factor?

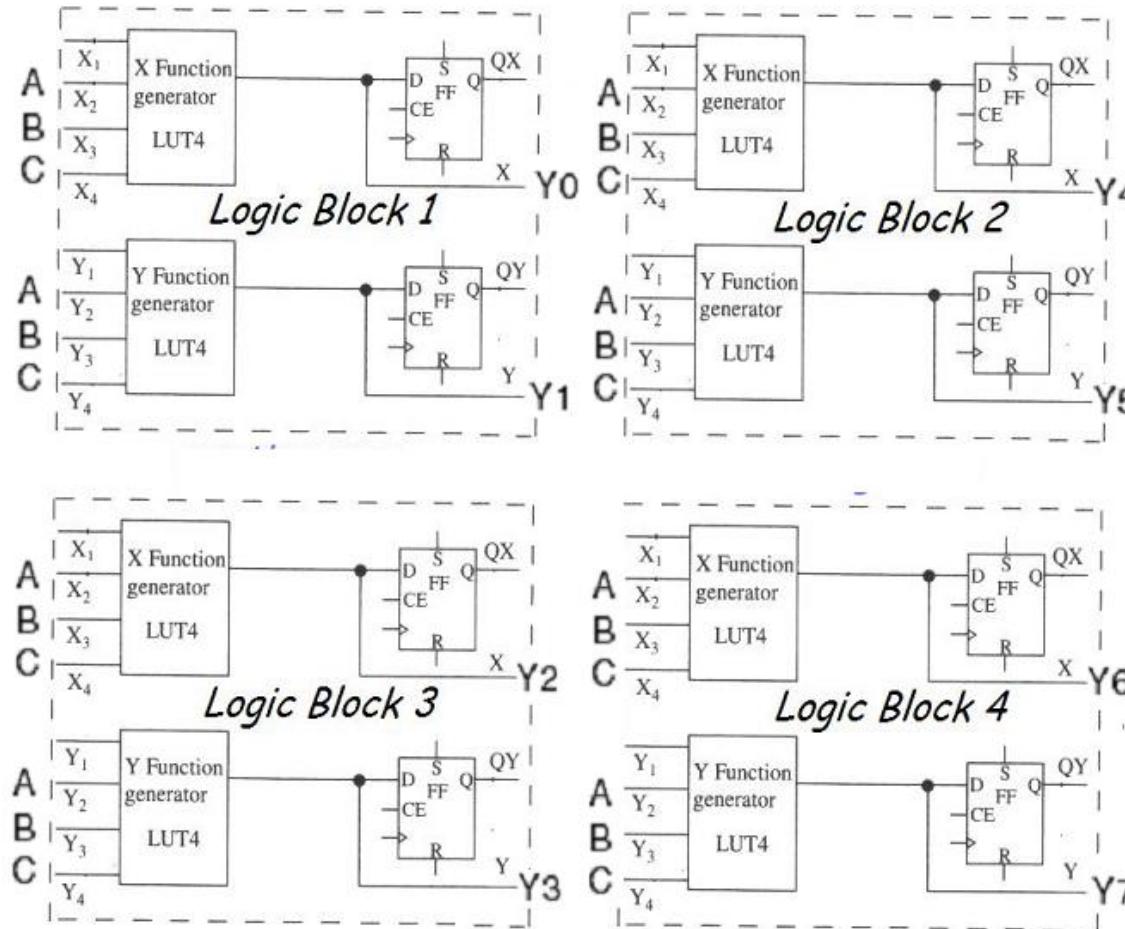
Example Exam 2 Question:



Implementing a Desired Function in an FPGA Hand-mapped 3-to-8 decoder



Implementing a Desired Function in an FPGA Hand-mapped 3-to-8 decoder



Note: Very Expensive Implementation
(uses 128 SRAM cells – 4 Logic Blocks – 8 FF wasted)

Shannon's Decomposition

$$Z(a, b, c, d, e, f) = a' \cdot Z(0, b, c, d, e, f) + a \cdot Z(1, b, c, d, e, f) = a'Z_0 + aZ_1$$

This expansion can be continued for any number of variables

Example: $Z = abcd'ef' + a'b'c'def' + b'cde'f$

Setting $a = 0$ gives

$$Z_0 = 0 \cdot bcd'ef' + 1 \cdot b'c'def' + b'cde'f = b'c'def' + b'cde'f$$

and setting $a = 1$ gives

$$Z_1 = 1 \cdot bcd'ef' + 0 \cdot b'c'def' + b'cde'f = bcd'ef' + b'cde'f.$$

$$Z = a'Z_0 + aZ_1$$

Shannon's Decomposition

(6 variable function to two 5 variable functions)

Example: $Z = abcd'ef' + a'b'c'def' + b'cde'f$

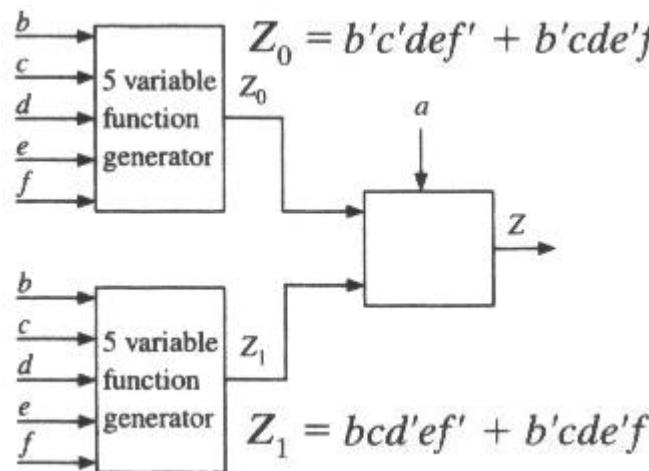
Setting $a = 0$ gives

$$Z_0 = 0 \cdot bcd'ef' + 1 \cdot b'c'def' + b'cde'f = b'c'def' + b'cde'f$$

and setting $a = 1$ gives

$$Z_1 = 1 \cdot bcd'ef' + 0 \cdot b'c'def' + b'cde'f = bcd'ef' + b'cde'f.$$

$$Z = a'Z_0 + aZ_1$$

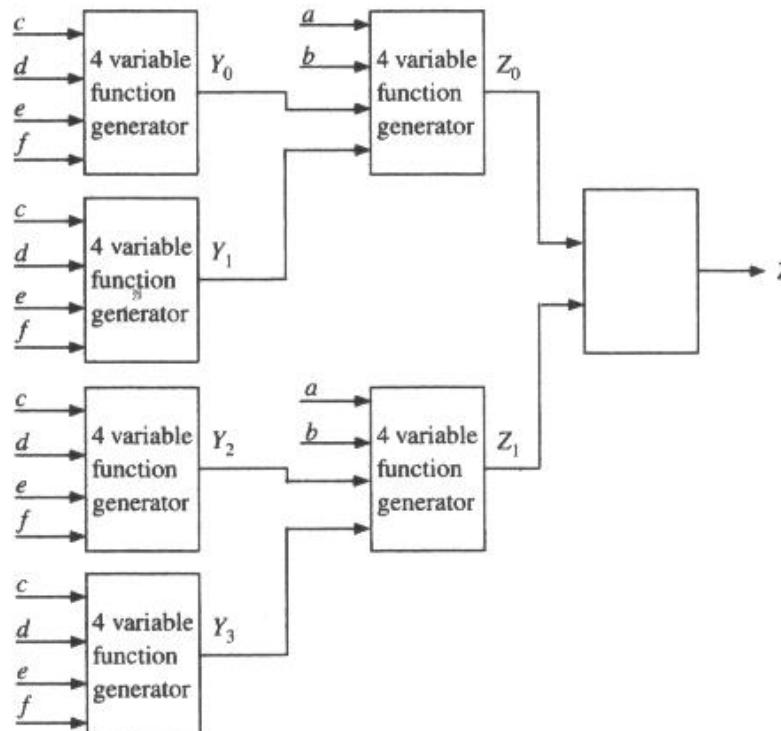


Shannon's Decomposition

(6 variable function to six 4 variable functions)

Example: $Z = abcd'ef' + a'b'c'def' + b'cde'f$

$$\begin{aligned} Z(a, b, c, d, e, f) &= a'b' \cdot Z(0, 0, c, d, e, f) + a'b \cdot Z(0, 1, c, d, e, f) \\ &\quad + ab' \cdot Z(1, 0, c, d, e, f) + ab \cdot Z(1, 1, c, d, e, f) \\ &= a'b' \cdot Y_0 + a'b \cdot Y_1 + ab' \cdot Y_2 + ab \cdot Y_3 \end{aligned}$$

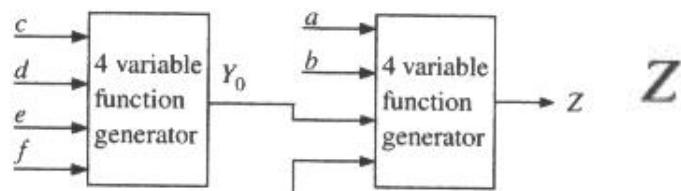


Shannon's Decomposition

(6 variable function to five 4 variable functions)

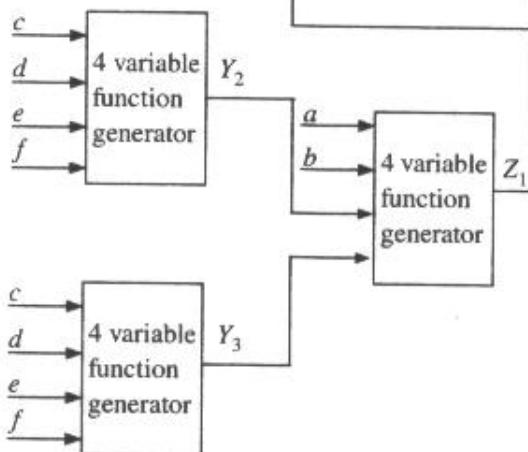
Example 2: $Z = abcd'ef' + a'b'c'def' + b'cde'f$

- Substituting $a = b = 0$ gives $Y_0 = c'def' + cde'f$
- Substituting $a = 0, b = 1$ gives $Y_1 = 0$
- Substituting $a = 1, b = 0$ gives $Y_2 = cde'f,$
- Substituting $a = b = 1$ gives $Y_3 = cd'e'f'$



$$Z = a'b' \cdot Y_0 + ab' \cdot Y_2 + ab \cdot Y_3$$

Possible because $Y_1 = 0$



Shannon's Decomposition

(7 variable function to six 5 variable functions)

$$\begin{aligned}Z(a, b, c, d, e, f, g) &= a'b' \cdot Z(0, 0, c, d, e, f, g) + a'b \cdot Z(0, 1, c, d, e, f, g) \\&\quad + ab' \cdot Z(1, 0, c, d, e, f, g) + ab \cdot Z(1, 1, c, d, e, f, g) \\&= a'b' \cdot Y_0 + a'b \cdot Y_1 + ab' \cdot Y_2 + ab \cdot Y_3\end{aligned}$$

Example 3:

$$Z = c'de'fg + bcd'e'fg' + a'c'def'g + a'b'd'ef'g' + ab'defg'$$

- Substituting $a = b = 0$ gives $Y_0 = c'de'fg + c'def'g + d'ef'g'$
- Substituting $a = 0, b = 1$ gives $Y_1 = c'de'fg + cd'e'fg' + c'def'g$
- Substituting $a = 1, b = 0$ gives $Y_2 = c'de'fg + defg'$
- Substituting $a = b = 1$ gives $Y_3 = c'de'fg + cd'e'fg'$

$$Z_0 = a'b' \cdot Y_0 + a'b \cdot Y_1 \quad Z = Z_0 + ab' \cdot Y_2 + ab \cdot Y_3$$

Shannon's Decomposition

(7 variable function to six 5 variable functions)

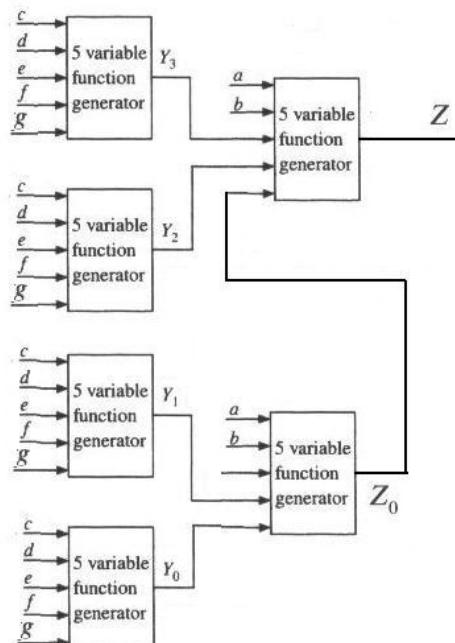
Example 3:

$$Z = c'de'fg + bcd'e'fg' + a'c'def'g + a'b'd'ef'g' + ab'defg'$$

- Substituting $a = b = 0$ gives $Y_0 = c'de'fg + c'def'g + d'ef'g'$
- Substituting $a = 0, b = 1$ gives $Y_1 = c'de'fg + cd'e'fg' + c'def'g$
- Substituting $a = 1, b = 0$ gives $Y_2 = c'de'fg + defg'$
- Substituting $a = b = 1$ gives $Y_3 = c'de'fg + cd'e'fg'$

$$Z_0 = a'b' \cdot Y_0 + a'b \cdot Y_1$$

$$Z = Z_0 + ab' \cdot Y_2 + ab \cdot Y_3$$



Any 7 Variable function can be implemented with six or fewer LUT5s!

Shannon's Decomposition

(Implementing 7 variable function using LUT4s w/ MUXES)

Note: to Reduce the number of LUTs needed many FPGAs provide Multiplexers in addition to LUT4s.

Implement a seven-variable function using four-input LUTs and 2-to-1 multiplexers.

Shannon's expansion can be used to obtain the following decompositions:

$$7\text{-variable function generator} = \text{two } 6\text{-variable function generators} + \text{a 2-to-1 mux} \dots \quad (\text{i})$$

$$6\text{-variable function generator} = \text{two } 5\text{-variable function generators} + \text{a 2-to-1 mux} \dots \quad (\text{ii})$$

$$5\text{ variable function generator} = \text{two } 4\text{-variable function generators} + \text{a 2-to-1 mux} \dots \quad (\text{iii})$$

Substituting (iii) into (ii), we obtain

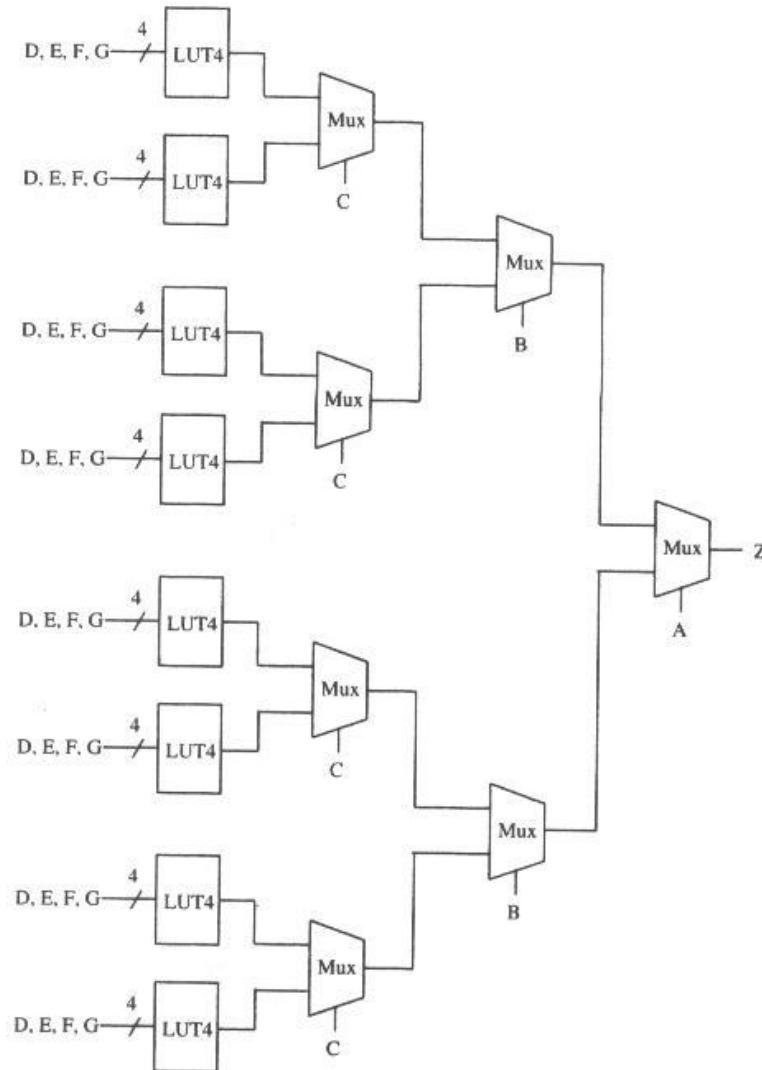
$$\begin{aligned} 6\text{-variable function generator} &= \text{four } 4\text{-variable function generators} \\ &\quad + \text{three 2-to-1 muxes} \dots \end{aligned} \quad (\text{iv})$$

Substituting (iv) into (i), we obtain

$$7\text{-variable function generator} = \text{eight } 4\text{-variable function generators} + \text{seven 2-to-1 muxes}$$

Shannon's Decomposition

(Implementing 7 variable function using LUT4s w/ MUXES)

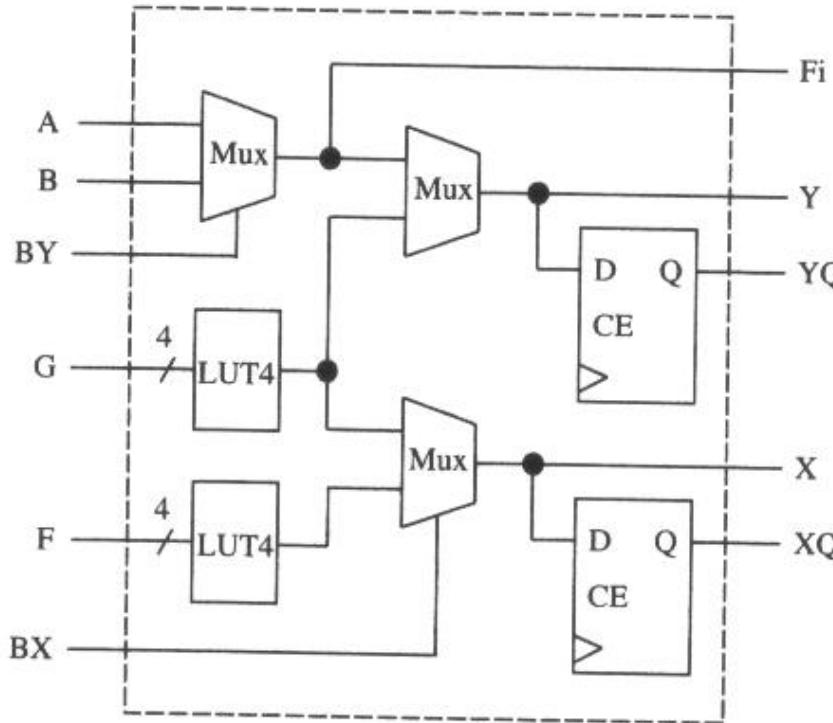


Shannon's Decomposition

(Implementing 7 variable function using 4 Xilinx Spartan Slices)

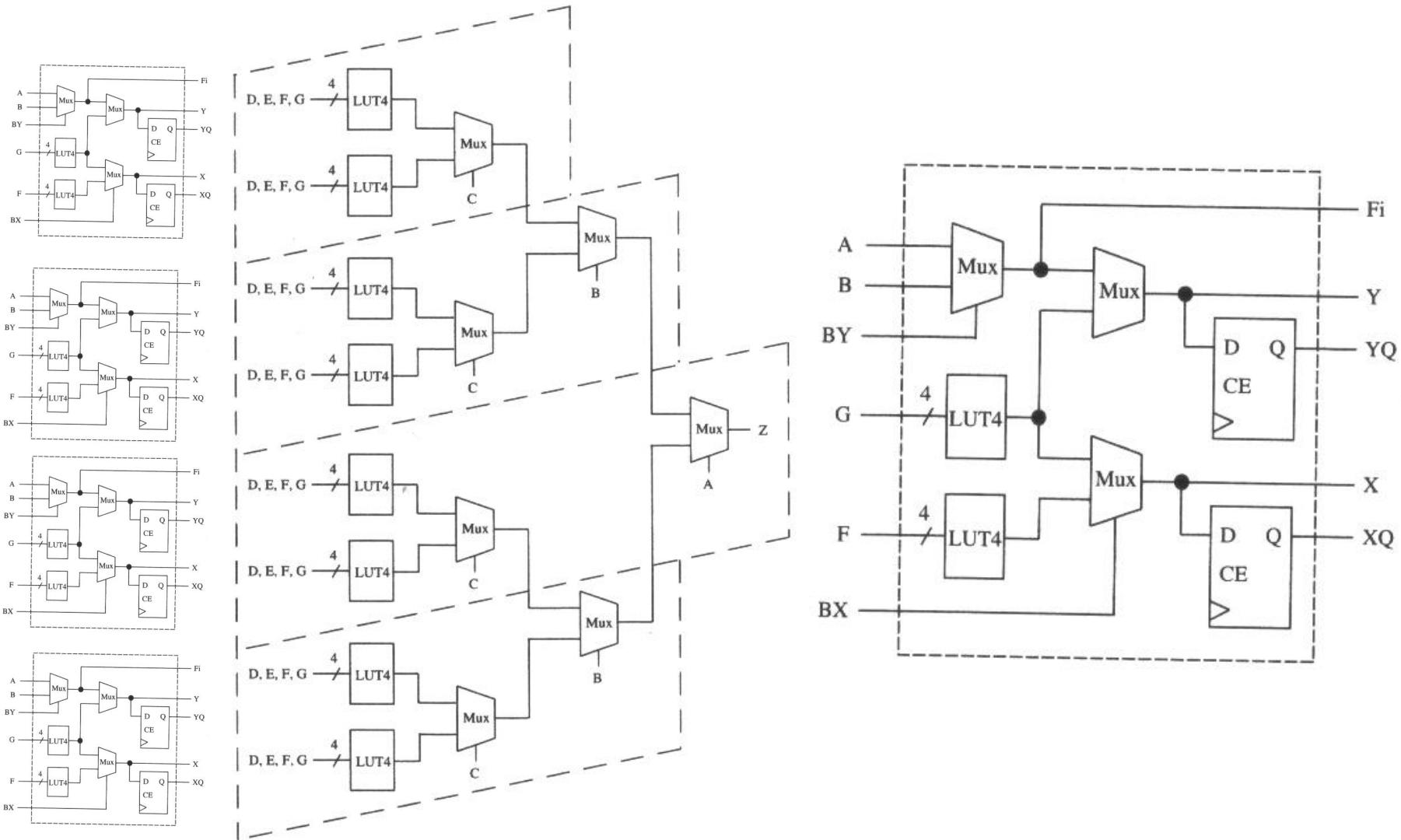
Example: Parity Generation Logic

$$Z = A \oplus B \oplus C \oplus D \oplus E \oplus F \oplus G$$

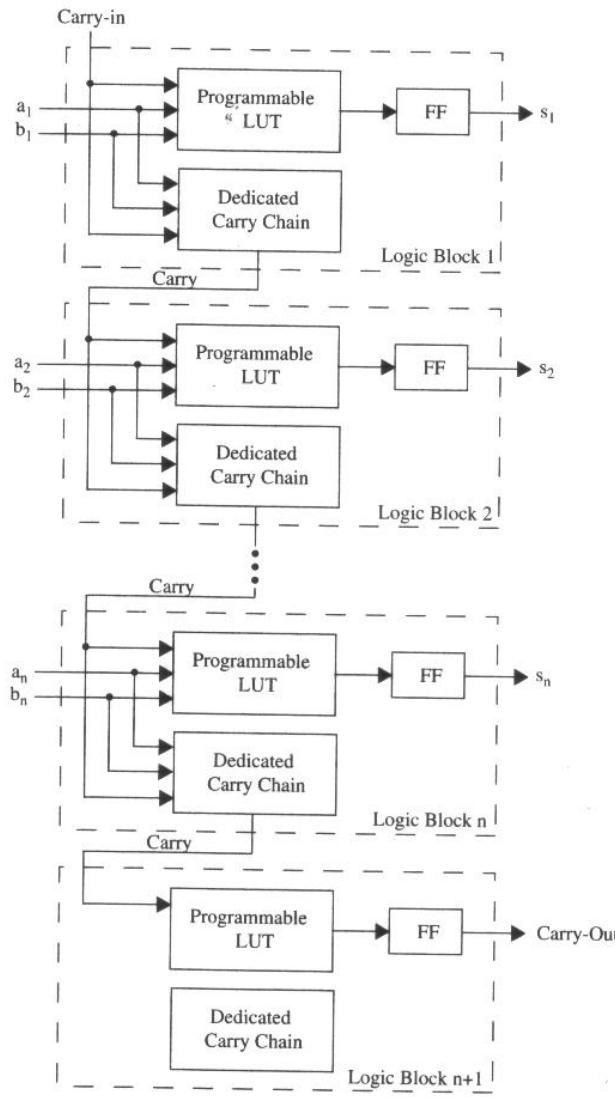


Shannon's Decomposition

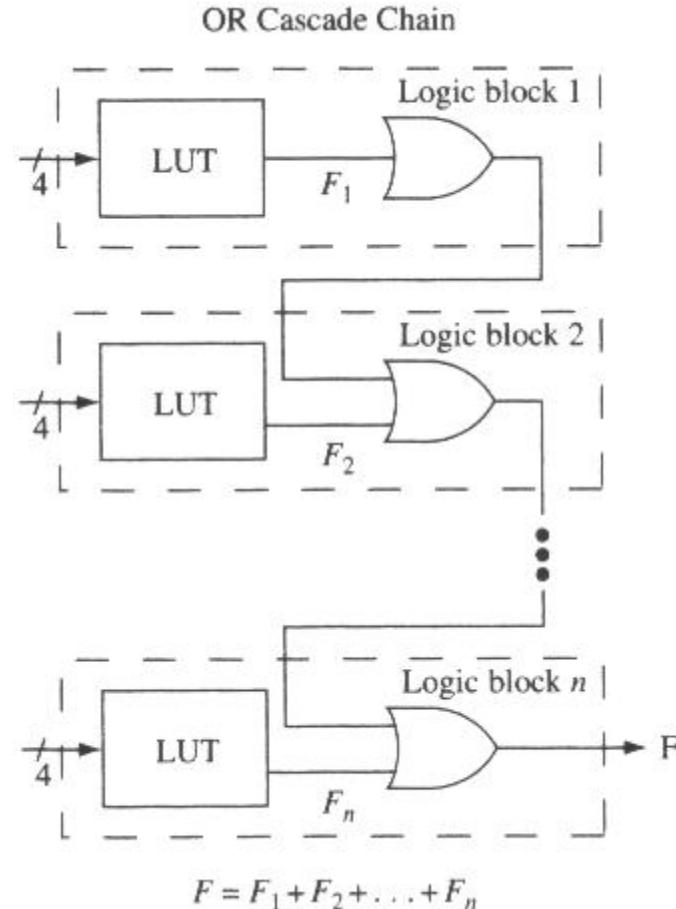
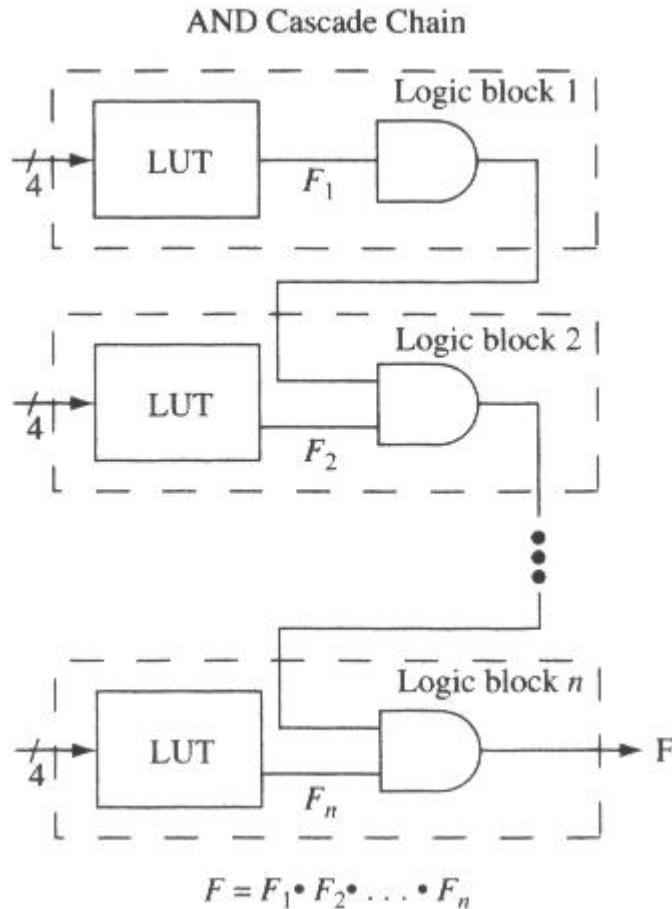
(Implementing 7 variable function using 4 Xilinx Spartan Slices)



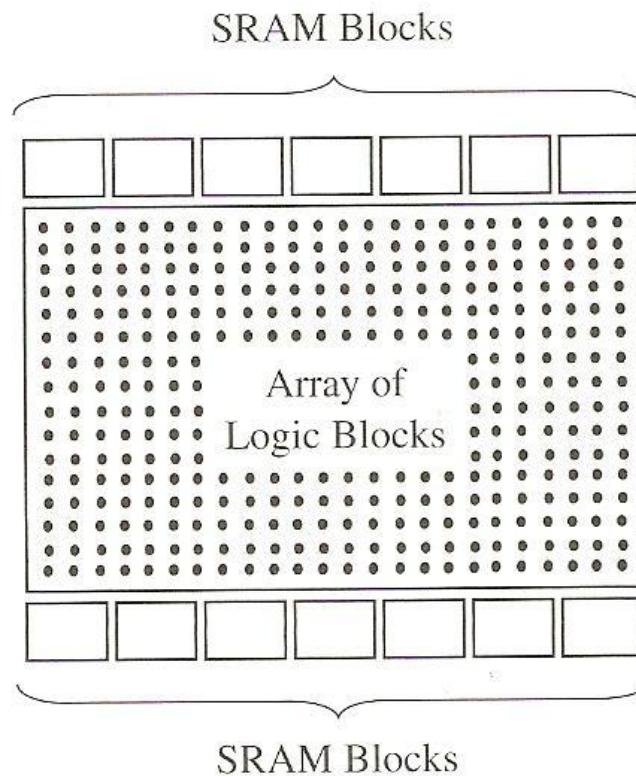
Commonplace Operation Support (Carry Chains in FPGAs)



Commonplace Operation Support (Cascade Chains in FPGAs)



Commonplace Operation Support (Dedicated Memory in FPGAs)



Commonplace Operation Support (Dedicated Memory in FPGAs)

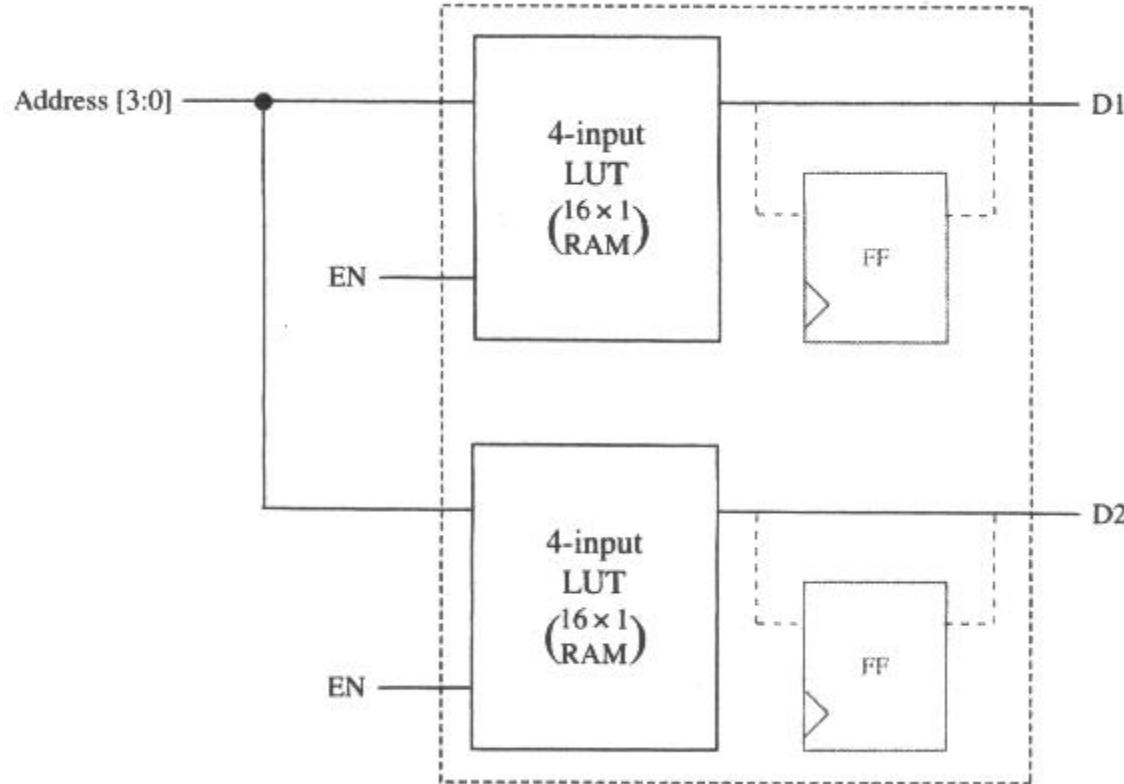
FPGA Family	Dedicated RAM Size (Kb)	Organization
Xilinx Virtex 5	1152–10368	64–576 18Kb blocks
Xilinx Virtex 4	864–9936	48–552 18Kb blocks
Xilinx Virtex-II	72–3024	4–168 18Kb blocks
Xilinx Spartan 3E	72–648	4–36 18Kb blocks
Altera Stratix II	409–9163	104–930 512b blocks 78–768 4Kb blocks 0–9 512Kb blocks
Altera Cyclone II	117–1125	26–250 4Kb blocks
Lattice SC	1054–7987	56–424 18Kb blocks
Actel Fusion	27–270	6–60 4Kb blocks

Dedicated Memory – Variable Data Widths

Width	Depth	Addr Bus	Data Bus
1	32K	15 bits	1 bit
2	16K	14 bits	2 bits
4	8K	13 bits	4 bits
8	4K	12 bits	8 bits
16	2K	11 bits	16 bits

Creating Memory from LUTs

(16 x 2 memory)



Creating Memory from LUTs

Verilog HDL Model that typically infers LUT-based memory

```
module Memory (input CLK, MemWrite, input [11:0] Address,
               input [31:0] Data_In, output [31:0] DATA_Out);

    reg [31:0] DataMEM [0:511];

    initial
        begin
            $readmemh("initial_ram.txt",DataMEM);
        end

    always @ (posedge CLK)
        begin
            if (MemWrite) DataMEM[Address] = Data_In; // Synchronous Write
        end

    assign DATA_Out = DataMEM[Address];           // Asynchronous Read

endmodule
```

Creating Memory from LUTs

Verilog HDL Model that typically infers LUT-based memory

```
module Memory (input CLK, MemWrite, input [11:0] Address,
               input [31:0] Data_In, output [31:0] DATA_Out);

    reg [31:0] DataMEM [0:511];

    initial
        begin
            $readmemh("initial_ram.txt",DataMEM);
        end

    always @ (posedge CLK)
        begin
            if (MemWrite) DataMEM[Address] = Data_In; // Synchronous Write
        end

    assign DATA_Out = DataMEM[Address];           // Asynchronous Read

endmodule
```

Flow Summary	
Flow Status	Successful - Mon Apr 07 19:48:32 2014
Quartus II 32-bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Full Version
Revision Name	Memory
Top-level Entity Name	Memory
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	23,890 / 33,216 (72 %)
Total combinational functions	23,069 / 33,216 (69 %)
Dedicated logic registers	16,384 / 33,216 (49 %)
Total registers	16384
Total pins	78 / 475 (16 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Creating Memory from LUTs

Verilog Model that typically infers Dedicated memory

```
module Memory (input CLK, MemWrite, input [11:0] Address,
               input [31:0] Data_In, output reg [31:0] DATA_Out);

    reg [31:0] DataMEM [0:4095];

    initial
        begin
            $readmemh("initial_ram.txt",DataMEM)
        end

    always @(posedge CLK)
        begin
            if (MemWrite) DataMEM[Address] = Data_In; // Synchronous Write
            DATA_Out = DataMEM[Address];           // Synchronous Read
        end

endmodule
```

Flow Summary	
Flow Status	Successful - Mon Apr 07 19:23:32 2014
Quartus II 32-bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Full Version
Revision Name	Memory
Top-level Entity Name	Memory
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	33 / 33,216 (< 1 %)
Total combinational functions	32 / 33,216 (< 1 %)
Dedicated logic registers	33 / 33,216 (< 1 %)
Total registers	33
Total pins	78 / 475 (16 %)
Total virtual pins	0
Total memory bits	131,072 / 483,840 (27 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Creating Memory from LUTs

Verilog Model that typically infers Dedicated memory

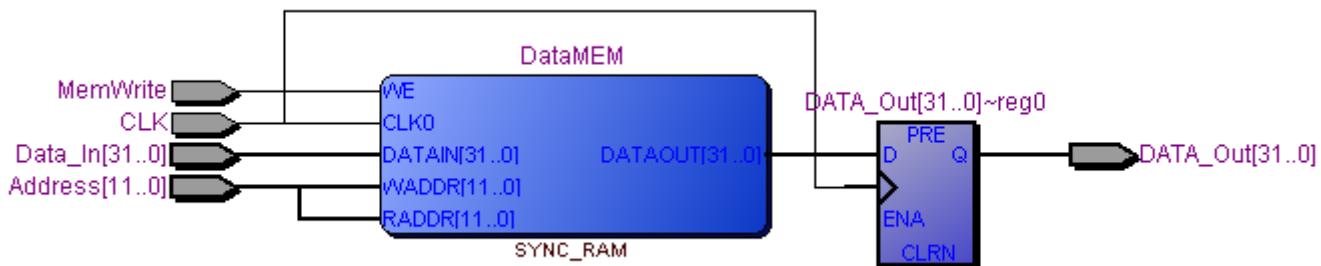
```
module Memory (input CLK, MemWrite, input [11:0] Address,
               input [31:0] Data_In, output reg [31:0] DATA_Out);

    reg [31:0] DataMEM [0:4095];

    initial
        begin
            $readmemh("initial_ram.txt",DataMEM);
        end

    always @ (posedge CLK)
        begin
            if (MemWrite) DataMEM[Address] = Data_In; // Synchronous Write
            DATA_Out = DataMEM[Address];           // Synchronous Read
        end

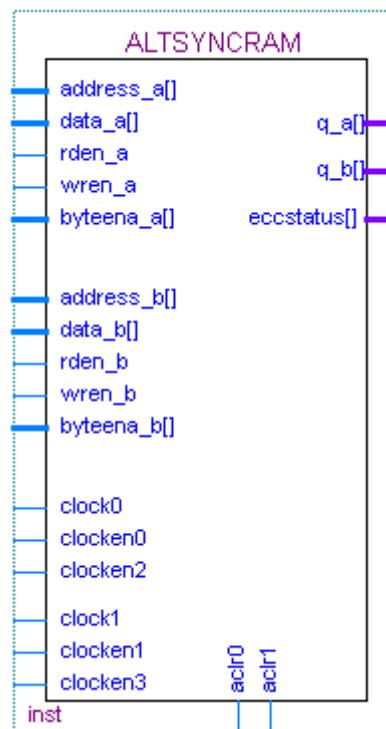
endmodule
```



Structural Technique for Implementing Memory Elements in the FPGA's Dedicated Block RAM Memory

```
// Memory module in Verilog HDL in a manner that explicitly utilizes
// the Altera Megafunction library in Quartus II. This will utilize
// 64 M4K dedicated memory elements for a total storage of
// 262144 bits (It also uses 48 LE's).
module rom (input [14:0] address, input clock, output [7:0] q);
    altsyncram C1 (
        .clock0 (clock),
        .address_a (address),
        .q_a (q),
        .aclr0 (1'b0),
        .aclr1 (1'b0),
        .address_b (1'b1),
        .addressstall_a (1'b0),
        .addressstall_b (1'b0),
        .byteena_a (1'b1),
        .byteena_b (1'b1),
        .clock1 (1'b1),
        .clocken0 (1'b1),
        .clocken1 (1'b1),
        .clocken2 (1'b1),
        .clocken3 (1'b1),
        .data_a ({8{1'b1}}),
        .data_b (1'b1),
        .eccstatus (),
        .q_b (),
        .rden_a (1'b1),
        .rden_b (1'b1),
        .wren_a (1'b0),
        .wren_b (1'b0));

```



```
defparam
    C1.clock_enable_input_a = "BYPASS",
    C1.clock_enable_output_a = "BYPASS",
    C1.init_file = "E:/example/rom.hex",
    C1.intended_device_family = "Cyclone II",
    C1.lpm_hint = "ENABLE_RUNTIME_MOD=NO",
    C1.lpm_type = "altsyncram",
    C1.numwords_a = 32768,
    C1.operation_mode = "ROM",
    C1.outdata_aclr_a = "NONE",
    C1.outdata_reg_a = "UNREGISTERED",
    C1.power_up_uninitialized = "FALSE",
    C1.ram_block_type = "M4K",
    C1.widthad_a = 15,
    C1.width_a = 8,
    C1.width_byteena_a = 1;
endmodule
```

Using LUTS to Implement a 4x4 Multiplier (using LUTs – i.e. distributed RAM)

```
module LUTmult(input [3:0] Mplier, Mcand, output reg [7:0] Product);

    reg [7:0] prod_rom [0:255];

    initial
        begin:ROM_LOAD
            reg [8:0] i;
            for (i=0;i<256;i=i+1)
                prod_rom[i] = {4'b0000,i[7:4]}*{4'b0000,i[3:0]};
        end

```

columns

Flow Summary	
rows	Flow Status
x"00", x"01",	Successful - Tue Apr 08 09:08:21 2014
x"00", x"02",	Quartus II 32-bit Version
x"00", x"03",	13.0.1 Build 232 06/12/2013 SP 1 SJ Full Version
x"00", x"04",	Revision Name
x"00", x"05",	LUTmult
x"00", x"06",	Top-level Entity Name
x"00", x"07",	LUTmult
x"00", x"08",	Family
x"00", x"09",	Cyclone II
x"00", x"0A",	Device
x"00", x"0B",	EP2C35F672C6
x"00", x"0C",	Timing Models
x"00", x"0D",	Final
x"00", x"0E",	Total logic elements
x"00", x"0F",	67 / 33,216 (< 1 %)
Total combinational functions	
Dedicated logic registers	
Total registers	67 / 33,216 (< 1 %)
Total pins	0 / 33,216 (0 %)
Total virtual pins	0 / 33,216 (0 %)
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

```
    always @ (Mplier or Mcand)
        Product = prod_rom[{Mplier,Mcand}]; // read Product LUT

    endmodule
```

Using LUTS to Implement a 4x4 Multiplier (using Dedicated Block RAM)

```
module LUTmult(input [3:0] clk, Mplier, Mcand, output reg [7:0] Product);

reg [7:0] prod_rom [0:255];

initial
begin:ROM_LOAD
reg [8:0] i;
for (i=0;i<256;i=i+1)
    prod_rom[i] = {4'b0000,i[7:4]}*{4'b0000,i[3:0]};
end
```

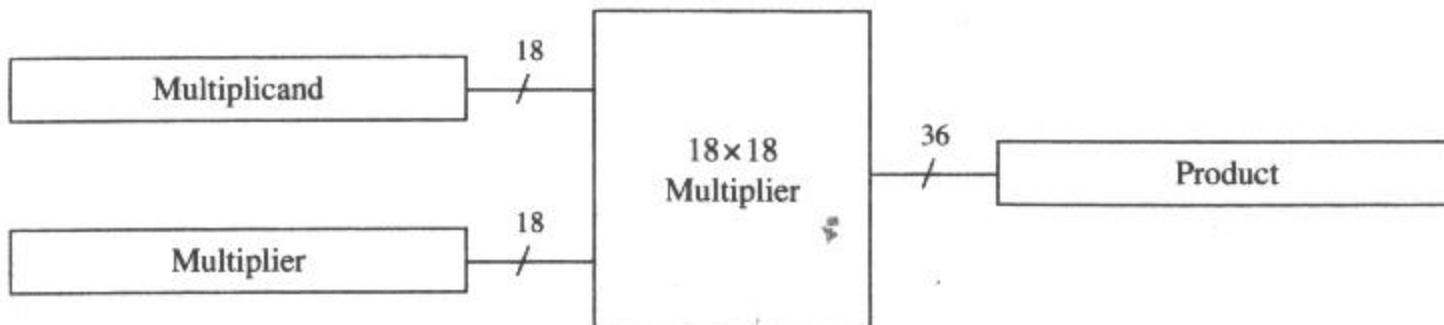
————— columns —————

Flow Summary			
rows	x"00", x"01", x"02" x"00", x"02", x"04" x"00", x"03", x"06" x"00", x"04", x"08" x"00", x"05", x"0A" x"00", x"06", x"0C" x"00", x"07", x"0E" x"00", x"08", x"10" x"00", x"09", x"12" x"00", x"0A", x"14" x"00", x"0B", x"16" x"00", x"0C", x"18" x"00", x"0D", x"1A" x"00", x"0E", x"1C" x"00", x"0F", x"1E"	Flow Status Quartus II 32-bit Version Revision Name Top-level Entity Name Family Device Timing Models Total logic elements ... Total combinational functions ... Dedicated logic registers Total registers Total pins Total virtual pins Total memory bits Embedded Multiplier 9-bit elements Total PLLs	Successful - Tue Apr 08 09:19:39 2014 13.0.1 Build 232 06/12/2013 SP 1 SJ Full Version LUTmult LUTmult Cyclone II EP2C35F672C6 Final 0 / 33,216 (0 %) 0 / 33,216 (0 %) 0 / 33,216 (0 %) 0 20 / 475 (4 %) 0 2,048 / 483,840 (< 1 %) 0 / 70 (0 %) 0 / 4 (0 %) 00", x"00",)E", x"0F", , x"1E", ?A", x"2D", 38", x"3C", 16", x"48", 54", x"5A", 52", x"69", 70", x"78", 7E", x"87", 3C", x"96", 9A", x"A5", A8", x"B4", B6", x"C3", C4", x"D2", D2", x"E1");

```
always @ (posedge clk)
Product = prod_rom[{Mplier,Mcand}]; // read Product LUT
// synchronously

endmodule
```

Common Operation Support (Multipliers in FPGAs)



```
module multiplier (input [31:0] A,B, output [63:0] C);  
    assign C = A * B;  
endmodule
```

Flow Summary	
Flow Status	Successful - Tue Apr 08 09:27:18 2014
Quartus II 32-bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Full Version
Revision Name	multiplier
Top-level Entity Name	multiplier
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	80 / 33,216 (< 1 %)
Total combinational functions	80 / 33,216 (< 1 %)
Dedicated logic registers	0 / 33,216 (0 %)
Total registers	0
Total pins	128 / 475 (27 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	8 / 70 (11 %)
Total PLLs	0 / 4 (0 %)

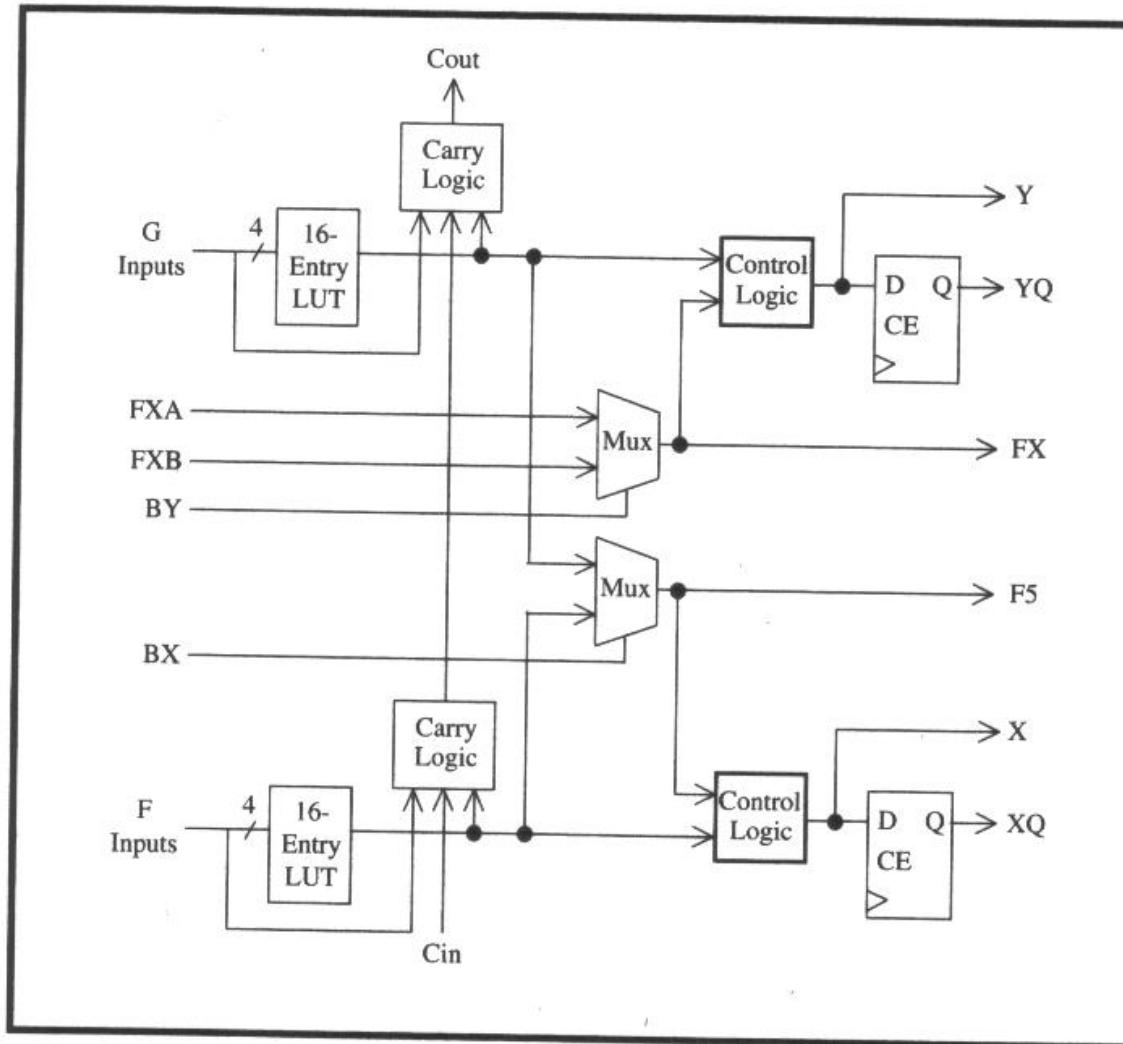
FPGA	Dedicated Multipliers
Xilinx Virtex-4, Virtex-II Pro/X, Spartan-3E, Spartan 3/3L	18×18 multipliers
Altera Stratix II Cyclone II	18×18 multipliers

Common Operation Support

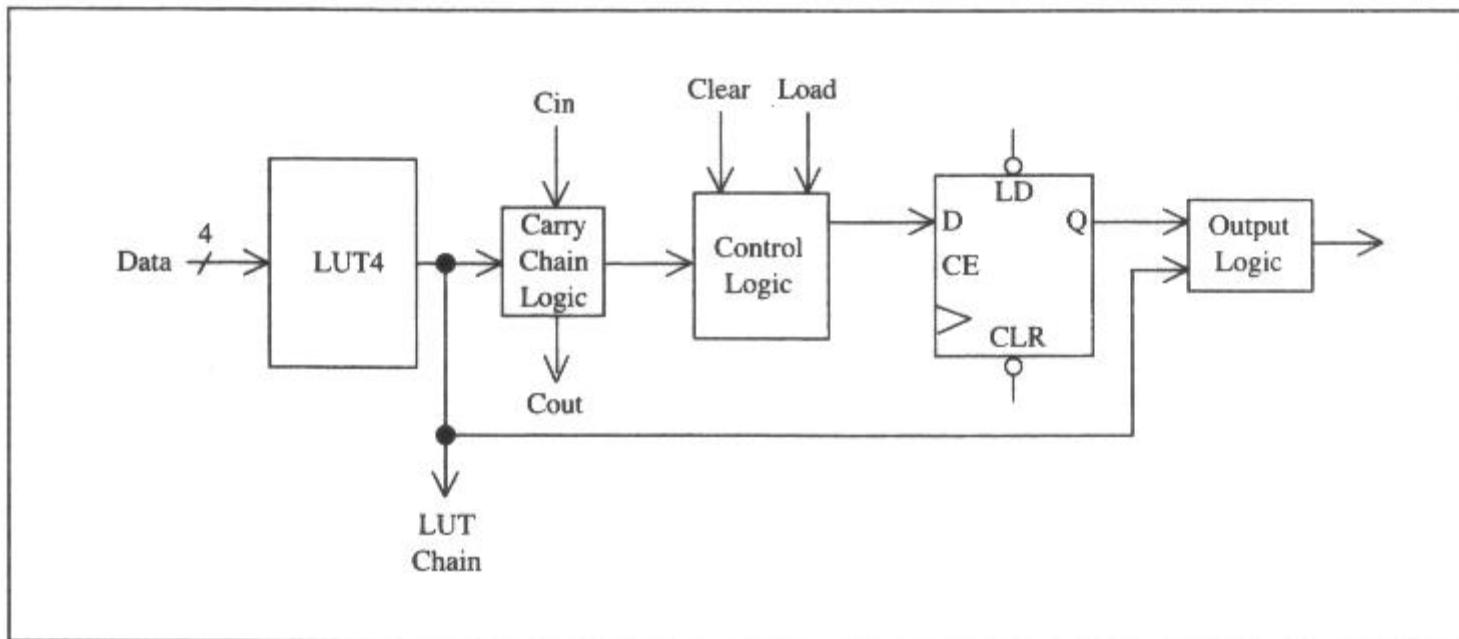
(high end logic, embedded processors, dsp, soft/hard ip cores)

FPGA	Embedded Processor
Xilinx Virtex-4, Virtex-II Pro/X	IBM 400 MHz PowerPC
Xilinx Spartan-3E, Spartan 3/3I	MicroBlaze PicoBlaze
Altera Stratix II Cyclone II	Nios II
Altera APEX APEX II	ARM, MIPS, Nios
Altera Excalibur	ARM 9
Actel Fusion	ARM7

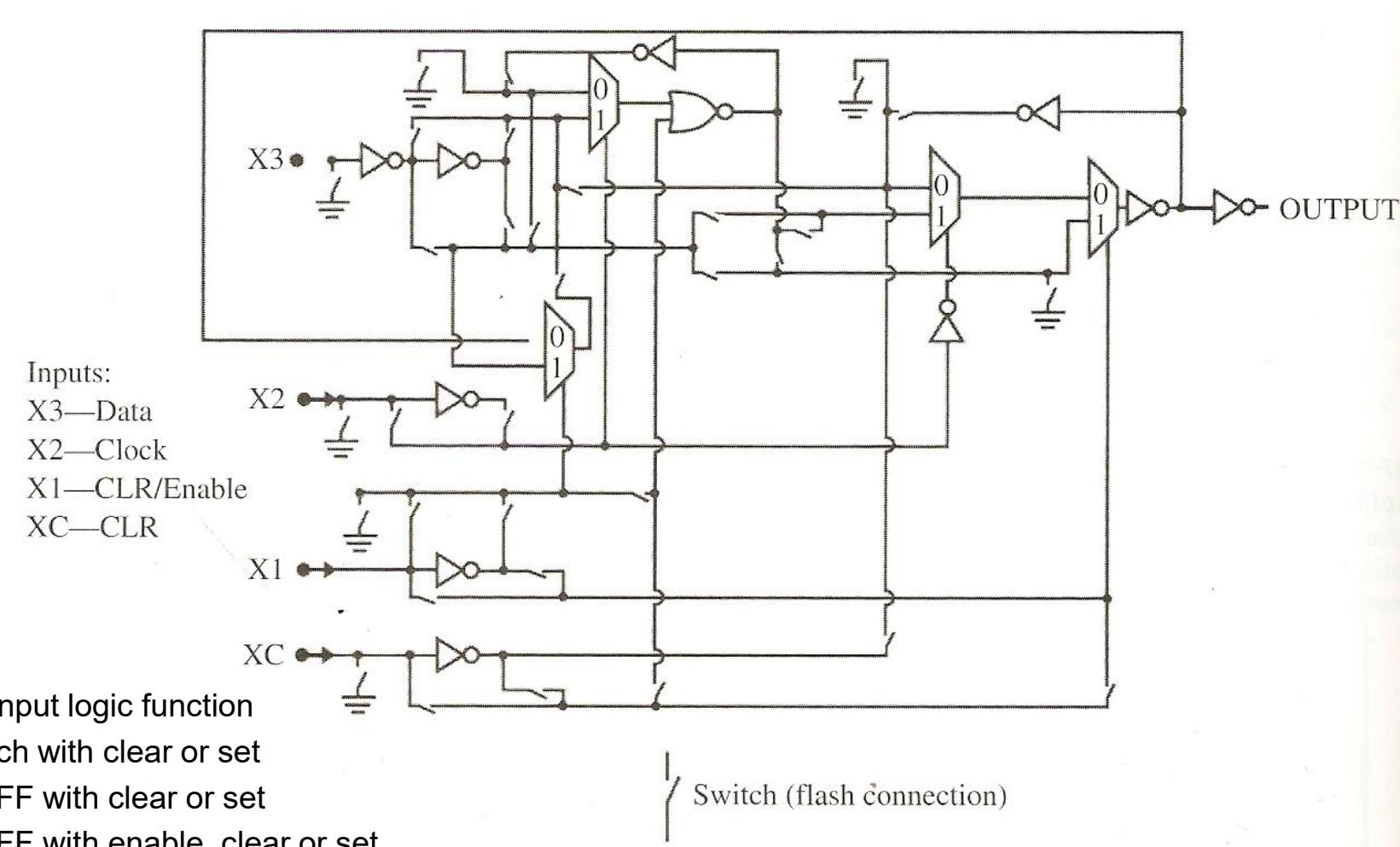
Logic Blocks of some Commercial FPGAs (Xilinx Spartan Slice)



Logic Blocks of some Commercial FPGAs (Altera Stratix Logic Element -- LE)

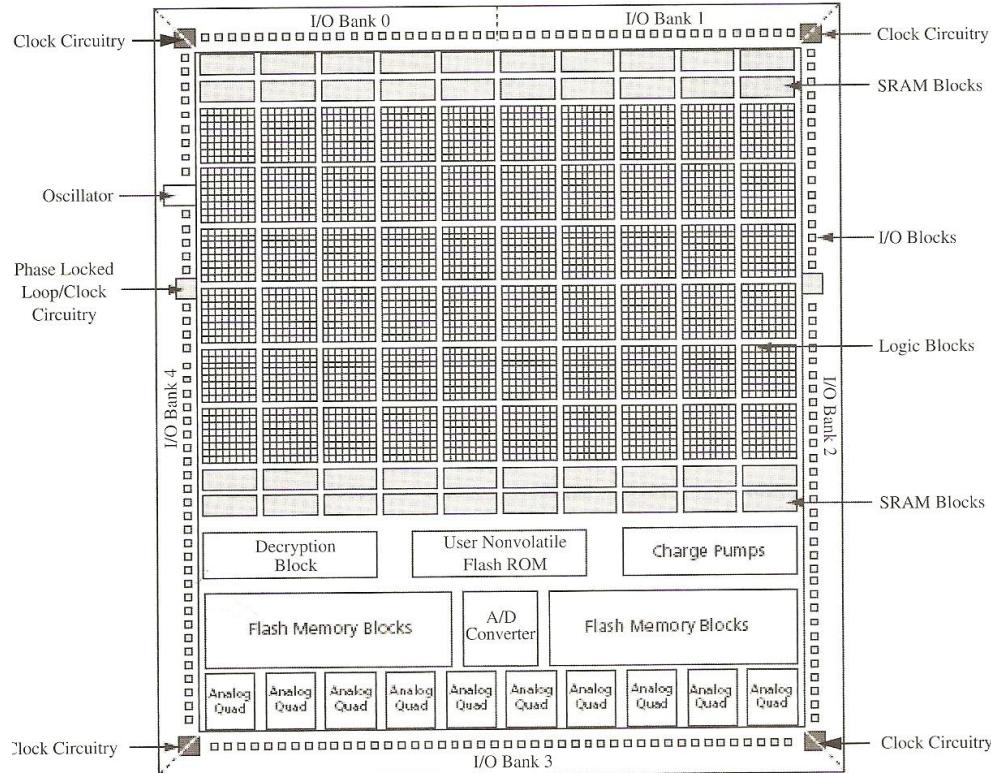


Logic Blocks of some Commercial FPGAs (Actel Fusion & ProAS1C Logic Block)



Notes on Actel Fusion Architecture

The Actel Fusion architecture, shown in Figure 3-40, provides several specialized components, including embedded RAM, decryption, and A/D converters. At the core of the chip are tiles of logic blocks (VersaTiles in Actel terminology). The embedded RAM is in the form of rows of SRAM blocks above and below the tiles of logic blocks. Several specialized components appear below the SRAM blocks in the bottom. There is a dedicated decryption unit that implements the AES decryption algorithm. (AES stands for Advanced Encryption Standard, which has been the cryptographic standard for the U.S. government since 2001.) There is an analog-to-digital converter (ADC) that accepts inputs from several analog quads, which are circuitry to condition analog signals received by the FPGA. The analog quads contain circuitry to monitor and condition signals according to voltage, current, and temperature.



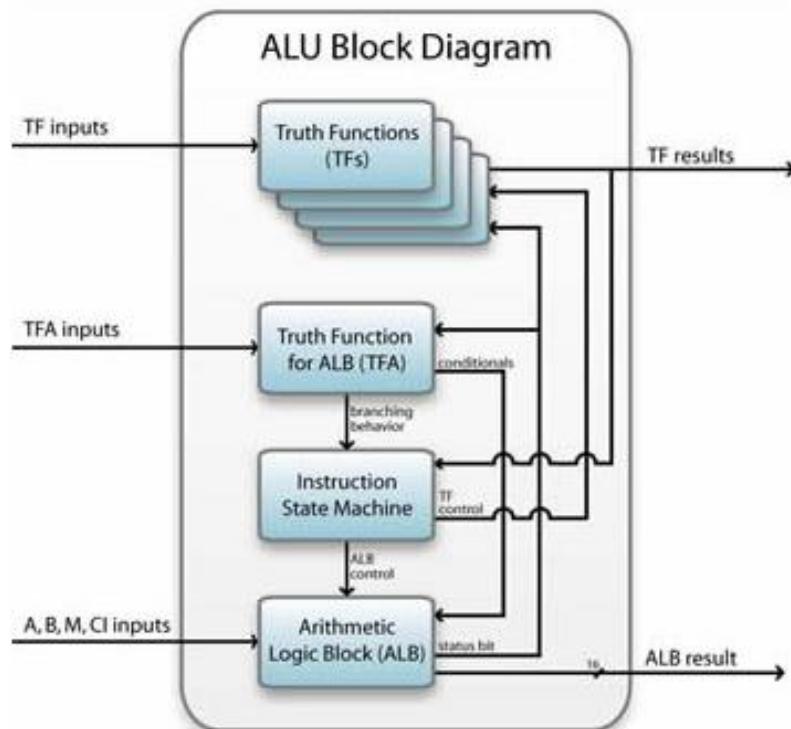
Granularity

- FPGA Granularity – working definition – complexity of the Logic Blocks
 - Fine grain; FPGAs with very simple logic blocks
 - Coarse grain; FPGAs with complex logic blocks

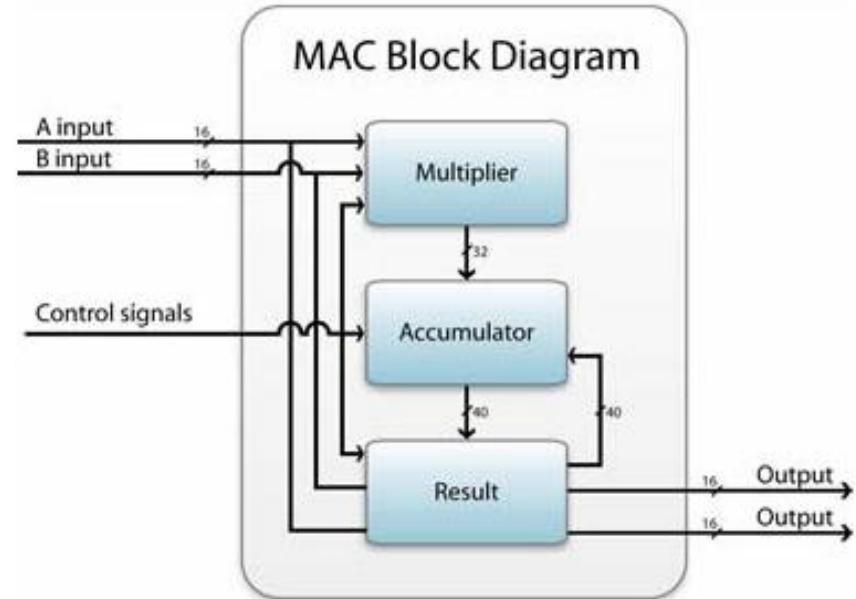
Fine grain	range	Coarse grain
simple LBs	(smaller DP)	complex LBs
more LBs in FPGA		less LBs in FPGA
easier to map		harder to map
harder to place/route		easier to place/route
can waste LBs		can waste internal LB resources
Actel Fusion	↔	Altera Stratix ↔ Xilinx Spartan

MathStar's Field Programmable Logic Array (Very Coarse-Grain Logic Blocks)

Arithmetic Logic Unit (ALU)

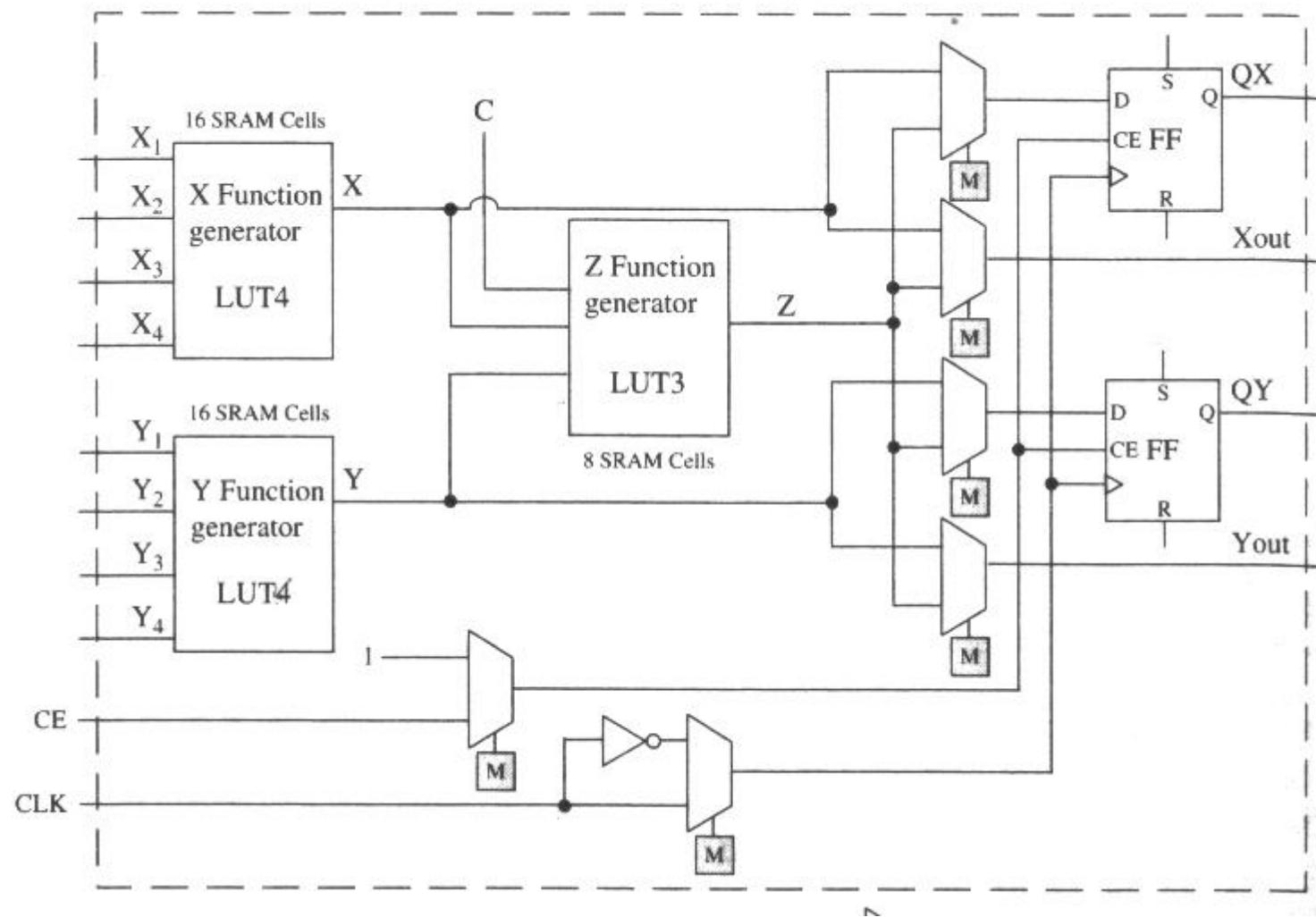


Multiply - Accumulator (MAC)



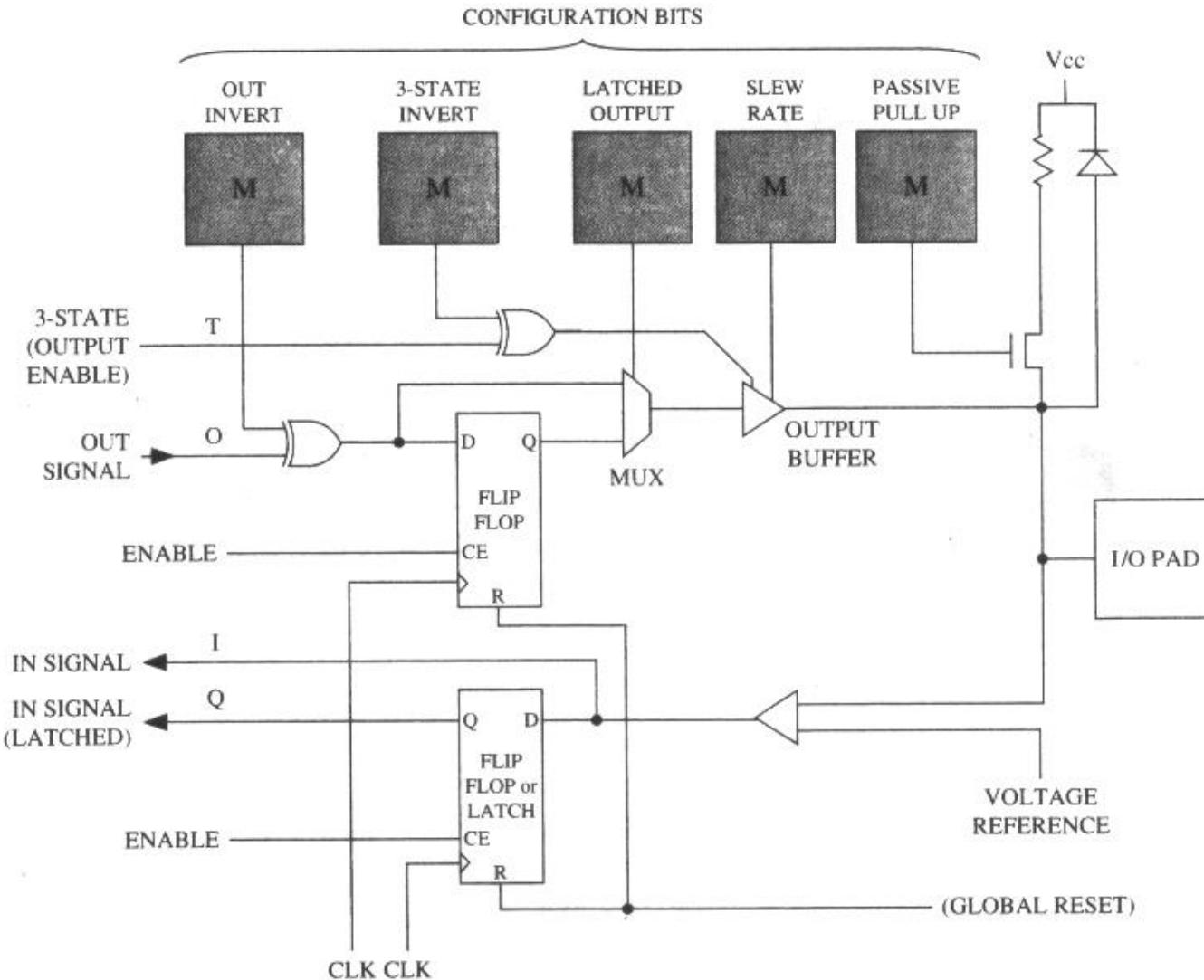
Cost of Reconfigurability

(programmable points in a logic block)



Cost of Reconfigurability

(programmable points in a FPGA I/O Block)



Cost of Reconfigurability

Vendor	Device Family	Device	# of Configuration Bits	# of Logic Blocks	# of LUTs	# Usable I/O Pins
Xilinx	Virtex-5	XC5VLX30	8.4M	4,800	19,200	400
		XC5VLX330	79.7M	51,840	207,360	1200
Xilinx	Virtex-II	XC2V40	0.3M	256	512	88
		XC2V8000	26.2M	46,592	93,184	1108
Xilinx	Spartan 3E	XC3S100E	0.6M	960	1,920	108
		XC3S1600E	6.0M	14,752	29,504	376
Altera	Stratix II	EP2S15	4.7M	6,240	12,480	366
		EP2S180	49.8M	71,760	143,520	1170
Altera	Stratix	EP1S10	3.5M	10,570	10,570	426
		EP1S80	23.8M	79,040	79,040	1238
Altera	Cyclone II	EP2C5	1.3M	4,608	4,608	158
		EP2C70	14.3M	68,416	68,416	622

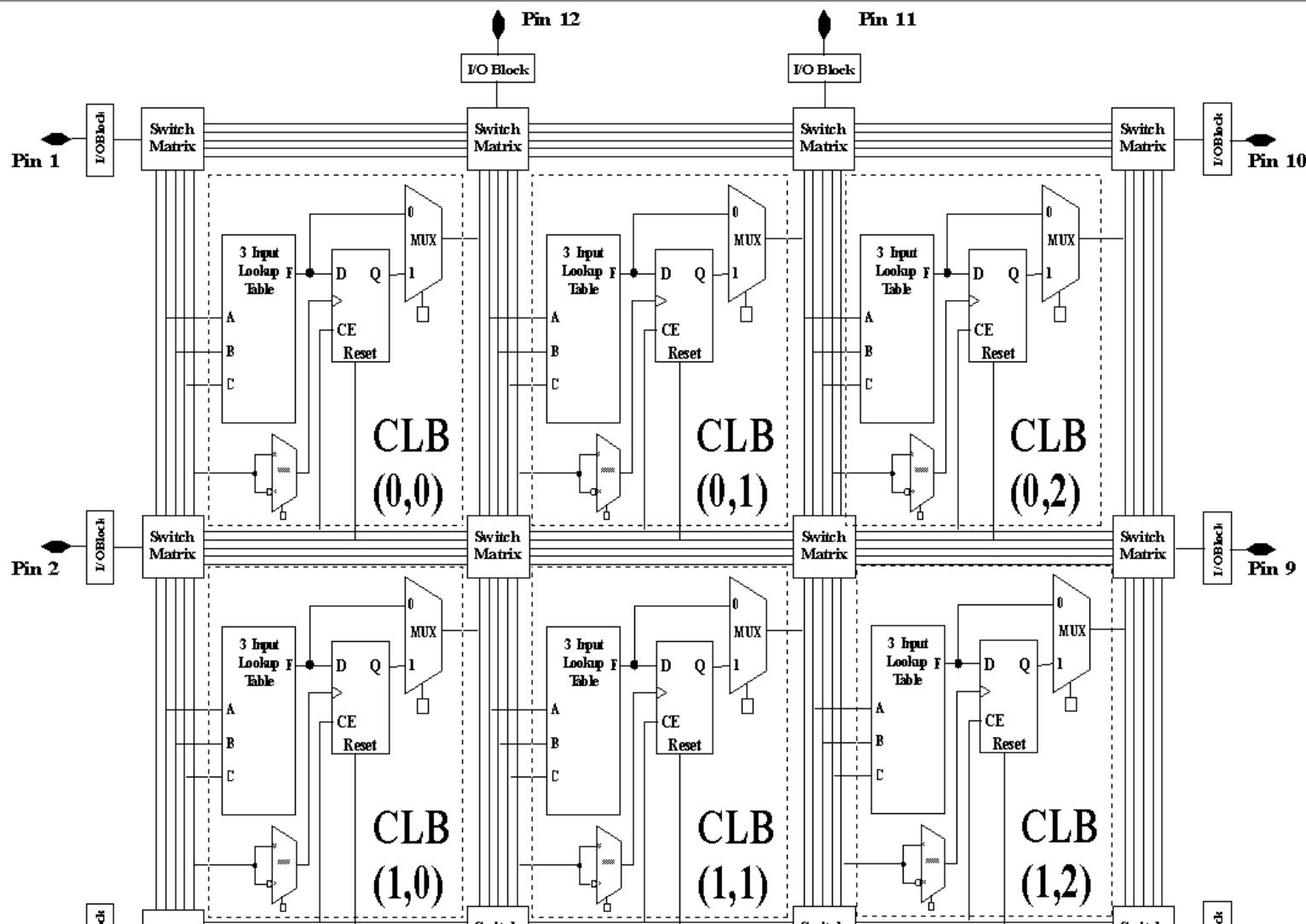
State Machine Implementation in FPGAs

- In FPGAs, it may not be important to minimize the number of flip-flops used in the design
- Goal should be to reduce the total number of logic cells used and the interconnections between the cells
- The manner in which the state assignments are performed can facilitate this goal.
- Often this means that we need to use more flip-flops than the minimum of $\log_2(N)$, where N is the number of states.

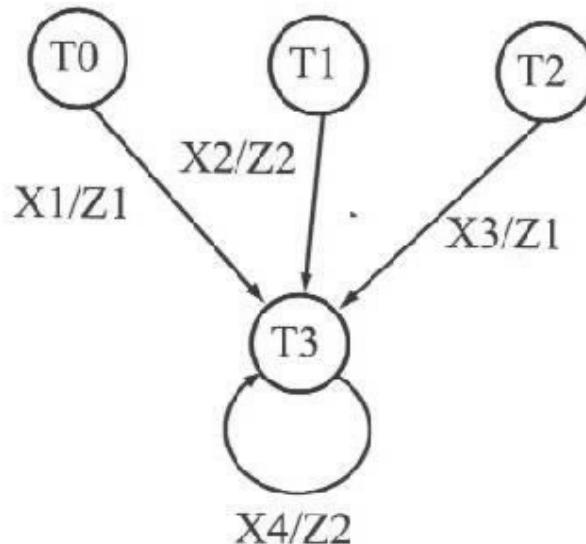
One-Hot State Assignment

- The one-hot assignment method uses one flip-flop for each state
 - State machine with N states requires N flip-flops (not $\log_2(N)$ flip-flops as before!)
- Often FPGAs have a flip-flop with each cell. This flip-flop cannot be used by other circuitry when any of the other components of the cell are used
 - Therefore the availability of usable flip-flops in the FPGA may not be the limiting constraint.
- Often one-hot state assignments result in simpler state assignments and output equations which are easier to route and fit in the lookup tables.

Simplified View of an FPGA Architecture



One-Hot State Assignment



One-hot state assignment for flip-flops Q0 Q1 Q2 Q3:

T0: 1000, T1: 0100, T2: 0010, T3: 0001

$$Q3^+ = X_1 Q_0 + X_2 Q_1 + X_3 Q_2 + X_4 Q_3$$

$$Z_1 = X_1 Q_0 + X_3 Q_2$$

$$Z_2 = X_2 Q_1 + X_4 Q_3$$

One-Hot State Assignment

- When a one-hot assignment is used, resetting the system requires that one flip-flop be set to 1 instead of being reset to 0.
- Could be a problem if flip-flops do not have a preset input
 - Two workarounds in this case
 - Replace one of the flip-flop variables with its complement in all the equations ($Q_3+ = X_1 Q_0'$ instead of $Q_3+ = X_1 Q_0$) then
So: $Q_0 Q_1 Q_2 Q_3 = 0000$, $Q_3+ = X_1 Q_0'$ instead of $Q_0 Q_1 Q_2 Q_3 = 1000$ and $Q_3+ = X_1 Q_0$.
 - Add (i.e. OR) the term $Q_0' Q_1' Q_2' Q_3'$ to the equation Q_0+ then after the first clock cycle Q_0+ will equal 1 and you will be in the correct starting state!

Metrics for FPGA Capacity

Vendor	FPGA Product	Capacity (Approx) in Gates/LUTs
Xilinx	Spartan-II	15K to 200K
	Spartan-IIE	50K to 600K
	Spartan-3	50K to 5M
	Virtex-5	19,200 to 207,360 LUTs
	Virtex	57,906 to 1,124,022
	Virtex-E	71,693 to 4,074,387
	Virtex-II	40K to 8M
Altera	ACEX 1K	56K to 257K
	APEX II	1.9M to 5.25M
	FLEX 10K	10K to 50K
	Stratix/Stratix II	10,570 to 132,540 logic elements
Lattice Semiconductor	LatticeECP2	6K to 68K LUTs
	Lattice SC	15.2K to 115.2K LUTs
	ispXPGA	139K to 1.25M
	MachXO	256 to 2280 LUTs
	LatticeECP	6.1K to 32.8K LUTs
Actel	Axcelerator	125K To 2M
	eX	3K to 12K
	ProASIC3	30K to 3M
	MX	3K to 54K
Quick Logic	Eclipse/EclipsePlus	248K to 662K
	Quick RAM	45K to 176K
	pASIC 3	5K to 75K
Atmel	AT40K	5K to 40K
	AT40KAL	5K to 50K

Metrics for FPGA Capacity

(equivalent gate count, etc)

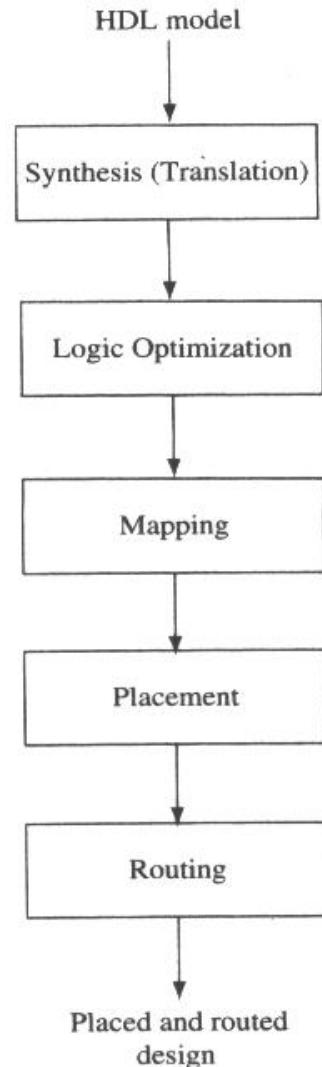
PREP Benchmarks

The **Programmable Electronics Performance Company (PREP)** was a non-profit organization that gathered and distributed a series of benchmarks for programmable ASICs. The nine PREP benchmark circuits in the PREP 1.3 suite were as follows:

1. An 8-bit datapath consisting of a 4-to-1 MUX, a register, and a shift register
2. An 8-bit timer-counter consisting of two registers, a 4-to-1 MUX, a counter, and a comparator
3. A small state machine (8 states, 8 inputs, and 8 outputs)
4. A larger state machine (16 states, 8 inputs, and 8 outputs)
5. An ALU consisting of a 4×4 multiplier, an 8-bit adder, and an 8-bit register
6. A 16-bit accumulator
7. A 16-bit counter with synchronous load and enable
8. A 16-bit prescaled counter with load and enable
9. A 16-bit address decoder

PREP's online information included Verilog and VHDL source code and test benches (provided by Synplicity). PREP also made additional synthesis benchmarks available, including a bit-slice processor, multiplier, and R4000 MIPS RISC microprocessor.

FPGA CAD Design Flow



Mapping

- Mapping: process of binding technology-dependent circuits of the target technology to the technology-independent circuits in the design (produced by translation/synthesis phase)
- In FPGAs it focuses upon dividing the synthesized design into sub-blocks where each sub-block can fit into a logic block, I/O block, or some other logic entity within the FPGA
- Where these blocks are to be placed and how they are connected within the FPGA is handled in the place and rout phase of the design flow

Placement

- Placement: Process of taking defined logic sub-blocks and assigning them to physical locations with the target implementation.
- In FPGAs it focuses upon assigning each sub-block to specific Logic blocks, I/O blocks, or dedicated memory

Routing

- Routing: process of interconnecting the sub-blocks in the design.
- Note: place and route are dependent upon each other but can be performed as separate steps.
- In FPGAs routing involves providing the interconnection between the Logic Blocks, I/O blocks, and the other hardware entities after the necessary logic functionality has been assigned to these entities.

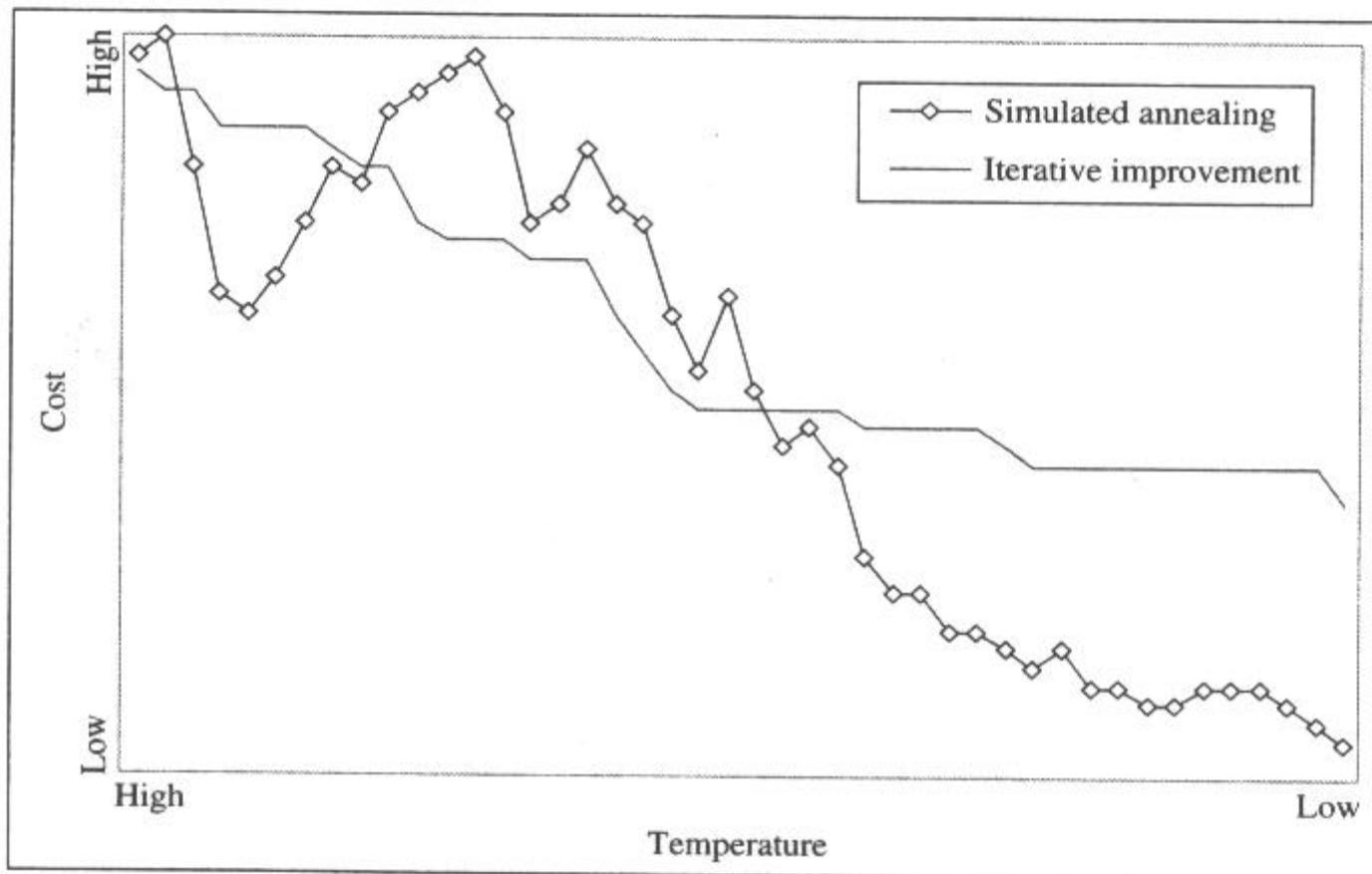
Mapping, Place and Route

- Automated Heuristics are used by the CAD tool to perform these operations.
- Most techniques create an initial solution to the problem and then try to improve upon the solution by selectively altering a portion of the solution.
- Algorithms based upon greedy/iterative improvement techniques employ a greedy strategy and will only accept a new solution if it is better than a previous one
- These algorithms can get trapped at a local optimization point that can be much worse than the global one

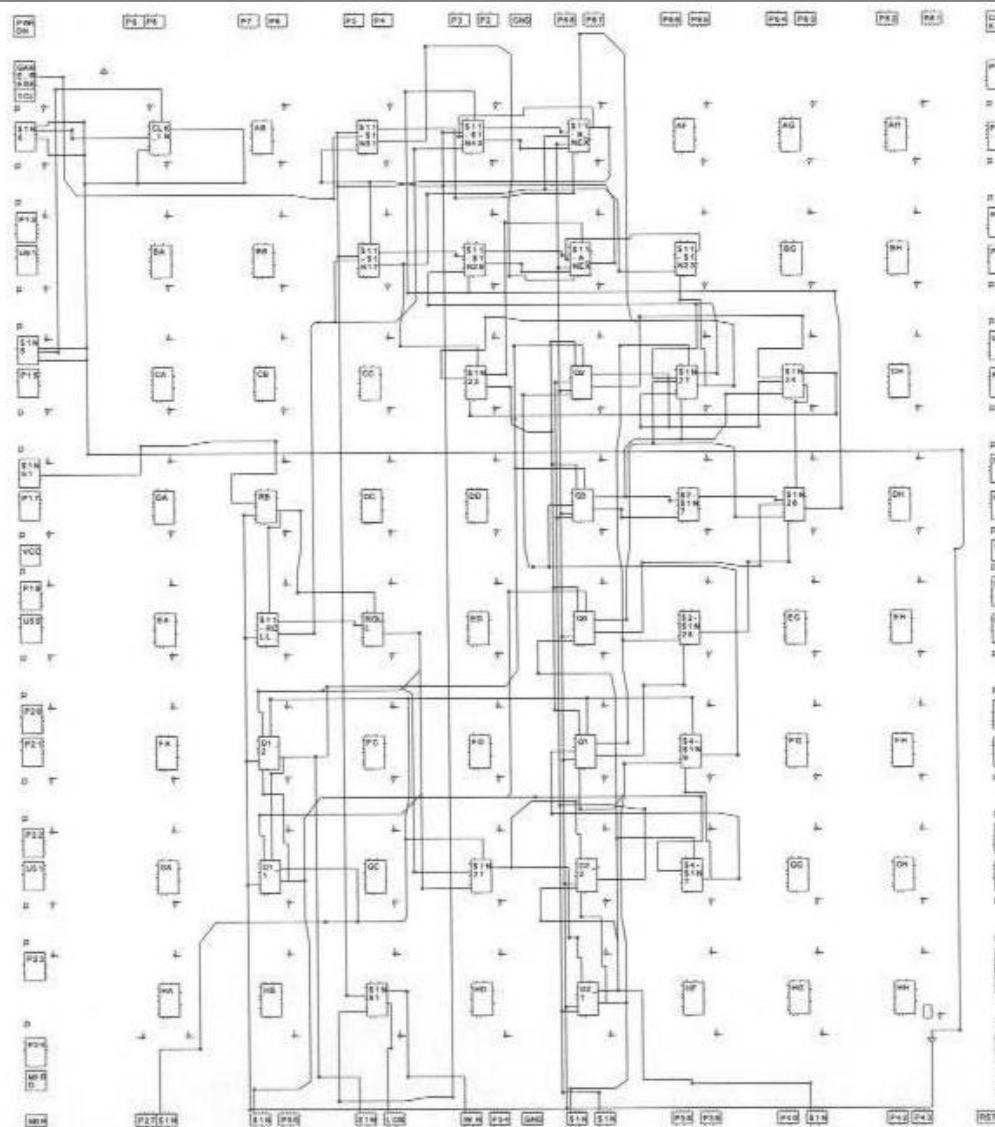
Mapping, Place and Route

- Hill Climbing Heuristics attempt overcome this problem – they are most often used in modern CAD tool environments
- Simulated Annealing is one such hill-climbing technique
- It is based upon the process of Annealing Molten Metal
- It allows solutions that are worse than a previous solution to be accepted in a probabilistic manner during different iterations.
- As the heuristic nears completion the algorithm reverts to a simple greedy iterative approach – only accepting solutions that are better than the current one.

Simulated Annealing for Place and Route



FPGA, Mapping, Place and Route

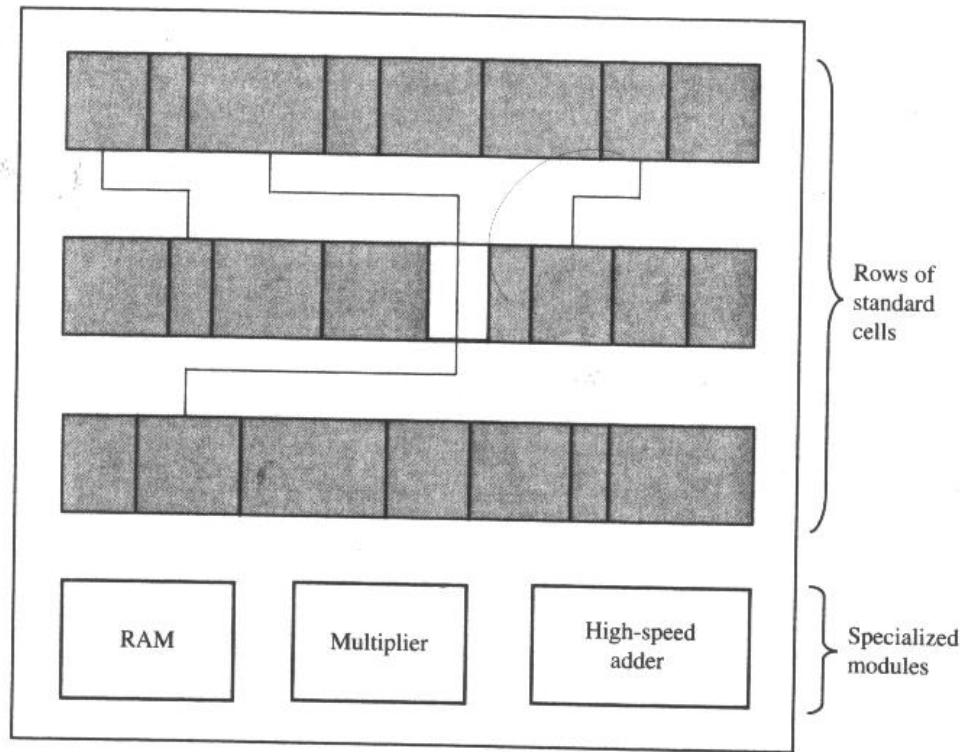


Mapping, Place and Route

- Of course the same design can be implemented in other non-FPGA based technologies as well.
- Example Standard Cell

Standard Cell Approach Standard cell design is a common technique for integrated circuit design. The design is mapped into a library of standard logic gates. Typically NOT, AND, NAND, OR, NOR, XOR, XNOR, and so on are available. CAD tools that support standard cell design methodology will also usually contain a library of complex functions and standard building blocks such as multiplexers, decoders, encoders, comparators, and counters. The design is mapped into a form that contains only cells available in the library. The cells are placed in rows that are separated by routing channels. Some cells may be used only for routing between rows of cells. Such cells are called feedthrough cells. For the standard cell methodology to be effective, the height of cells should be the same. But it is possible to include memory modules, specialized arithmetic modules, and so on.

Mapping, Placement, and Routing



Place and Route

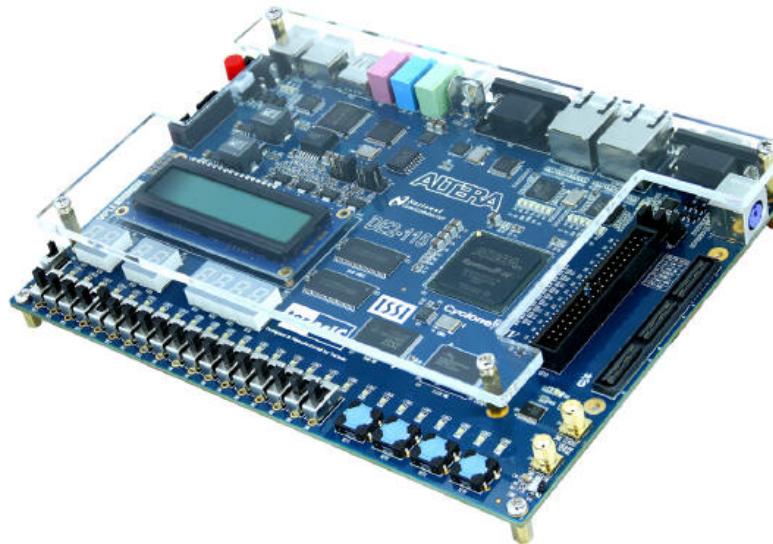
- Placement: taking the defined logic block and I/O blocks

CPE/EE 422/522

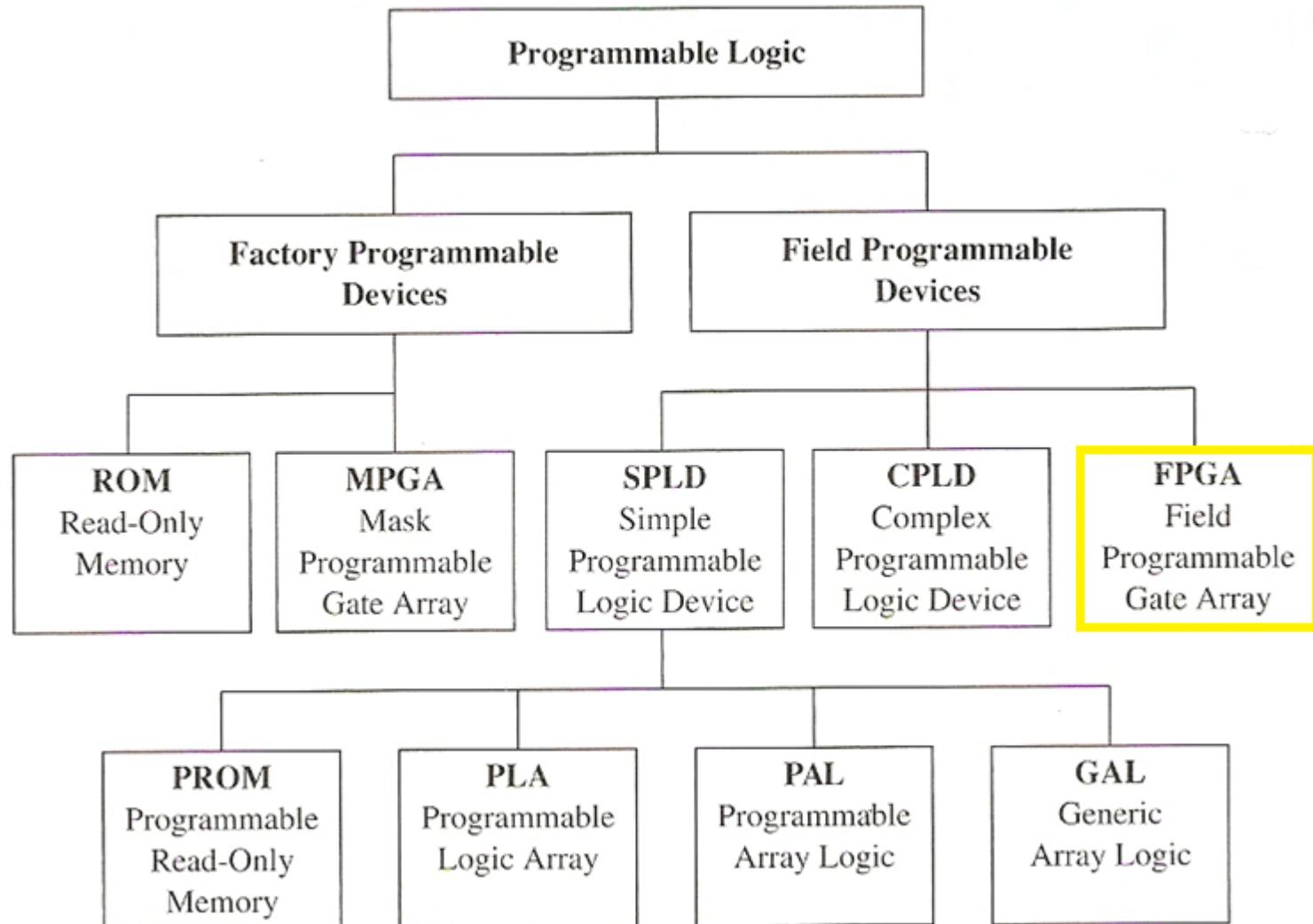
Advanced Logic Design

Electrical and Computer Engineering
UAH

Overview of FPGA Architectures & Technology
Considerations



Types of Programmable Logic Devices



Field Programmable Gate Arrays

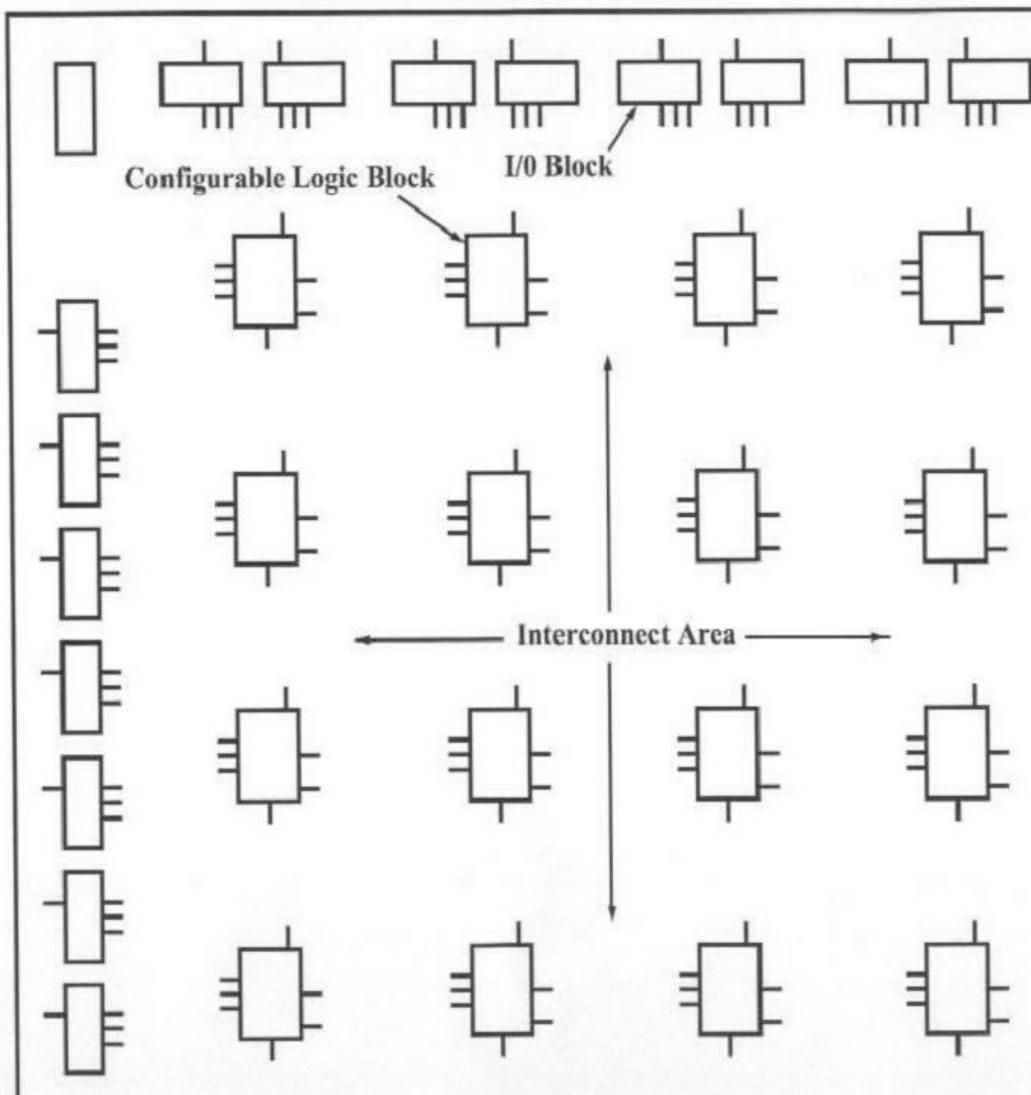
- FPGAs are pre-fabricated integrated circuits that can be configured after fabrication to emulate the functionality of a wide variety of digital systems
- Uses Include:
 - Rapid Prototyping of new hardware designs
 - Faster time to market for low to medium volume productions than Application Specific Integrated Circuits
 - Adaptable/Evolutionary Designs
 - Easy upgrade of hardware
 - Nonintrusive updates through partial reconfiguration
 - Reconfigurable Computing
 - Non-Instruction Set Architecture approaches to High-Performance or Energy Aware Computing

FPGA

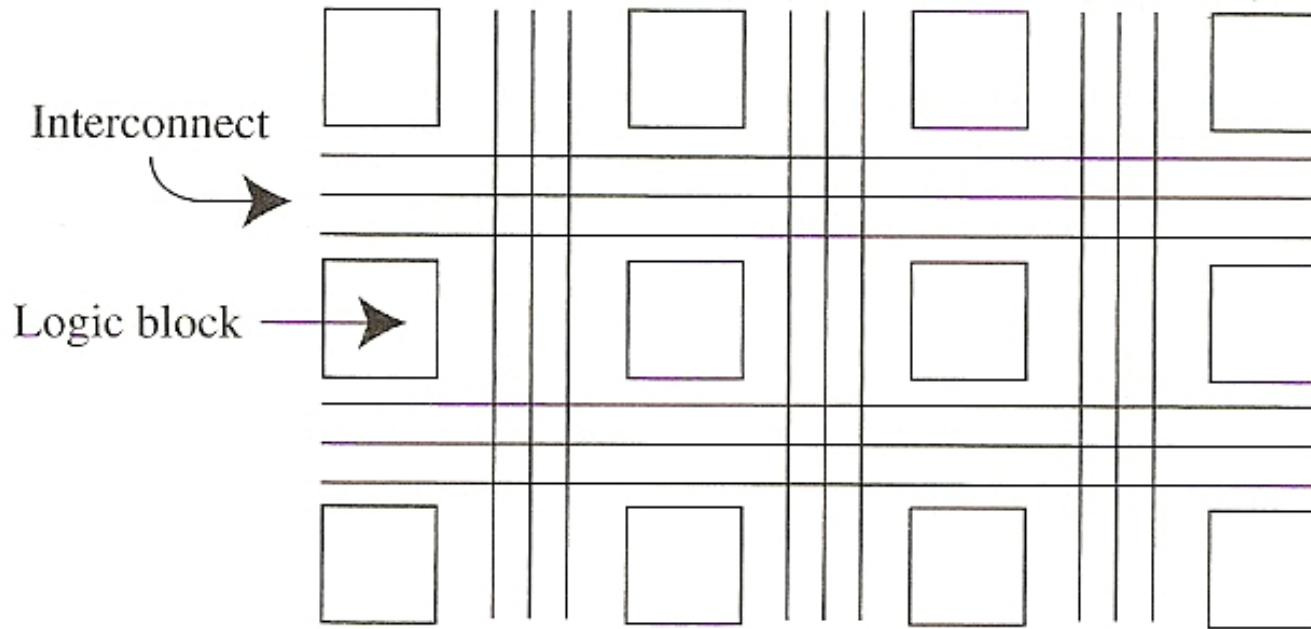
Field programmable Gate Arrays are large programmable logic devices that are comprised of

- Configurable logic blocks where small sequential and combinational logic elements of the design can be implemented.
- Programmable routing channels that interconnects these logic blocks.
- I/O blocks that are connected to logic blocks through routing interconnect and that make off-chip connections

Typical FPGA Layout



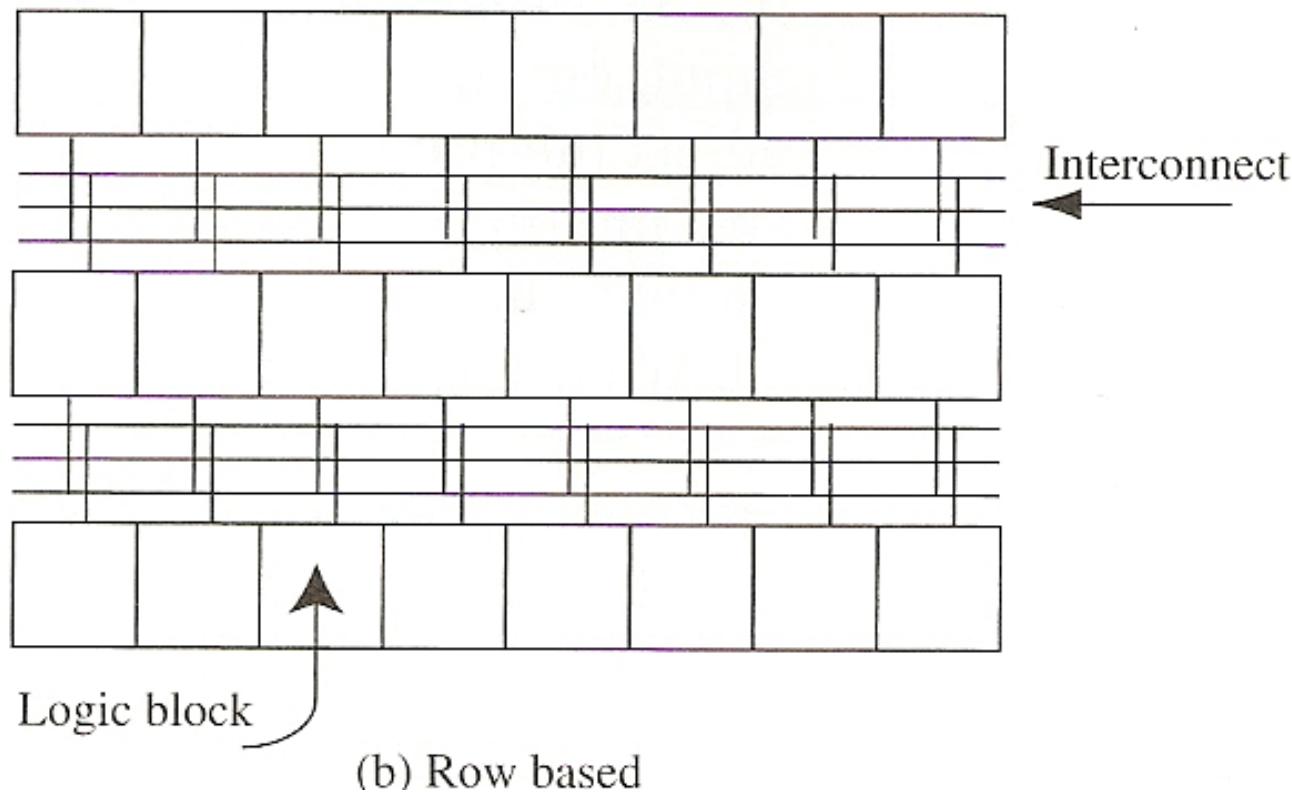
Typical FPGA Architectures



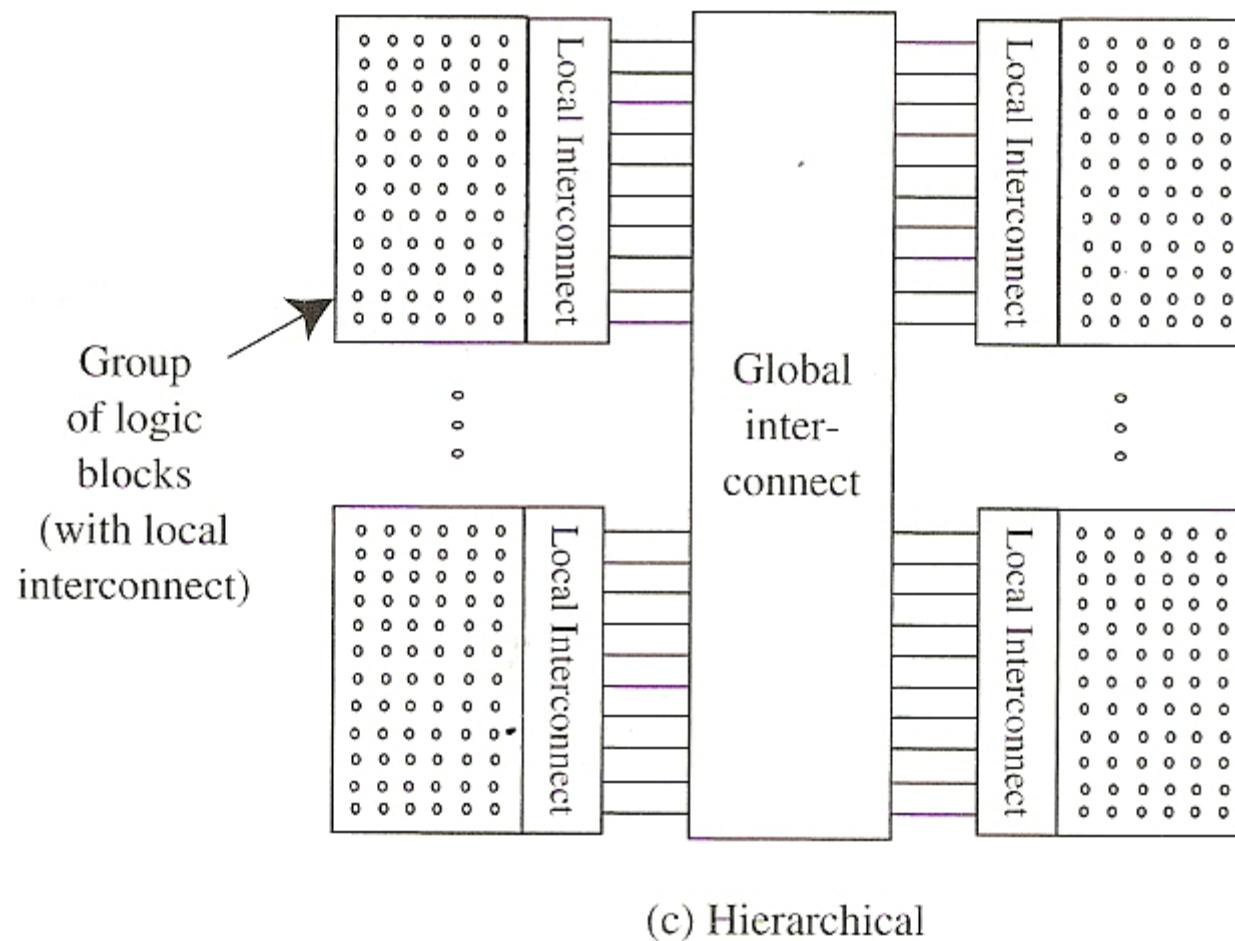
(a) Matrix based (symmetrical array)

Also called Island Style Architecture

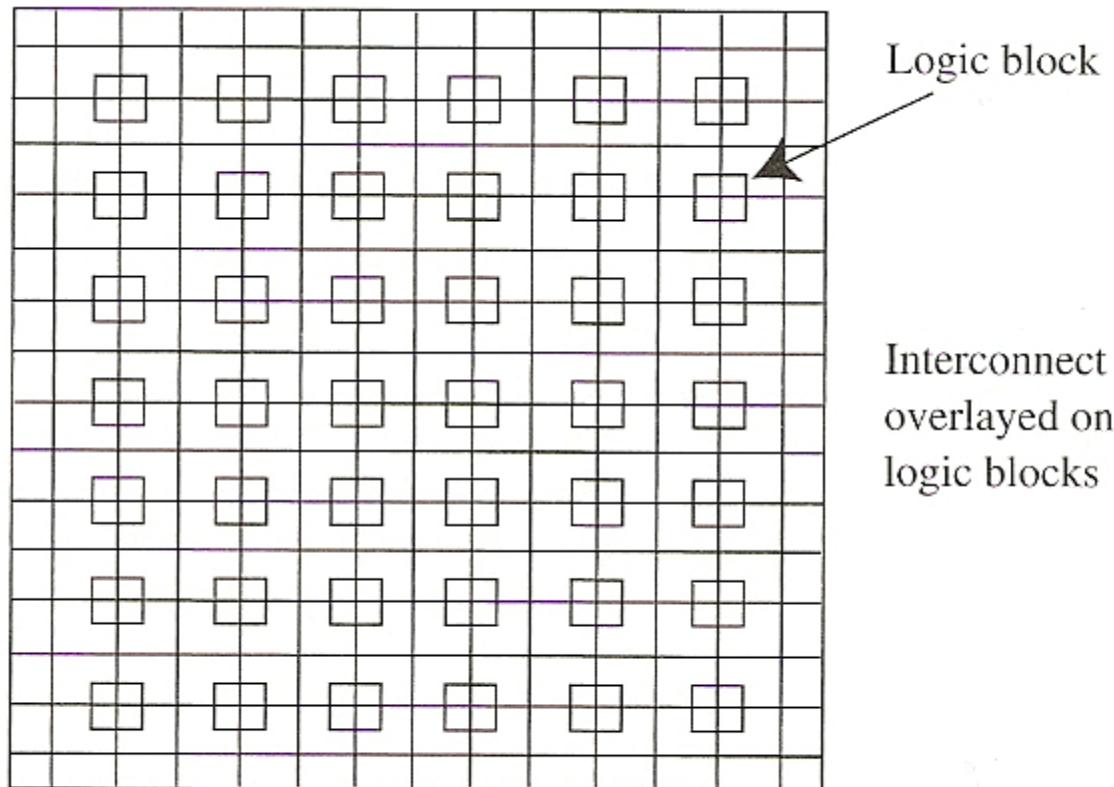
Typical FPGA Architectures



Typical FPGA Architectures



Typical FPGA Architectures



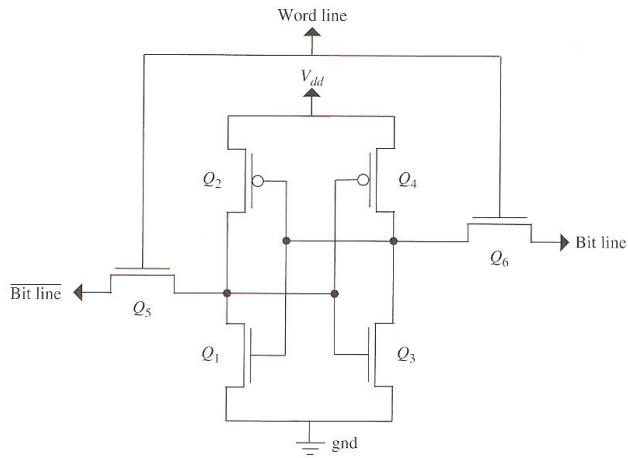
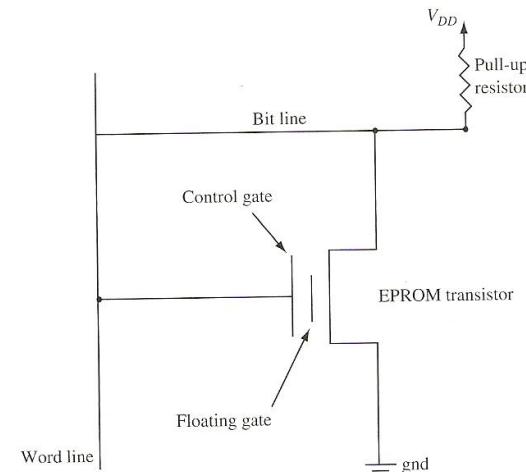
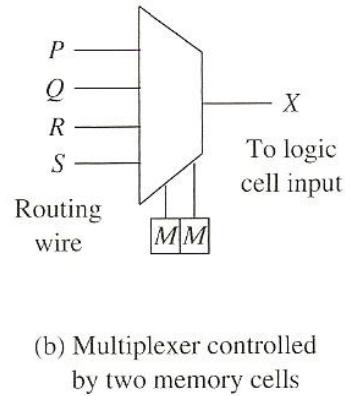
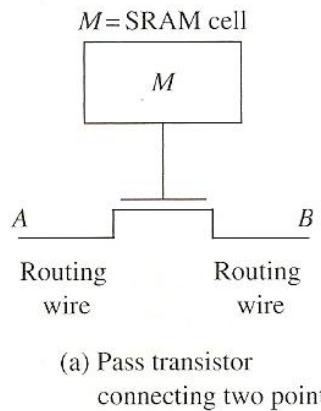
(d) Sea of gates

FPGA Configuration Technologies

Programming Technology	Volatility	Programmability	Area Overhead	Resistance	Capacitance
SRAM	Volatile	In-circuit reprogrammable	Large	Medium to high	High
EPROM	Nonvolatile	Out-of-circuit reprogrammable	Small	High	High
EEPROM	Nonvolatile	In-circuit reprogrammable	Medium to high	High	High
Antifuse	Nonvolatile	Not reprogrammable	Small	Small	Small

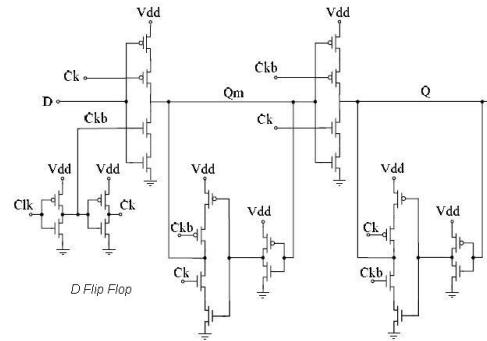
- Configuration Technology is used to
 - Configure the routing interconnect of the FPGA
 - Configure the Configurable Logic Blocks and the IO blocks

FPGA Routing Technology



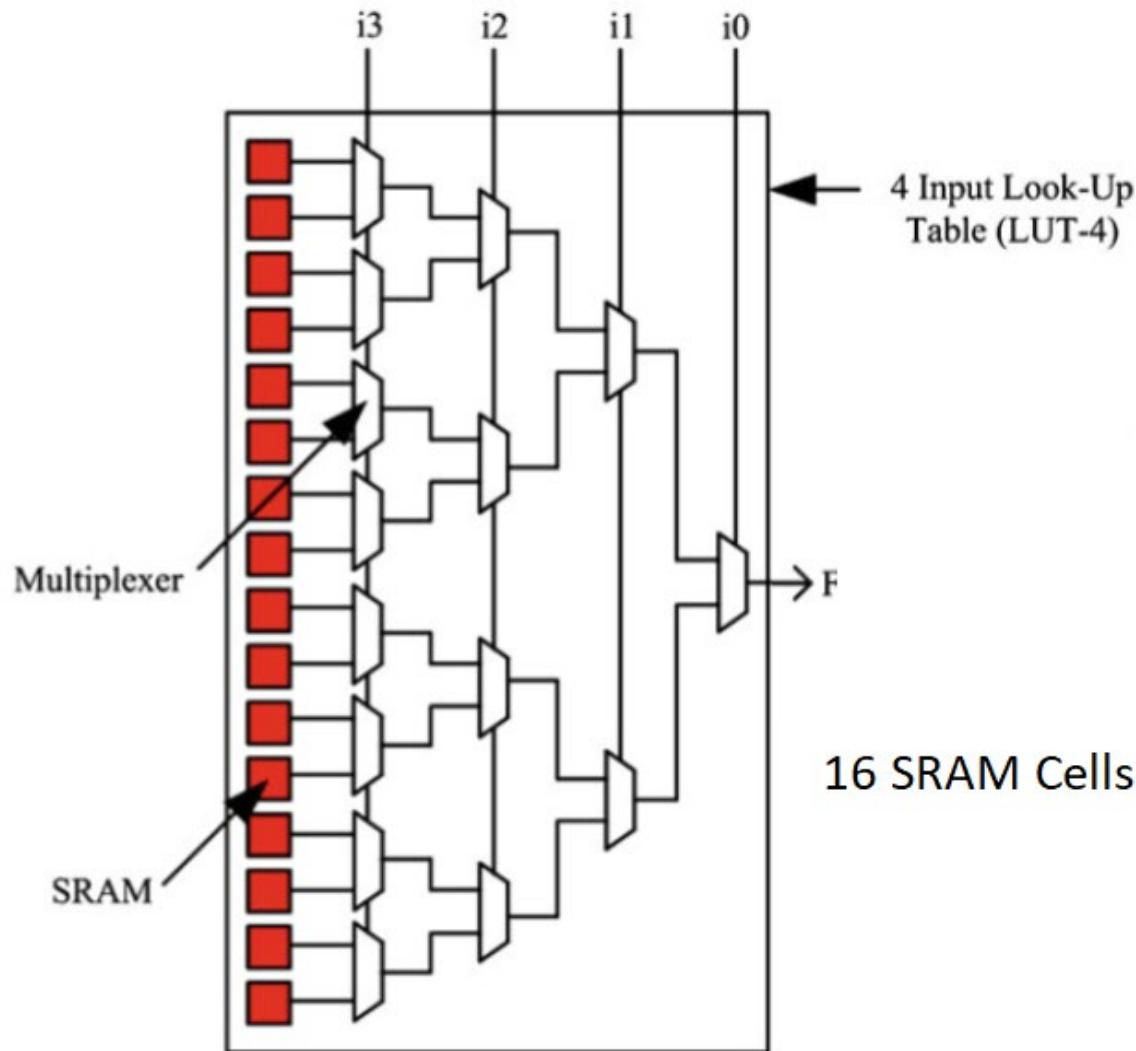
SRAM

EPROM, EEPROM, FLASH



D-Flip Flop

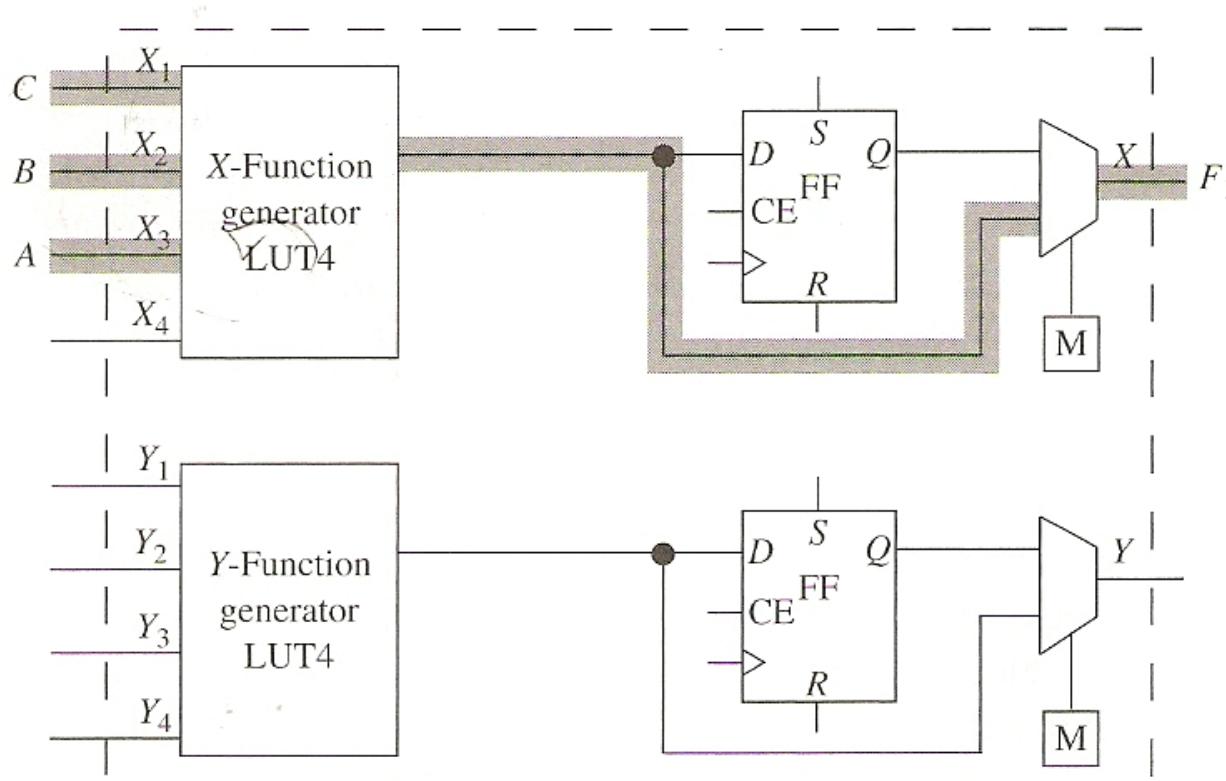
4 Input Lookup Table (LUT-4) Internal Configuration



Lookup Table Based FPGAs

$$F_1 = A'B'C + A'BC' + AB$$

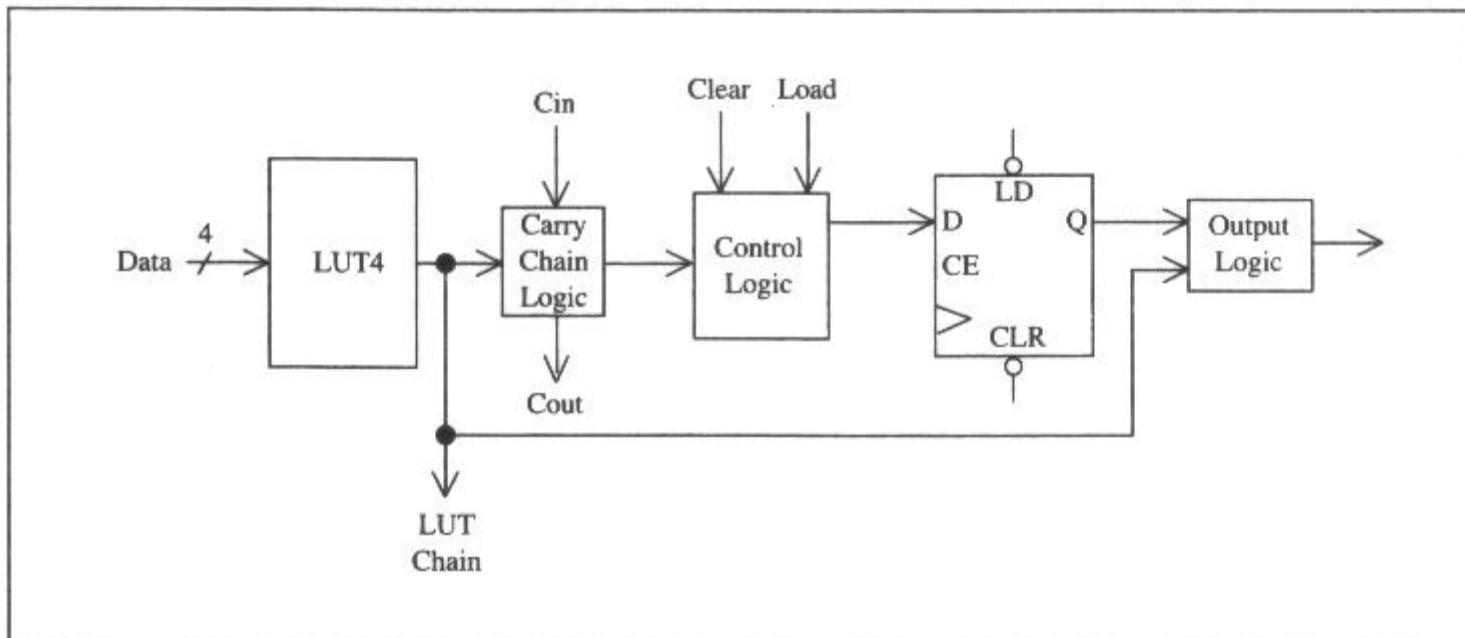
A	B	C	F ₁
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



X4	X3	X2	X1	X	F ₁
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	1	1
0	0	1	1	0	0
0	1	0	0	0	0
0	1	0	1	0	0
0	1	1	0	1	1
1	0	0	0	0	0
1	0	0	1	1	1
1	0	1	0	1	1
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	0	0
1	1	1	0	1	1
1	1	1	1	1	1

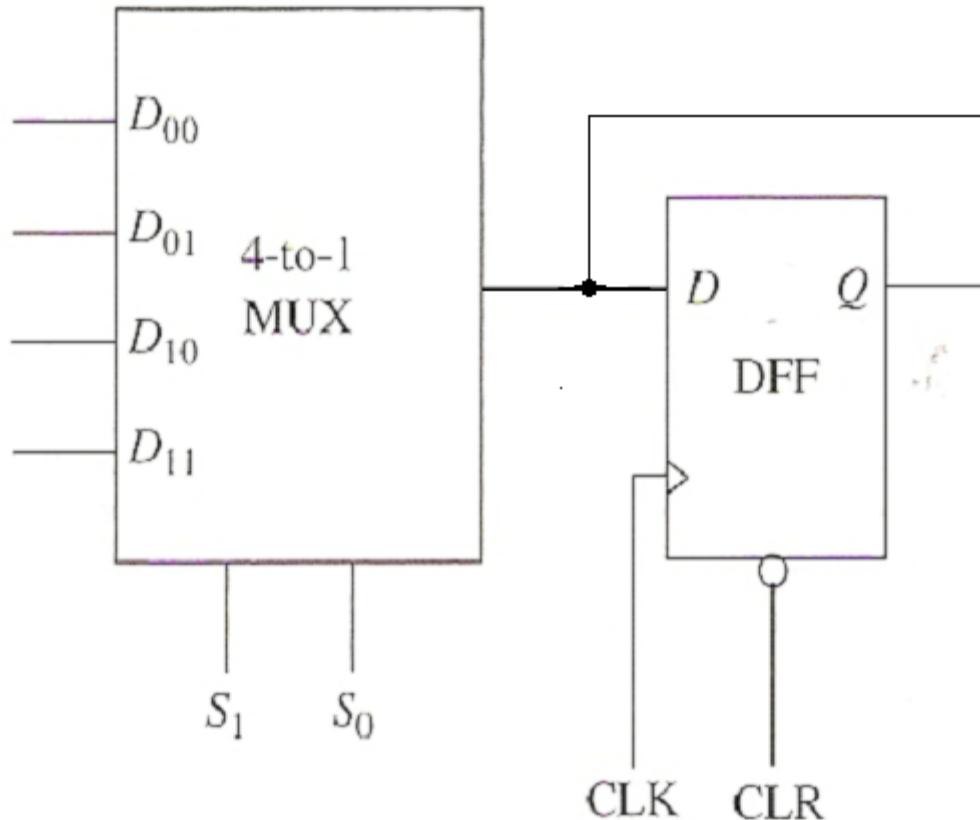
Contents of LUT4

IntelFPGA Stratix Logic Element -- LE



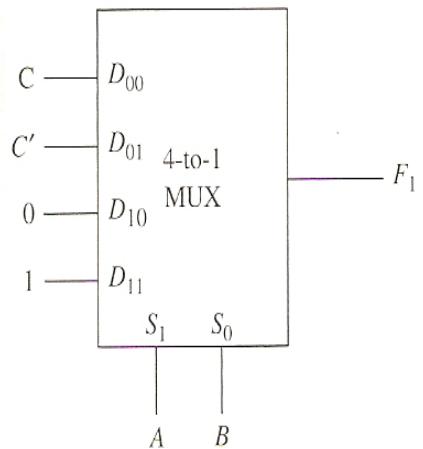
Multiplexer Based FPGAs

$$F_1 = A'B'C + A'BC' + AB$$

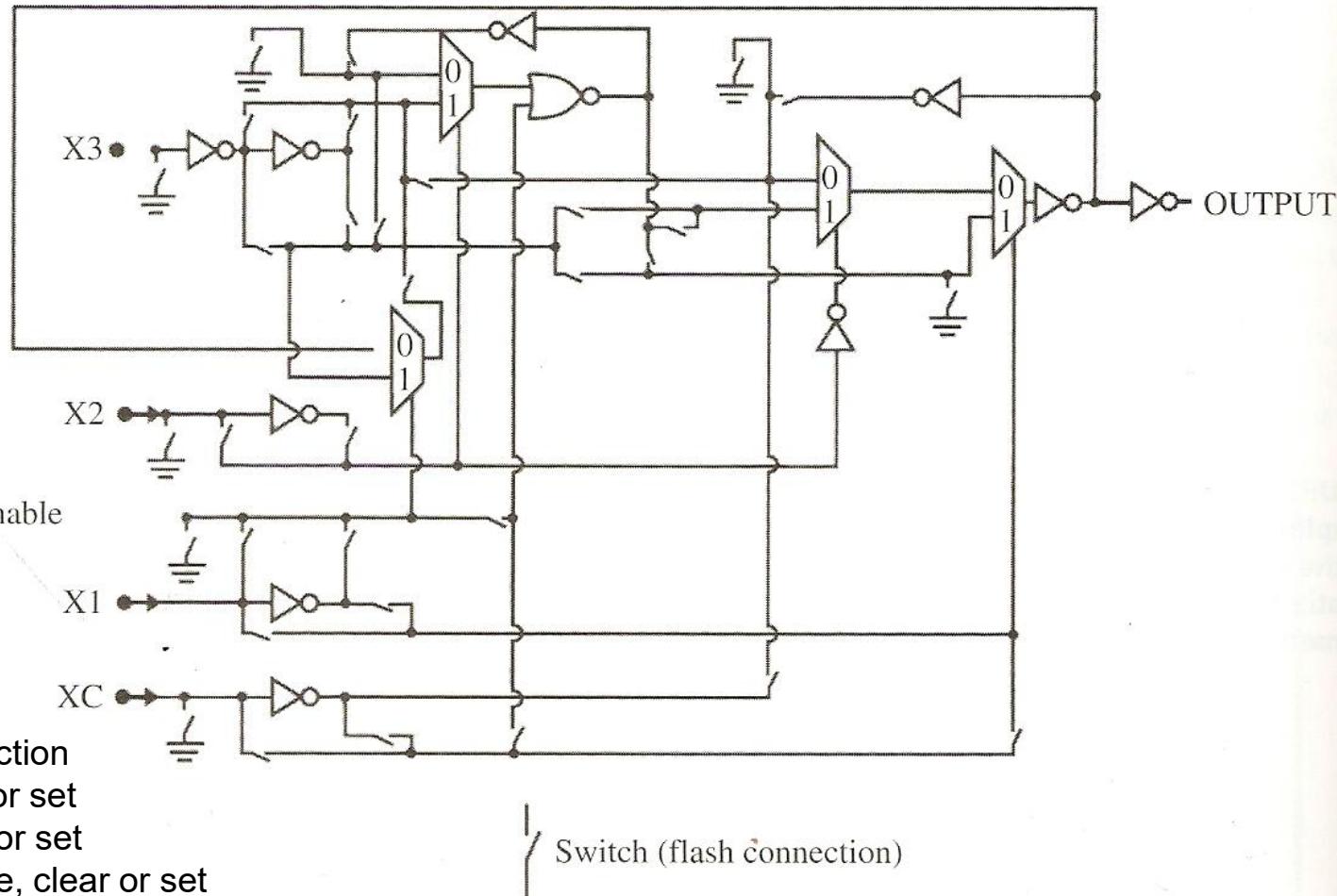


A	B	C	F_1
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

A	B	C	F	Mux Input in Terms of {0, 1, C, C'}
0	0	0	0	{ } C
0	0	1	1	{ } C'
0	1	0	1	{ } 0
0	1	1	0	{ } 1
1	0	0	0	
1	0	1	0	
1	1	0	1	
1	1	1	1	



Microsemi Fusion & ProAS1C Logic Block



Field Programmable Gate Array Capacities

Vendor	FPGA Product	Capacity (Approx) in Gates/LUTs
Xilinx	Kintex 7	41,000 to 298,600 LUTs
	Artix 7	63,400 to 134,600 LUTs
	Virtex-6	46,560 to 474,240 LUTs
	Spartan-3	50K to 5M
	Virtex-5	19,200 to 207,360 LUTs
IntelFPGA (Altera)	Arria V	76,800 to 516,096 LUTs
	Arria II	45,125 to 256,500 LUTs
	ACEX 1K	56K to 257K
	APEX II	1.9M to 5.25M
	FLEX 10K	10K to 50K
Lattice	Stratix/Stratix II	10,570 to 132,540 logic elements
	LatticeECP2	6K to 68K LUTs
	Lattice SC	15.2K to 115.2K LUTs
	ispXPGA	139K to 1.25M
Microsemi	MachXO	256 to 2280 LUTs
	LatticeECP	6.1K to 32.8K LUTs
	Fusion	90K to 1.5M system gates
	IGLOO	15K to 3M system gates
	Axcelerator	125K To 2M
Quick Logic	eX	3K to 12K
	ProASIC3	30K to 3M
	MX	3K to 54K
	Eclipse/EclipsePlus	248K to 662K
Atmel	Quick RAM	45K to 176K
	pASIC 3	5K to 75K
	AT40K	5K to 40K
	AT40KAL	5K to 50K

Architecture, Technology, and Logic Block Type

Company	Device Names	General Architecture	Logic Block Type	Programming Technology
Microsemi	IGLOO	Sea of Tiles	LUT	Flash
	ProASIC/ ProASIC3/ ProASIC ^{plus}	Sea of Tiles	Multiplexers & Basic Gates	Flash, SRAM
	SX/SXA/eX/MX	Sea of Modules	Multiplexers & Basic Gates	Antifuse
	Axcelerator	Sea of Modules	Multiplexers & Basic Gates	SRAM
	Fusion	Sea of Tiles	Multiplexers & Basic Gates	Flash, SRAM
Xilinx	Kintex	Symmetrical Array	LUT	SRAM
	Virtex	Symmetrical Array	LUT	SRAM
	Spartan	Symmetrical Array	LUT	SRAM

Architecture, Technology, and Logic Block Type

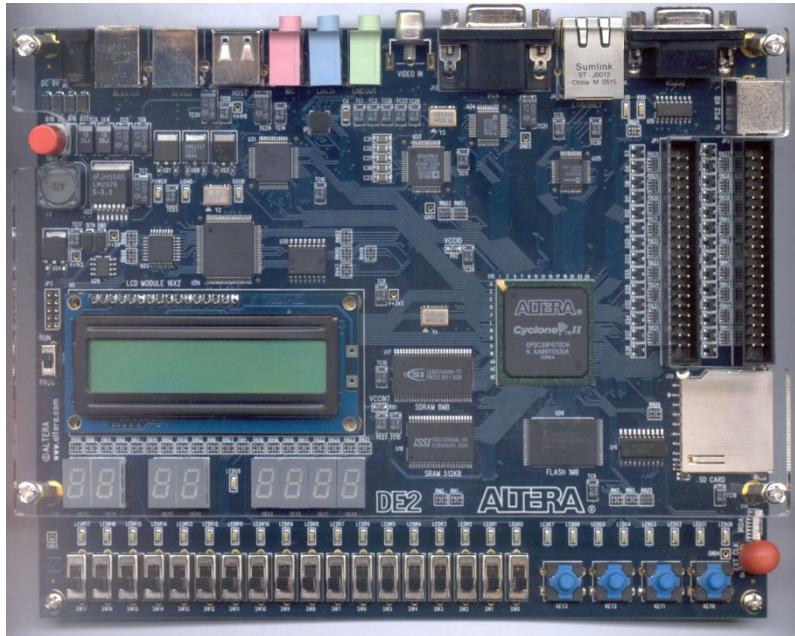
Company	Device Names	General Architecture	Logic Block Type	Programming Technology
Atmel	AT40KAL	Cell-Based	Multiplexers & Basic Gates	SRAM
QuickLogic	Eclipse II PolarPro	Flexible Clock Cell-Based	LUT LUT	SRAM SRAM
Intel FPGA (Altera)	Cyclone IVE	Two-Dimensional Row and Column Based	LUT	SRAM
	Stratix II	Two-Dimensional Row and Column Based	LUT	SRAM
	APEX II	Row and Column, But Hierarchical Interconnect	LUT	SRAM

CPE 322

Digital Hardware Design Fundamentals

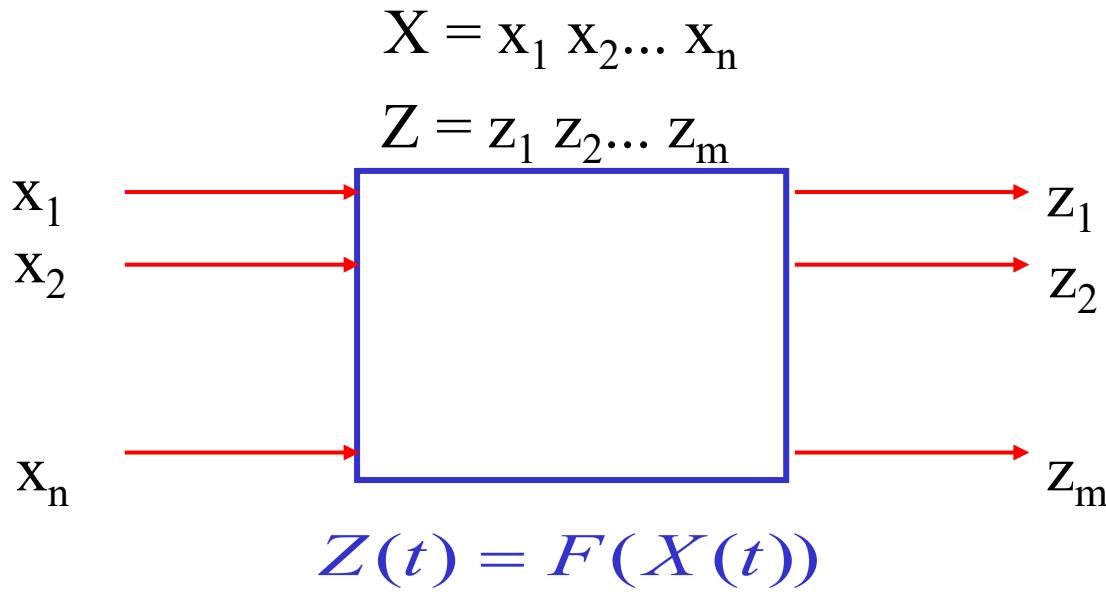
Electrical and Computer Engineering
University of Alabama in Huntsville

Hazard Detection and Prevention in Combinational Logic



Combinational Networks

- Has no memory (state)
 - Present is only a function of the current input, past history is not important



No memory (no feedback) but there often propagation delays in real-world networks!

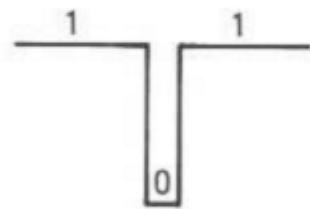
Hazards in Combinational Networks

- Occur when different paths from input to output have different propagation delays
- Static 1-hazard
 - a network output momentarily go to the 0 when it should remain a constant 1
- Static 0-hazard
 - a network output momentarily go to the 1 when it should remain a constant 0
- Dynamic hazard
 - if an output change three or more times, when the output is supposed to change from 0 to 1 (1 to 0)

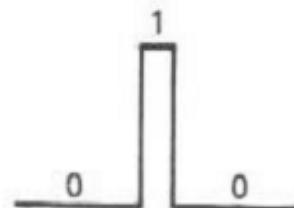
Hazards in Combinational Networks

- When an input to a combinational network changes, unwanted switching transients may appear at the network output.
- These transients occur because different paths through the network from input to output may have different propagation delays.

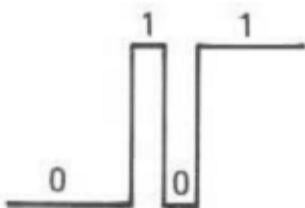
Types of Hazards:



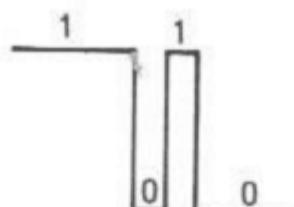
(a) static 1-hazard



(b) static 0-hazard



(c) dynamic hazards



Hazards in Combinational Networks (CNs)

- We only consider hazards which occur when a single input variable changes
- Analysis begins by determining the *Transient Output Function*, F^t , which represents the behavior of the network under transient conditions.

The *Transient Output Function*, F^t , is determined in the same way as ordinary (steady-state) output Functions except each variable and its complement are treated as independent variables because under transient conditions these variables may assume the same values.

Laws and Theorems of Boolean Algebra

Boolean manipulation of F^t involves the use of theorems that do not involve a variable and its complement on the same side of the expression.

Operations with 0 and 1:

$$X + 0 = X \quad (1-5)$$

$$X + 1 = 1 \quad (1-6)$$

$$X \cdot 1 = X \quad (1-5D)$$

$$X \cdot 0 = 0 \quad (1-6D)$$

Idempotent laws:

$$X + X = X \quad (1-7)$$

$$X \cdot X = X \quad (1-7D)$$

~~Involution law:~~

$$(X')' = X \quad (1-8)$$

~~Laws of complementarity~~

$$X + X' = 1 \quad (1-9)$$

$$X \cdot X' = 0 \quad (1-9D)$$

Commutative laws:

$$X + Y = Y + X \quad (1-10)$$

$$XY = YX \quad (1-10D)$$

Associative laws:

$$\begin{aligned} (X + Y) + Z &= X + (Y + Z) \\ &= X + Y + Z \end{aligned} \quad (1-11)$$

$$(XY)Z = X(YZ) = XYZ \quad (1-11D)$$

Distributive laws:

$$X(Y + Z) = XY + XZ \quad (1-12)$$

$$X + YZ = (X + Y)(X + Z) \quad (1-12D)$$

Laws and Theorems of Boolean Algebra

Simplification theorems:

$$\cancel{XY + XY' = X} \quad (1-13)$$

$$X + XY = X \quad (1-14)$$

$$\cancel{(X + Y')Y = XY} \quad (1-15)$$

$$\cancel{(X + Y)(X + Y')} = X \quad (1-13D)$$

$$X(X + Y) = X \quad (1-14D)$$

$$\cancel{XY' + Y = X + Y} \quad (1-15D)$$

DeMorgan's laws:

$$(X + Y + Z + \dots)' = X'Y'Z' \dots \quad (1-16) \quad (XYZ\dots)' = X' + Y' + Z' + \dots \quad (1-16D)$$

$$[f(X_1, X_2, \dots, X_n, 0, 1, +, \cdot)]' = f(X_1', X_2', \dots, X_n', 1, 0, \cdot, +) \quad (1-17)$$

Duality:

$$(X + Y + Z + \dots)^D = XYZ\dots \quad (1-18) \quad (XYZ\dots)^D = X + Y + Z + \dots \quad (1-18D)$$

$$[f(X_1, X_2, \dots, X_n, 0, 1, +, \cdot)]^D = f(X_1, X_2, \dots, X_n, 1, 0, \cdot, +) \quad (1-19)$$

Theorem for multiplying out and factoring:

$$\cancel{(X + Y)(X' + Z) = XZ + X'Y} \quad (1-20)$$

$$\cancel{XY + X'Z = (X + Z)(X' + Y)} \quad (1-20D)$$

Consensus theorem:

$$\cancel{XY + YZ + X'Z = XY + X'Z} \quad (1-21)$$

$$\cancel{\begin{aligned} &(X + Y)(Y + Z)(X' + Z) \\ &= (X + Y)(X' + Z) \end{aligned}} \quad (1-21D)$$

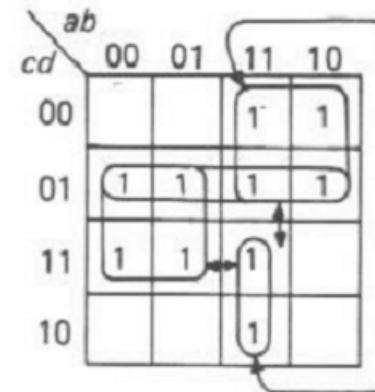
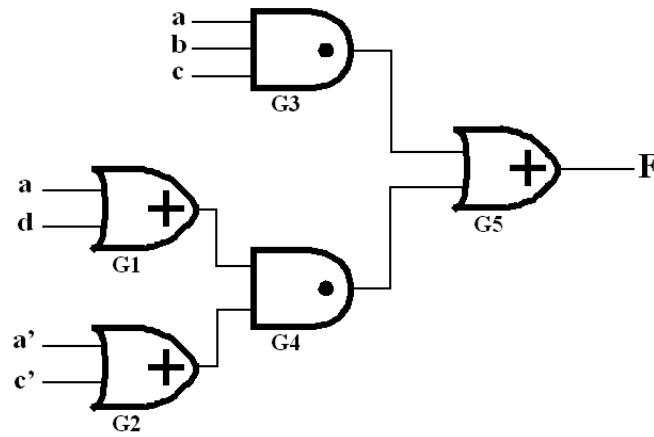
Boolean manipulation of F^t involves the use of theorems that do not involve a variable and its complement on the same side of the expression.

Detection of Static 1 Hazards

- Each product term of F^t is called a 1-term.
- The 1-terms of F^t are plotted on a Karnaugh map
- If two 1's in adjacent squares on the map of F^t are covered by the same 1-term, changing the input to the network between the corresponding two input states cannot cause a hazard.
- If two 1's in adjacent squares on the map are not covered by a single 1-term, a hazard is present.

Detection of Static 1 Hazards

$$F^t = abc + (a+d)(a' + c') = abc + aa' + ac' + a'd + c'd$$
$$X(Y + Z) = XY + XZ \quad (1-12)$$



- 1-Hazard exists for case where **a** changes (from 0 to 1 or 1 to 0) and **b=1, c=1, d=1**.
- 1-Hazard exists for case where **c** changes (from 0 to 1 or 1 to 0) and **a=1, b=1, d=1**.
- 1-Hazard exists for case where **c** changes (from 0 to 1 or 1 to 0) and **a=1, b=1, d=0**.

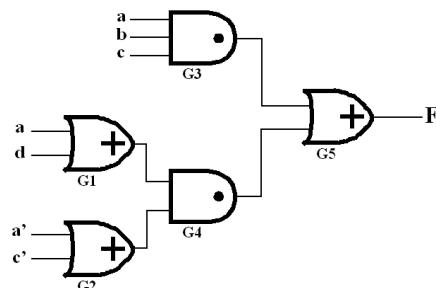
Detection of Static 0 Hazards

- Each sum term of F^t is called a 0-term.
- The 0-terms of F^t are plotted on a Karnaugh map
- If two 0's in adjacent squares on the map of F^t are covered by the same 0-term, changing the input to the network between the corresponding two input states cannot cause a hazard.
- If two 0's in adjacent squares on the map are not covered by a single 0-term, a hazard is present.

Detection of Static 0 Hazards

$$F^t = abc + (a+d)(a' + c') = [(a+d)(a'+c')+a][(a+d)(a'+c')+bc]$$

$$X + YZ = (X+Y)(X+Z) \quad (1-12D)$$



$$[(a+d+a)(a'+c'+a)][((a+d)(a'+c')+b)((a+d)(a'+c')+c)]$$

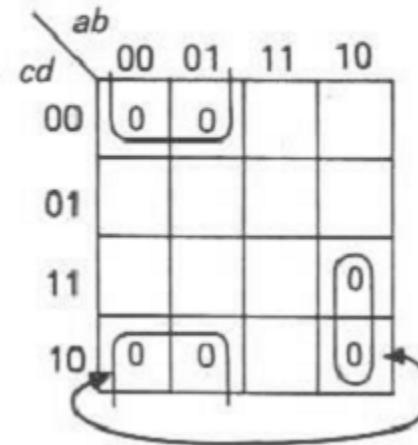
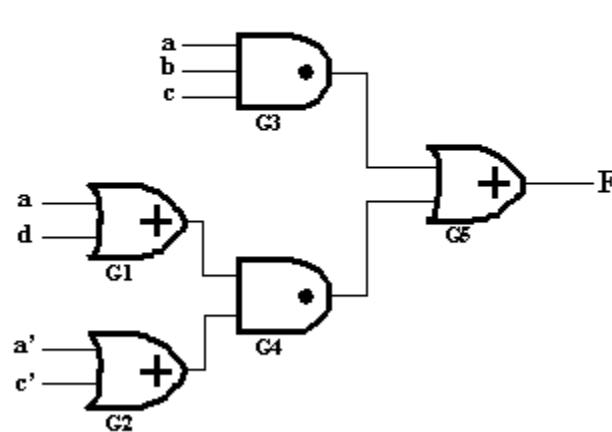
$$X + YZ = (X+Y)(X+Z) \quad (1-12D)$$

$$(a+d)(a'+a+c')(a+d+b)(a'+c'+b)(a+c+d)(a'+c'+c)$$

$$X(X+Y) = X \quad (1-14D)$$

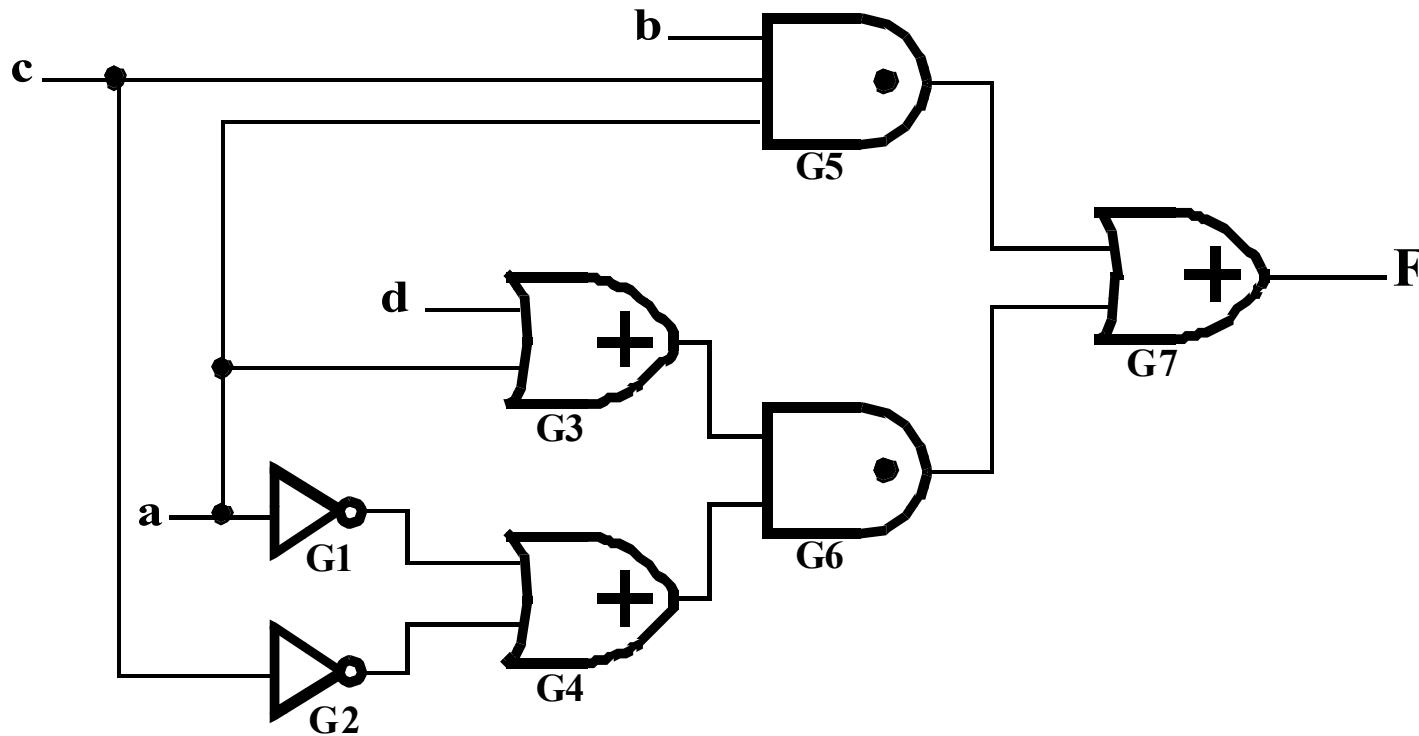
Detection of Static 0 Hazards

$$F^t = abc + (a+d)(a' + c') = (a+d)(a+a'+c')(b+a'+c')(c+a'+c')$$



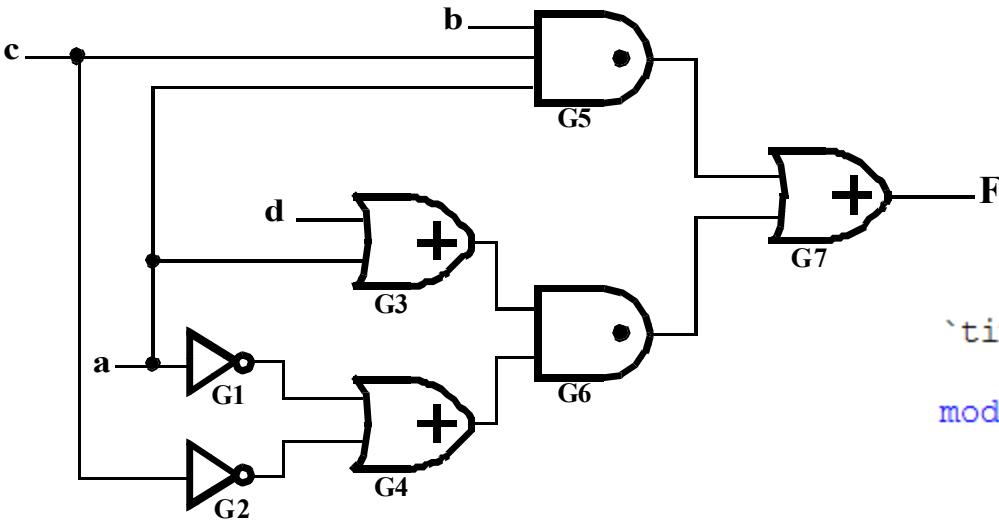
- 0-Hazard exists for case where **a** changes (from 0 to 1 or 1 to 0) and **b=0, c=1, d=0**.

Using Static Timing to Observe the Effect of Hazards



Assuming 1 ns rise/fall times for each gate
(AND, OR, and NOT gates)

Using Static Timing to Observe the Effect of Hazards



Assuming 1 ns rise/fall times for each gate
(AND, OR, and NOT gates)

```
'timescale 1ns/100ps

module example (input a,b,c,d, output F);

wire a_n,c_n,n1,n2,n3,n4;

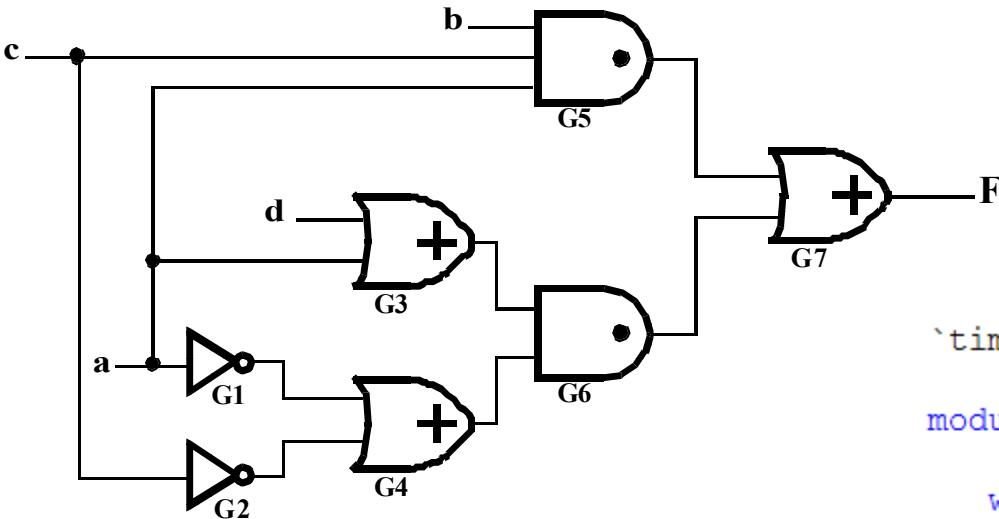
not #(1) G1(a_n,a);
not #(1) G2(c_n,c);

or  #(1) G3(n1,d,a);
or  #(1) G4(n2,a_n,c_n);

and # (1) G5(n3,b,c,a);
and # (1) G6(n4,n1,n2);
or   #(1) G7(F,n3,n4);

endmodule
```

Using Static Timing to Observe the Effect of Hazards



Assuming 1 ns rise/fall times for each gate
(AND, OR, and NOT gates)

```
'timescale 1ns/100ps

module example (input a,b,c,d, output F);

wire a_n,c_n,n1,n2,n3,n4;

assign #1 a_n = ~a;
assign #1 c_n = ~c;

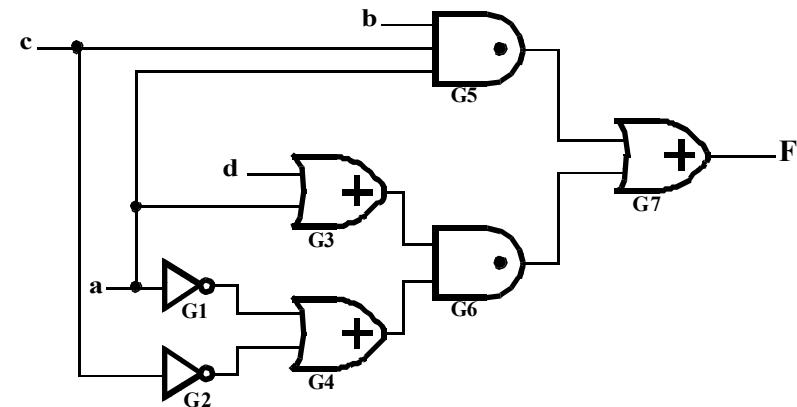
assign #1 n1 = d | a;
assign #1 n2 = a_n | c_n;

assign #1 n3 = b & c & a;
assign #1 n4 = n1 & n2;
assign #1 F = n3 | n4;

endmodule
```

Driving the input to Detect Hazards

- 1-Hazard exists for case where **c** changes (from 0 to 1 or 1 to 0) and **a=1, b=1, d=1**.
- 1-Hazard exists for case where **c** changes (from 0 to 1 or 1 to 0) and **a=1, b=1, d=0**.
- 1-Hazard exists for case where **a** changes (from 0 to 1 or 1 to 0) and **b=1, c=1, d=1**.



Assuming 1 ns rise/fall times for each gate
(AND, OR, and NOT gates)

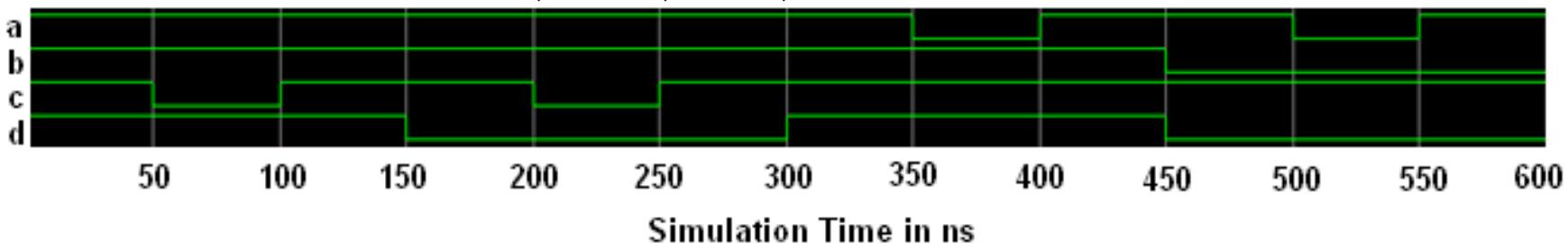
- 0-Hazard exists for case where **a** changes (from 0 to 1 or 1 to 0) and **b=0, c=1, d=0**.

force a 1 0 ns, 0 350 ns, 1 400 ns, 0 500 ns, 1 550 ns

force b 1 0 ns, 0 450 ns

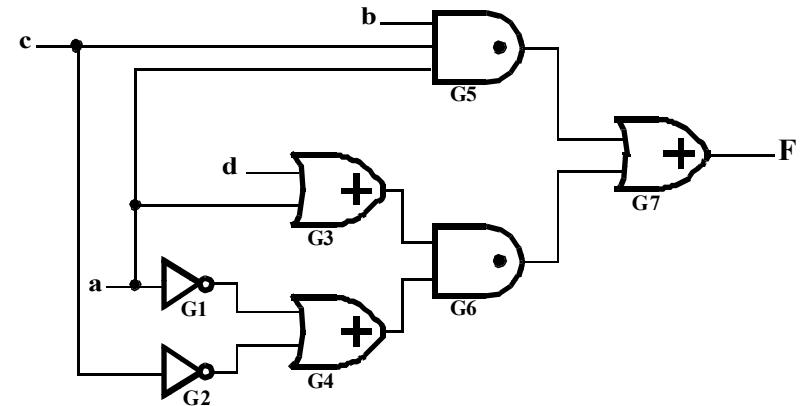
force c 1 0 ns, 0 50 ns, 1 100 ns, 0 200 ns, 1 250 ns

force d 1 0 ns, 0 150 ns, 1 300 ns, 0 450 ns

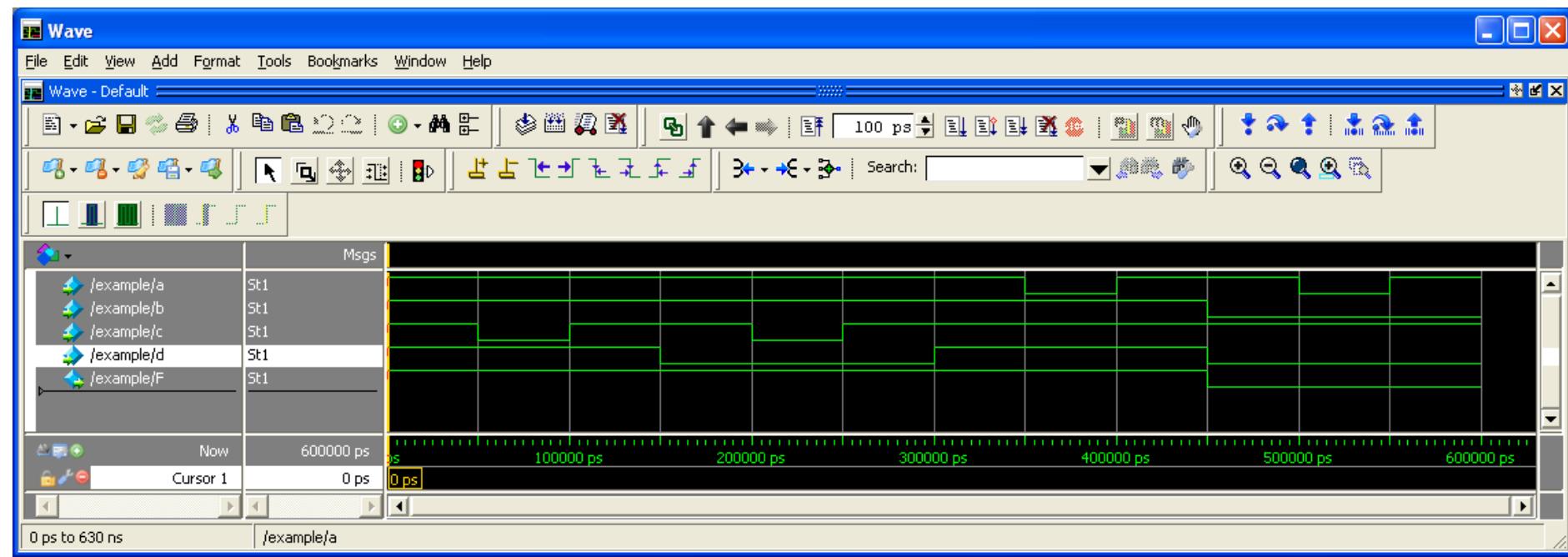


RTL Simulation (zero delay)

```
'timescale 1ns/100ps
a=1; b=1; c=1; d=1;
#50 c=0; #50 c=1;
#50 d=0; #50 c=0; #50 c=1; #50 d=1;
#50 a=0; #50 a=1;
#50 b=0; d=0;
#50 a=0; #50 a=1;
run 600 ns
```



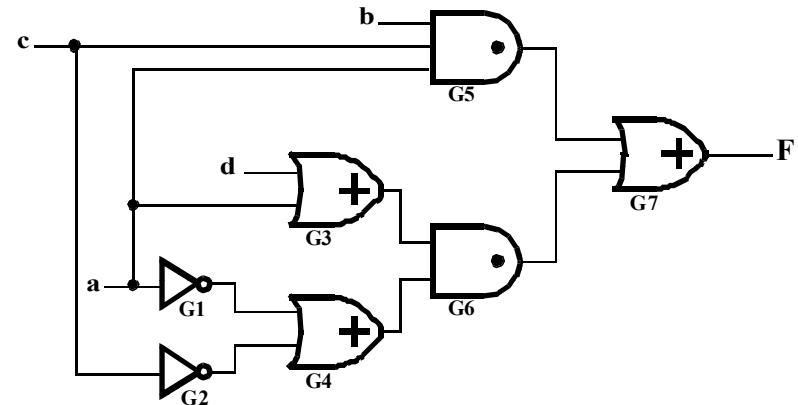
Assuming 1 ns rise/fall times for each gate
(AND, OR, and NOT gates)



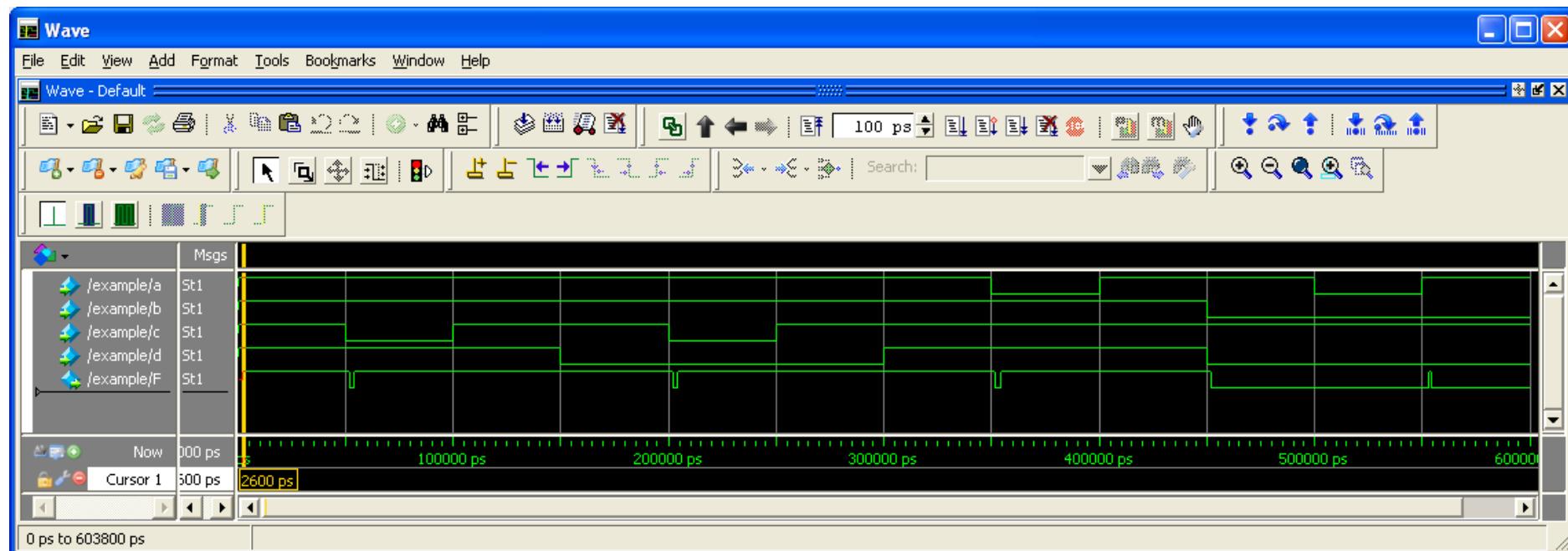
Static Timing Simulation

(1ns Delay each gate)

```
'timescale 1ns/100ps  
a=1; b=1; c=1; d=1;  
#50 c=0; #50 c=1;  
#50 d=0; #50 c=0; #50 c=1; #50 d=1;  
#50 a=0; #50 a=1;  
#50 b=0; d=0;  
#50 a=0; #50 a=1;  
run 600 ns
```



Assuming 1 ns rise/fall times for each gate
(AND, OR, and NOT gates)



Static Timing Simulation

(1ns Delay each gate)

- In this case we were able to see the effect of all four hazards!
 - Not always the case – hazards may cause a glitch but whether or not one occurs depends upon the actual timing values.
- There is automated help in ModelSim™ to allow for hazard checking.

Design of Hazard Free Networks

To design a network that is free of static and dynamic hazards, the following procedure may be used:

1. Find a sum-of-products expression (F') for the output in which every pair of adjacent 1s is covered by a 1-term. (The sum of all prime implicants will always satisfy this condition.) A two-level AND-OR network based on this F' will be free of 1-, 0-, and dynamic hazards.
2. If a different form of network is desired, manipulate F' to the desired form by simple factoring, DeMorgan's laws, etc. Treat each x_i and x'_i as independent variables to prevent introduction of hazards.

Alternatively, you can start with a product-of-sums expression in which every pair of adjacent 0s is covered by a 0-term.

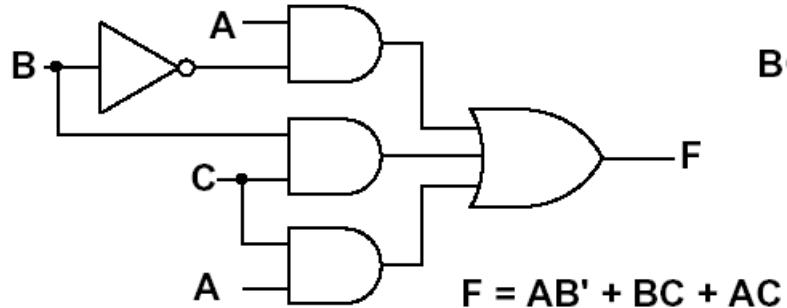
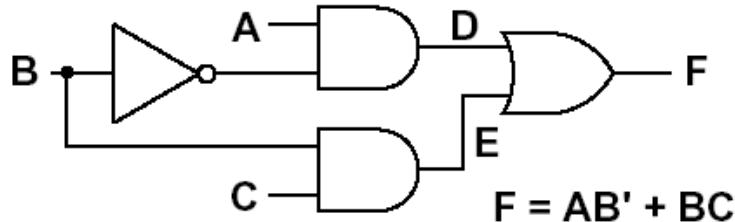
Hazards in Combinational Circuits

AB	00	01	11	10
C	0			1
	1	1	1	1

$$f = AB' + BC$$

AB	00	01	11	10
C	0			1
	1	1	1	1

$$f = AB' + BC + AC$$



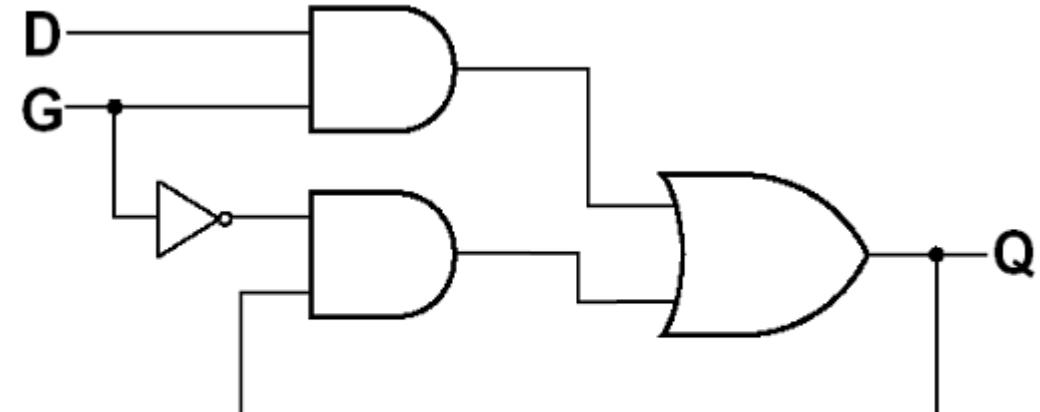
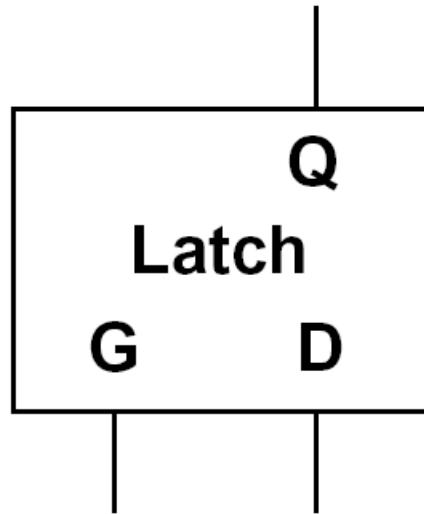
To avoid hazards:
every pair of adjacent 1s should be covered by a 1-term

Hazards in Combinational Circuits

Why do we care about hazards?

- Combinational networks
 - don't care – the network will function correctly
- Synchronous sequential networks
 - don't care - the input signals must be stable within setup and hold time of flip-flops
- Asynchronous sequential networks
 - hazards can cause the network to enter an incorrect state
 - circuitry that generates the next-state variables must be hazard-free
- Dynamic Power consumption is proportional to the number of transitions

Transparent D Latch with Hazard

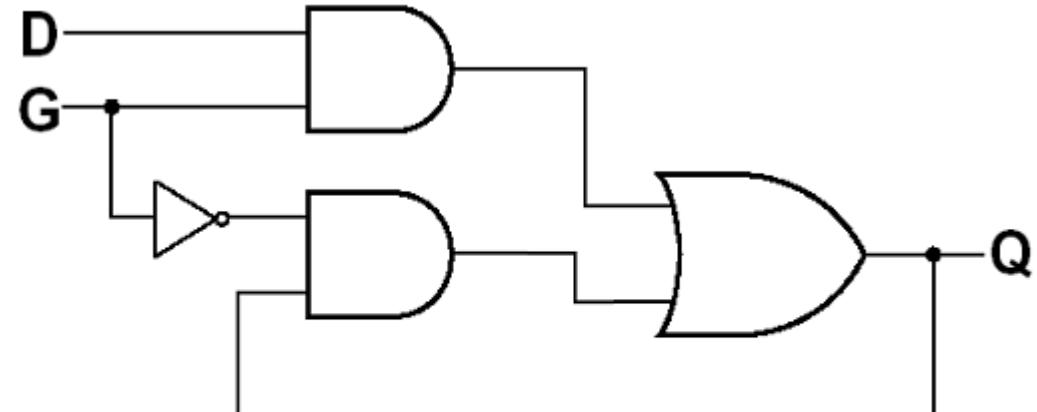
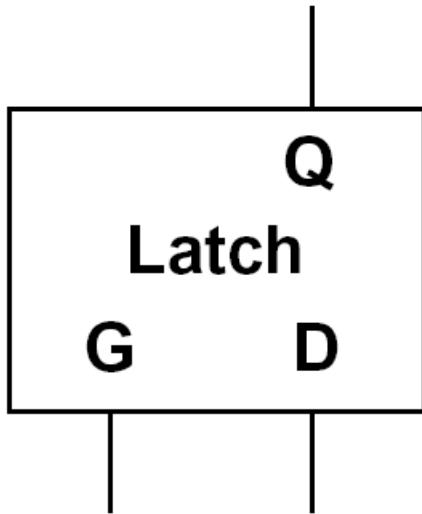


A state transition table for the transparent D latch. The columns are labeled Q and Q+. The rows are labeled 00, 01, 11, and 10. The first column (Q) shows the current state, and the second column (Q+) shows the next state. The table is partially filled with blue lines.

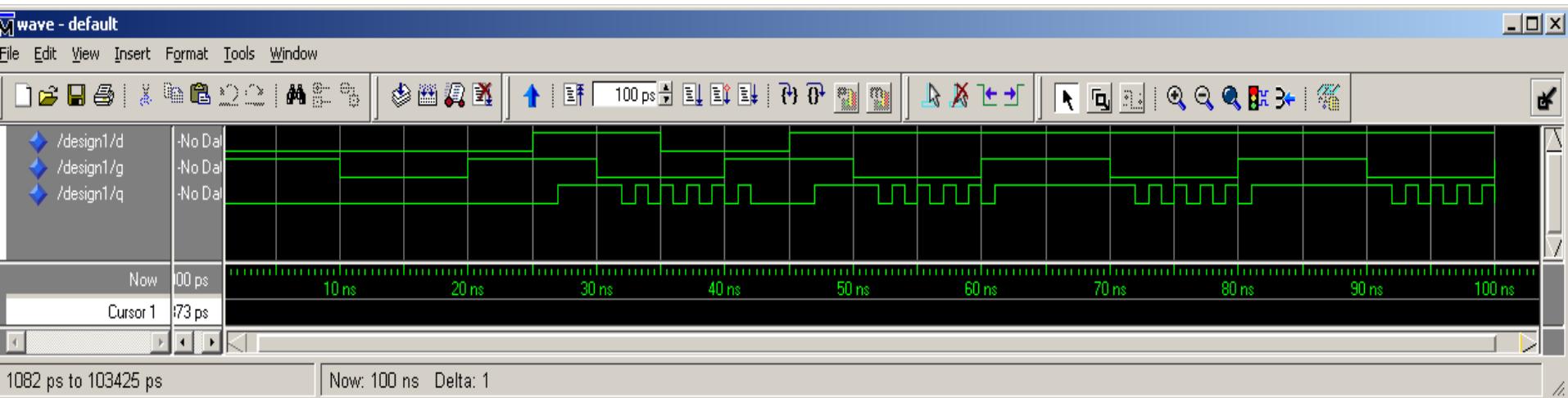
	Q	Q+
00	0	
01	0	
11	0	
10	0	

$$Q^+ = DG + G'Q$$

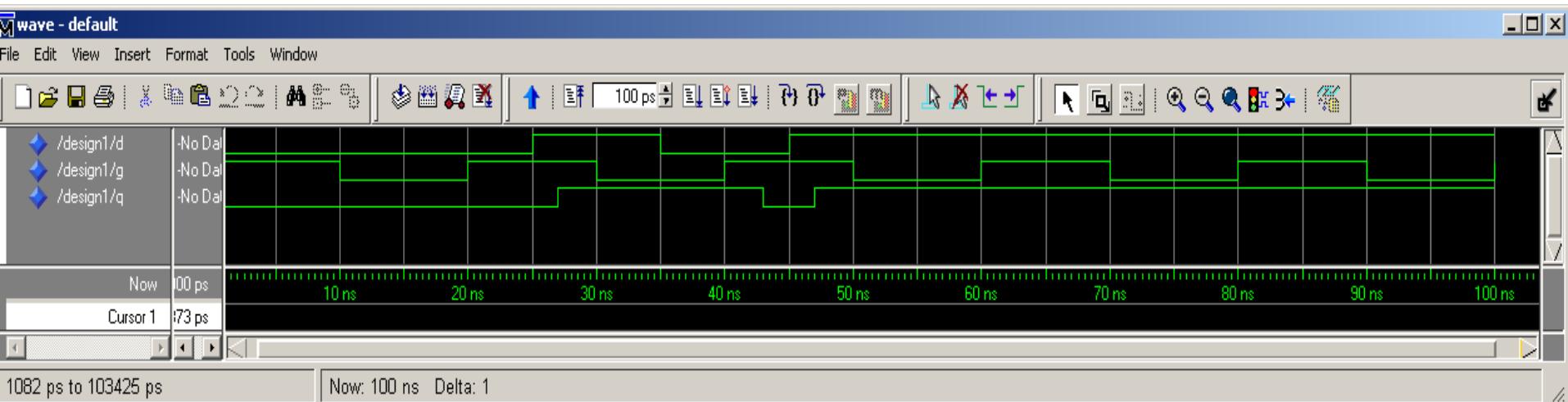
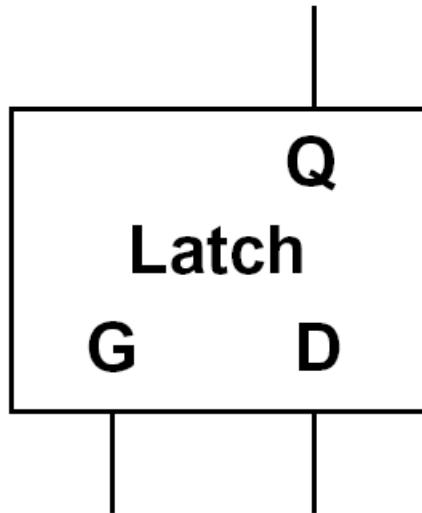
Transparent D Latch with Hazard (1 ns Delay all gates)



$$Q^+ = DG + G'Q$$



Transparent D Latch (hazard removed) (1 ns Delay all gates)



Hazards In a FPGA-based Design

- Hazards can appear in any combinational logic section of designs that are implemented in reconfigurable logic devices such as your Altera Cyclone II device on the DE-2 board
- In such devices all combinational logic is implemented using an interconnection of 1, 2, and 4 input ROM type lookup tables.
- The wiring delay between lookup table and the uneven propagation delay of the lookup tables themselves can result in different paths for signals taken from input to output that have different propagation delays (just as in the case of multi-gate-level logic).

Hazards In a FPGA-based Design

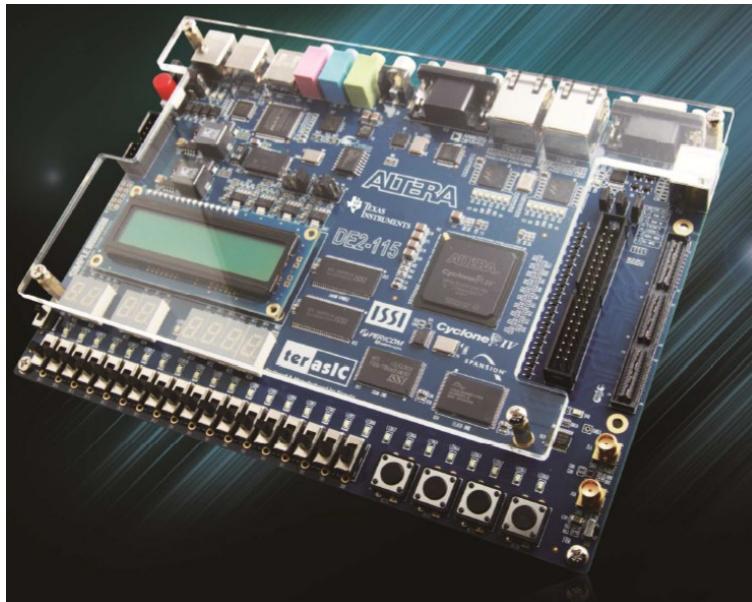
- Analysis of hazards in programmable logic is difficult and beyond the scope of this class.
- Hazards are not a problem, though, if there is time to reach a steady state before the final value is stored in a flip-flop (synchronous design techniques).
- Hazards though can be very serious issue if we use combinational control logic to gate the clock!
 - (we will discuss this a bit later)

CPE 322 Fundamentals of Hardware Design

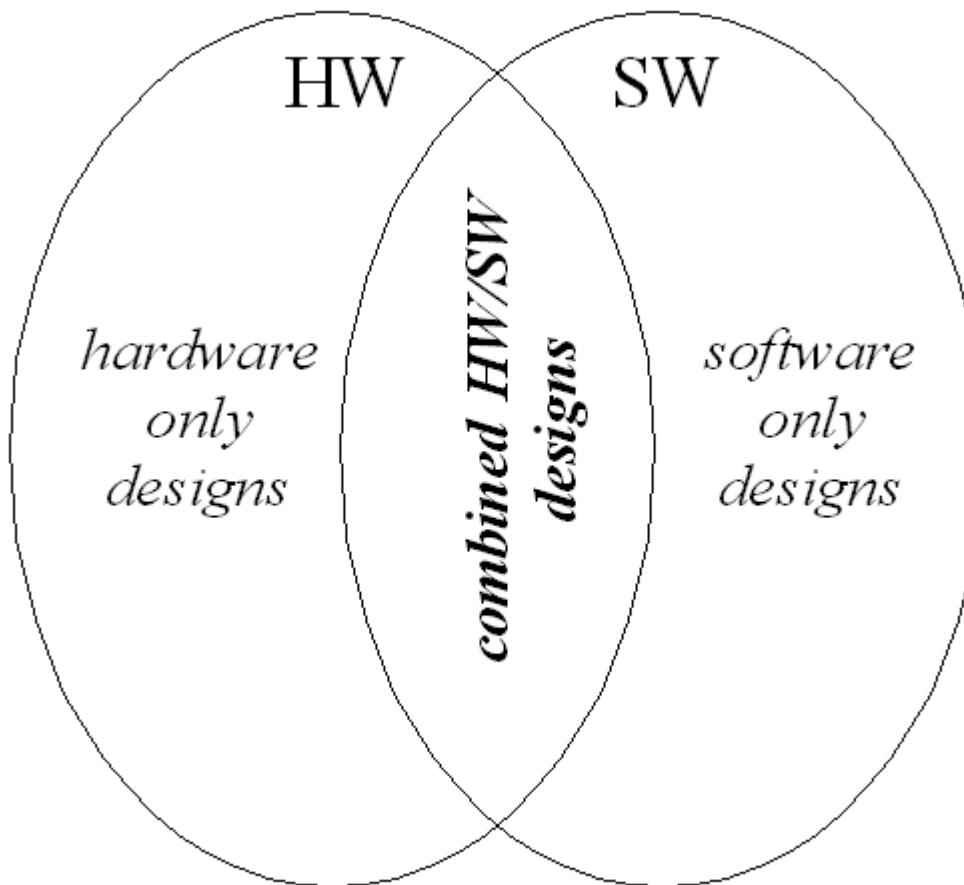
Electrical and Computer Engineering
University of Alabama in Huntsville

Big Picture

General Hardware/Software Design Trade-offs and
HW/SW Co-Design Paradigm



Hardware/Software Design Space Continuum



Hardware/Software Trade-offs

- Performance of hardware for a given function can be much faster than a software implementation because
 - There is little or no instruction fetch overhead like there is in a traditional instruction set processor
 - It can make use of the available fine grain concurrency and execute many operations in parallel
- Software is better suited for irregularly-structured control structures
- Traditionally software was considered to be more flexible and field upgradable when compared to hardware
 - FPGA's blurres this distinction though

Hardware/Software Trade-offs

- Cost of implementing a function in hardware is usually much greater than implementing the same function in software
 - Requires additional chip area in an Application Specific Integrated Circuit or board area in a PC board or reconfigurable logic area in an FPGA.
 - Usually takes more time to create a hardware design than a software one
- Both hardware and software resources are limited but hardware resources tend to be more tightly constrained and should be reserved for elements that have the greatest effect on the performance (or possibly other important constraints such as reliability)

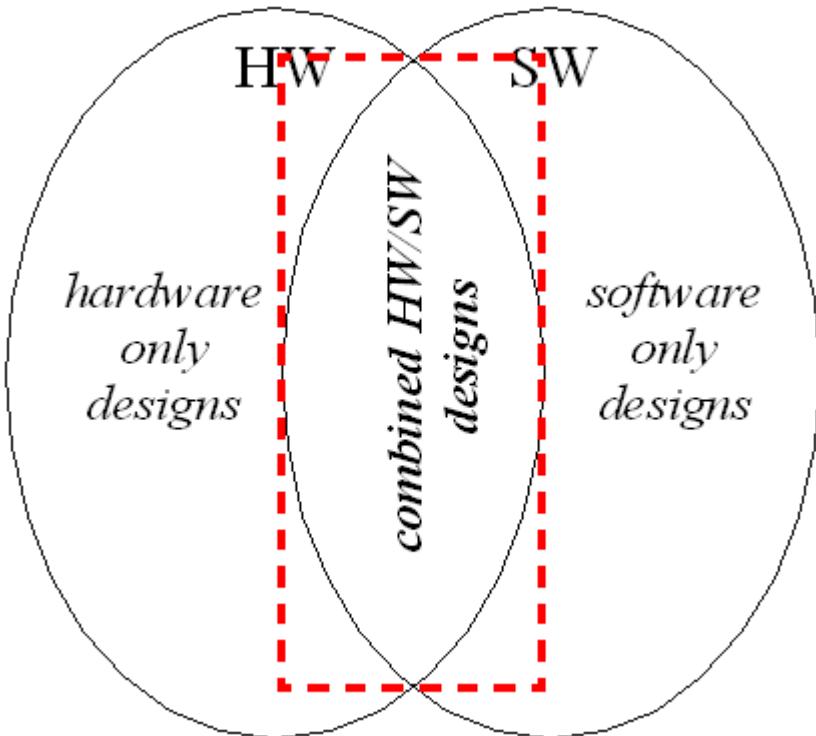
Hardware/Software Trade-offs

- Both hardware and software solutions allow for reduced energy consumption
 - Dynamic and Static Voltage Frequency Scaling, DVFS
 - Dynamic power management
 - Internal Power gating with Data Retention modes
 - Clock Gating (single and multi-level)
- Specific hardware energy consumption strategies can be applied at a finer grain which can result in extremely low power consumption for custom hardware.
- These features are also often dynamically controllable by software at a courser grain using commercially available hardware for much less cost.

Differences between ISP Programming and FPGA Reconfiguration

- Instruction Set Processors, ISPs, such as embedded controllers, allow you to change the program or programs that execute in one or more thread control units.
- FPGAs and other reconfigurable hardware devices allow you to reconfigure both the control units and the data paths of the device.
- In both cases this allows one to increase the usability of the devices involved.
- The allowable number of configurations of internal hardware is much smaller for an ISP when compared to an FPGA but the size of the program (reconfiguration information) is also much smaller!

Embedded Systems



- Application-specific systems which contain hardware and software tailored for a particular task
- Generally part of a larger system
- Responds to external inputs and drives external outputs
- Often must respond in real-time
- Must adhere to strict area (footprint), performance, and power consumption constraints

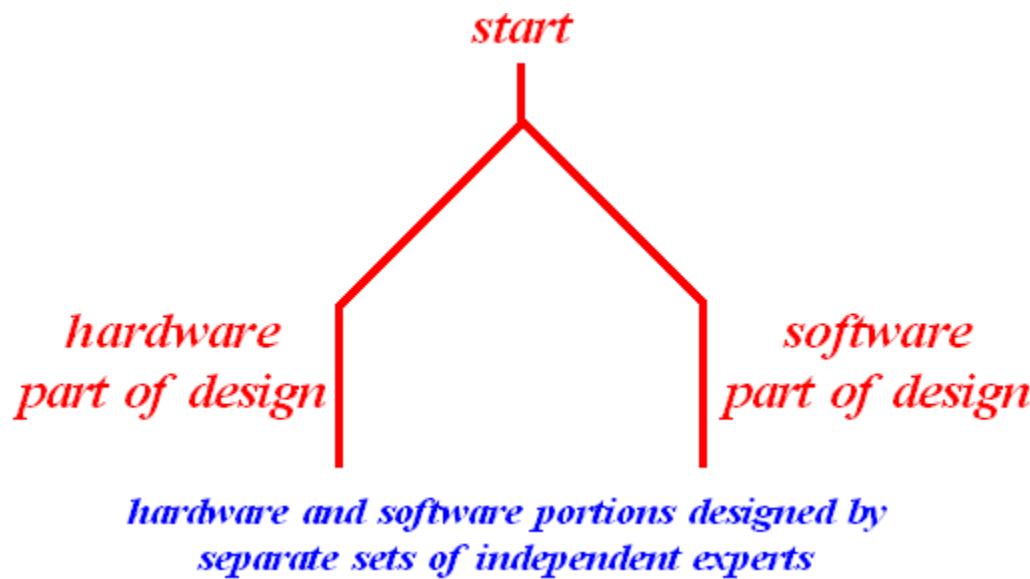
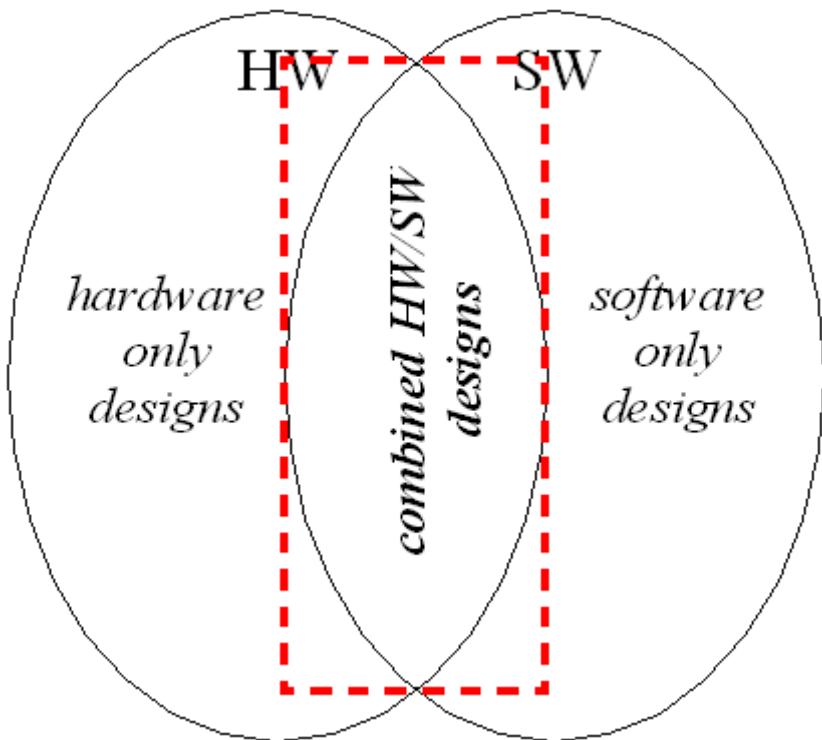
Embedded Systems Examples

- Automotive: Anti-lock breaks, engine control, dynamic ride control, engine diagnostics
- Consumer Electronics: cameras, DVD Controllers, Microwave Ovens, Toasters, Refrigerators, Washers, Smart Card Controllers, RFID systems, Stand Alone GPS systems, TV remote controller, landline telephones, smart scales,
- Industrial Process Controllers: motor control systems , electronic data acquisition and supervisory control, automated laboratory instrument control
- Computer Peripherals: printers, external disk controllers, FAX controllers
- Cell Phones and Peripherals: wireless headsets, wifi bridging devices
- Medical Applications: ECG display and diagnostics, blood cell recorder/ analyzer, patient monitor system
- Network Systems: Routers, network switches, external firewalls

Traditional HW/SW Design Process

- System requirements are developed and then analyzed to determine the “level” of technology required to fulfill these needs.
- Hardware and software teams then independently develop their designs
 - combining of design efforts occur late in the development cycle during the first prototype testing.
- Often leads to a generalized hardware platform being selected and all specialization occurring in the software.

Traditional HW/SW Design Process

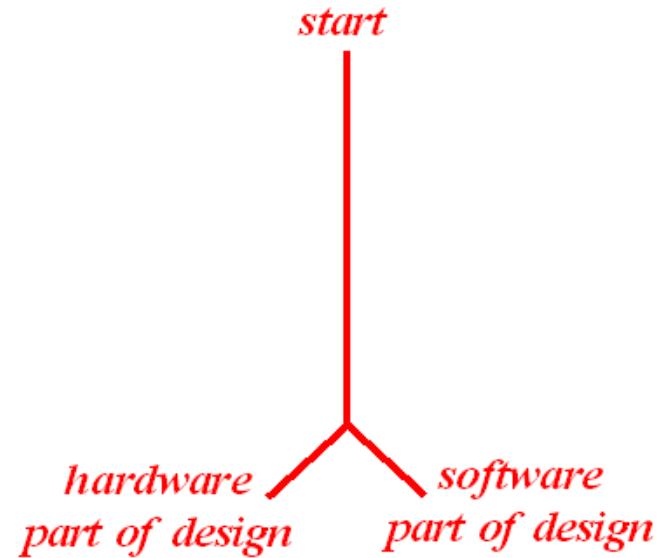
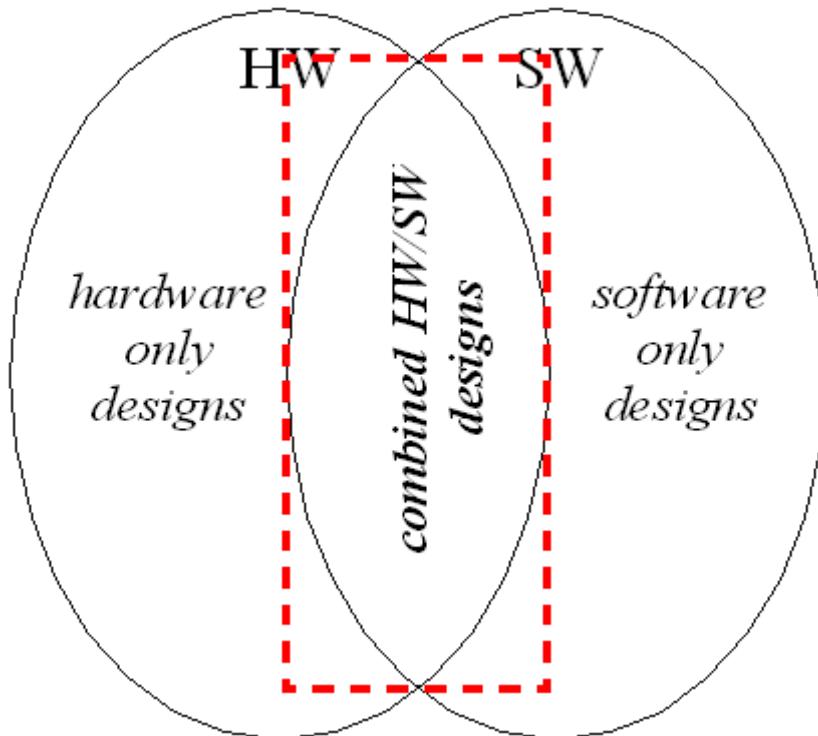


Hardware/Software Co-Design

Meeting system level objectives by exploiting the synergism of hardware and software through their concurrent design.

"Hardware/Software Co-Design", Giovanni De Micheli, and Rajesh K. Gupta, Readings in Hardware/Software Codesign Academic Press, 2002

Hardware/Software Codesign



*Hardware and software portions designed by the same group of experts with cooperation with one another.
Placement of functionality can occur later in the design cycle*

Hardware/Software Co-Design

Meeting system level objectives by exploiting the synergism of hardware and software through their concurrent design.

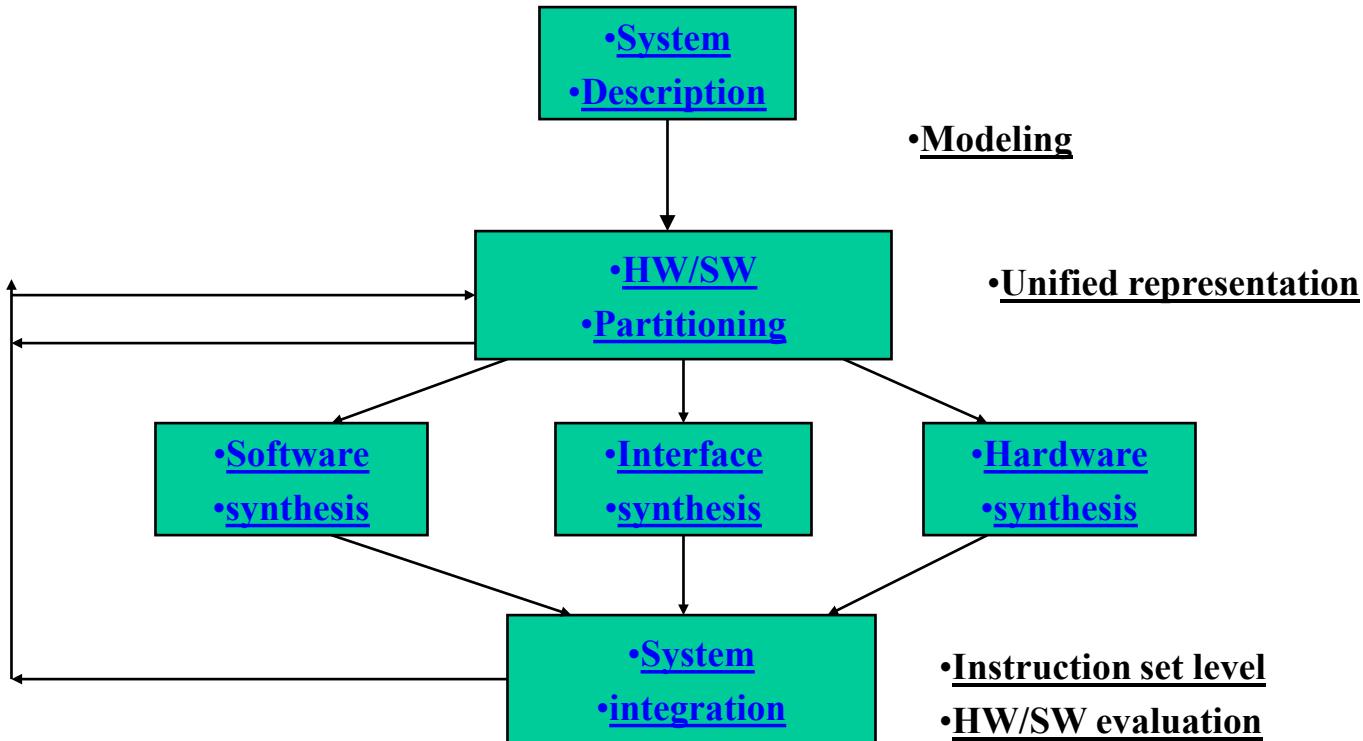
"Hardware/Software Co-Design", Giovanni De Micheli, and Rajesh K. Gupta, Readings in Hardware/Software Codesign Academic Press, 2002

Hardware/Software Co-Design

Design Flow for HW/SW Co-Design

- specification
- modeling
- design space exploration and partitioning
- synthesis and optimization
- validation
- implementation

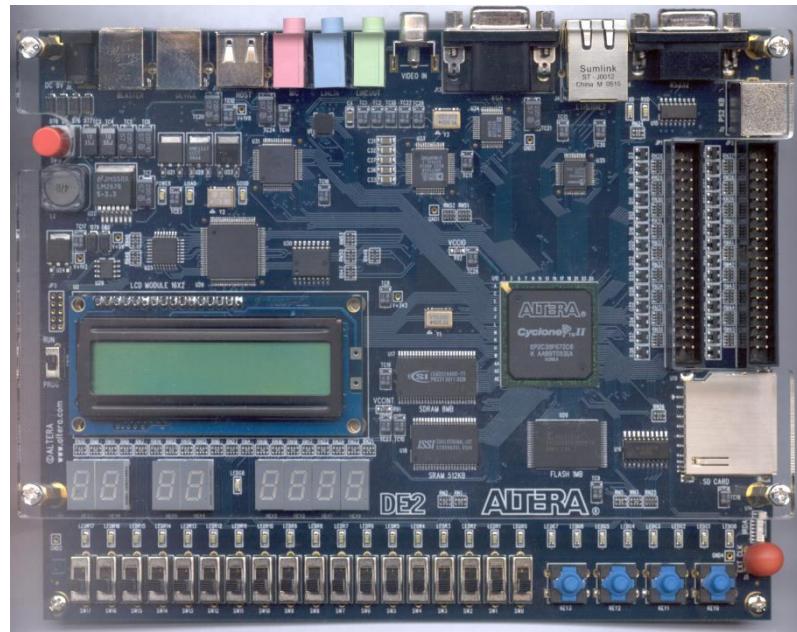
Hardware/Software Co-Design



Digital Hardware Design Fundamentals

Electrical and Computer Engineering UAH

Teaching Philosophy & Combinational Logic Review



My Teaching Philosophy

I believe that my role as an instructor of engineering students is not simply to train the next generation of technologists, but rather to produce truly *educated* engineers who can advance the state-of-the-art in their discipline while continuously adapting to an ever changing technical landscape.

To accomplish this goal, I believe that there should be an appropriate balance between theory and practice -- the ability to view things in a general and abstract manner should be complemented by the hands-on experience gained by solving specific, nontrivial real world engineering problems.

Theory & Practice

Theory



practice may change significantly over one's career but first principles (theory) are time invariant

Practice



- Number Representation
- Boolean Algebra
- Timing Issues
- Modeling Issues
- Testability Issues



Quartus II, Design Entry
Simulation, and FPGA
Circuit Synthesis

ModelSim Design Entry
and Simulation

DE-2 Rapid Prototyping
Environment

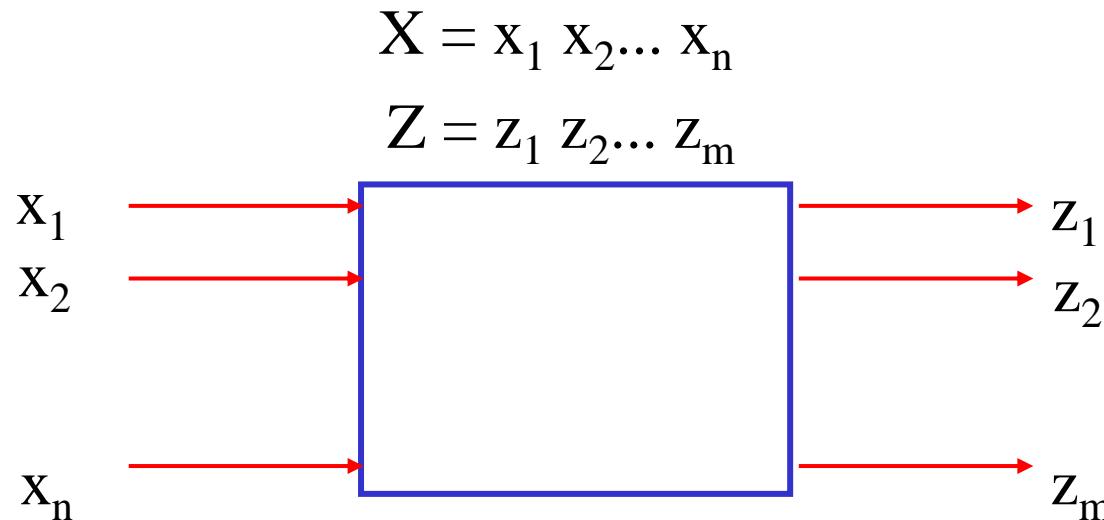
Architectural alternatives
for programmable logic

Course Goals

- Students will understand and appreciate that Verilog is not a *programming language*, but rather is a *hardware description language*.
- Students will understand the differences, advantages, and disadvantages between embedded software and programmable logic
- CPE and EE students with affinity towards hardware will discover and develop skills to enable them to bet their paycheck on these skills

Combinational Logic

- Has no memory =>
present state depends only on the present input



$$Z(t) = F(X(t))$$

Note:

Positive Logic – low voltage corresponds to a logic 0, high voltage to a logic 1

Negative Logic – low voltage corresponds to a logic 1, high voltage to a logic 0

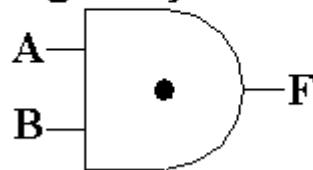
Boolean Algebra

- Basic mathematics used for logic design
 - A ‘first principle’ of ECE
- Terminology:
 - Logic value (two value -- high/low, 1/0)
 - Operator (AND, OR, NOT)
 - Variable (an input or output that can change value)
 - Literal (the appearance of a variable or its complement)
 - Constant (a logic high or low that does not change with time)
 - Boolean Expression (a logical expression that contains one or more Boolean operators, variables, or constants)
 - Note: Boolean Expressions can be converted into logic circuitry

Basic Boolean Logic Gates (operators)

AND

Logic Symbol



Truth Table

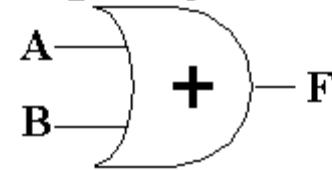
A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

Boolean Equation

$$F = AB \text{ or } F = A \cdot B$$

OR

Logic Symbol



Truth Table

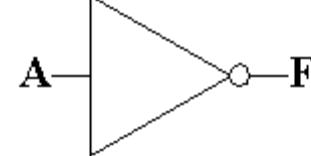
A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

Boolean Equation

$$F = A + B$$

NOT

Logic Symbol



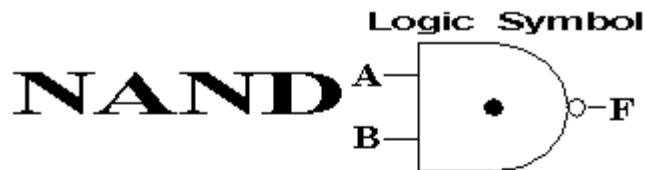
Truth Table

A	F
0	1
1	0

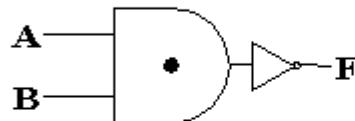
Boolean Equation

$$F = A'$$
 or $F = \bar{A}$

Common Derived Logic Gates (operators)



Equivalent Representations



or

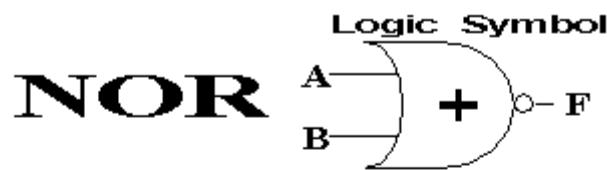


Truth Table

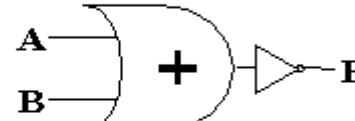
A	B	F
0	0	1
0	1	1
1	0	1
1	1	0

Boolean Equation

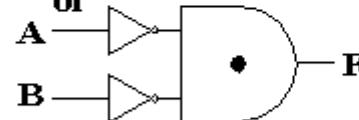
$$F = (AB)'$$



Equivalent Representations



or



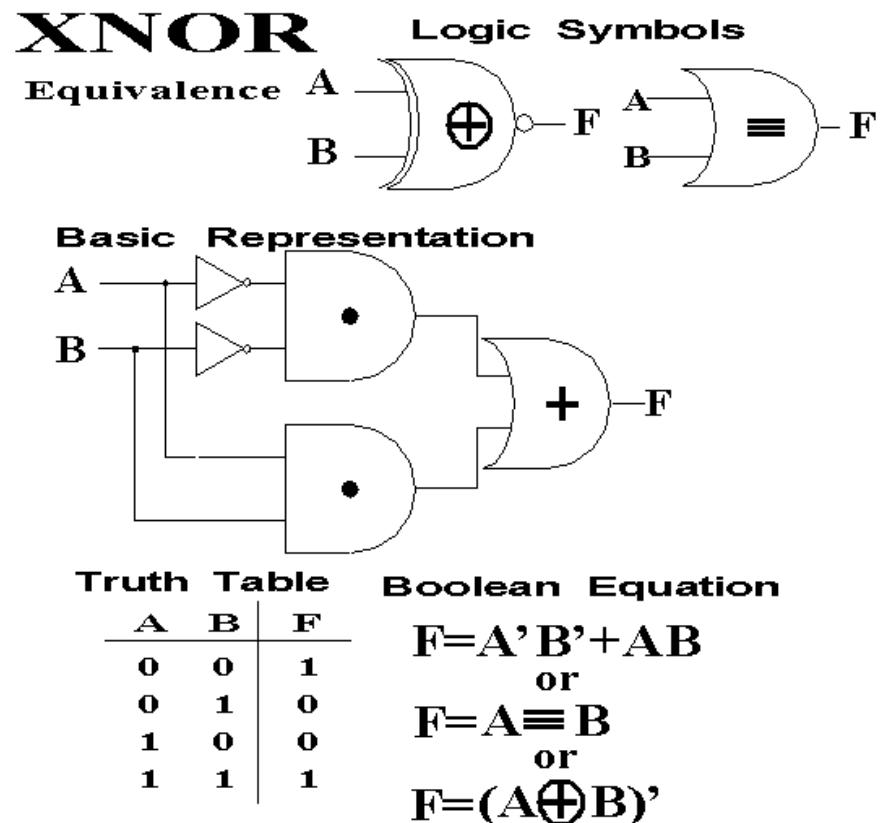
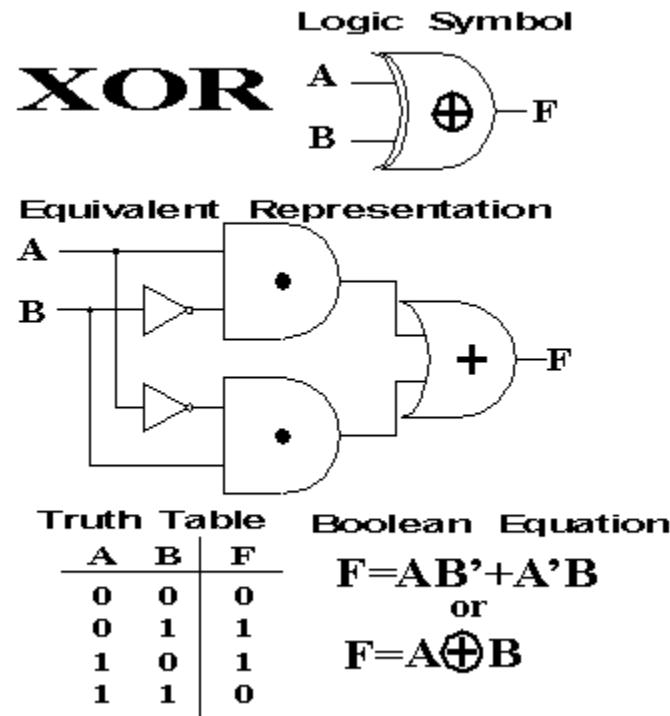
Truth Table

A	B	F
0	0	1
0	1	0
1	0	0
1	1	0

Boolean Equation

$$F = (A+B)'$$

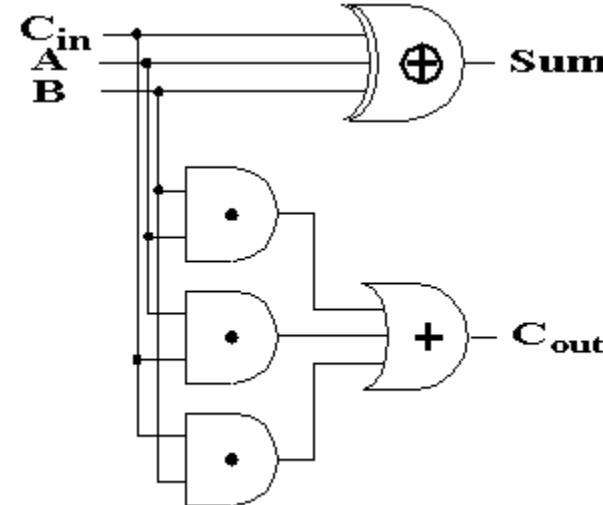
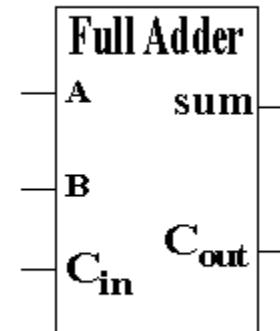
Common Derived Logic Gates (operators)



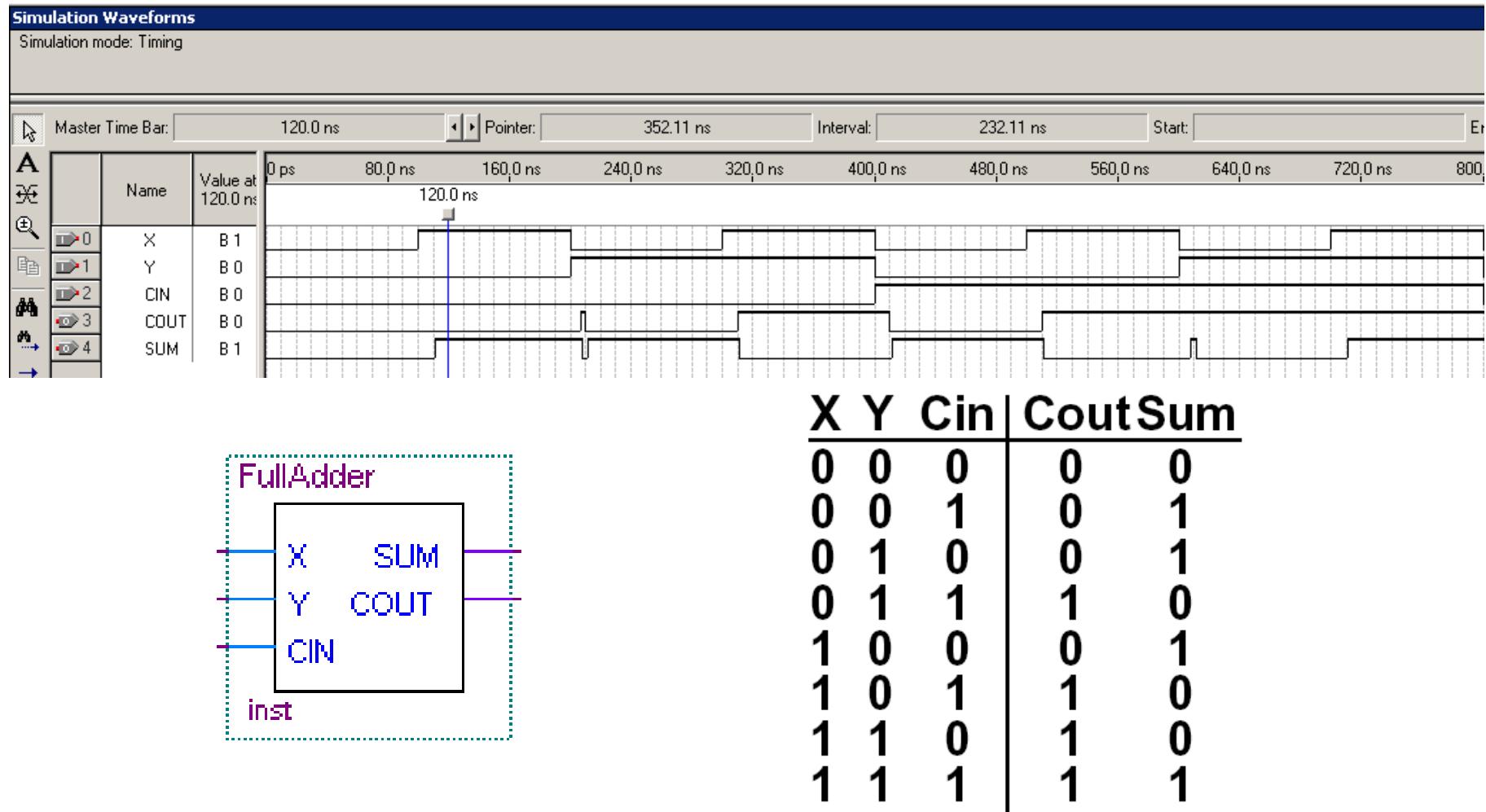
Full Adder

C_{in}	A	B	Sum
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

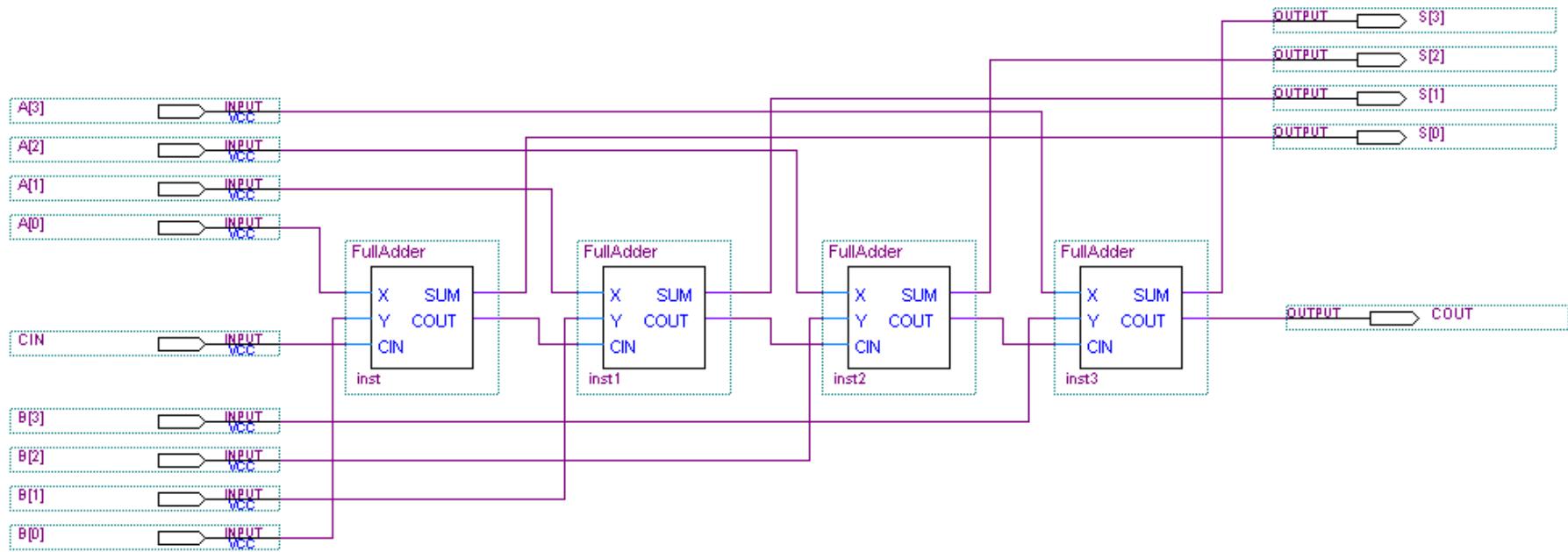
C_{in}	A	B	C_{out}
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



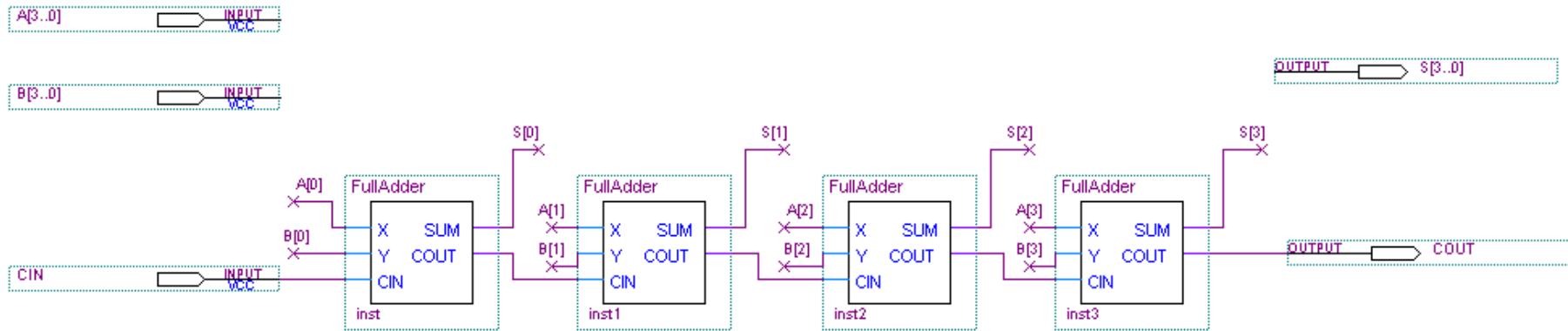
Quartus II Post Synthesis/Place and Route Simulation of Full Adder



Hierarchical Design of a 4-Bit Adder

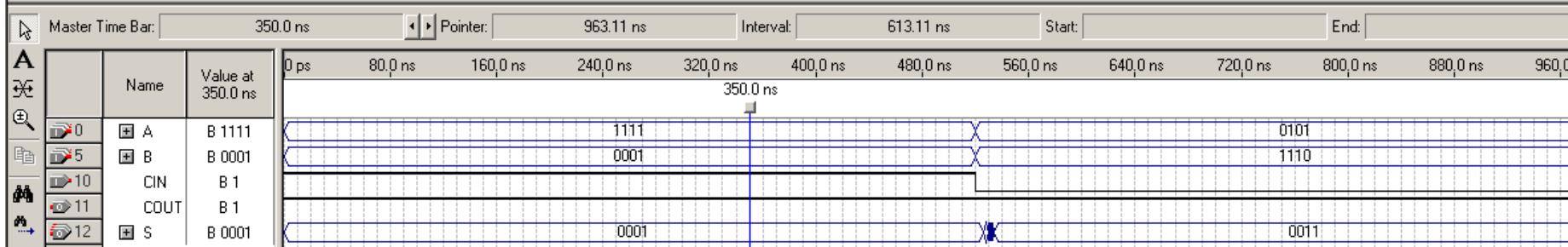


Same Design using Signal Naming Convention Instead of Wires



Simulation Waveforms

Simulation mode: Timing



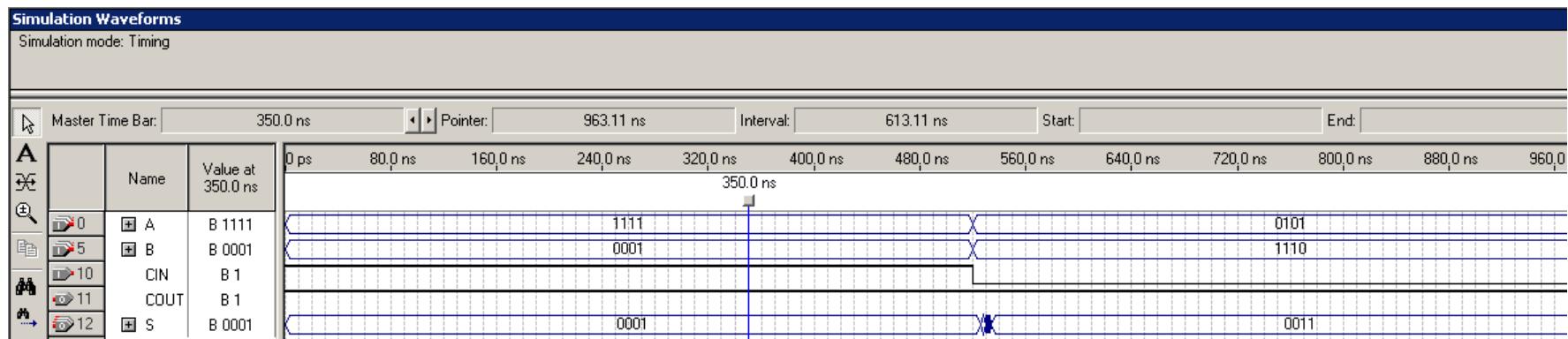
Quartus II Simulation of 4-bit Adder

$$\begin{array}{r} \textcolor{red}{1} \text{ } \textcolor{red}{1} \text{ } \textcolor{red}{1} \text{ } \textcolor{red}{1} \\ 1 \text{ } 1 \text{ } 1 \text{ } 1 \quad A \\ 0 \text{ } 0 \text{ } 0 \text{ } 1 \quad B \\ + \qquad \qquad \qquad \text{CIN} \\ \hline 10001 \quad S \end{array}$$

COUT

$$\begin{array}{r} \textcolor{red}{1} \text{ } \textcolor{red}{1} \text{ } \textcolor{red}{1} \text{ } \textcolor{red}{1} \\ 0 \text{ } 1 \text{ } 0 \text{ } 1 \quad A \\ 1 \text{ } 1 \text{ } 1 \text{ } 0 \quad B \\ + \qquad \qquad \qquad \text{CIN} \\ \hline 10011 \quad S \end{array}$$

COUT



Inversion

- DeMorgan's Laws:

$$(X + Y)' = X'Y'$$

The complement of the sum is the product of the complements

$$(XY)' = X'+Y'$$

The complement of the product is the sum of the complements

Complement (inverse) is formed by replacing each variable with its complement, replacing AND with OR, OR with AND, 0 with 1, and 1 with 0 while keeping the same precedence of operation

Inversion

$$F = X + E'K[C(AB+D') \bullet 1 + WZ'(G'H + 0)]$$



$$F' = X'(E + K' + [C' + (A' + B') D + 0] [W' + Z + (G + H') \bullet 1])$$

Complement (inverse) is formed by replacing each variable with its complement, replacing AND with OR, OR with AND, 0 with 1, and 1 with 0 while keeping the same precedence of operation

Duality Principle

A Dual of a Boolean Expression is obtained by swapping the two multi-input operators (**AND** and **OR**) at the same time swapping the values (0 & 1) while keeping the same precedence of operation.

Note: Unlike Inversion the Dual does not involve complementing the variables

Duality Principle states that if two Boolean expressions are equal then the duals are also equal.

Duality and Positive and Negative Logic

- Positive Logic
 - low voltage corresponds to a logic 0
 - high voltage to a logic 1
- Negative Logic
 - Low voltage corresponds to a logic 1
 - High voltage to a logic 0
- A circuit that implements a Boolean expression in Positive Logic will Implement the Dual of that expression in negative logic

Laws and Theorems of Boolean Algebra

Operations with 0 and 1:

$$X + 0 = X \quad (1-5)$$

$$X + 1 = 1 \quad (1-6)$$

$$X \cdot 1 = X \quad (1-5D)$$

$$X \cdot 0 = 0 \quad (1-6D)$$

Idempotent laws:

$$X + X = X \quad (1-7)$$

$$X \cdot X = X \quad (1-7D)$$

Involution law:

$$(X')' = X \quad (1-8)$$

Laws of complementarity

$$X + X' = 1 \quad (1-9)$$

$$X \cdot X' = 0 \quad (1-9D)$$

Commutative laws:

$$X + Y = Y + X \quad (1-10)$$

$$XY = YX \quad (1-10D)$$

Associative laws:

$$\begin{aligned} (X + Y) + Z &= X + (Y + Z) \\ &= X + Y + Z \end{aligned} \quad (1-11)$$

$$(XY)Z = X(YZ) = XYZ \quad (1-11D)$$

Distributive laws:

$$X(Y + Z) = XY + XZ \quad (1-12)$$

$$X + YZ = (X + Y)(X + Z) \quad (1-12D)$$

Laws and Theorems of Boolean Algebra

Simplification theorems:

$$XY + XY' = X \quad (1-13)$$

$$X + XY = X \quad (1-14)$$

$$(X + Y')Y = XY \quad (1-15)$$

$$(X + Y)(X + Y') = X \quad (1-13D)$$

$$X(X + Y) = X \quad (1-14D)$$

$$XY' + Y = X + Y \quad (1-15D)$$

DeMorgan's laws:

$$(X + Y + Z + \dots)' = X'Y'Z' \dots \quad (1-16) \quad (XYZ\dots)' = X' + Y' + Z' + \dots \quad (1-16D)$$

$$[f(X_1, X_2, \dots, X_n, 0, 1, +, \cdot)]' = f(X_1', X_2', \dots, X_n', 1, 0, \cdot, +) \quad (1-17)$$

Duality:

$$(X + Y + Z + \dots)^D = XYZ\dots \quad (1-18) \quad (XYZ\dots)^D = X + Y + Z + \dots \quad (1-18D)$$

$$[f(X_1, X_2, \dots, X_n, 0, 1, +, \cdot)]^D = f(X_1, X_2, \dots, X_n, 1, 0, \cdot, +) \quad (1-19)$$

Theorem for multiplying out and factoring:

$$(X + Y)(X' + Z) = XZ + X'Y \quad (1-20) \quad XY + X'Z = (X + Z)(X' + Y) \quad (1-20D)$$

Consensus theorem:

$$XY + YZ + X'Z = XY + X'Z \quad (1-21) \quad \begin{aligned} & (X + Y)(Y + Z)(X' + Z) \\ & = (X + Y)(X' + Z) \end{aligned} \quad (1-21D)$$

Simplifying Logic Expressions

- Combining terms
 - Use $XY+XY'=X$ (1-13) , $X+X=X$ (1-7)

$$\mathbf{ABC'D' + ABCD' = \underline{ABC'D'} + \underline{ABCD'} = ABD'}$$

$$Cout = X'Y\text{Cin} + XY'\text{Cin} + XY\text{Cin}' + XY\text{Cin}$$

$$= (X'Y\text{Cin} + XY\text{Cin}) + (XY'\text{Cin} + XY\text{Cin}) + (XY\text{Cin}' + XY\text{Cin})$$

$$= Y\text{Cin} + X\text{Cin} + XY$$

Simplifying Logic Expressions

- Combining terms

- Use $XY+XY' = X$ (1-13) , DeMorgan's Laws (1-16)

$$(A+BC)(D + E') + A'(B'+C')(D+E') =$$

$$\underline{[A'(B'+C')]'}(D + E') + \underline{A'(B'+C')}(D+E') =$$

$$D+E'$$

Simplifying Logic Expressions

- Eliminating terms
 - Use $X+XY=X$ (1-14) , $X^*X=X$ (1-7D),
 - Consensus theorem $XY +YZ +X'Z = XY + X'Z$ (1-21)

$$A'B + A'BC = A'B$$

$$\begin{aligned} A'BC' + BCD + A'BD &= \underline{C'A'B} + \underline{CBD} + \underline{A'BD} = \\ &= A'BC' + BCD \end{aligned}$$

Simplifying Logic Expressions

- Eliminating literals

– Use $X(Y+Z) = XY + XZ$ (1-12) , $XY' + Y = X + Y$ (1-15D)

$$A'B + A'B'C'D' + ABCD' = A'(B + B'C'D') + ABCD'$$

$$A'(B + C'D') + ABCD'$$

$$A'B + A'C'D' + ABCD'$$

$$B(A' + ACD') + A'C'D'$$

$$B(A' + CD') + A'C'D'$$

$$A'B + BCD' + A'C'D'$$

Simplifying Logic Expressions

- Adding Redundant terms
 - Add $X \cdot X' (=0)$ or Multiply by $(X+X') (=1)$ or add in consensus term ($[XY + YZ + X'Z = XY + X'Z (1-21)]$)

$$\underline{WX} + \underline{XY} + \underline{X'Z'} + \underline{WY'Z'} =$$

$$\underline{WX} + \underline{XY} + \underline{X'Z'} + \underline{WY'Z'} + \underline{WZ'}$$

$$\underline{WX} + \underline{XY} + \underline{X'Z'} + \cancel{\underline{WY'Z'}} + \underline{WZ'}$$

$[X + XY = X (1-14)]$

$$\underline{WX} + \underline{XY} + \underline{X'Z'} + \cancel{\underline{WZ'}}$$

$$\underline{WX} + \underline{XY} + \underline{X'Z'}$$

Theorems to Apply to Exclusive-OR

$$X \oplus 0 = X$$

$$X \oplus 1 = X'$$

$$X \oplus X = 0$$

$$X \oplus X' = 1$$

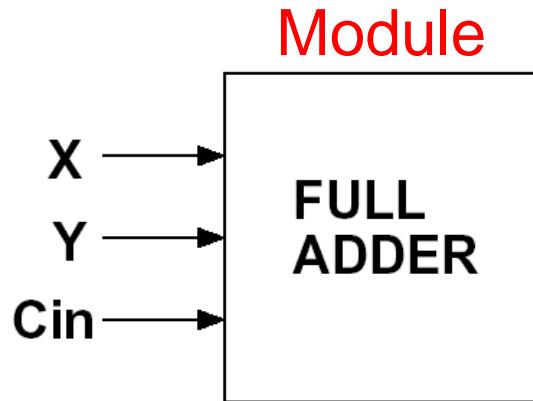
$$X \oplus Y = Y \oplus X \quad (\text{Commutative law})$$

$$(X \oplus Y) \oplus Z = X \oplus (Y \oplus Z) \quad (\text{Associative law})$$

$$X(Y \oplus Z) = XY \oplus XZ \quad (\text{Distributive law})$$

$$(X \oplus Y)' = X \oplus Y' = X' \oplus Y = XY + X'Y'$$

Full Adder



Truth table

X	Y	Cin	Cout	Sum
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

Algebraic expressions

F(inputs for which the function is 1):

Minterms

$$\text{Sum} = X'Y'Cin + X'Y'Cin' + XY'Cin' + XYCin$$

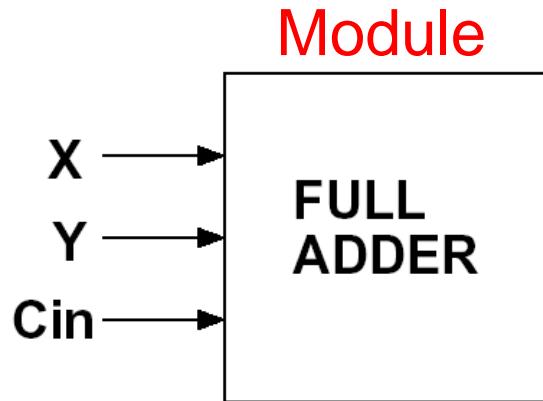
$$\text{Cout} = X'YCin + XY'Cin + XYCin' + XYCin$$

m-notation

$$\text{Sum} = m_1 + m_2 + m_4 + m_7 = \sum m(1, 2, 4, 7)$$

$$\text{Cout} = m_3 + m_5 + m_6 + m_7 = \sum m(3, 5, 6, 7)$$

Full Adder (cont'd)



Truth table

X	Y	Cin	Cout	Sum
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

Algebraic expressions

F(inputs for which the function is 0):

$$\text{Sum} = (X + Y + \text{Cin})(X + Y' + \text{Cin}')(X' + Y + \text{Cin}')(X' + Y' + \text{Cin})$$

$$\text{Cout} = (X + Y + \text{Cin})(X + Y + \text{Cin}')(X + Y' + \text{Cin})(X' + Y + \text{Cin})$$

M-notation

$$\text{Sum} = M_0 \cdot M_3 \cdot M_5 \cdot M_6 = \prod M(0, 3, 5, 6)$$

$$\text{Cout} = M_0 \cdot M_1 \cdot M_2 \cdot M_4 = \prod M(0, 1, 2, 4)$$

Karnaugh Maps

- Convenient way to simplify logic functions of 3, 4, 5, (6) variables
- Four-variable K-map
 - each square corresponds to one of the 16 possible minterms
 - 1 - minterm is present;
0 (or blank) – minterm is absent;
 - X – don't care
 - the input can never occur, or
 - the input occurs but the output is not specified
 - adjacent cells differ in only one value => can be combined

Location
of minterms

msb lsb

$F(A,B,C,D) =$

AB \ CD	00	01	11	10
00	0	4	12	8
01	1	5	13	9
11	3	7	15	11
10	2	6	14	10

Kmap

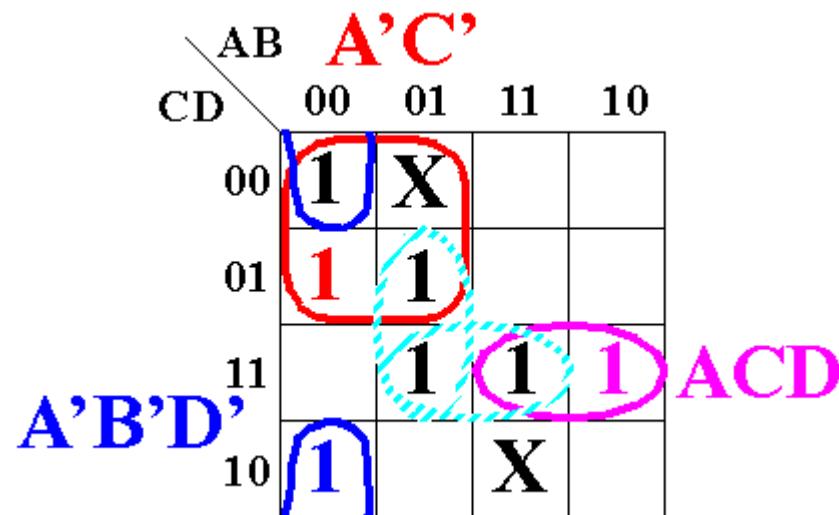
Sum-of-Products Representation

- Function consists of a sum of Prime Implicants
- **Prime Implicant**
 - a group of one, two, four, eight 1s on the Kmap represents a prime implicant if it cannot be combined with another group of 1s to eliminate a variable
- Prime Implicant is **Essential** if it contains a 1 that is not contained in any other Prime Implicant

Selection of Prime Implicants

- 1. Choose a minterm (a 1) that has not been covered yet
- 2. Find all 1s and Xs adjacent to that minterm
- 3. If a single term covers the minterm and all adjacent 1s and Xs, then that term is an essential prime implicant, so select that term
- 4. Repeat steps 1, 2, 3 until all essential prime implicants have been chosen
- 5. Find a minimum set of prime implicants that cover the remaining 1s on the map. If there is more than one such set, choose a set with a minimum number of literals

$$F(A,B,C,D) = \sum m(0,1,2,5,7,11,15) + \sum d(4,14)$$



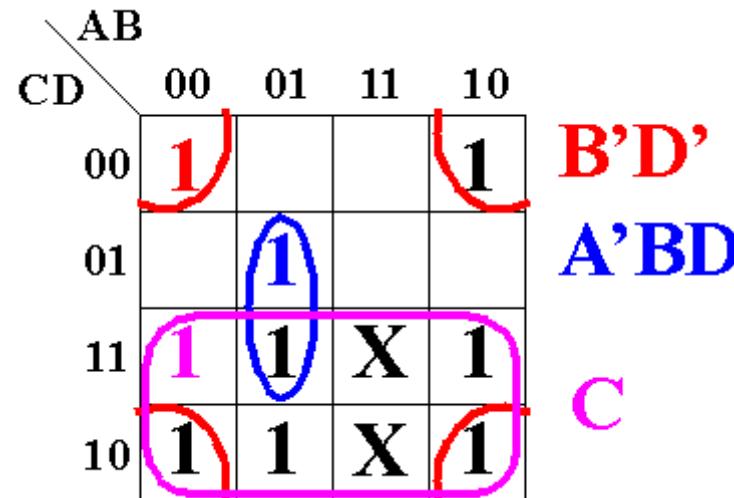
Two minimum forms

$$\begin{aligned} F &= A'C' + A'B'D' + ACD + A'BD \\ &\text{or} \\ F &= A'C' + A'B'D' + ACD + BCD \end{aligned}$$

Minimum Sum-of-Products Representation

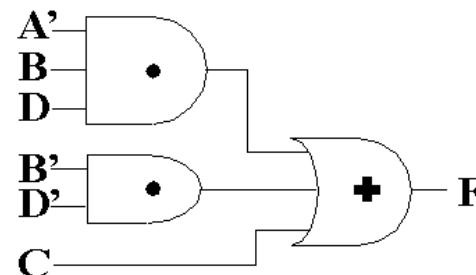
$$F(A,B,C,D) = \sum m(0,2,3,5,6,7,8,10,11) + \sum d(14,15)$$

- 1. Choose a minterm (a 1) that has not been covered yet
- 2. Find all 1s and Xs adjacent to that minterm
- 3. If a single term covers the minterm and all adjacent 1s and Xs, then that term is an essential prime implicant, so select that term
- 4. Repeat steps 1, 2, 3 until all essential prime implicants have been chosen
- 5. Find a minimum set of prime implicants that cover the remaining 1s on the map. If there is more than one such set, choose a set with a minimum number of literals



$$F(A,B,C,D) = B'D' + A'BD + C$$

*Minimum
SOP
Representation*



Inversion

- DeMorgan's Laws:

$$(X + Y)' = X'Y'$$

The complement of the sum is the product of the complements

$$(XY)' = X' + Y'$$

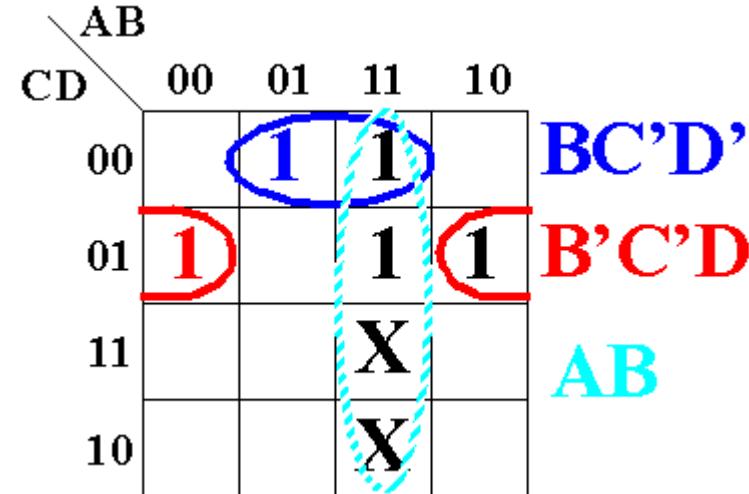
The complement of the product is the sum of the complements

Minimum Product-of-Sums Representation

$$F(A,B,C,D) = \sum m(0,2,3,5,6,7,8,10,11) + \sum d(14,15)$$

$$F'(A,B,C,D) = \sum m(1,4,9,12,13) + \sum d(14,15)$$

- 1. Choose a minterm (a 1) that has not been covered yet
- 2. Find all 1s and Xs adjacent to that minterm
- 3. If a single term covers the minterm and all adjacent 1s and Xs, then that term is an essential prime implicant, so select that term
- 4. Repeat steps 1, 2, 3 until all essential prime implicants have been chosen
- 5. Find a minimum set of prime implicants that cover the remaining 1s on the map. If there is more than one such set, choose a set with a minimum number of literals



$$F'(A,B,C,D) = BC'D' + B'C'D + AB$$

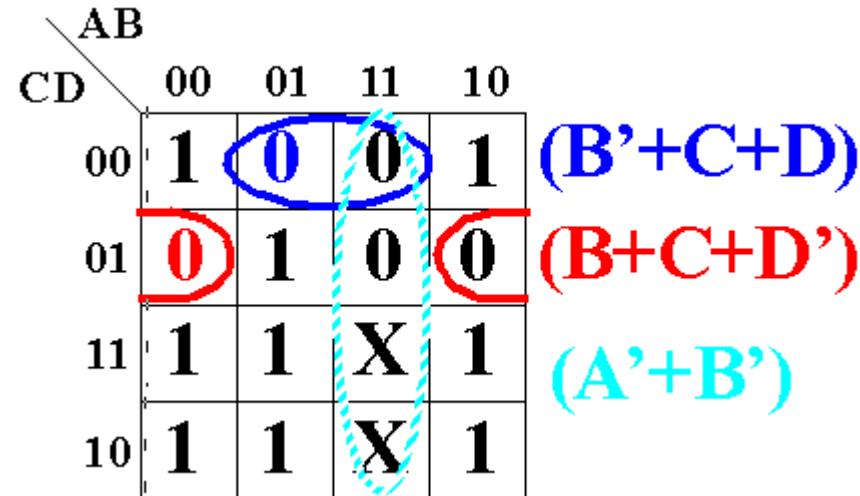
$$F(A,B,C,D) = [F'(A,B,C,D)]' \text{ (Involution law)}$$

$$F(A,B,C,D) = (B'+C+D)(B+C+D')(A'+B') \text{ (DeMorgan's Laws)}$$

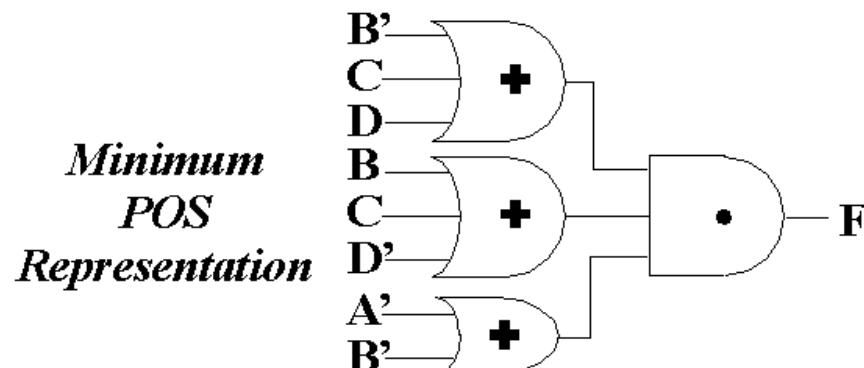
Minimum Product-of-Sums Representation

$$F(A,B,C,D) = \sum m(0,2,3,5,6,7,8,10,11) + \sum d(14,15)$$

- 1. Choose a 0 term that has not been covered yet
- 2. Find all 0s and Xs adjacent to that minterm
- 3. If a single term covers the 0 term and all adjacent 0s and Xs, then that term is essential, so select that term
- 4. Repeat steps 1, 2, 3 until all essential groupings have been chosen
- 5. Find a minimum set of groupings that cover the remaining 0s on the map. If there is more than one such set, choose a set with a minimum number of literals
- 6. Interpret results in POS manner



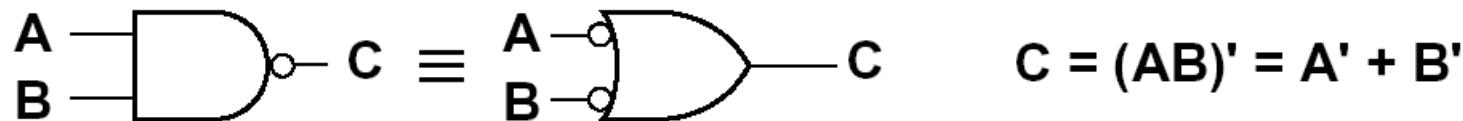
$$F(A,B,C,D) = (B' + C + D)(B + C + D')(A' + B')$$



Designing with NAND and NOR Gates (1)

- Implementation of NAND and NOR gates is easier than that of AND and OR gates (e.g., CMOS)

NAND:



NOR:



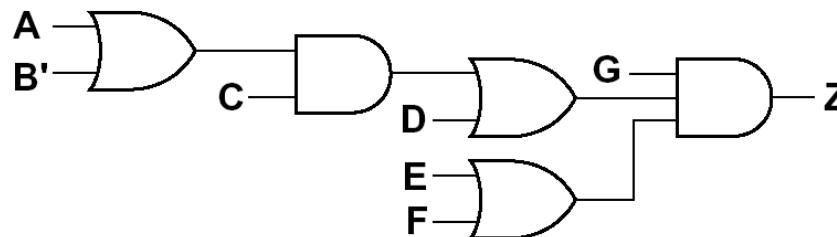
Designing with NAND and NOR Gates (2)

- Any logic function can be realized using only NAND or NOR gates => NAND/NOR is functionally complete
 - NAND function is complete – can be used to generate any logical function;
 - 1: $(a * a')' = (0)' = 1$
 - 0: $[(a * a')' * (a * a')'] = [1 * 1]' = [1]' = 0$
 - a' : $(a * a)' = (a)' = a'$
 - ab : $[(a * b)' * (a * b)']' = [(a * b)']' = a * b = ab$
 - $a+b$: $(a' * b')' = a + b$

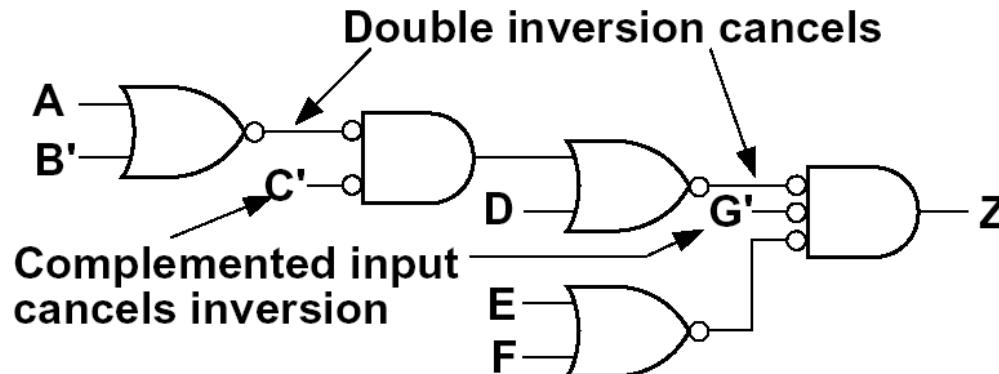
Using only NOR Gates to Implement Equivalent Multi-Level AND/OR Logic

- Example:

NOR:
 $A \text{---} B \text{---} C \equiv A \text{---} B \text{---} C$



(a) AND-OR network

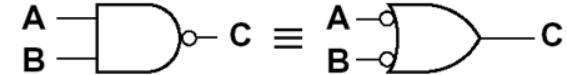


(b) Equivalent NOR-gate network

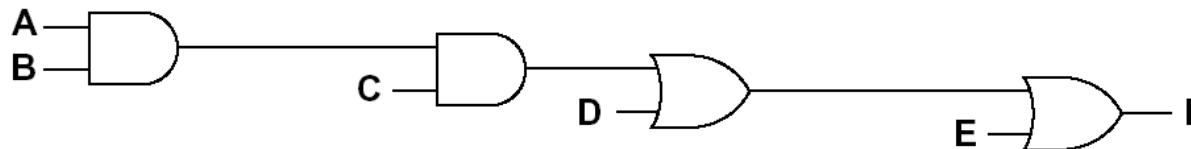
Using only NAND Gates to Implement Equivalent Multi-Level AND/OR Logic

- Example

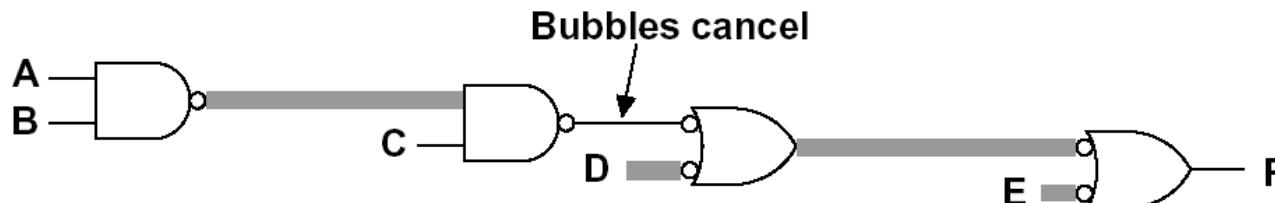
NAND:



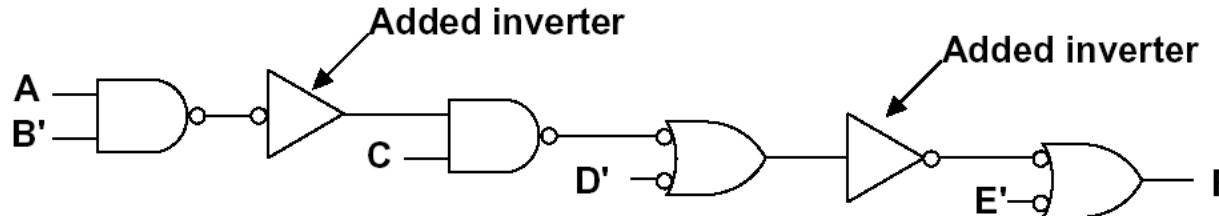
A \overline{AB} \equiv $\overline{A} \cdot \overline{B}$



(a) AND_OR network



(b) First step in NAND conversion

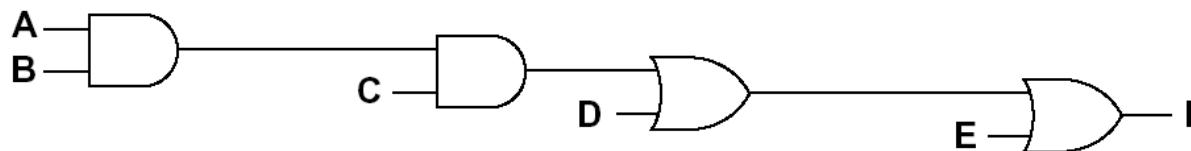


(c) Completed conversion

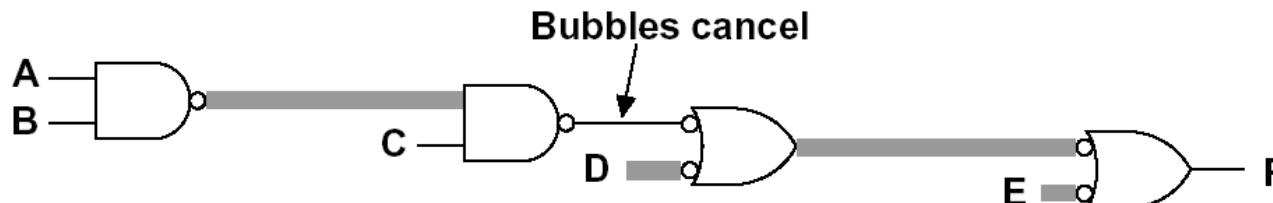
Using only NAND Gates to Implement Equivalent Multi-Level AND/OR Logic

- Example

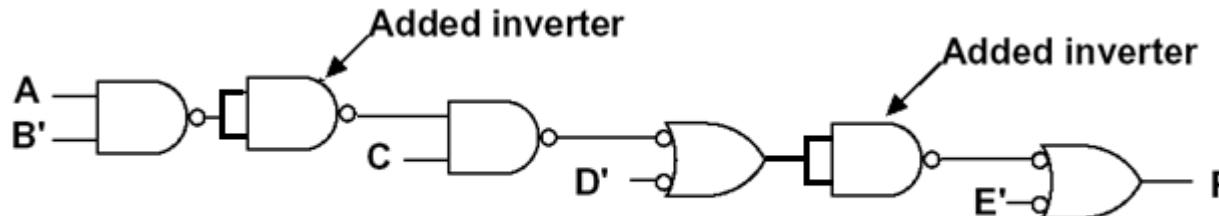
NAND:

$$A \text{---} B \text{---} \text{NAND gate} \text{---} C \equiv A \text{---} B \text{---} \text{Inverter} \text{---} C$$


(a) AND_OR network



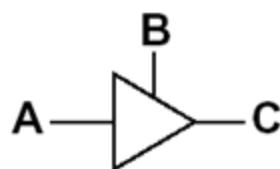
(b) First step in NAND conversion



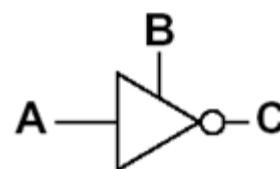
(c) Completed conversion

Tristate Logic and Busses

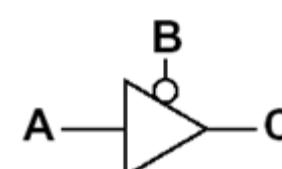
- Four kinds of tristate buffers
 - B is a control input used to enable and disable the output
 - Tristate Hi-Z state can be viewed simplistically as disconnecting the output C from the rest of the logic (more accurate description will be discussed later)



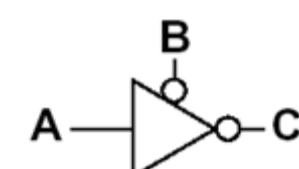
B	A	C
0	0	Hi-Z
0	1	Hi-Z
1	0	0
1	1	1



B	A	C
0	0	Hi-Z
0	1	Hi-Z
1	0	1
1	1	0



B	A	C
0	0	0
0	1	1
1	0	Hi-Z
1	1	Hi-Z



B	A	C
0	0	1
0	1	0
1	0	Hi-Z
1	1	Hi-Z

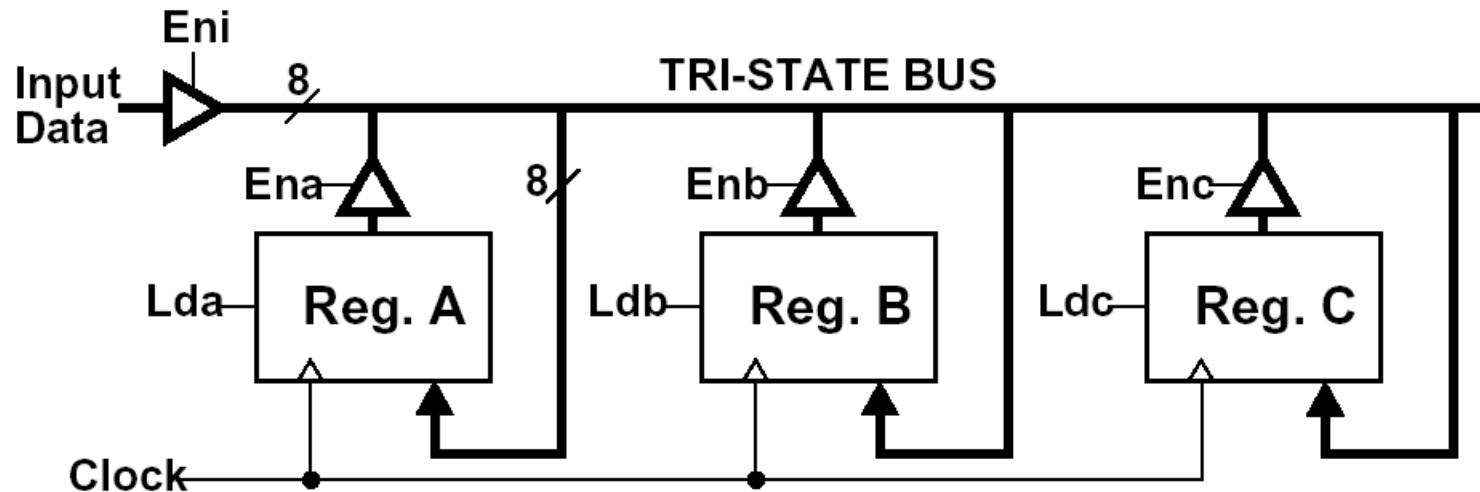
(a)

(b)

(c)

(d)

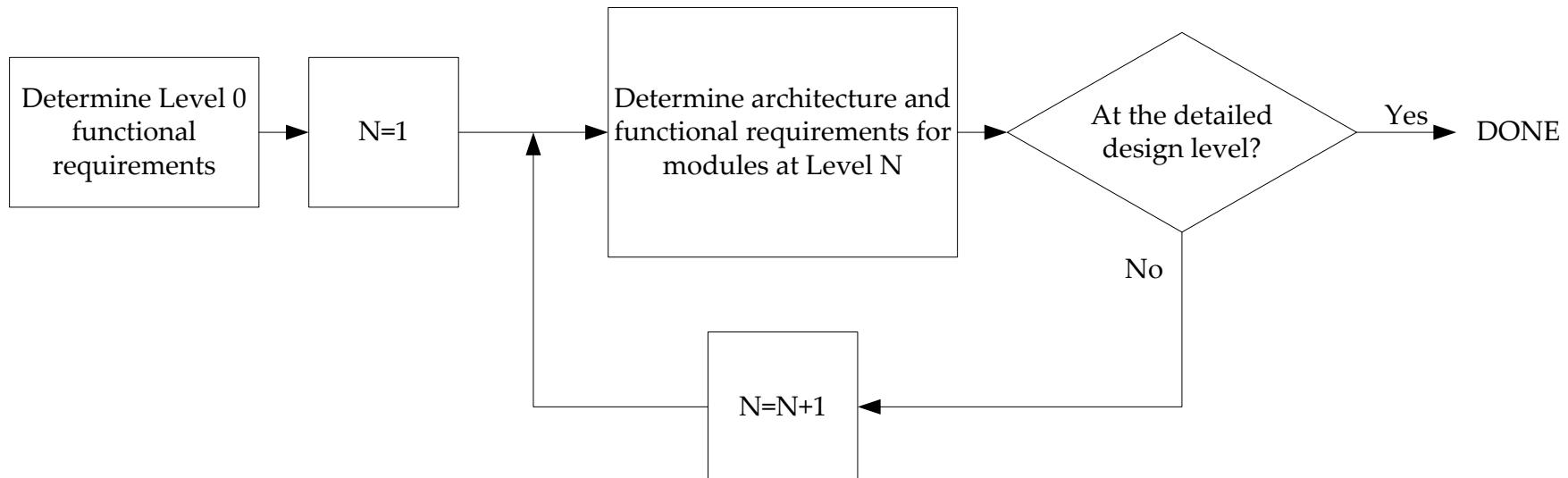
Data Transfer Using Tristate Bus



Hierarchical Design Using Schematic Capture

Form of Functional Decomposition

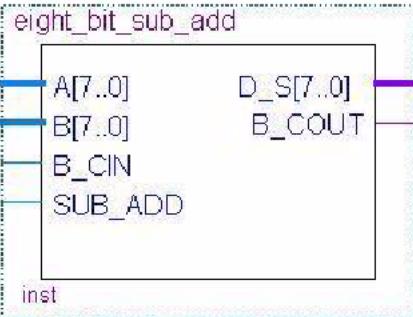
- **Recursively divide and conquer**
 - **Split a module into several sub-modules**
 - **Define the input, output, and behavior**
 - **Stop when you reach realizable components**



Example Lab 2 Part 1

(8 bit Subtractor/Adder Module, *eight_bit_sub_add* symbol)

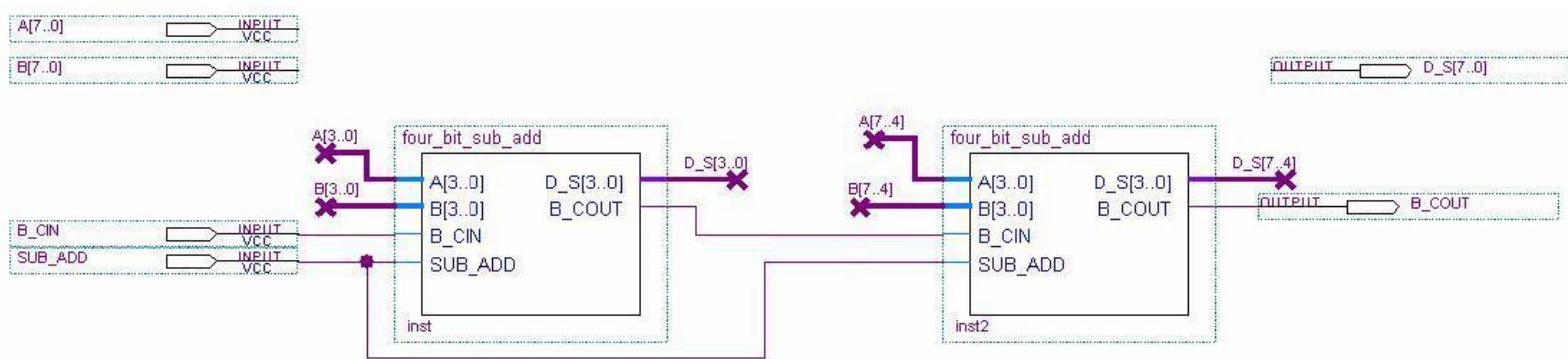
Function:



```
if (SUB_ADD) = '1' then
    D_S <= A - B - B_Cin --subtract B and B_Cin from A
        if (B>A) B_Cout <= '1' {borrow}
        else B_Cout <= '0' {no borrow}
    else { -- if SUB_ADD='0'
        D_S <= (A + B )[7:0]+B_Cin; {lower 8 bits of A + B + B_Cin}
        B_Cout <= (A+B+B_Cin)[8]; {most significant bit of A+B+B_Cin}
    }
```

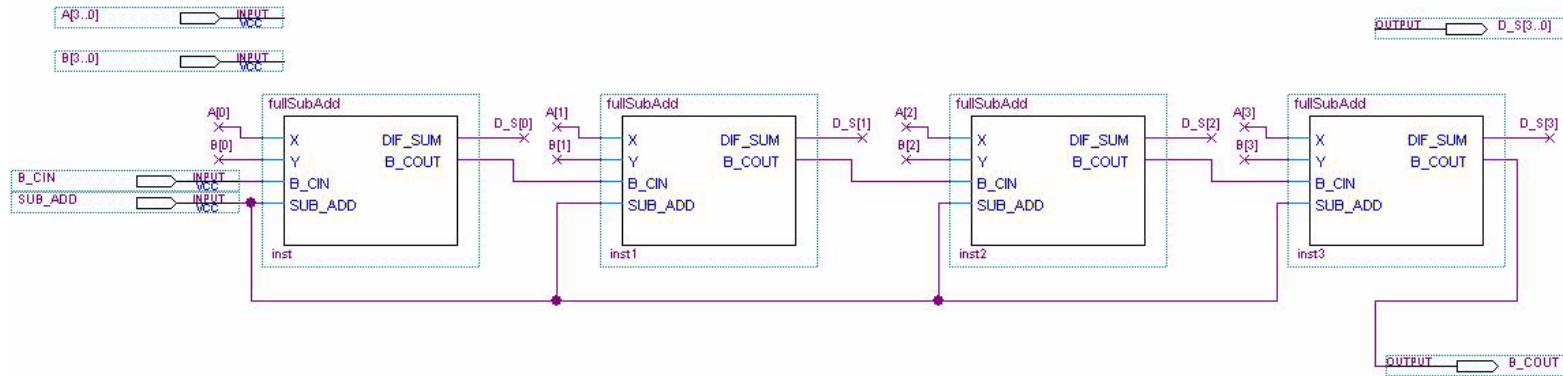
Example Lab 2 Part 1

(8 bit Subtractor/Adder Module, Level 1 -- *eight bit sub add module*)



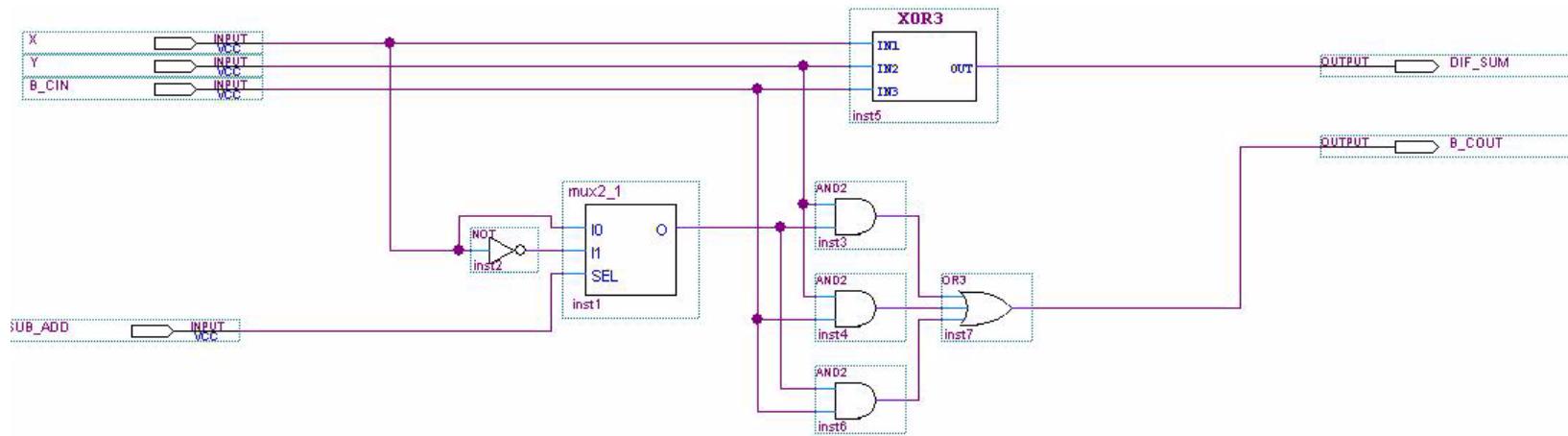
Example Lab 2 Part 1

(8 bit Subtractor/Adder Module, Level 2 -- four_bit_sub_add module)

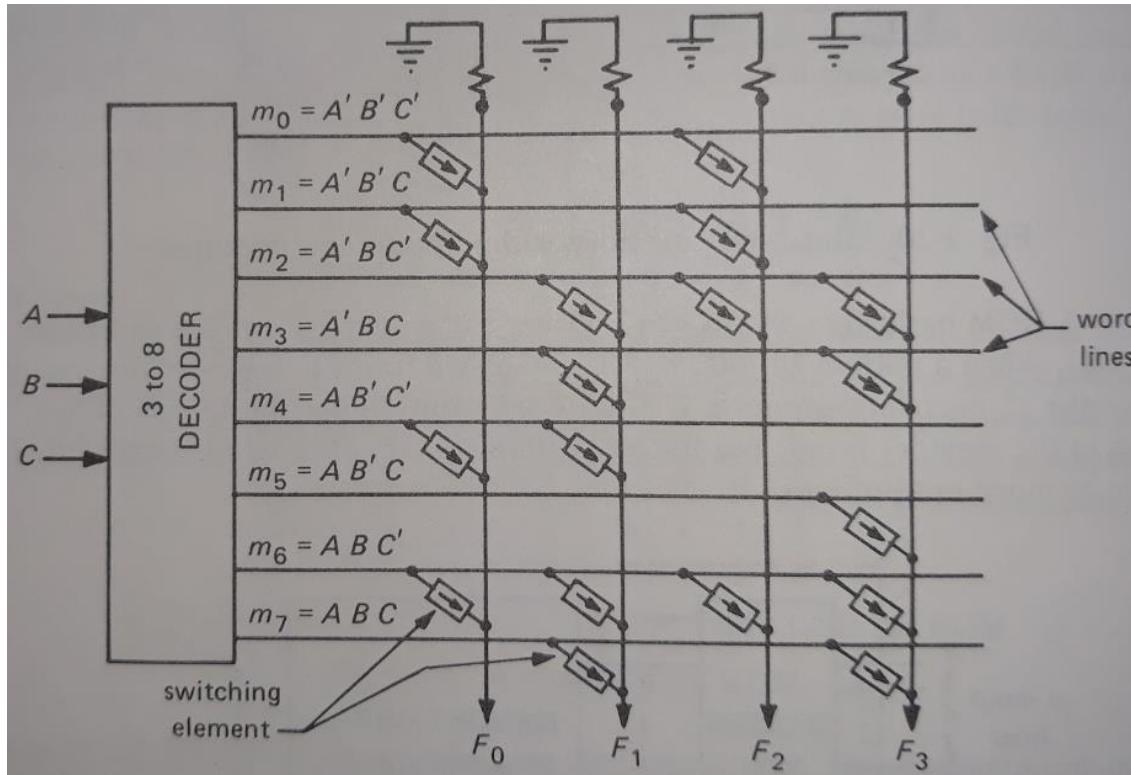


Example Lab 2 Part 1

(8 bit Subtractor/Adder Module, Level 3 -- *fullSub_Add module*)



8 word by 4 bit ROM used to Implement 4 Boolean Functions



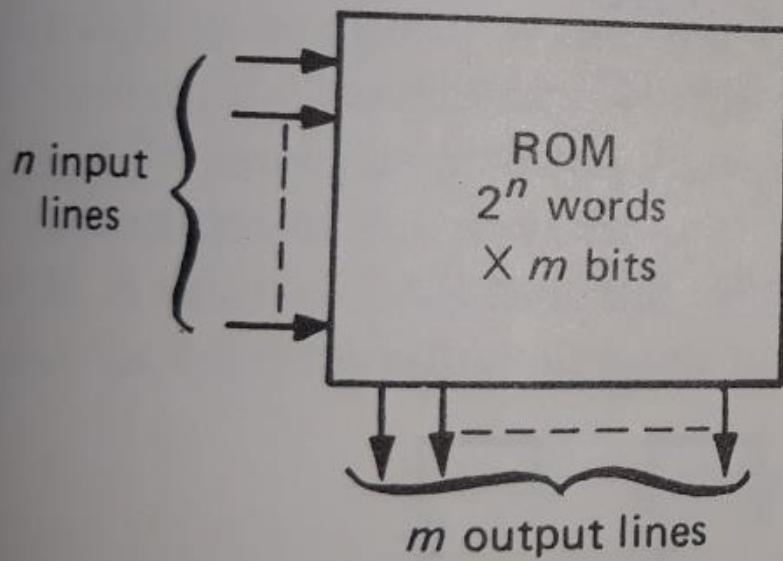
$$F_0 = \Sigma m(0, 1, 4, 6) = A'B' + AC'$$

$$F_1 = \Sigma m(2, 3, 4, 6, 7) = B + AC'$$

$$F_2 = \Sigma m(0, 1, 2, 6) = A'B' + BC'$$

$$F_3 = \Sigma m(2, 3, 5, 6, 7) = AC + B$$

ROM Memory with n Inputs and m Outputs



n input variables	m output variables
00 ... 00	100 ... 110
00 ... 01	010 ... 111
00 ... 10	101 ... 101
00 ... 11	110 ... 010
.	.
.	.
.	.
11 ... 00	001 ... 011
11 ... 01	110 ... 110
11 ... 10	011 ... 000
11 ... 11	111 ... 101

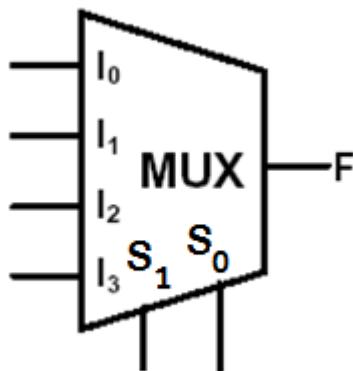
typical data array stored in ROM (2^n words of m bits each)

Using a 4-to-1 MUX to Implement a Boolean Combinational Logic Function

$$F = A'B' + B'C$$

Using a 4-to-1 MUX to Implement a Boolean Combinational Logic Function

$$F = A'B' + B'C$$

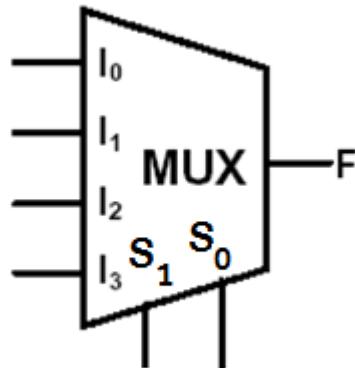


Using a 4-to-1 MUX to Implement a Boolean Combinational Logic Function

$$F = A'B' + B'C$$

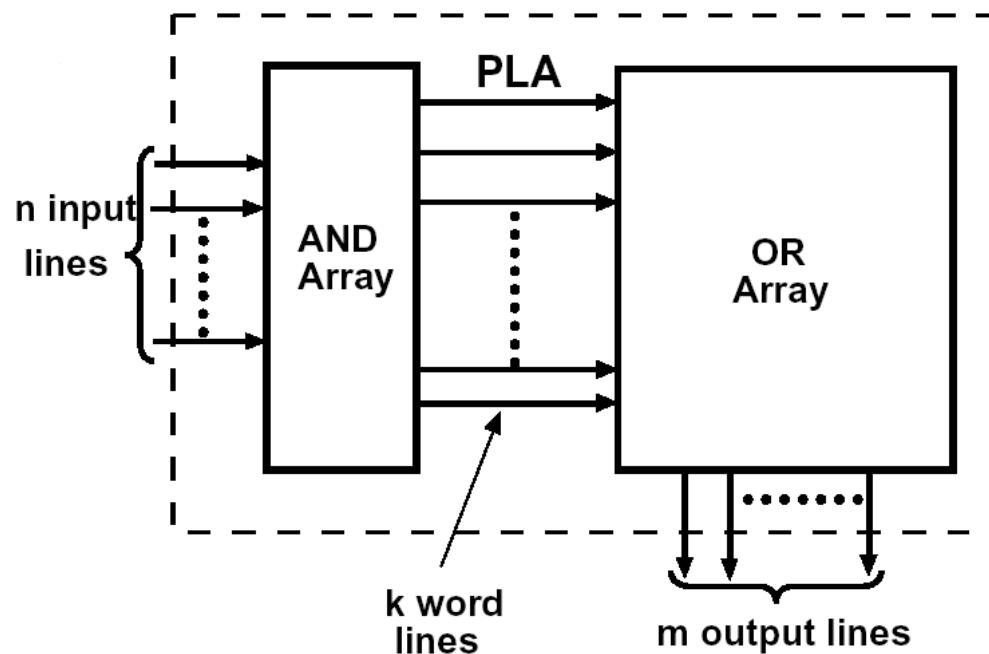
Truth Table

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

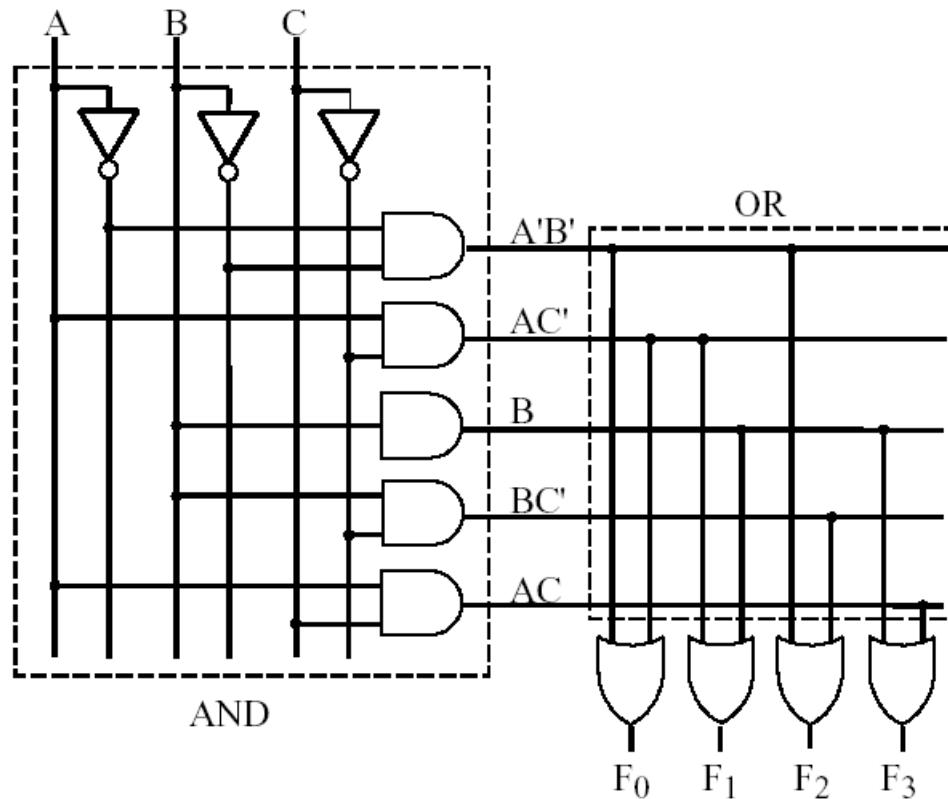


Programmable Logic Arrays (PLAs)

- Legacy Programmable Logic Device that incorporates two-level logic
 - n inputs and m outputs – m functions of n variables
 - AND array – realizes product terms of the input variables
 - OR array – ORs together the product terms



AND-OR Array Equivalent

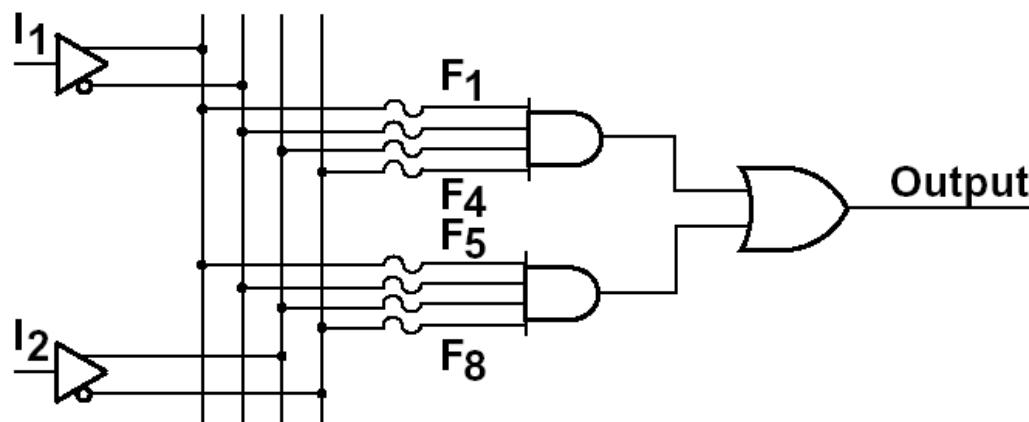
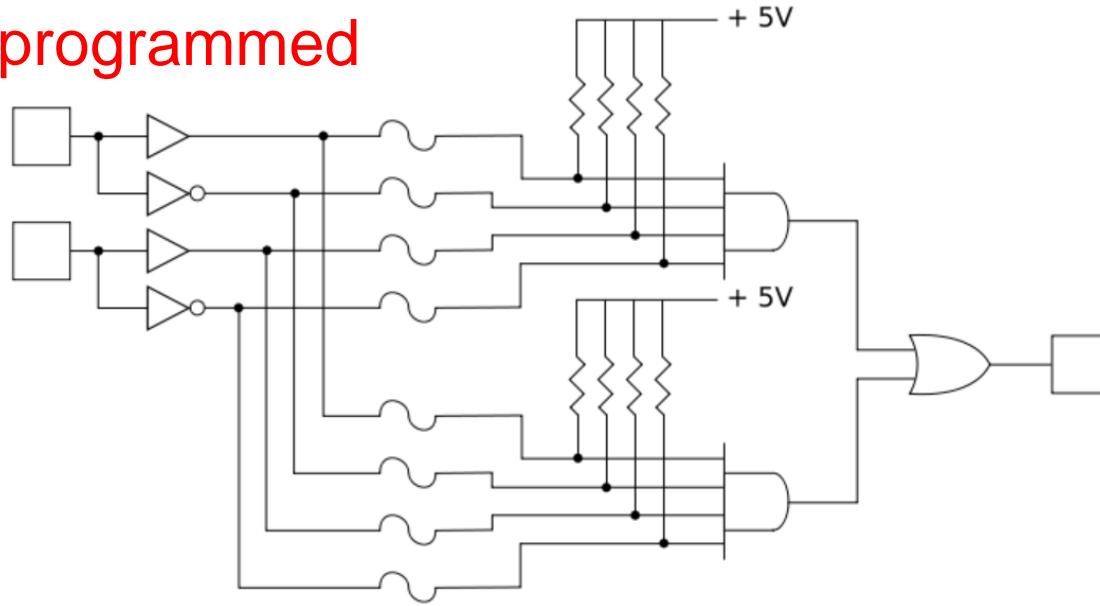


Programmable Array Logic (PALs)

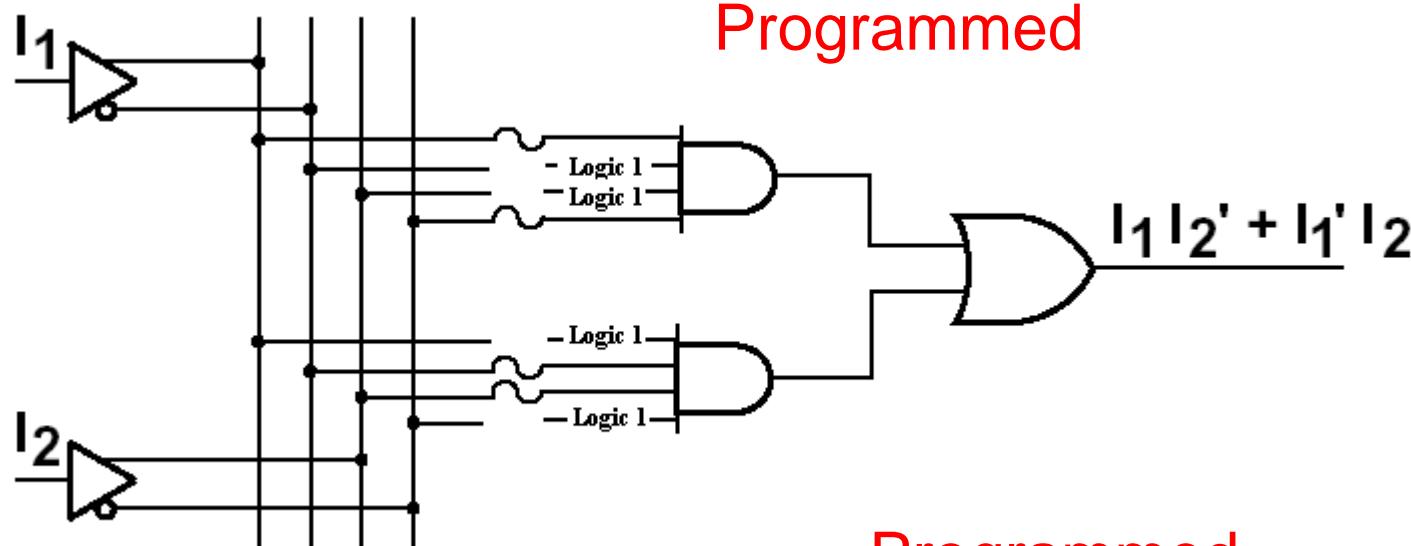
- PAL is a special case of PLA
 - AND array is programmable and OR array is fixed
- PAL is
 - less expensive
 - easier to program

Programmable Array Logic (PALs)

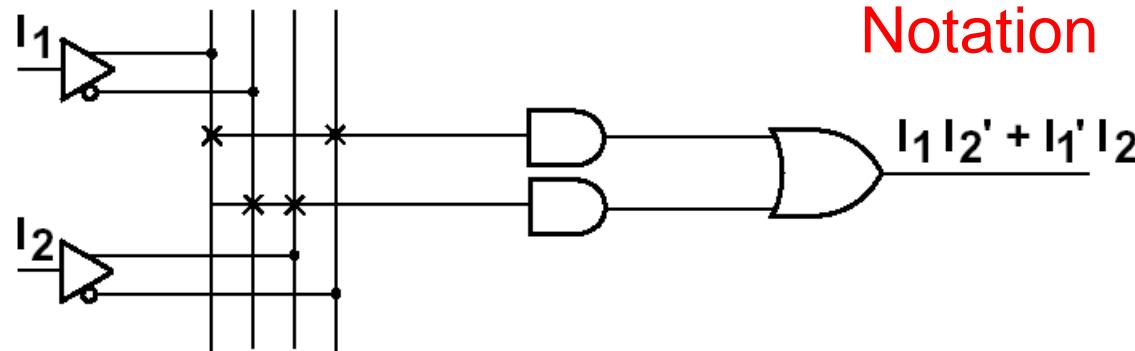
Unprogrammed



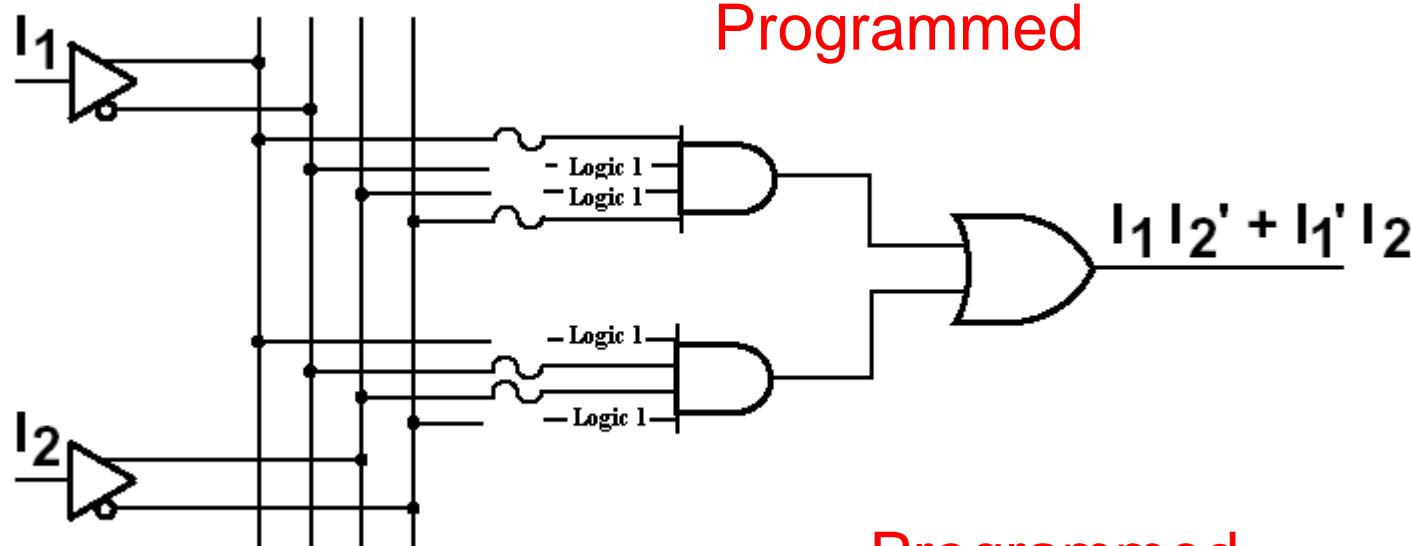
Programmable Array Logic (PALs)



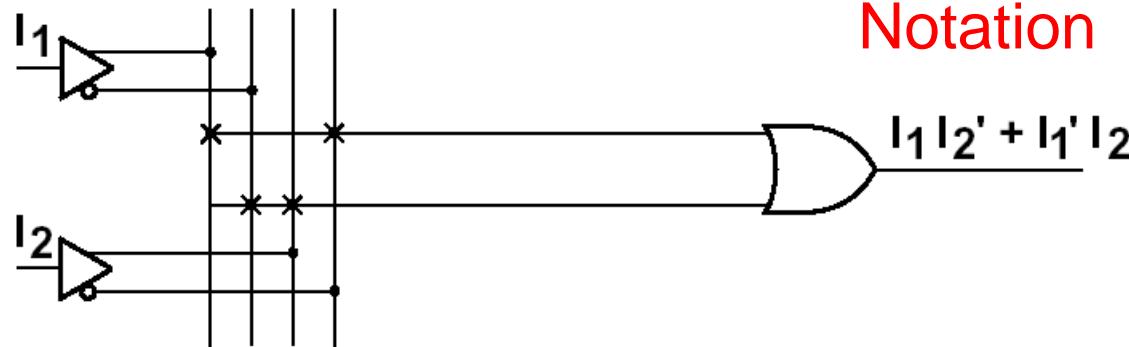
Programmed
Shorthand
Notation



Programmable Array Logic (PALs)



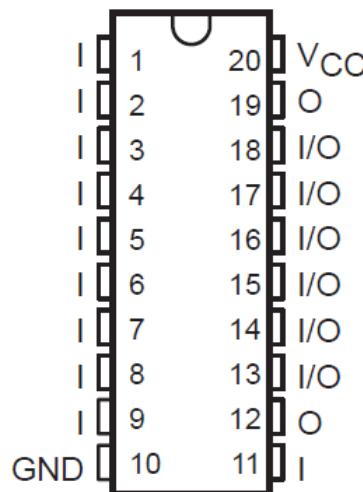
Programmed
Shorthand
Notation



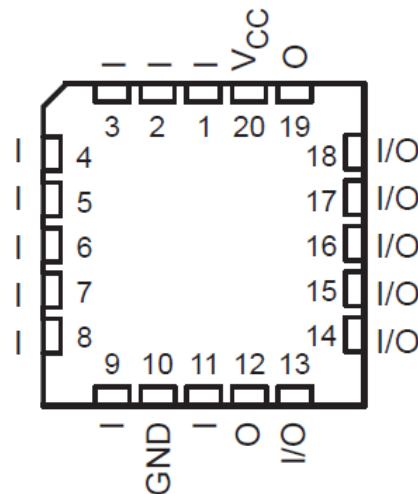
PALs

- Typical PALs have
 - from 10 to 20 inputs
 - from 2 to 10 outputs
 - from 2 to 8 AND gates driving each OR gate
 - often include tri-state logic

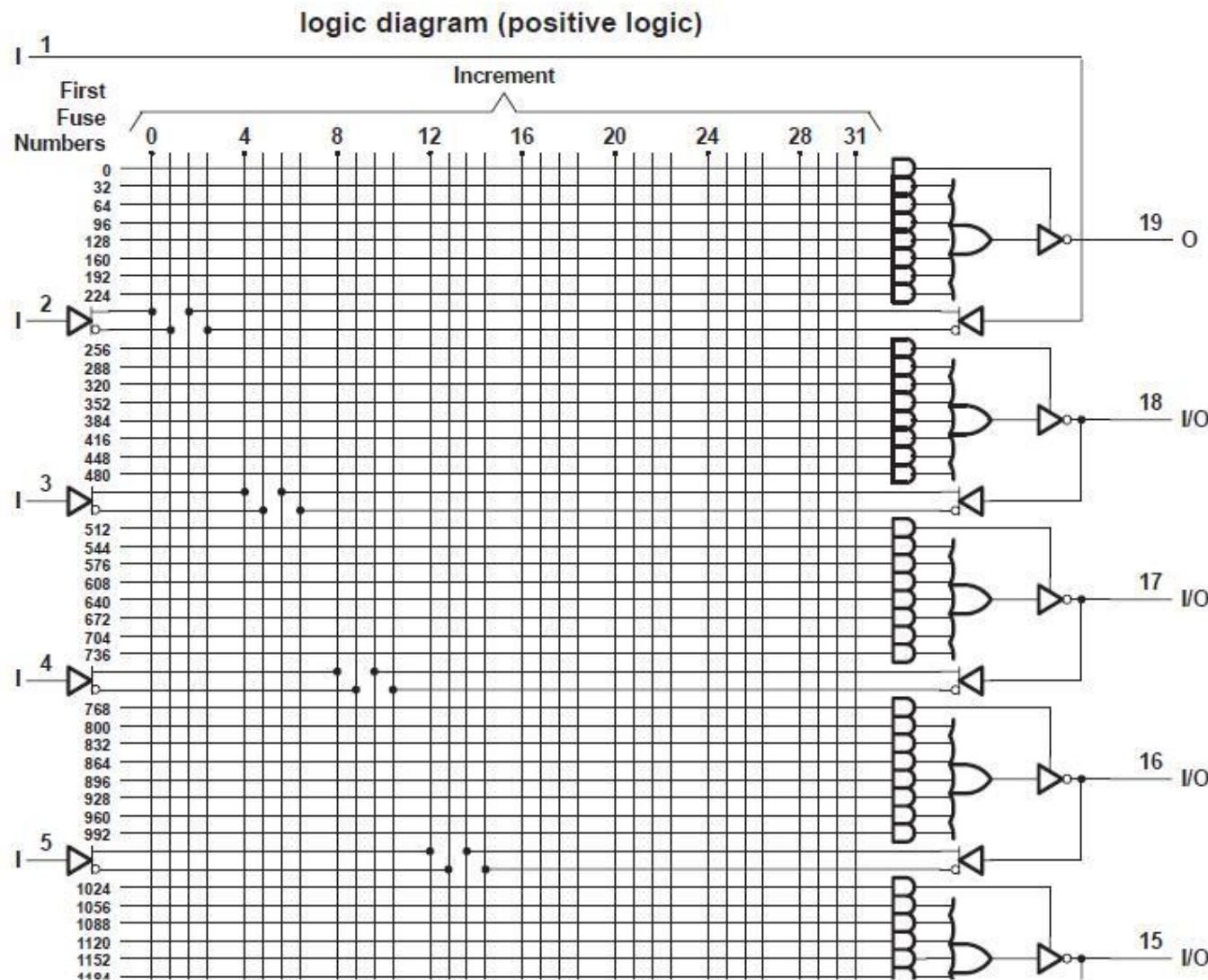
TIBPAL16L8'
C SUFFIX . . . J OR N PACKAGE
M SUFFIX . . . J OR W PACKAGE
(TOP VIEW)



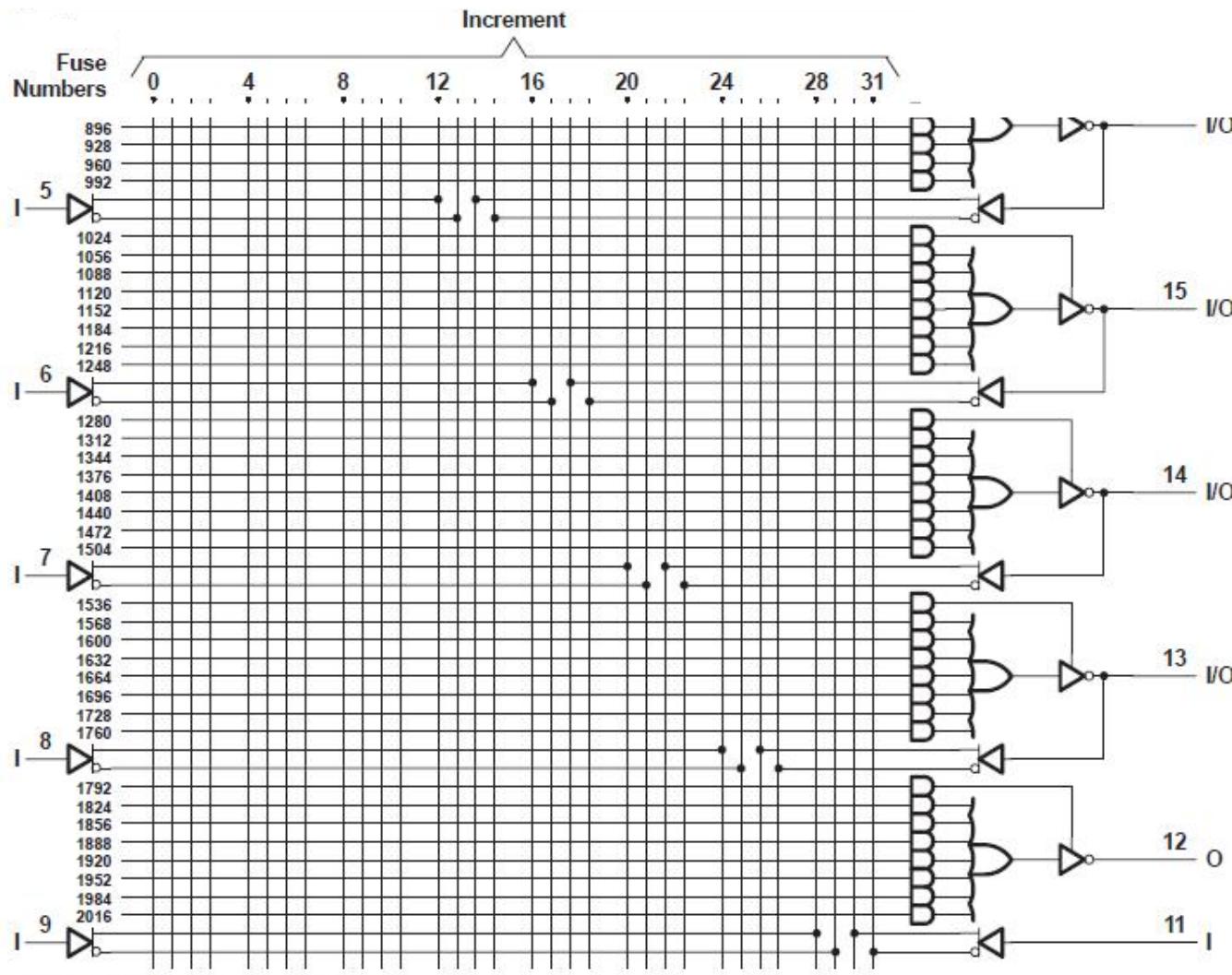
TIBPAL16L8'
C SUFFIX . . . FN PACKAGE
M SUFFIX . . . FK PACKAGE
(TOP VIEW)



Logic Diagram for 16L8 PAL



Logic Diagram for 16L8 PAL

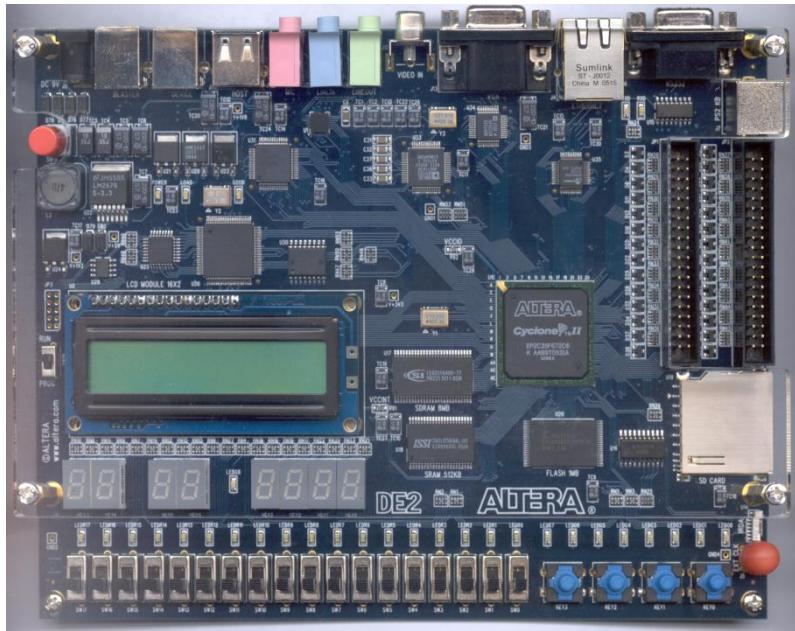


CPE 322

Digital Hardware Design Fundamentals

Electrical and Computer Engineering
UAH

Introduction to Verilog



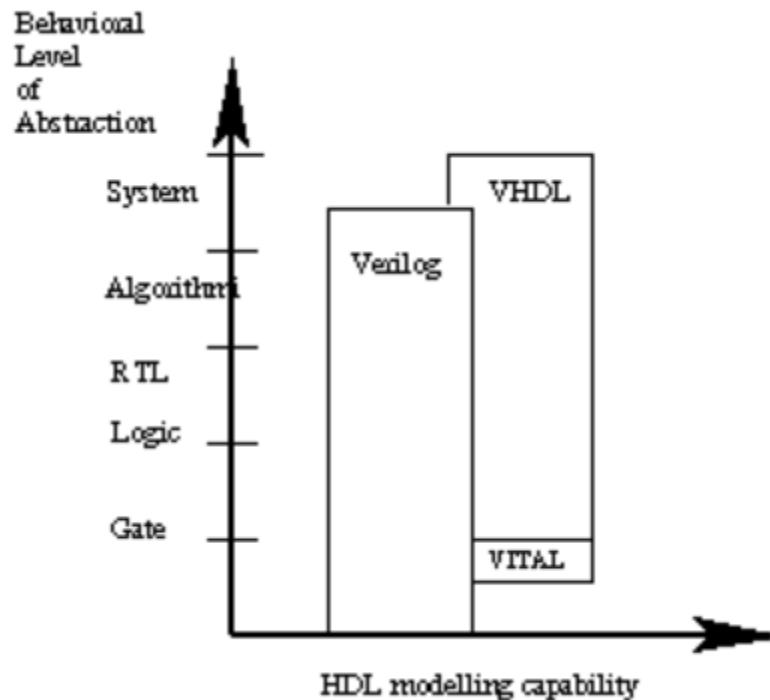
Hardware Description Languages

- Modeling Language that supports
 - Design Entry
 - Simulation
 - Rapid Prototyping of Designs
 - Synthesis (design creation)
- Promotes top-down design methodologies
- Promote the creation of technology independent Intellectual Property

Hardware Description Languages

- Support modeling at different levels of abstraction:
 - Structural (specifying interconnections of the gates)
 - Dataflow (specifying design using Boolean Logic)
 - Equations that specify output signals in terms of input signals.
 - Behavioral (specifying high-level behavior)

Common Modeling Constructs



<http://www.angelfire.com/in/rajesh52/verilogvhdl.html>

Verilog HDL Levels of Abstraction

- Structural Design (hierarchal design approach)
 - Data Flow (equation based)
 - Behavioral Design (modeling using highly abstract language constructs)
-
- **Most often Verilog HDL models contain aspects of all three levels of abstraction!**

Names (Identifiers)

- Must begin with alphabetic or underscore characters a-z A-Z _
- May contain the characters a-z A-Z 0-9 _ and \$
- May use any character by escaping with a backslash (\) at the beginning of the identifier, and terminating with a white space.

Examples	Notes
adder	legal identifier name
XOR	uppercase identifier is unique from xor keyword
\reset*	an escaped identifier (must be followed by a white space)

Verilog Logic Values

- Verilog HDL support 4 Logic Values

Logic Value	Description
0	zero, low, or false
1	one, high, or true
z or Z	high impedance (tri-stated or floating)
x or X	unknown or uninitialized

Verilog Logic Strengths

- Verilog HLD has 8 strengths: 4 driving, 3 capacitive, and 1 high impedance (no strength)

Strength Level	Strength Name	Specification Keyword		Display Mnemonic	
7	Supply Drive	supply0	supply1	Su0	Su1
6	Strong Drive	strong0	strong1	St0	St1
5	Pull Drive	pull0	pull1	Pu0	Pu1
4	Large Capacitive	large		La0	La1
3	Weak Drive	weak0	weak1	We0	We1
2	Med. Capacitive	medium		Me0	Me1
1	Small Capacitive	small		Sm0	Sm1
0	High Impedance	highz0	highz1	Hiz0	Hiz1

Integer Representation

size'base value – sized integer in the specified base

size -- number of bits in the number (optional)

'base -- number of bits in the number (optional)

value

Base	Symbol	Legal Values
binary	b or B	0, 1, x, X, z, Z, ?, _
octal	o or O	0-7, x, X, z, Z, ?, _
decimal	d or D	0-9, _
hexadecimal	h or H	0-9, a-f, A-F, x, X, z, Z, ?, _

Notes: ? Is same as Z or z; _ (underscore) is ignored

Integer Representation (Examples)

Examples	Size	Base	Binary Equivalent
10	unsized	decimal	0...01010 (32-bits)
'o7	unsized	octal	0...00111 (32-bits)
1'b1	1 bit	binary	1
8'hc5	8 bits	hex	11000101
6'hFO	6 bits	hex	110000 (truncated)
6'hF	6 bits	hex	001111 (zero filled)
6'hZ	6 bits	hex	zzzzzz (Z filled)

Verilog Model Structure

```
module module_name (port list );
    port declarations;
    ...
    variable declaration;
    ...
    description of behavior
endmodule
```

```
module module_name (port declarations and list);
    ...
    variable declaration;
    ...
    description of behavior
endmodule
```

Systems are described as a set of modules

Modules

- Provides external interface to other modules and/or the outside world
 - *inputs*, *outputs*, and *inouts*
- Contains internal logic that performs the module's function
- Encapsulates the internal signals, logic, and sub-modules that are referenced by the module.

Verilog Signal Types

- Two Main Types of Signals:
 - “Wire” is a simple connection between two components
 - Should have just one driver
 - Value is not retained unless driven – i.e. no persistence
 - Wire is the default if not specified in port declarations
 - i.e. **input** x; is the same as **input wire** x;
 - “Reg” stores its value until it is updated again
 - Required in behavioral designs that have procedural statements

Verilog Net Data Types

- The ‘wire’ type is an instance of the allowable net types.

Keyword	Functionality
<code>wire or tri</code>	Simple interconnecting wire
<code>wor or trior</code>	Wired outputs OR together
<code>wand or triand</code>	Wired outputs AND together
<code>tri0</code>	Pulls down when tri-stated
<code>tri1</code>	Pulls up when tri-stated
<code>supply0</code>	Constant logic 0 (supply strength)
<code>supply1</code>	Constant logic 1 (supply strength)
<code>trireg</code>	Stores last value when tri-stated (capacitance strength)

The ‘wire’ type is of primary importance in this course.

Verilog Net Data Types

- In general, Net data types are used to connect structural components together.
 - They transfer both logic values and logic strengths.
 - They must be used when
 - A signal is driven by the output of some device.
 - A signal is also declared as an *input* port or an *inout* port.
 - A signal is on the LHS of a continuous assignment statement.

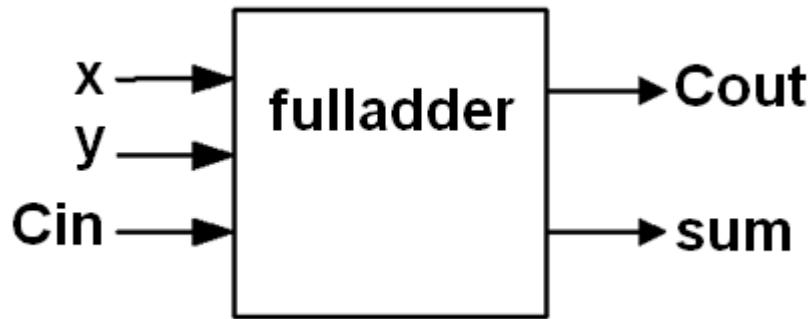
Structural Design

- Structural Modeling is a hierachal design approach that is analogous to schematic capture.
- The design is partitioned into manageable hardware blocks that are called *components*.
- Each subcomponent is modeled separately but can reside in the same file with other component modules.
- Multiple instances of these subcomponents are then be incorporated into the module and interconnected together to create a more complex design.
- The resulting design then becomes a component itself that can be used as one of the building blocks of an even more complex design.

Lower-Level Component Model

(Structural Decomposition)

Full Adder Example



```
module fulladder(x,y,Cin,Cout,sum);
    input x,y,Cin;
    output Cout,sum;

    wire n0,n1,n2; // internal nodes

    xor G0(sum,x,y,Cin);
    and G1(n0,x,y);
    and G2(n1,x,Cin);
    and G3(n2,y,Cin);
    or  G4(Cout,n0,n1,n2);

endmodule
```

```
module fulladder(input x,y,Cin, output Cout,sum);

    wire n0,n1,n2; // internal nodes

    xor G0(sum,x,y,Cin);
    and G1(n0,x,y);
    and G2(n1,x,Cin);
    and G3(n2,y,Cin);
    or  G4(Cout,n0,n1,n2);

endmodule
```

Verilog's Built-in Primitives

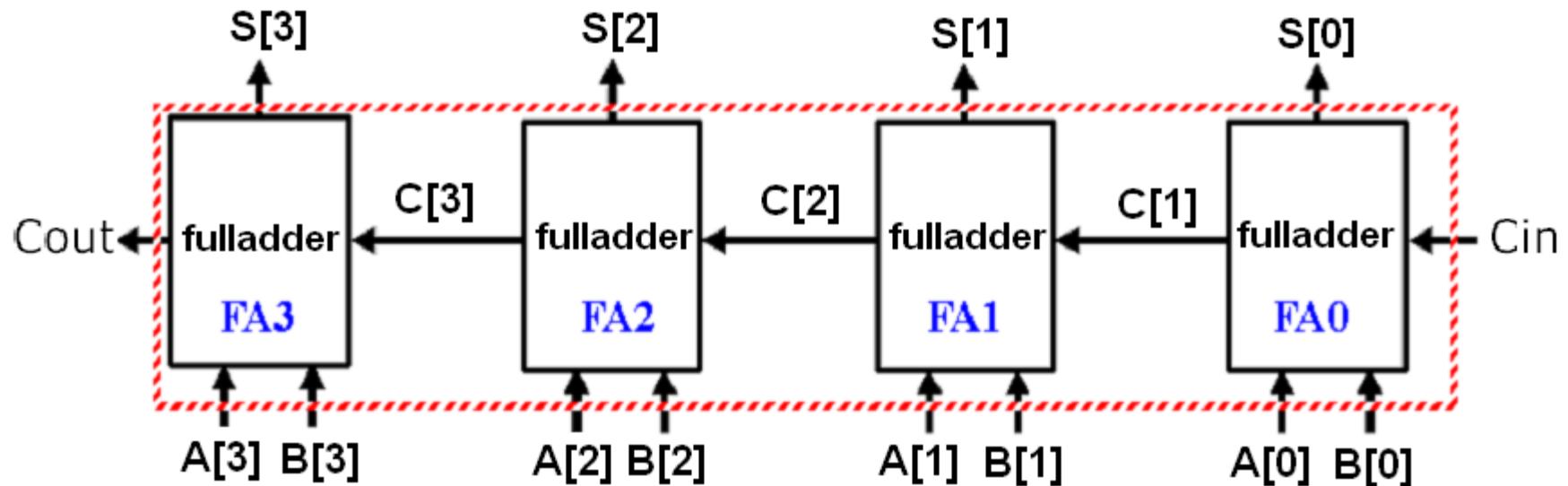
Gate Type		Terminal Order
and or xor	nand nor Xnor	(1_output, 1-or-more_inputs)
buf	not	(1-or-more_outputs, 1_input)
bufif0 bufif1	notif0 notif1	(1_output, 1_input, 1_control)
pullup	pulldown	(1_output)
<u>user-defined-primitives</u>		(1_output, 1-or-more_inputs)

Switch Type		Terminal Order
pmos nmos	rpmos rnmos	(1_output, 1_input, 1_control)
cmos	rcmos	(1_output, 1_input, n_control, p_control)
tran	rtran	(2_bidirectional-inouts)
tranif0 rtranif0	rtranif1 rtranif1	(2_bidirectional-inouts, 1_control)

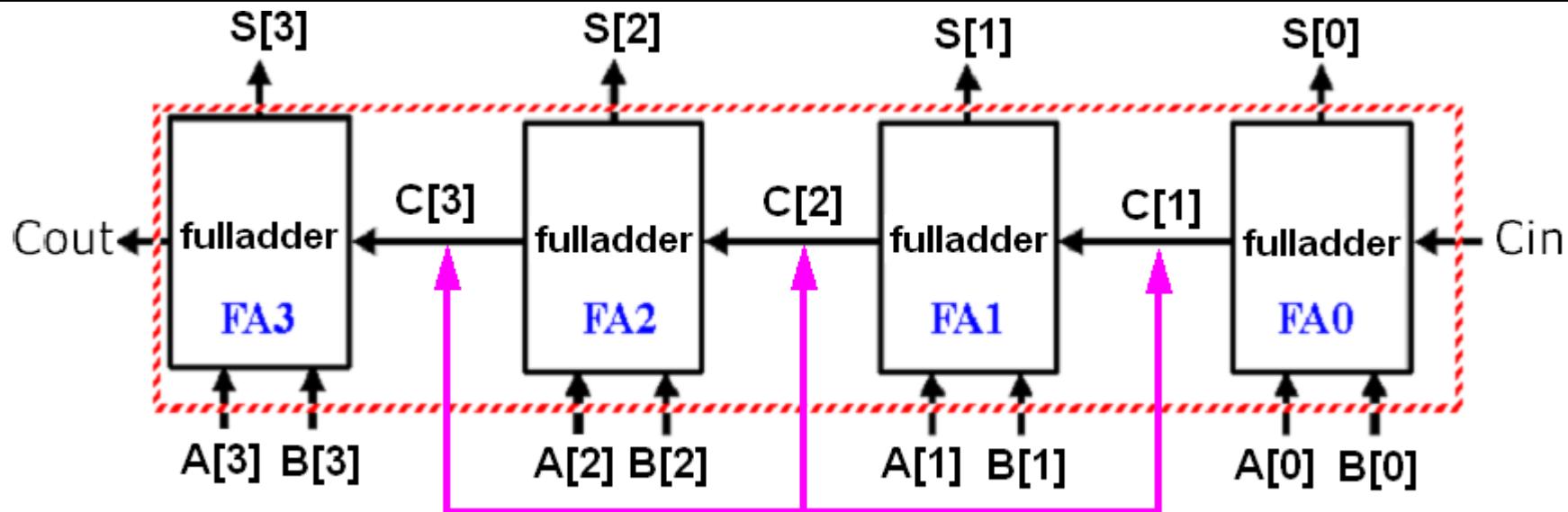
Structural Design

- Components are individually modeled as a separate modules
- A new module is then created that contains this set of components.
- Subcomponent modules are “wired” together by connecting by interconnecting them together using *actual signals*.
 - The input and output signals of the components (component port names) are called *formal signals*
- Instances of a subcomponent module are made by specifying the component type, instance name, and port/signal association parameters.
- Either *Port Order* or *Port Name* association is used to bind the component I/O port *formal signals* to the module’s own internal *actual signals*.

4-bit Adder



4-bit Adder (cont'd)



Wires that are visible only within the *fulladder* module

```
module adder4(input Cin, input [3:0] A,B, output Cout, output [3:0] S);

    wire [3:1] C;
    *
    *
    *

endmodule
```

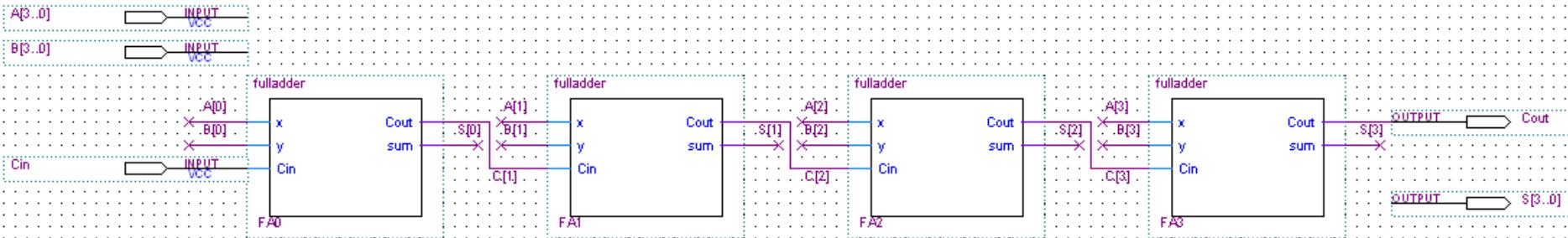
Structural Design

Model Instance Port Order Association

- The *actual* signals in the model instance statement must be placed in the same positions as the corresponding *formal* signals were listed in the component declaration.
 - *module_name instance_name(signal, signal, ...);*
- This is the method most often referenced in the textbook.

4-bit Adder (cont'd)

(port order association method)



```
module adder4(input Cin, input [3:0] A,B, output Cout, output [3:0] S);

wire [3:1] C;

fulladder FA0 (A[0],B[0],Cin,C[1],S[0]);
fulladder FA1 (A[1],B[1],C[1],C[2],S[1]);
fulladder FA2 (A[2],B[2],C[2],C[3],S[2]);
fulladder FA3 (A[3],B[3],C[3],Cout,S[3]);

endmodule
```

Note: *actual* signals must be placed in exactly the same positions in the model instance statement as the corresponding *formal* signals were listed in the component's model declaration.

```
module fulladder(input x,y,Cin, output Cout,sum);

wire n0,n1,n2; // internal nodes

xor G0(sum,x,y,Cin);
and G1(n0,x,y);
and G2(n1,x,Cin);
and G3(n2,y,Cin);
or G4(Cout,n0,n1,n2);

endmodule
```

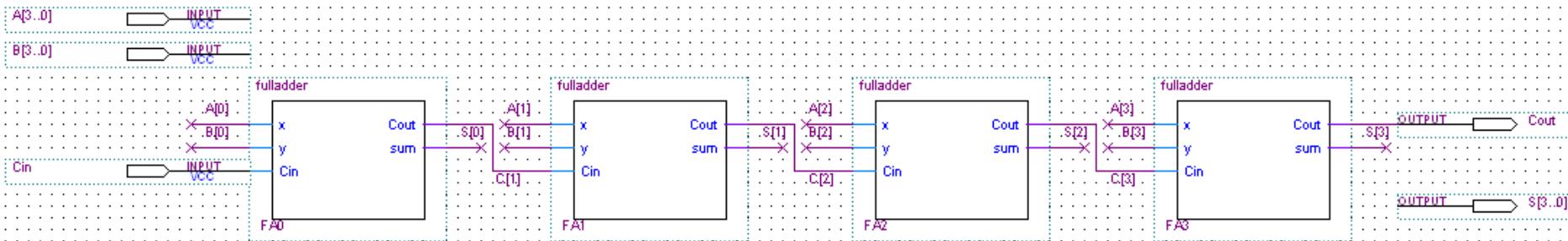
Structural Design

Module Instance Port Name Association

- The *formal signals* are those *inputs/outputs/or inout* signals that are in the port declarations of the given submodule.
- These signals must be liked (wired) to the *actual signals* in the current module.
 - This is done during component instantiation using the `.` operator that precedes the formal signal name and then encapsulating the *actual signal name* in parenthesis in the component instantiation,
 - `module_name instance_name(.port_name(signal), .port_name(signal), ...);`

4-bit Adder Structural Design

(port signal name association method)



```
module adder4(input Cin, input [3:0] A,B, output Cout, output [3:0] S);

    wire [3:1] C;

    fulladder    FA0 (.x(A[0]),.y(B[0]),.Cin(Cin),.Cout(C[1]),.sum(S[0]));
    fulladder    FA1 (.x(A[1]),.y(B[1]),.Cin(C[1]),.Cout(C[2]),.sum(S[1]));
    fulladder    FA2 (.x(A[2]),.y(B[2]),.Cin(C[2]),.Cout(C[3]),.sum(S[2]));
    fulladder    FA3 (.x(A[3]),.y(B[3]),.Cin(C[3]),.Cout(Cout),.sum(S[3]));

endmodule
```

- It does not matter the order that these signals are placed in the port map statement.
- More intuitive for many and less error prone.

Data Flow

- Method primarily employs RTL (register transfer language) Boolean Equations to describe the functionality of the design
- Method is suitable to model both combinational (memory-less) and sequential logic
- Represents a relatively low level of abstraction. Complexity is very high for complex designs.
- Can be combined with structural methodologies to incorporate a hierarchy that will aid in managing this complexity.

Verilog Operators

Verilog Operator	Name	Functional Group
[]	bit-select or part-select	
()	parenthesis	
!	logical negation	logical
~	negation	bit-wise
&	reduction AND	reduction
	reduction OR	reduction
~&	reduction NAND	reduction
~	reduction NOR	reduction
^	reduction XOR	reduction
~^ or ~~	reduction XNOR	reduction
+	unary (sign) plus	arithmetic
-	unary (sign) minus	arithmetic
{ }	concatenation	concatenation
{()}	replication	replication
*	multiply	arithmetic
/	divide	arithmetic
%	modulus	arithmetic
+	binary plus	arithmetic
-	binary minus	arithmetic

Verilog Operator	Name	Functional Group
<<	shift left	shift
>>	shift right	shift
>	greater than	relational
>=	greater than or equal to	relational
<	less than	relational
<=	less than or equal to	relational
==	logical equality	equality
!=	logical inequality	equality
==	case equality	equality
!=	case inequality	equality
&	bit-wise AND	bit-wise
^	bit-wise XOR	bit-wise
~^ or ~~	bit-wise XNOR	bit-wise
	bit-wise OR	bit-wise
&&	logical AND	logical
	logical OR	logical
?:	conditional	conditional

Verilog Operators Order of Precedence

Operator	Precedence
+,-,! ,~ (unary)	Highest
* , / , %	
+,- (binary)	
<<., >>	
<,, <=,, >,, >=	
=,, ==,, !=	
====,, !==	
&,, ~&	
^,, ^~	
,, ~	
&&	
?:	Lowest

Continuous Assignment Statements

(Implicit form)

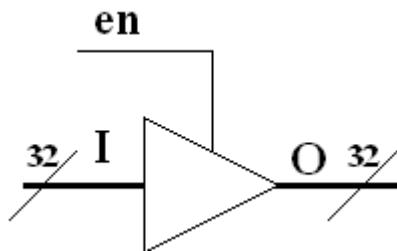
- *net_declaration* = expression;
 - Example:
 - wire sum = x ^ y ^ Cin;

Continuous Assignment Statements (Explicit form)

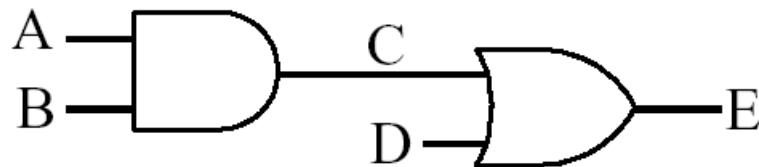
- Format: **assign** net = expression;
- Can use conditional operator ?
 - <condition> ? <if true> : <else>;
If the Condition is true, the value of <if true> will be taken,
otherwise the value of <else> will be taken.

Example:

assign O = en ? I : 32'bz;



Data Flow Representation



Continuous Assignment Statements

```
assign C = A & B;
```

```
assign E = C | D;
```

Order of continuous assignment statements is not important

```
assign C = A & B;
```

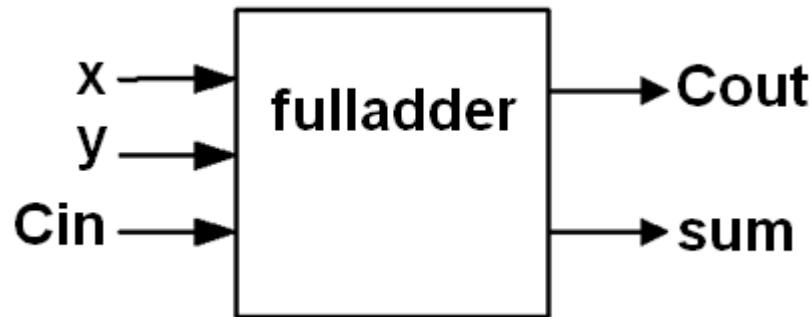
```
assign E = C | D;
```

- These signal are assumed to be evaluated by the Verilog HDL circuit synthesizer or simulator concurrently.

Low-Level Component Model

(Data Flow Example)

Full Adder Example



```
module fulladder(input x,y,Cin,output Cout,sum);  
    assign sum = x ^ y ^ Cin;  
    assign Cout = (x & y) | (x & Cin) | (y & Cin);  
endmodule
```

Behavioral Modeling using Procedural Constructs

- Two Procedural Constructs
 - **initial** Statement
 - **always** Statement
- **initial** Statement : Executes only once
- **always** Statement : Executes in a loop
- Example:

...
initial begin
Sum = 0;
Carry = 0;
end
...

...
always @(A or B) begin
Sum = A ^ B;
Carry = A & B;
end
...

Event Control

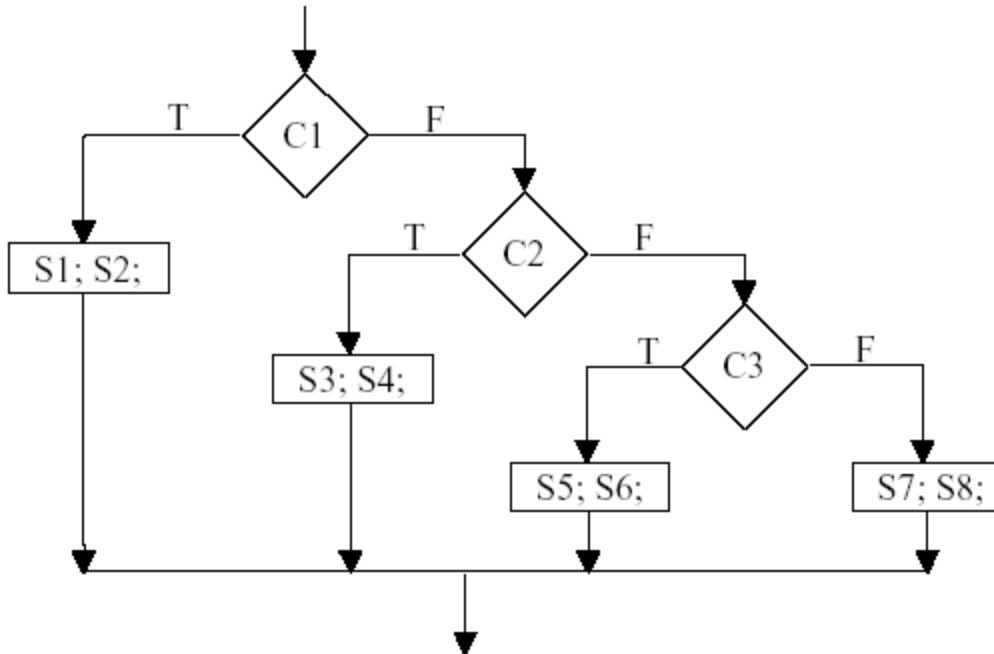
- Event Control
 - Edge Triggered Event Control
 - Level Triggered Event Control
- Edge Triggered Event Control
 - @ (posedge CLK) //Positive Edge of CLK

Curr_State = Next_state;

@ negedge	@ posedge
1 → x	0 → x
1 → z	0 → z
1 → 0	0 → 1
x → 0	x → 1
z → 0	z → 1

- Level Triggered Event Control
 - @ (A or B) //change in values of A or B
- Out = A & B;

IF Statements in Verilog HDL



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

Case Statement in Verilog HDL

The case statement has the general form:

case (expression)

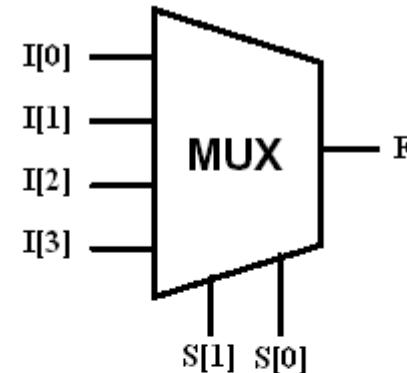
choice1 : procedural statement;

choice2 : procedural statement;

 • • •

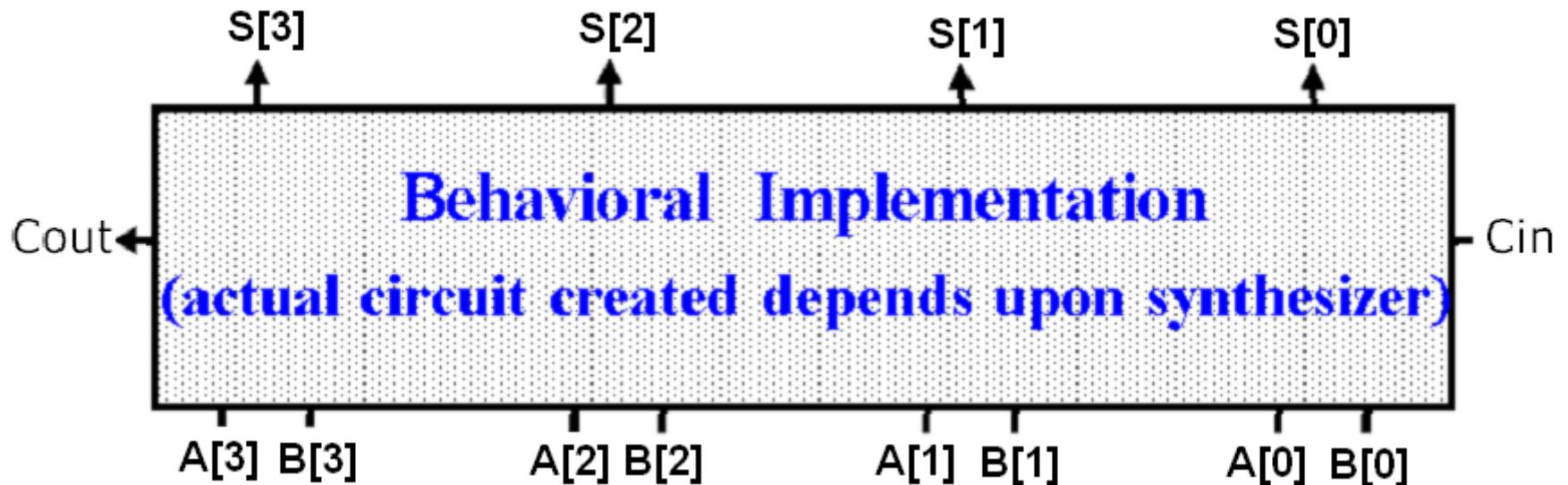
default : procedural statement;

endcase



```
case (S)
    0 : F = I[0];
    1 : F = I[1];
    2 : F = I[2];
    3 : F = I[3];
    default : F = 1'bx;
endcase
```

Behavioral Implementation



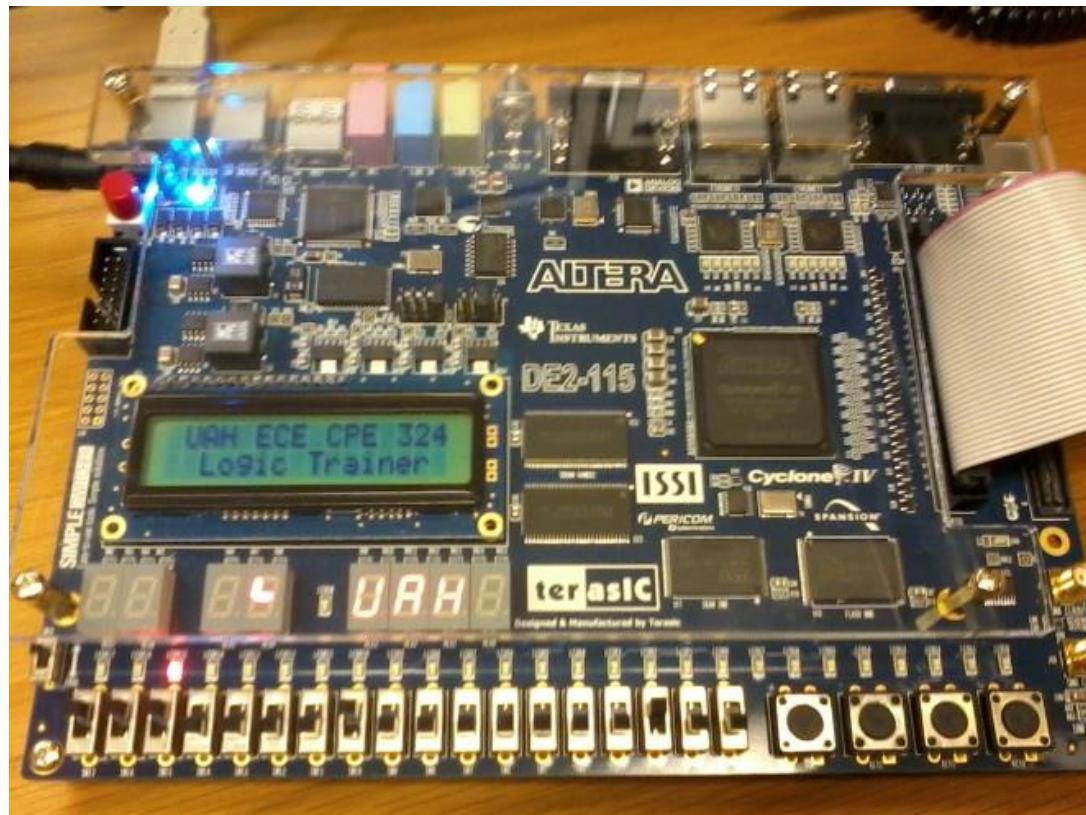
```
module adder4(input Cin, input [3:0] A,B, output Cout, output [3:0] S);  
  always @ (A or B or Cin)  
    {Cout,S} = A + B + Cin;  
endmodule
```

CPE 322

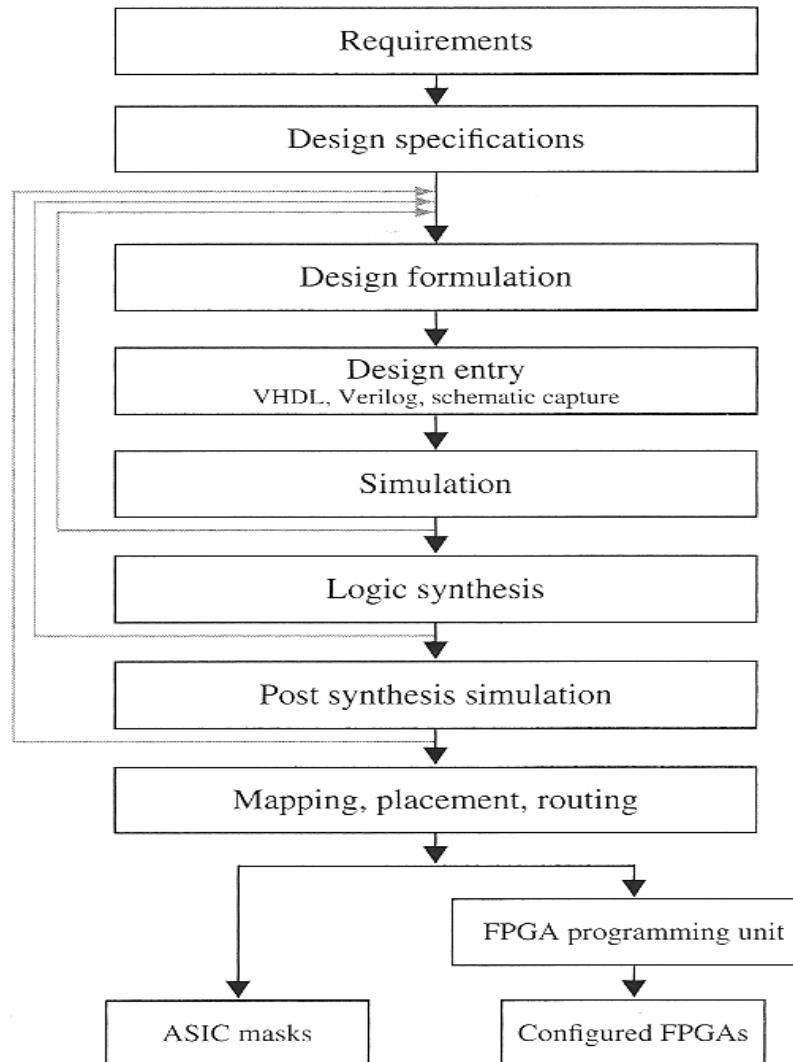
Digital Hardware Design Fundamentals

Electrical and Computer Engineering
UAH

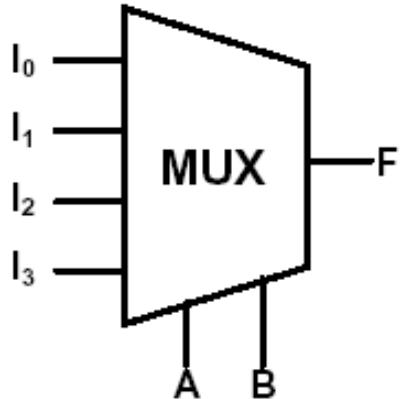
Introduction to Verilog Part II



Steps in Modern Digital System Design



Verilog Models for a MUX



Structural Design using AND, OR, NOT primitives

```
module MUX(input A, B, I0, I1, I2, I3, output F);

    wire A_,B_; // A negation, B negation nodes
    wire n0,n1,n2,n3; // internal nodes of AND terms

    not G0 (A_,A);
    not G1 (B_,B);
    and G2 (n0,A_,B_,I0);
    and G3 (n1,A_,B,I1);
    and G4 (n2,A,B_,I2);
    and G5 (n3,A,B,I3);
    or  G6 (F,n0,n1,n2,n3);

endmodule
```

Data flow design using continuous (concurrent signal) assignment statement

```
module MUX(input A, B, I0, I1, I2, I3, output F);

    assign F = (~A & ~B & I0) | (~A & B & I1) |
               (A & ~B & I2) | (A & B & I3);

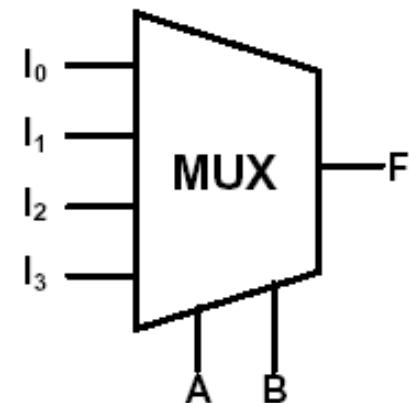
endmodule
```

Verilog Models for a MUX

- Conditional continuous (concurrent signal) assignment statement format:
 - `<condition> ? <if true> : <else>;`
If the Condition is true, the value of `<if true>` will be taken, otherwise the value of `<else>` will be taken.

Data flow design of MUX using conditional continuous (concurrent signal) assignment statement

```
module MUX(input A, B, I0, I1, I2, I3, output F);
    assign F = (A) ? (B ? I3 : I2) : (B ? I1 : I0);
endmodule
```



Behavioral Modeling using Procedural Constructs

- Two Procedural Constructs
 - **initial** Statement
 - **always** Statement
- **initial** Statement : Executes only once
- **always** Statement : Executes in a loop
- Example:

```
...
initial
  begin
    Sum = 0;
    Cout = 0;
  end
...
```

```
...
always @(A or B)
  begin
    Sum = A ^ B;
    Cout = A & B;
  end
...
```

Event Control

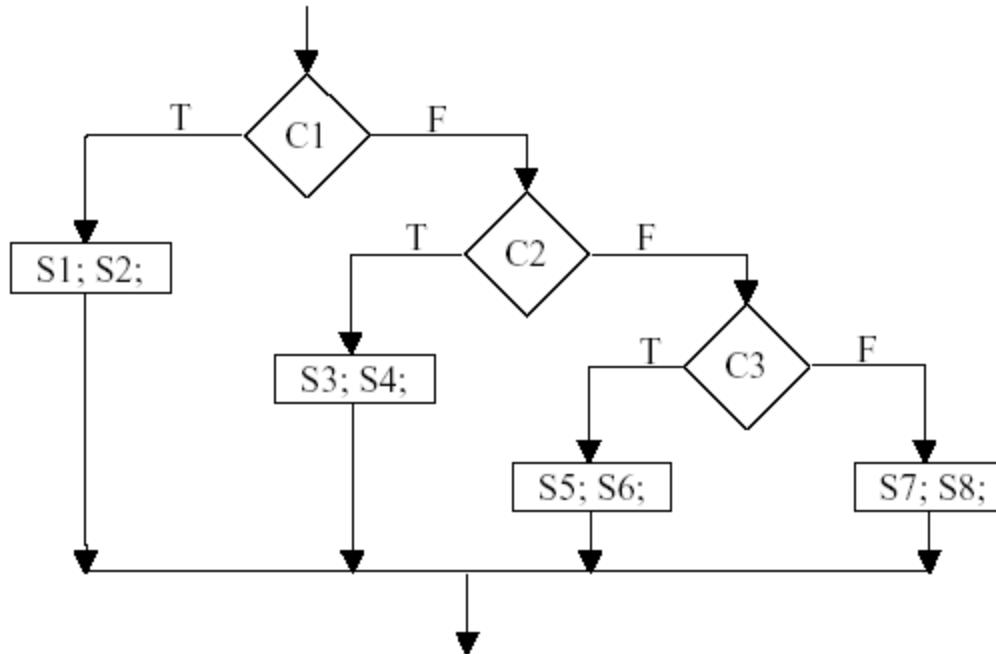
- Event Control
 - Edge Triggered Event Control
 - Level Triggered Event Control
- Edge Triggered Event Control
 - @ (posedge CLK) //Positive Edge of CLK

Curr_State = Next_state;

@ negedge	@ posedge
1 → x	0 → x
1 → z	0 → z
1 → 0	0 → 1
x → 0	x → 1
z → 0	z → 1

- Level Triggered Event Control
 - @ (A or B) //change in values of A or B
- Out = A & B;

IF Statements in Verilog HDL



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

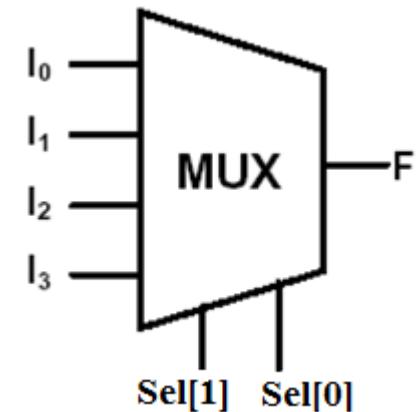
Verilog Models for a MUX

Data flow design of MUX using nested **if** statement placed inside an **always** block

```
module MUX(input [1:0] Sel, I0, I1, I2, I3, output reg F);

  always @ (Sel, I0, I1, I2, I3)
    begin
      if      (Sel == 2'b00) F = I0;
      else if (Sel == 2'b01) F = I1;
      else if (Sel == 2'b10) F = I2;
      else if (Sel == 2'b11) F = I3;
    end

endmodule
```



Verilog Models for a MUX

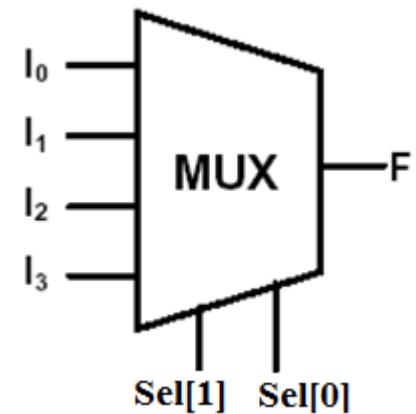
- The case statement has the general form:

```
case expression
    choice1 : sequential statements1
    choice2 : sequential statements2
    ...
    [default : sequential statements]
endcase
```

Data flow design of MUX using a **case** statement placed inside an **always** block

```
module MUX(input [1:0] Sel, I0, I1, I2, I3, output reg F);

    always @ (Sel, I0, I1, I2, I3)
        case (Sel)
            2'b00 : F = I0;
            2'b01 : F = I1;
            2'b10 : F = I2;
            2'b11 : F = I3;
        endcase
    endmodule
```

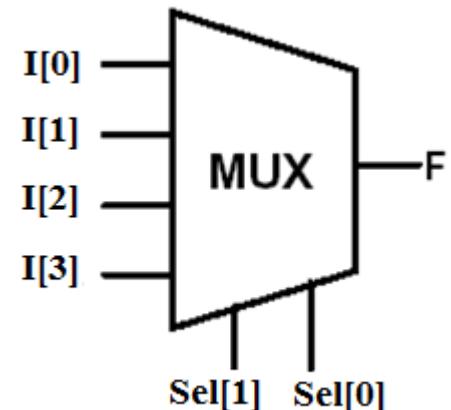


Verilog Models for a MUX

Data flow design of MUX using array declaration of the two sets of inputs

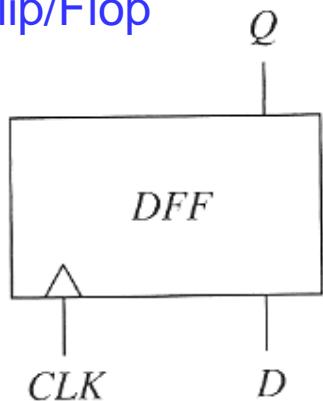
- The **Sel** input is used as the index into the **I** inputs.

```
module MUX(input [1:0] Sel, input [3:0] I, output reg F);  
  
    always @ (Sel, I)  
        begin  
            F = I[Sel];  
        end  
  
    endmodule
```



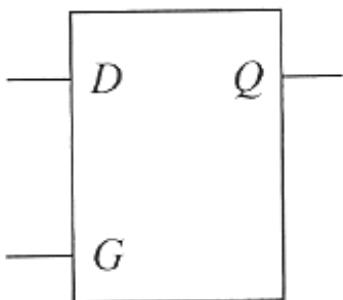
Modeling Simple Sequential Elements

D Flip/Flop



```
module DFF (input D, CLK, output reg Q);  
  
    always @ (posedge CLK)  
        begin  
            Q <= D;  
        end  
  
endmodule
```

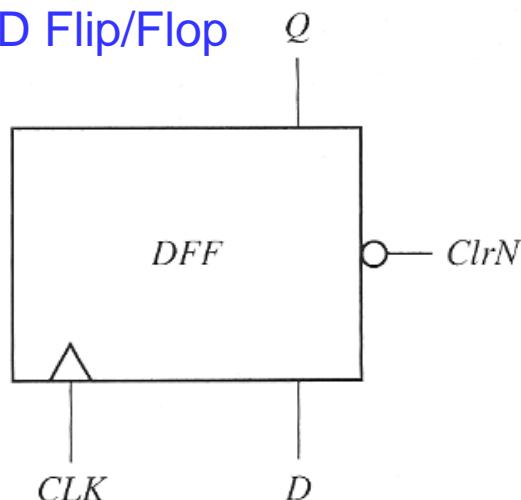
D Latch



```
module DLatch (input D, G, output reg Q);  
  
    always @ (D, G)  
        begin  
            if (G) Q <= D;  
        end  
  
endmodule
```

Modeling Simple Sequential Elements

D Flip/Flop



```
module DFF (input D, CLK, ClrN, output reg Q);  
  
    always @ (posedge CLK)  
        begin  
            if (~ClrN)  
                Q <= 0;  
            else  
                Q <= D;  
        end  
  
endmodule
```

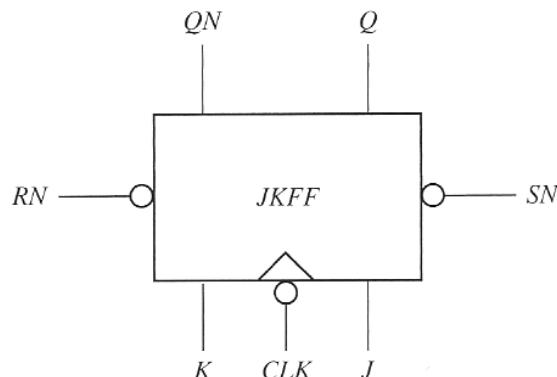
Synchronous ClrN

- Note: should not mix level triggered and edge triggered inputs on sensitivity lists

```
module DFF (input D, CLK, ClrN, output reg Q);  
  
    always @ (posedge CLK, negedge ClrN)  
        begin  
            if (~ClrN)  
                Q <= 0;  
            else  
                Q <= D;  
        end  
  
endmodule
```

Asynchronous ClrN

Modeling Simple Sequential Elements



Asynchronous SN and RN

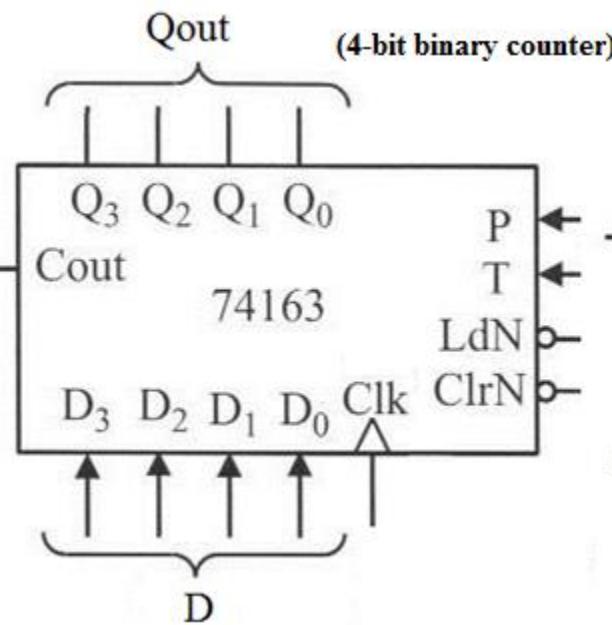
```
module JKFF (input SN, RN, J, K, CLK, output Q, QN);
    reg Qint;
    always @ (negedge CLK, negedge RN, negedge SN)
        begin
            if (~RN)
                #8 Qint <= 0;
            else if (~SN)
                #8 Qint <= 1;
            else
                Qint <= #10 ((J & ~Qint) | (~K & Qint));
        end
        assign Q = Qint;
        assign QN = ~Qint;
    endmodule
```

- Text incorrectly mixed level triggered and edge triggered inputs on the same sensitivity list which is not recommended practice for asynchronous case

Synchronous SN and RN

```
module JKFF (input SN, RN, J, K, CLK, output Q, QN);
    reg Qint;
    always @ (negedge CLK)
        begin
            if (~RN)
                #8 Qint <= 0;
            else if (~SN)
                #8 Qint <= 1;
            else
                Qint <= #10 ((J & ~Qint) | (~K & Qint));
        end
        assign Q = Qint;
        assign QN = ~Qint;
    endmodule
```

Modeling more Complex Sequential Elements



74163 FULLY SYNCHRONOUS COUNTER

Control Signals			Next State				
ClrN	LdN	P•T	Q ₃ [*]	Q ₂ [*]	Q ₁ [*]	Q ₀ [*]	
0	X	X	0	0	0	0	(clear)
1	0	X	D ₃	D ₂	D ₁	D ₀	(parallel load)
1	1	0	Q ₃	Q ₂	Q ₁	Q ₀	(no change)
1	1	1	present state + 1				(increment count)

If T=1, the counter generates a carry, C_{out}, when in state 15; consequently

$$C_{out} = Q_3 \ Q_2 \ Q_1 \ Q_0 \ T$$

```
// 74163 FULLY SYNCHRONOUS COUNTER
module c74163(input LdN, ClrN, P, T, Clk, input reg [3:0] D, output Cout, output [3:0] Qout);

reg [3:0] Q;

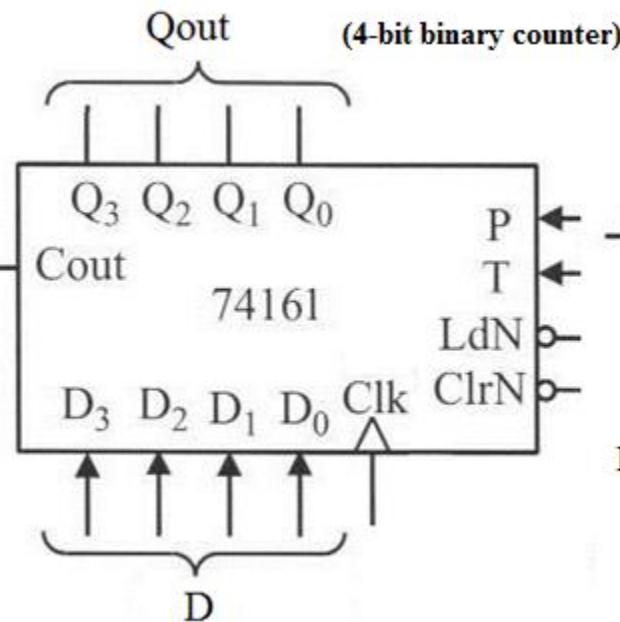
assign Qout = Q;
assign Cout = Q[3] & Q[2] & Q[1] & Q[0] & T;

always @ (posedge Clk)
begin
if (~ClrN) Q <= 4'b0000;
else if (~LdN) Q <= D;
else if (P & T) Q <= Q + 1;
end
endmodule
```

Modeling more Complex Sequential Elements

- Synchronous inputs, such as the ClrN signal on the 74163 take effect after the active edge of the clock.
- Asynchronous inputs are level triggered not edge triggered.
- To ensure correct simulation properties, an always block should always contain the same type of triggering in the sensitivity list
 - If one signal is edge-sensitive, then all signals should be edge sensitive.

Modeling more Complex Sequential Elements



74161 SYNCHRONOUS COUNTER
with asynchronous Clear (ClrN)

Control Signals			Next State			
ClrN	LdN	P•T	Q3*	Q2*	Q1*	Q0*
0	X	X	0	0	0	0
1	0	X	D3	D2	D1	D0
1	1	0	Q3	Q2	Q1	Q0
1	1	1	present state + 1			

If T=1, the counter generates a carry, C_{out} , when in state 15; consequently

$$C_{out} = Q_3 \ Q_2 \ Q_1 \ Q_0 \ T$$

```
// 74161 SYNCHRONOUS COUNTER with asynchronous clear
module c74163(input LdN, ClrN, P, T, Clk, input reg [3:0] D, output Cout, output [3:0] Qout);

reg [3:0] Q;

assign Qout = Q;
assign Cout = Q[3] & Q[2] & Q[1] & Q[0] & T;

always @(posedge Clk, negedge ClrN)
begin
if (~ClrN) Q <= 4'b0000;
else if (~LdN) Q <= D;
else if (P & T) Q <= Q + 1;
end
endmodule
```

Blocking versus Nonblocking Procedural Assignment Statements

Blocking Assignments

```
module dummy (input Trigger,  
              output reg [15:0] sum=0);  
  
    reg [15:0] var1=1,var2=2,var3=3;  
  
    always @ (Trigger)  
    begin  
        var1 = var2 + var3;  
        var2 = var1;  
        var3 = var2;  
        sum = var1 + var2 + var3;  
    end  
  
endmodule
```

Sum = 15 (16'd15)

Nonblocking Assignments

```
module dummy (input Trigger,  
              output reg [15:0] sum=0);  
  
    reg [15:0] var1=1,var2=2,var3=3;  
  
    always @ (Trigger)  
    begin  
        var1 <= var2 + var3;  
        var2 <= var1;  
        var3 <= var2;  
        sum <= var1 + var2 + var3;  
    end  
  
endmodule
```

Sum = 6 (16'd6)

Data Flow Representation Verilog HDL

8421 BCD to Excess3 Code Converter (Mealy FSM)

```
// bcd to excess 3 converter
// Mealy Implementation
// Data Flow Model
module bcd_ex3_mealy(input X,CLK,output Z);

reg Q1=0,Q2=0,Q3=0;

// FF State Update Portion of design
// Active only on rising edge of clock
always @(posedge CLK)
begin
    Q1 <= ~Q2;
    Q2 <= Q1;
    Q3 <= (Q1 & Q2 & Q3) | (~X & Q1 & ~Q3) | (X & ~Q1 & ~Q2);
end

// Output Equation -- Continuous Assignment that is
// a function only of the state variables Q1,Q2,Q3
assign Z = (~X & ~Q3) | (X & Q3);

endmodule
```

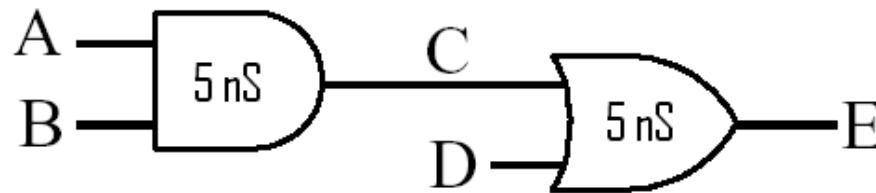
$$Q_1^+ = Q_2'$$

$$Q_2^+ = Q_1$$

$$Q_3^+ = Q_1 Q_2 Q_3 + X' Q_1 Q_3' + X Q_1' Q_2'$$

$$Z = X' Q_3' + X Q_3$$

Data Flow Representation



- If you are using Verilog HDL for simulation, then you can assign gate and wiring type delays:
 - Inertial (delays to operations, i.e. gate type delays)
 - Transport (delays between operations, i.e. wire delays)
- But user-specified delays are ignored if you are using Verilog HDL for synthesis (i.e. creating circuitry)!

Types of Delay

- Transport: Signal will assume its new value after the specified delay time.
- Inertial: Signal will assume its new value after specified delay time if the signal is in the same state at least as long as the specified propagation delay.

Setting the Timing Scale

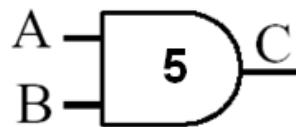
- Unless set by the ``timescale` construct units of time in Verilog are generic and measured in ‘time units’
- To associate these generic time-units with physical time use the *timescale* construct
 - ``timescale` time-unit/time-precision
 - Example ``timescale 1ns/100 ps`
 - Sets the generic time unit specified in the Verilog file to equal 1 ns
 - Sets the precision of time to be 100ps (0.1 ns)
 - Thus a time value of 10.52765 ns would be rounded to 10.5 ns

Inertial Delay

(using Continuous Assignment Statements)

- Provides for specification of the propagation delay and minimum input pulse width, i.e. ‘inertia’ of output:
 - Continuous Assignment statement example

```
wire <#delay> [net_name];  
or  
assign <#delay> [net] = Boolean Expression;
```



Examples below have the same timing – delaying any change in output **C** by 5 time units

```
wire C;  
assign #5 C = A & B; //delay in the continuous assignment  
  
wire #5 C = A & B; //delay in the implicit assignment  
  
wire #5 C; //delay in the wire declaration  
assign C = A & B;
```

Inertial Delay (using parametrized primitive instantiations)

- By default the timing delay for Verilog gate primitives is always zero.
- Rising and falling delays can be specified during primitive instantiation using the #(rise, fall) delay operator.
- Turn-off delay (transition to high impedance state Z) for tri-state primitives can also be defined by using the #(rise, fall, off) delay operator.

```
and #(10) G1(C,A,B); //rise=10,fall=10
and #(10,11) G2(C,A,B); //rise=10,fall=11
notif0 #(10,11,25) G3(c,d,en) //rise=10,fall=11,
                                //off=25 (when en==0)
```

Inertial Delay

- For each rise, fall, and turn-off delay times:
 - *Minimum*, Typical, and Maximum delay values can be entered using a colon to separate the values as shown below:

```
//min:typ:max values defined for the (rise, fall) delays  
and #(6:10:12, 9:11:13) G2(C,A,B);  
or  
assign #(6:10:12, 9:11:13) C = A & B;
```

- ModelSim™ the Typical delay values are normally used.

Min/Max/Typical Delay Selection in ModelSim™

- You can explicitly specify which delays are to be used at the start of the simulation.
- To do this utilize the following command line options
 - +maxdelays {for utilization of maximum delay values}
 - +mindelays {for utilization of minimum delay values}
 - +typdelays {for utilization of typical delay values -- the default}
- These can be placed on the vsim command line or by selecting *max*, *min*, or *typ* under *Delay Selection* pull down widget of the *Verilog* tab of the *Start Simulation Menu*.

Adding Delays to a module

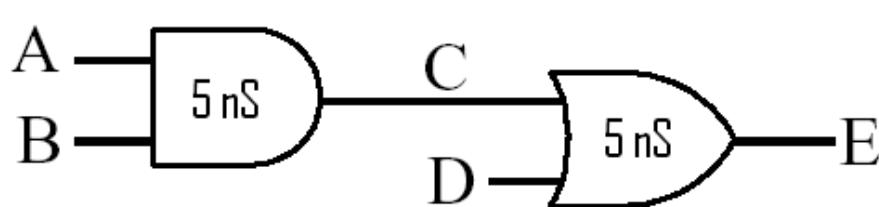
- By default the timing inside a module is controlled by the design of the module itself.
- Many modules are created that have parameterized delays similar to the #(rise,fall) delay operator used with gate primitives.
- This is accomplished using the **parameter** keyword to create delay variables.
- Note that parameters can also be used to change other scalar values in the module.

Parameters

Parameters are run-time constant for storing integers, real numbers, time, delays, or ASCII strings. Parameters may be redefined for each instance of a module.

- Syntax parameter declaration
 - **parameter** identifier1 = constant_expression,
identifier2 = constant_expression, ..., ;
- Syntax parameter redefinition
 - *Explicit*
`defparam hierarchy_path.parameter_name = value;`
 - *Implicit Parameter Redefinition*
`module_name #(value) instance_name (signals);`

Changing Timing Parameters of Modules

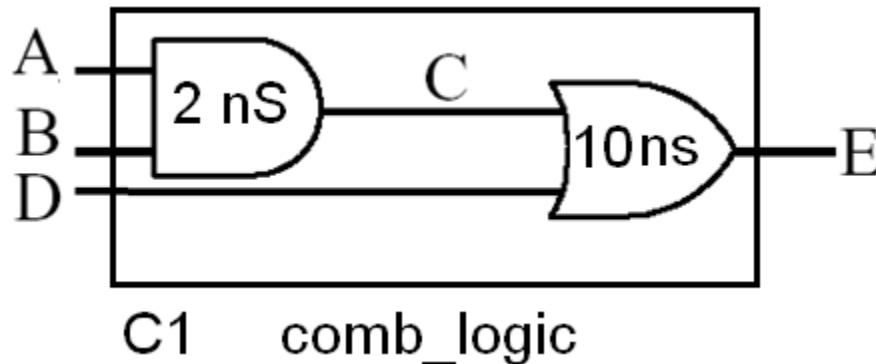


for
`timescale 1ns/100 ps;

```
module comb_logic(input A,B,D, output E);
parameter and_delay = 5, // default rise/fall delays of 5
           or_delay = 5; // for both internal gates
assign #or_delay C = A & B;
assign #and_delay E = C & D;
endmodule
```

- When the module is instantiated then you can choose to override the delay values using the `#(first_parameter, second_parameter)` notation

Changing Timing Parameters of Modules



for
'timescale 1ns/100 ps;

- For example to specify a 2ns rise/fall for the internal OR gate and a 10ns rise/fall for the internal AND gate you could instantiate the component as follows:

```
comb_logic #(2,10) C1(A,B,D,E);
```

Assigning Inertial Delay to Complex Combinational Logic that is Modeled Behaviorally

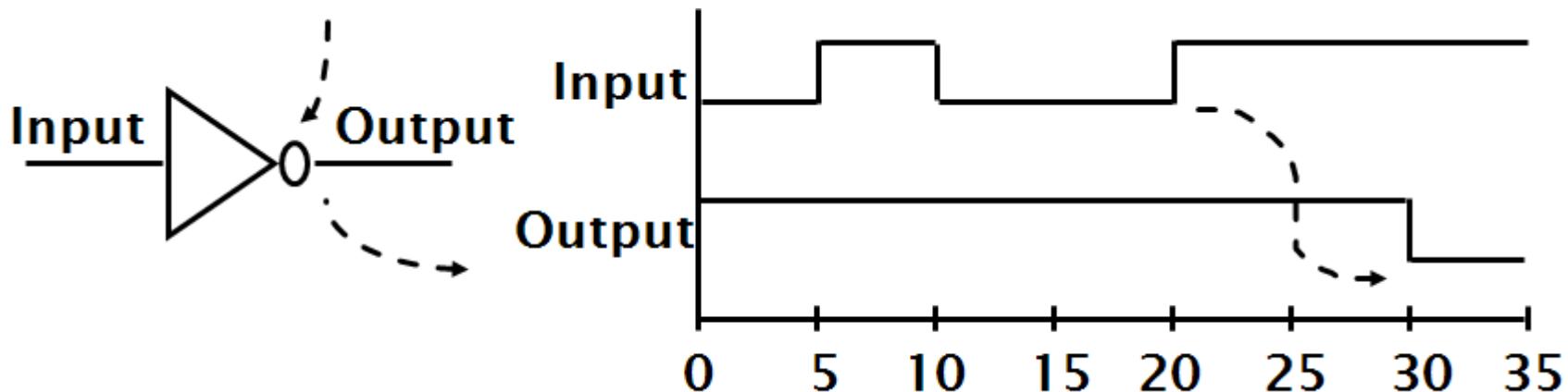
Best Practices:

- When using an **always** block to model complex combinational logic behaviorally
 - Model this logic without assigning delay parameters (i.e. zero delay logic).
 - The outputs from this block should then be passed through as inputs to continuous assignment statements.
 - Assign the appropriate delays to these statements.
- The delay values should then reflect true inertial type delays.

Inertial Delay Example

```
assign #10 Output = ~Input; // using continuous assignments  
or  
not #(10) I1 (Output, Input); // using primitives
```

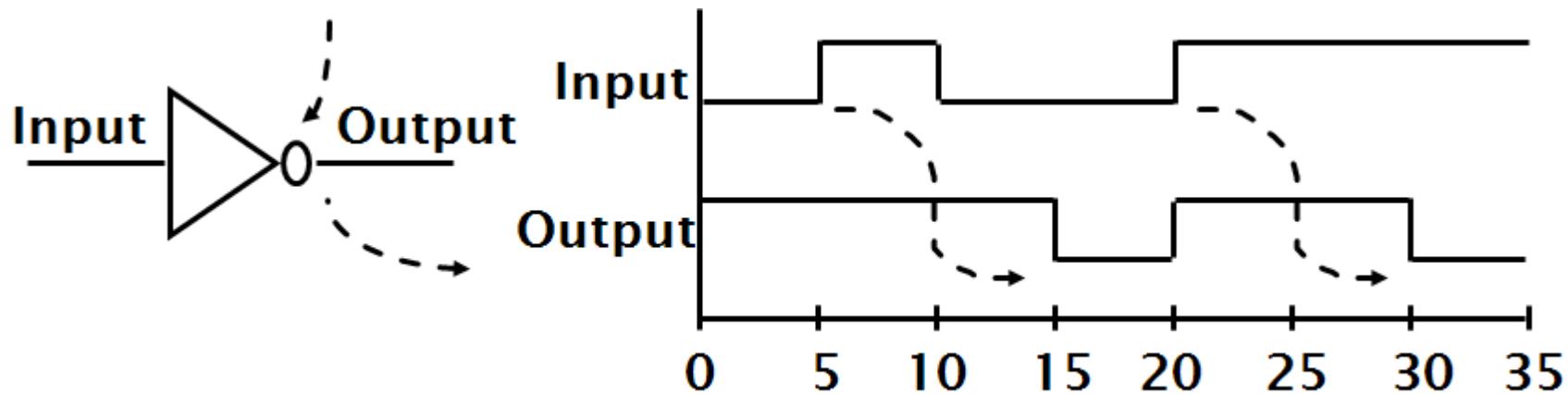
- Example of gate with ‘inertia’ smaller than propagation delay
 - e.g. Inverter with propagation delay of 10ns which suppresses pulses shorter than 5ns



Transport Delay

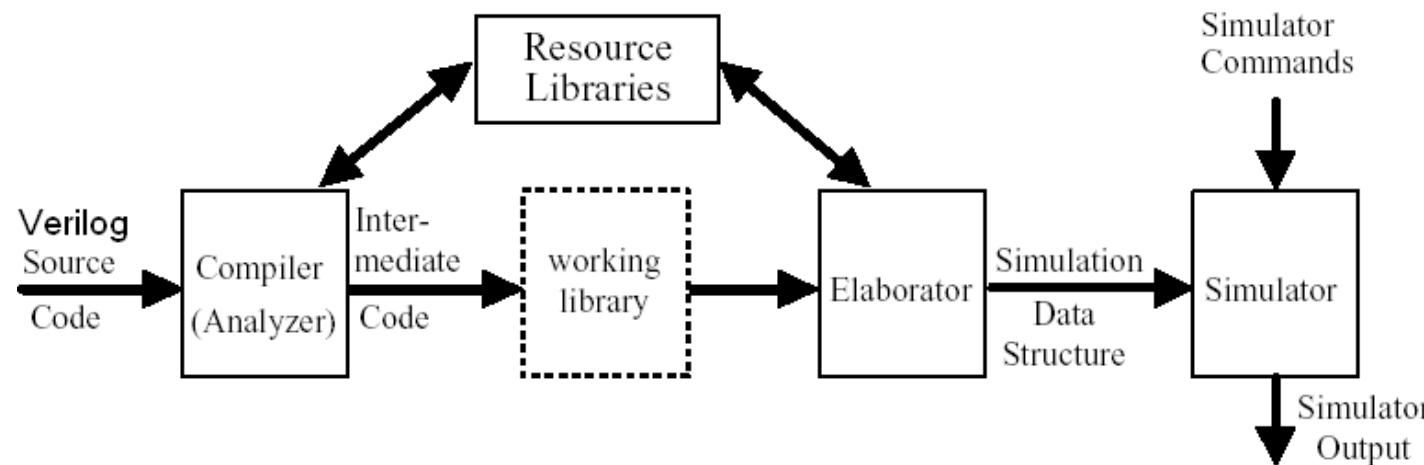
- Best practices for modeling Transport delays in Verilog is to place the delay on the RHS of a non-blocking procedural assignment statement.

```
// Transport delay example
always @(Input)
    Output <= #10 ~Input;
```



Compilation & Simulation of Verilog HDL Models

- Compiler (Semantic Analyzer) – checks the Verilog HDL source model
 - does it conforms with Verilog HDL syntax and semantic rules
 - are references to libraries correct
- Elaboration
 - creates an intermediate form used by a simulator or by a synthesizer
 - create ports, allocate memory storage, create interconnections, ...
 - establish mechanism for executing of Verilog HDL processes



Logic Simulation

- **Discrete Event Simulation** -- where the passage of time is simulated in discrete event driven steps.
 - **Simulation time** – the time value maintained by the simulator to model the actual time it would take for the circuit being simulated
 - Event driven
 - **Update Event** – an actual change in value of a net or variable in the circuit being simulated.
 - **Evaluation Event** – evaluation of a possible future change in value of a net or variable.
 - All process that are sensitive to the update event are evaluated in an arbitrary order.

Simulation

- **Event Queue** –list of pending events that is ordered based upon increasing simulation time that the event is to occur.
 - The Verilog event queue is segmented into five different regions
- **Scheduling an event** – the placement of an event on one of the regions of the event queue.
- **Simulation Cycle** – the processing of all active events.

Verilog Event Queue Regions

- **Active Event Region:** Events that are scheduled to occur at the current simulation time.
 - This is the only region that events can be retired.
 - *Continuous assignment & procedural continuous assignment* constructs add events to this region
- **Inactive Event Region:** Events that occur at the current simulation time but are required to wait until all events in the active region have been processed.
 - When all active events are processed events in this region are placed in active event region.
 - *Blocking assignments* with zero delay are placed here.

Verilog Event Queue Regions

- **Non-blocking Assign Update Region:** Events that were evaluated at some previous simulation time but are scheduled to occur at the current simulation time.
 - Events in this region are processed only after events in the **active** and **inactive** regions have been retired.
- **Monitor Event Region:** Events scheduled for the current time step that are continuously reenabled in every successive time step by the `$monitor` and `$strobe` system tasks.
 - Events in this region are processed only after events in the **active**, **inactive**, and **non-blocking** regions have been retired.

Verilog Event Queue Regions

- **Future Event Region:** Events that occur at some future simulation time.
 - Two sub-regions:
 - **Future Inactive Events:**
 - Events generate by Blocking assignments with nonzero delay
 - **Future Non-blocking Assignment Update Events:**
 - Events generated by Non-blocking assignments with nonzero delay
 - Processed after all other regions with events chosen based upon those that have the earliest scheduled simulation time.

Simulation Example

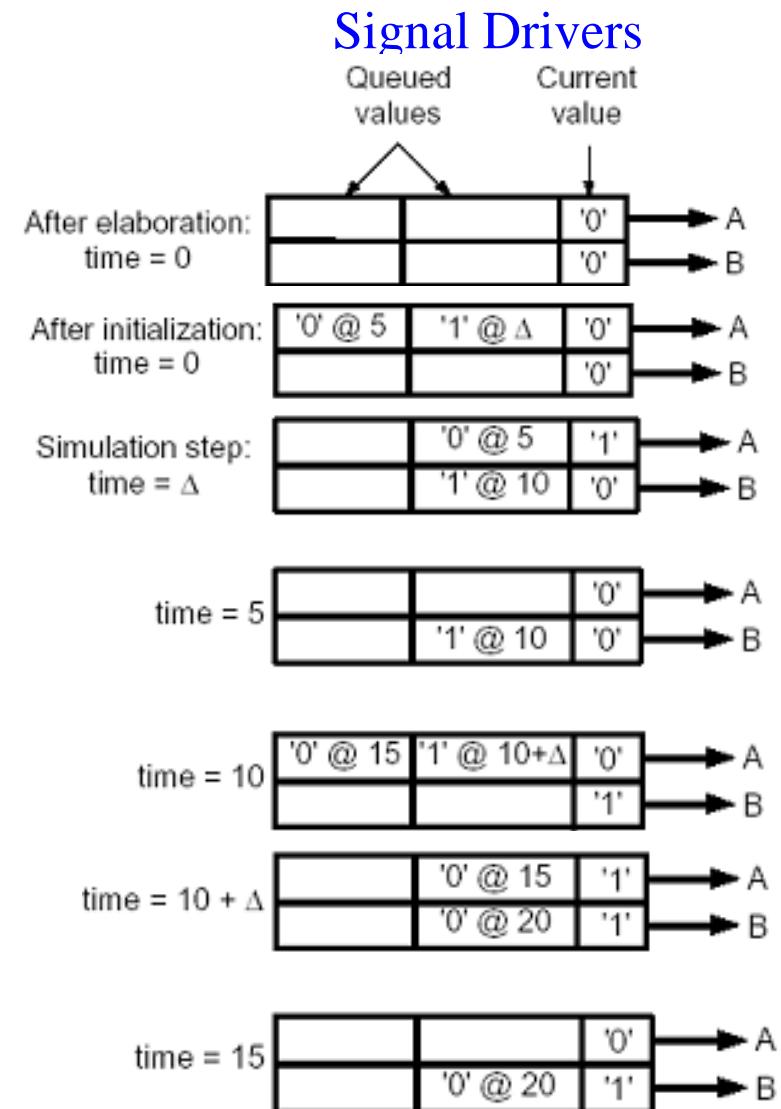
```
`timescale 1ns/100ps

module simulation_example;
    reg A=0,B=0;

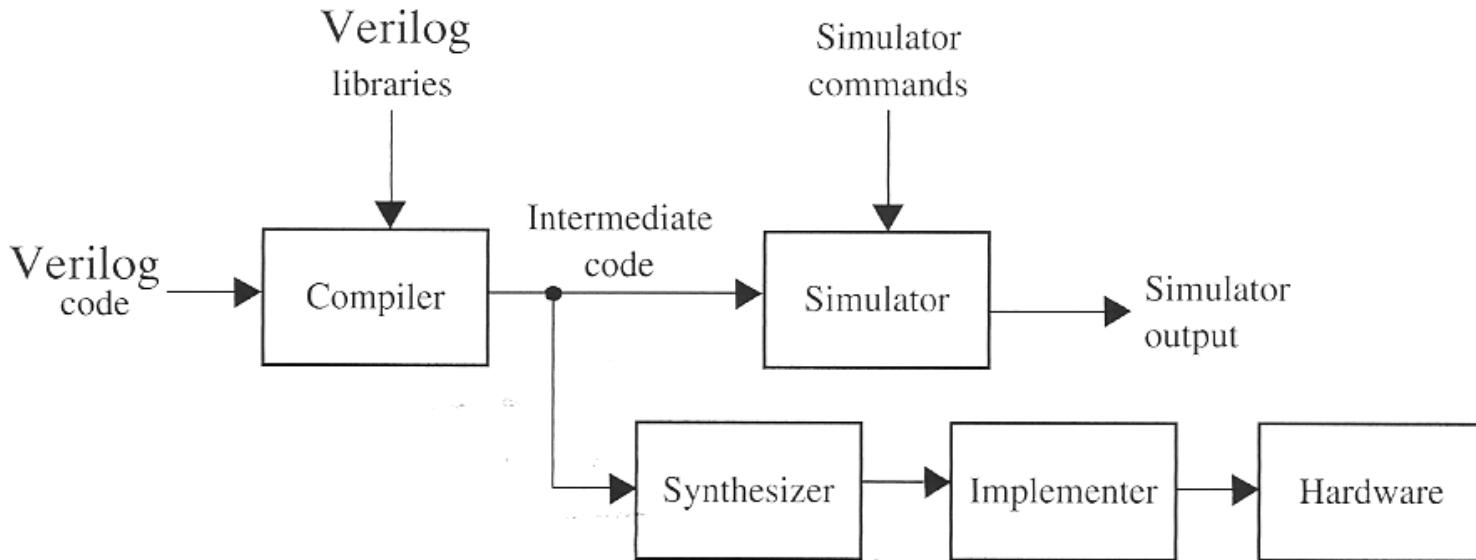
    always @ (B)
        begin
            A <= 1;
            A <= #5 0;
        end

    always @ (A)
        if (A) B <= #10 ~B;

endmodule
```

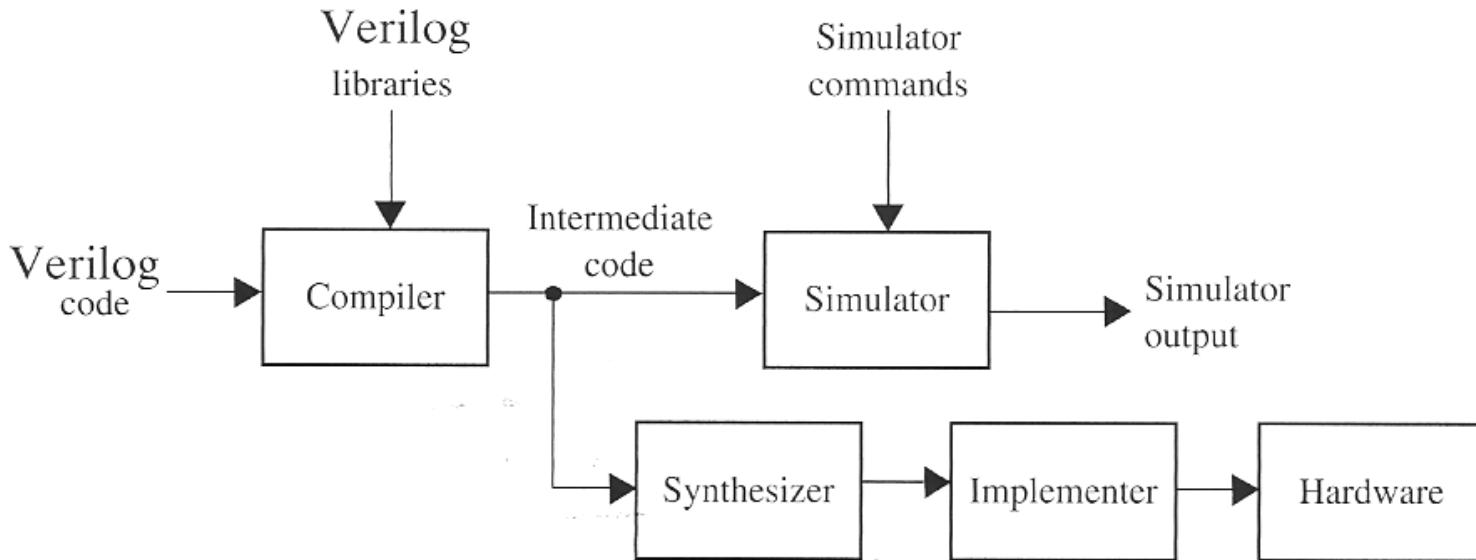


Synthesis of Verilog Code



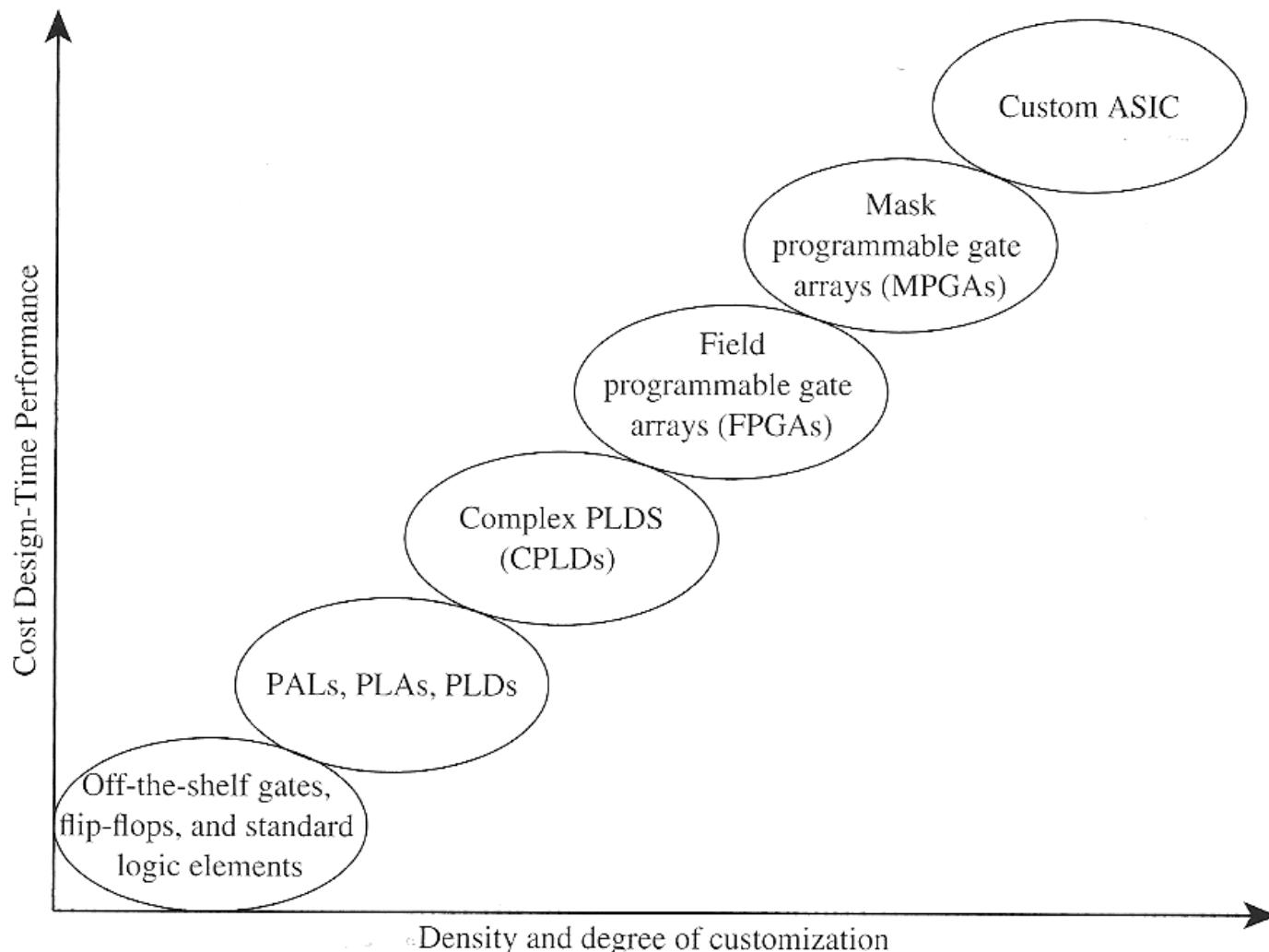
- Synthesis – automatic create hardware from a Verilog Description
 - Intermediate code is created through the same steps of analysis and elaboration used in simulation
 - Often simulation is performed first to catch errors before hardware is created

Synthesis of Verilog Code



- **Synthesis**
 - Produces a **netlist** that includes a list of required components and their interconnections
 - Output of **synthesizer** is used by the targeted **implementer** module to create the specific CPLD, FPGA, or ASIC hardware.

Spectrum of Possible Technologies for Design Synthesis

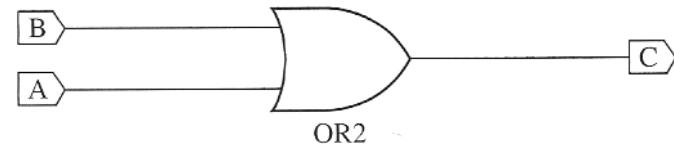


Synthesis Issues and Examples

What logic circuit will be synthesized?

```
module Q1 (input A, B, output reg C);  
  
    always @ (A)  
        C = #5 A | B;  
  
endmodule
```

Most synthesizers will give a warning that B is not in sensitivity list but synthesize an OR gate.



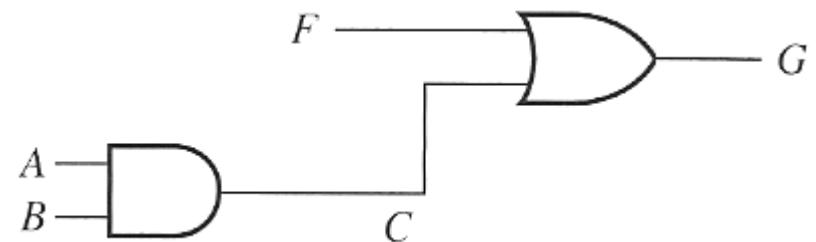
- In this case the synthesizer “guessed” at what the user wanted.
- This means the simulation of the circuit will behave differently from the actual circuit that is synthesized!
 - Check the warnings – synthesizer may have helped or it may have hurt you
- (The synthesizer will also ignore the #5 delay)

Synthesis Issues and Examples

What logic circuit will be synthesized?

Is it?

```
module Q1 (input A, B, F, CLK, output reg G);  
  
reg C;  
  
always @ (posedge CLK)  
begin  
    C <= A & B;  
    G <= C | F;  
end  
  
endmodule
```



- An edge-triggered clock is implied (by the **posedge** or **negedge**)
- This means LHS variables C and G will need to be remembered after the clocks edge.
- Non-blocking assignments mean that both statements function concurrently

Synthesis Issues and Examples

What logic circuit will be synthesized?

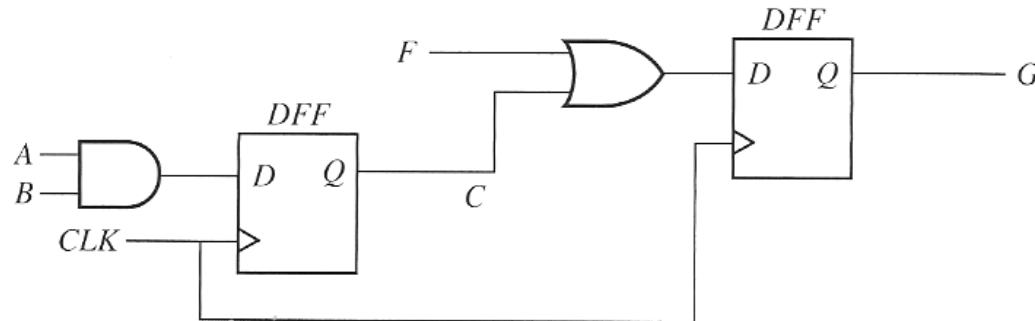
```
module Q1 (input A, B, F, CLK, output reg G);

reg C;

always @ (posedge CLK)
begin
C <= A & B;
G <= C | F;
end

endmodule
```

Actual Circuit

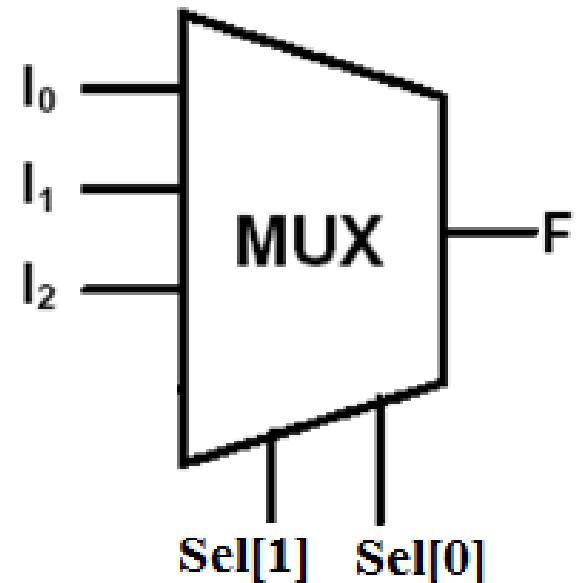


Avoiding Unwanted Latches

```
module MUX(input [1:0] Sel, I0, I1, I2, output reg F);

  always @ (Sel, I0, I1, I2)
    begin
      if      (Sel == 2'b00) F = I0;
      else if (Sel == 2'b01) F = I1;
      else if (Sel == 2'b10) F = I2;
    end

endmodule
```



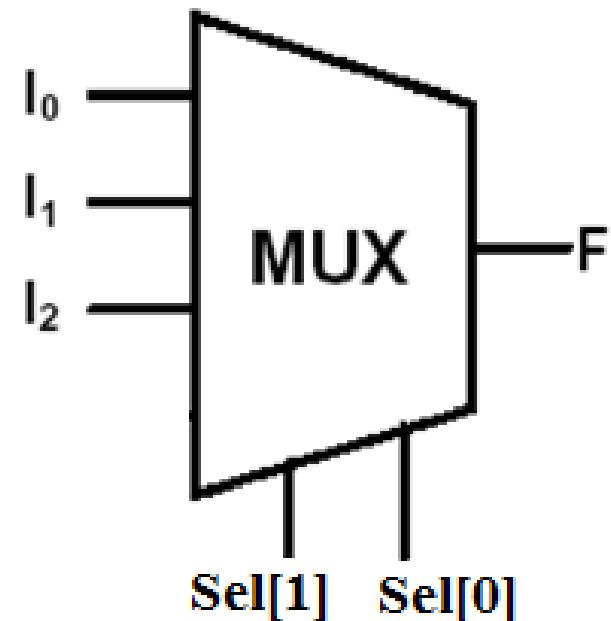
Unwanted latch created

Avoiding Unwanted Latches

```
module MUX(input [1:0] Sel, I0, I1, I2, output reg F);

  always @ (Sel, I0, I1, I2)
    begin
      if      (Sel == 2'b00) F = I0;
      else if (Sel == 2'b01) F = I1;
      else if (Sel == 2'b10) F = I2;
      else F = 0;
    end

endmodule
```

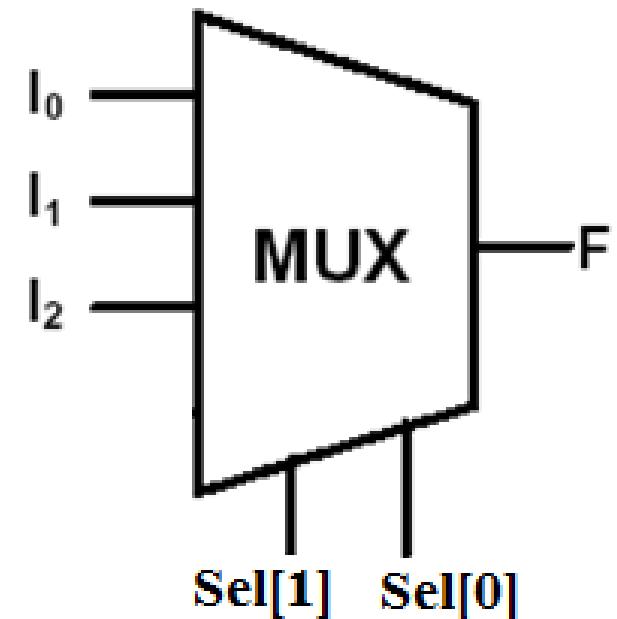


Avoiding Unwanted Latches

```
module MUX(input [1:0] Sel, I0, I1, I2, output reg F);

  always @ (Sel, I0, I1, I2)
    begin
      F = 0;
      if      (Sel == 2'b00) F = I0;
      else if (Sel == 2'b01) F = I1;
      else if (Sel == 2'b10) F = I2;
    end

endmodule
```

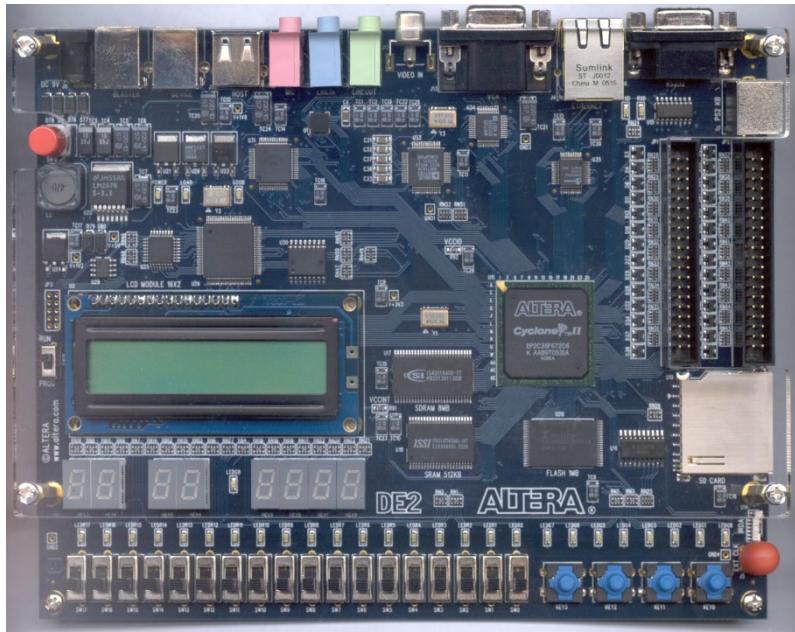


CPE 322

Digital Hardware Design Fundamentals

Electrical and Computer Engineering
UAH

Review of Basic Number Representation



Review of Number Representations

Unsigned Number

Positional Notation $\dots K_2 K_1 K_0 \bullet K_{-1} K_{-2} \dots$ where $K_i = \text{symbols}$

The diagram shows the positional notation of a number. It consists of two parts: the Integer Part and the Fractional Part, separated by a vertical line with a dot at the top labeled 'Radix Point'. The Integer Part is on the left, starting with K_2 , followed by K_1 , and ending with K_0 . The Fractional Part is on the right, starting with K_{-1} , followed by K_{-2} , and so on. An upward-pointing arrow is positioned between the two parts, pointing to the Radix Point.

for base R

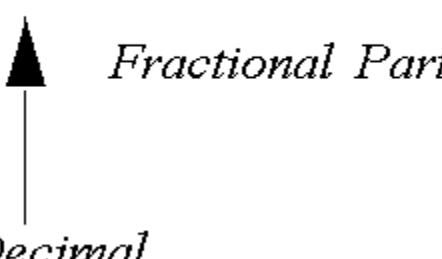
$$\text{Number} = \dots + K_2 R^2 + K_1 R^1 + K_0 R^0 + K_{-1} R^{-1} + K_{-2} R^{-2} + \dots$$

Review of Number Representations

Example: Unsigned Decimal Number

Positional
Notation

$\dots K_2 K_1 K_0 \bullet K_{-1} K_{-2} \dots$ where $K_i = \text{digits } 0 - 9$



base R=10

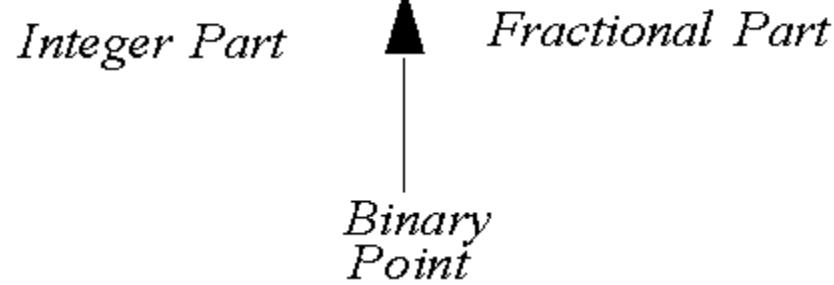
Number = 123.45 =

$$1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 + 4 \times 10^{-1} + 5 \times 10^{-2}$$

Review of Number Representations

Example: Unsigned Binary Number

Positional Notation $\dots K_2 K_1 K_0 \bullet K_{-1} K_{-2} \dots$ where $K_i = \text{symbols 0 \& 1}$



base R=2

$$\text{Number} = 101.11_2$$

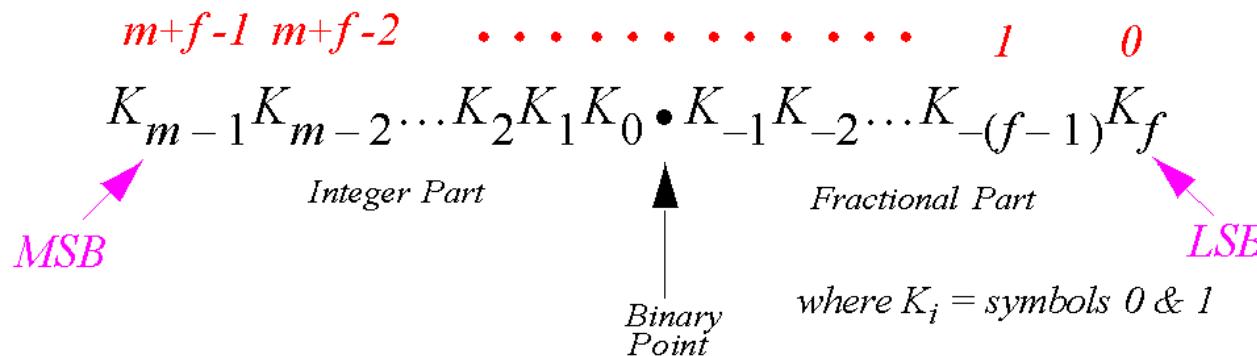
$$= 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2}$$

$$= 5.75_{10}$$

Review of Number Representations

Unsigned Binary Number -- Finite Length (fixed point)

- Digital logic representation of number have a set number of bits.



- In general, there are m bits to represent the integer (magnitude) portion of the number, f bits used to represent the fraction portion of the number.

Review of Number Representations

Signed Binary Number -- Finite Length (fixed point)

- Two's Complement is the most common signed binary number format.
 - Two's complement uses standard positional notation but with the most significant bit having a negative weighting

The diagram illustrates a floating-point number representation. It consists of a sequence of symbols: a red '1' for the sign, followed by a red '0' for the exponent, and then a series of black dots representing the fraction. Above this sequence, red numbers $m+f-1$, $m+f-2$, and so on down to 0 are aligned with the first symbol of each group of four black dots. Below the sequence, three pink arrows point upwards from labels: 'MSB' points to the first symbol of the first group; 'Integer Part' points to the first symbol of the second group; and 'LSB' points to the last symbol of the last group. A vertical pink arrow between the second and third groups is labeled 'Binary Point'. To the right of the sequence, the text 'where $K_i = \text{symbols } 0 \& 1$ ' is written.

Review of Number Representations

Signed Binary Number -- Finite Length (fixed point)

The diagram illustrates a floating-point number representation with the following components:

- Sign Bit:** The first bit, labeled $m+f-1$, indicates the sign of the number.
- Integer Part:** The bits from $m+f-2$ down to K_0 represent the integer part of the number.
- Fractional Part:** The bits from K_{-1} down to $K_{-(f-1)}$ represent the fractional part of the number.
- Binary Point:** A vertical line with an arrow pointing downwards, labeled "Binary Point", separates the integer and fractional parts.
- MSB (Most Significant Bit):** An arrow points to the first bit, labeled "MSB".
- LSB (Least Significant Bit):** An arrow points to the last bit, labeled "LSB".

Below the diagram, the text states: "where $K_i = \text{symbols } 0 \& 1$ ".

$$-K_{m-1}2^{m-1} + K_{m-2}2^{m-2} + \dots + K_02^0 + K_{-1}2^{-1} + \dots + K_{-(f-1)}2^{-(f-1)} + K_{-f}2^f$$



*MSB
negative
weighting* *LSB*

Integers are just the special case where $f = 0$

Review of Number Representations

Two's Complement Representation of Integers:

- Range: $2^{m-1}-1$ positive numbers

- 2^{m-1} negative numbers

- 1 representation of zero

MSB => sign bit: 0 = positive (or zero) 1 = negative

Review of Number Representations

Two's Complement Representation of Integers:

- To represent positive or zero numbers, simply enter binary value
i.e. for m=8 then $15_{10} = 00001111_2$
- To represent negative numbers, first enter the binary form of the numbers absolute value, then complement each bit and add 1.
i.e. for m=8 then -15_{10} is found by
 $\text{comp}(00001111) + 1 = 11110000 + 1$
 $= 11110001_2$

Review of Number Representations

Signed binary number – Finite Length (floating point):

The format for an IEEE 32-bit floating-point number is shown below

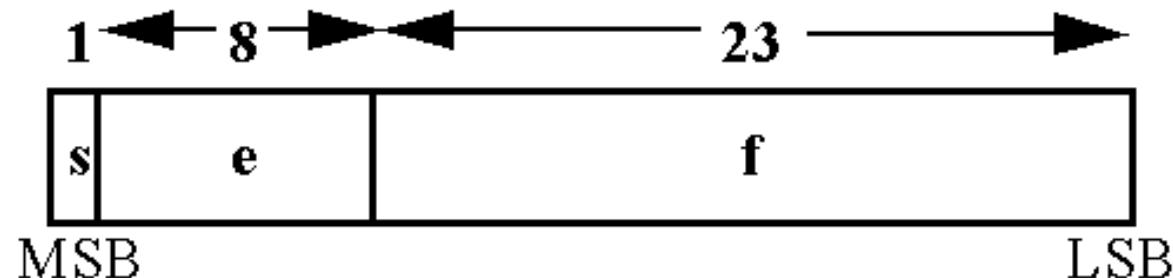
$$\text{Floating Point Number} = -1^s \times 1.f \times 2^{e-b}$$

where:

s=sign bit (0=positive mantissa, 1=negative mantissa)

e=exponent biased by b (b= 127 for IEEE 32-bit format)

f=fractional mantissa



Review of Number Representations

Signed binary number – Finite Length (floating point):

Special Case Numbers

Type	Sign	Exp	Exp+Bias	Exponent	Significand	Value
Zero	0	-127		0 0000 0000	000 0000 0000 0000 0000 0000 0000	0.0
One	0	0		127 0111 1111	000 0000 0000 0000 0000 0000 0000	1.0
Minus One	1	0		127 0111 1111	000 0000 0000 0000 0000 0000 0000	-1.0
Smallest denormalized number	*	-127		0 0000 0000	000 0000 0000 0000 0000 0000 0001	$\pm 2^{-23} \times 2^{-126} = \pm 2^{-149} \approx \pm 1.4 \times 10^{-45}$
"Middle" denormalized number	*	-127		0 0000 0000	100 0000 0000 0000 0000 0000 0000	$\pm 2^{-1} \times 2^{-126} = \pm 2^{-127} \approx \pm 5.88 \times 10^{-39}$
Largest denormalized number	*	-127		0 0000 0000	111 1111 1111 1111 1111 1111 1111	$\pm(1-2^{-23}) \times 2^{-126} \approx \pm 1.18 \times 10^{-38}$
Smallest normalized number	*	-126		1 0000 0001	000 0000 0000 0000 0000 0000 0000	$\pm 2^{-126} \approx \pm 1.18 \times 10^{-38}$
Largest normalized number	*	127		254 1111 1110	111 1111 1111 1111 1111 1111 1111	$\pm(2-2^{-23}) \times 2^{127} \approx \pm 3.4 \times 10^{38}$
Positive infinity	0	128		255 1111 1111	000 0000 0000 0000 0000 0000 0000	$+\infty$
Negative infinity	1	128		255 1111 1111	000 0000 0000 0000 0000 0000 0000	$-\infty$
Not a number	*	128		255 1111 1111	non zero	NaN

* Sign bit can be either 0 or 1 .

Review of Number Representations

Binary Coded Decimal (BCD) Numbers:

- Encoding where each ‘digit’ (0-9) is represented by its own 4-bit binary sequence (0000_2 — 1001_2)
- Advantage: Easy to convert to the decimal digits needed for I/O devices and in some cases faster decimal calculations
- Disadvantage: Requires more bits than an equivalent binary representation. Also requires more complex circuitry for arithmetic operations.

Review of Number Representations

Binary Coded Decimal (BCD) Numbers:

example:

for m=8 bit encoding:

$$01111000_2 \Rightarrow 0111 \ 1000 \Rightarrow 78_{10}$$

Review of Number Representations

Conversion of Binary to Octal Numbers (and conversely) :

Converting binary numbers to Octal Numbers can be done by inspection

Each octal digit corresponds to 3 bits.

Just begin at the binary replace each group of three bits with the corresponding octal digit (assume leading and lagging 0's).

example:

$01101011110.0011_2 =$

011 010 111 110 . 001 100 =
3 2 7 6 . 1 4₈

Review of Number Representations

Conversion of Binary to Hexadecimal Numbers (and conversely) :

Converting binary numbers to Hexadecimal Numbers can also be done by inspection

Each hex digit corresponds to 4 bits.

Just begin at the binary point and replace each group of four bits with the corresponding hexadecimal symbol (0-F).

example:

$01101011110.0011_2 =$

0110 1011 1110 . 0011 =
6 B E . 3₁₆

Review of Number Representations

- Note: Conversion from binary to hexadecimal (or octal) is so easy hexadecimal and octal are often considered to be short-hand notation form binary!

Review of Number/Character Representations

ASCII Table and Description

ASCII stands for American Standard Code for Information Interchange. Computers can only understand numbers, so an ASCII code is the numerical representation of a character such as 'a' or '@' or an action of some sort. ASCII was developed a long time ago and now the non-printing characters are rarely used for their original purpose. Below is the ASCII character table and this includes descriptions of the first 32 non-printing characters. ASCII was actually designed for use with teletypes and so the descriptions are somewhat obscure. If someone says they want your CV however in ASCII format, all this means is they want 'plain' text with no formatting such as tabs, bold or underscoring - the raw format that any computer can understand. This is usually so they can easily import the file into their own applications without issues. Notepad.exe creates ASCII text, or in MS Word you can save a file as 'text only'

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	Ø	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[[123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

Source: www.LookupTables.com

Review of Number/Character Representations

- Other encodings of characters and numbers are also used. Examples, EBSIDIC, Gray codes, etc.

Gray Code				Position	Binary			
2^3	2^2	2^1	2^0		2^3	2^2	2^1	2^0
0	0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	1
0	0	1	1	2	0	0	1	0
0	0	1	0	3	0	0	1	1
0	1	1	0	4	0	1	0	0
0	1	1	1	5	0	1	0	1
0	1	0	1	6	0	1	1	0
0	1	0	0	7 ►	0	1	1	1
1	1	0	0	8 ►	1	0	0	0
1	1	0	1	9	1	0	0	1
1	1	1	1	10	1	0	1	0
1	1	1	0	11	1	0	1	1
1	0	1	0	12	1	1	0	0
1	0	1	1	13	1	1	0	1
1	0	0	1	14	1	1	1	0
1	0	0	0	15	1	1	1	1

Floating Point Representations

- A floating point binary number consists of three parts:
 - sign (S), exponent (E), and mantissa (M).
 - Each (S, E, M) pattern uniquely identifies a floating point number.
- For each bit pattern, its IEEE floating-point value is derived as:
 - $\text{value} = (-1)^S * M * \{2^E\}$, where $1.0 \leq M < 10.0_B$
- S=0 results in a positive number and S=1 a negative number.

Floating Point Representations

(Mantissa part, M)

- Specifying that $1.0_B \leq M < 10.0_B$ makes the mantissa value for each floating point number unique.
 - For example, the only one mantissa value allowed for 0.5_D is $M = 1.0$
 - $0.5_D = 1.0_B * 2^{-1}$
 - Neither $10.0_B * 2^{-2}$ nor $0.1_B * 2^0$ qualifies
- Because all mantissa values are of the form $1.XX\dots$, one can omit the “1.” part in the representation.
 - The mantissa value of 0.5_D in a 2-bit mantissa is 00, which is derived by omitting “1.” from 1.00.
 - Mantissa without the implied 1 is called the *fraction*

Floating Point Representations (Exponent, E)

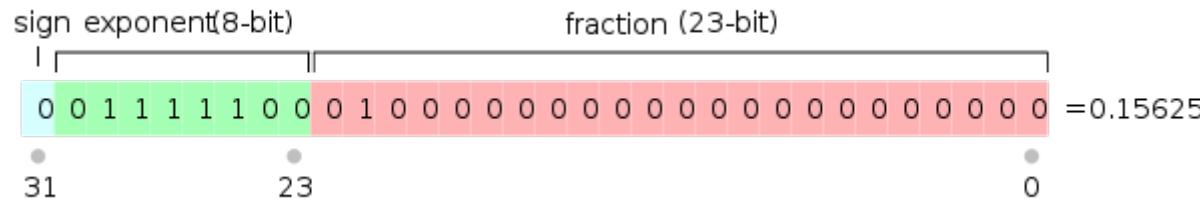
- In an n-bits exponent representation, $2^{n-1}-1$ is added to its 2's complement representation to form its excess representation.
 - See Table for a 3-bit exponent representation
- A simple unsigned integer comparator can be used to compare the magnitude of two FP numbers
- Symmetric range for +/- exponents (case where Exponent is all 1's is reserved)

2's complement	Actual decimal	Excess-3
000	0	011
001	1	100
010	2	101
011	3	110
100	(reserved pattern)	111
101	-3	000
110	-2	001
111	-1	010

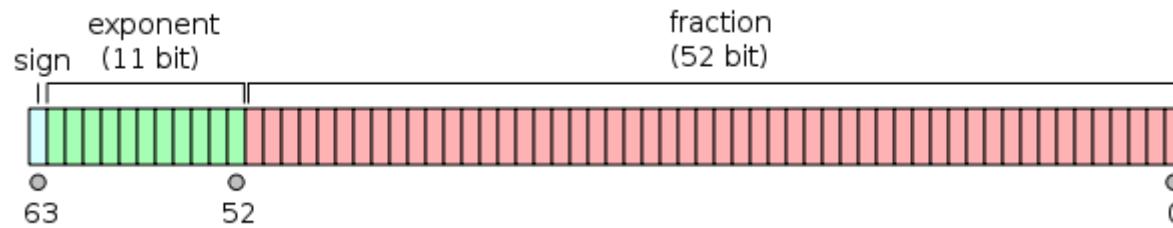
Floating Point Representations

IEEE 754 Format

- **Single Precision**
 - **1 bit sign, 8 bit exponent (bias-127), 23 bit fraction**



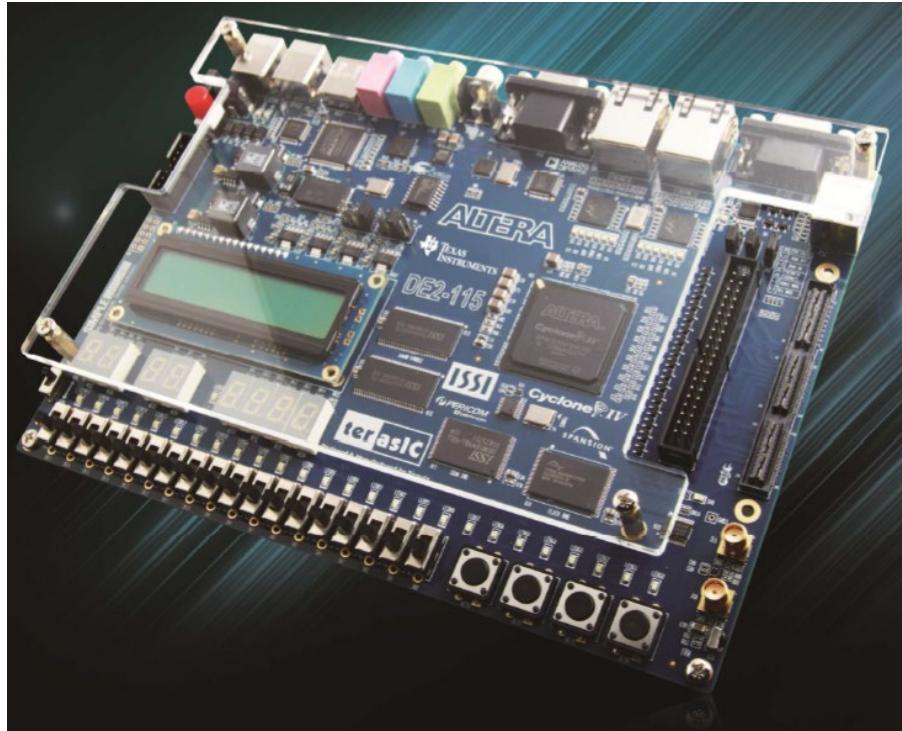
- **Double Precision**
 - 1 bit sign, 11 bit exponent (1023-bias), 52 bit fraction



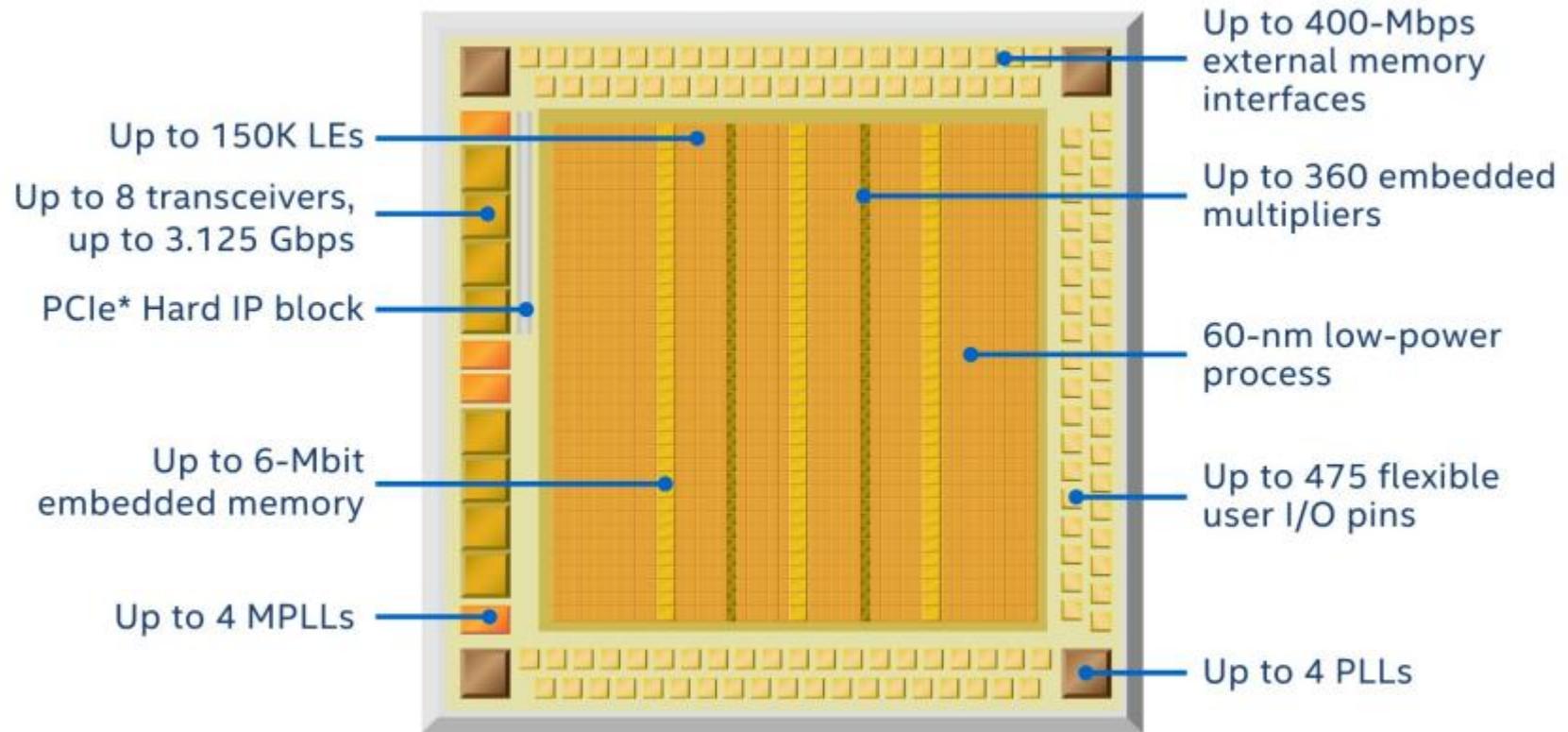
CPE 322

Digital Hardware Design Fundamentals

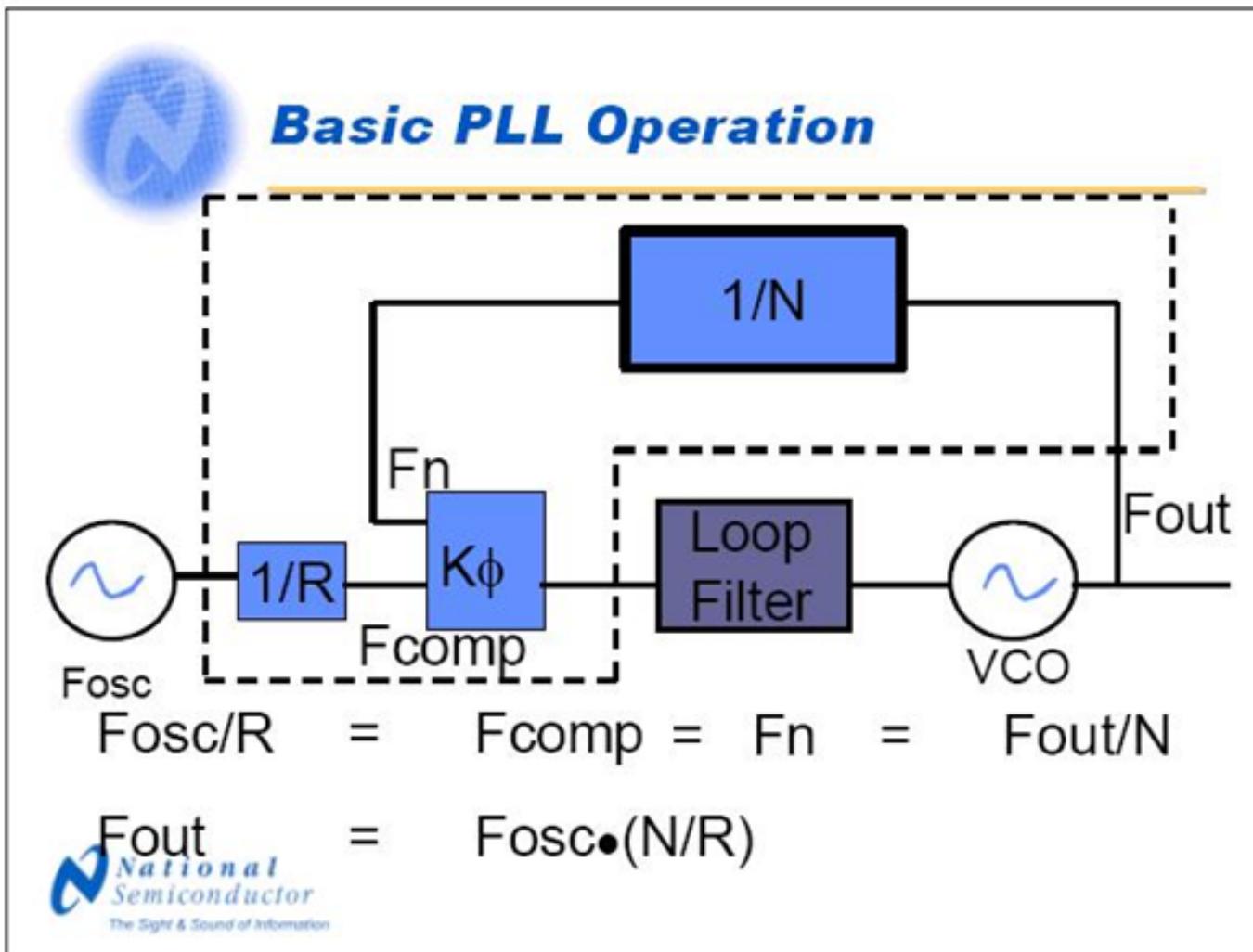
Non CLB FPGA Building Blocks:
Embedded Phase-Locked-Loops



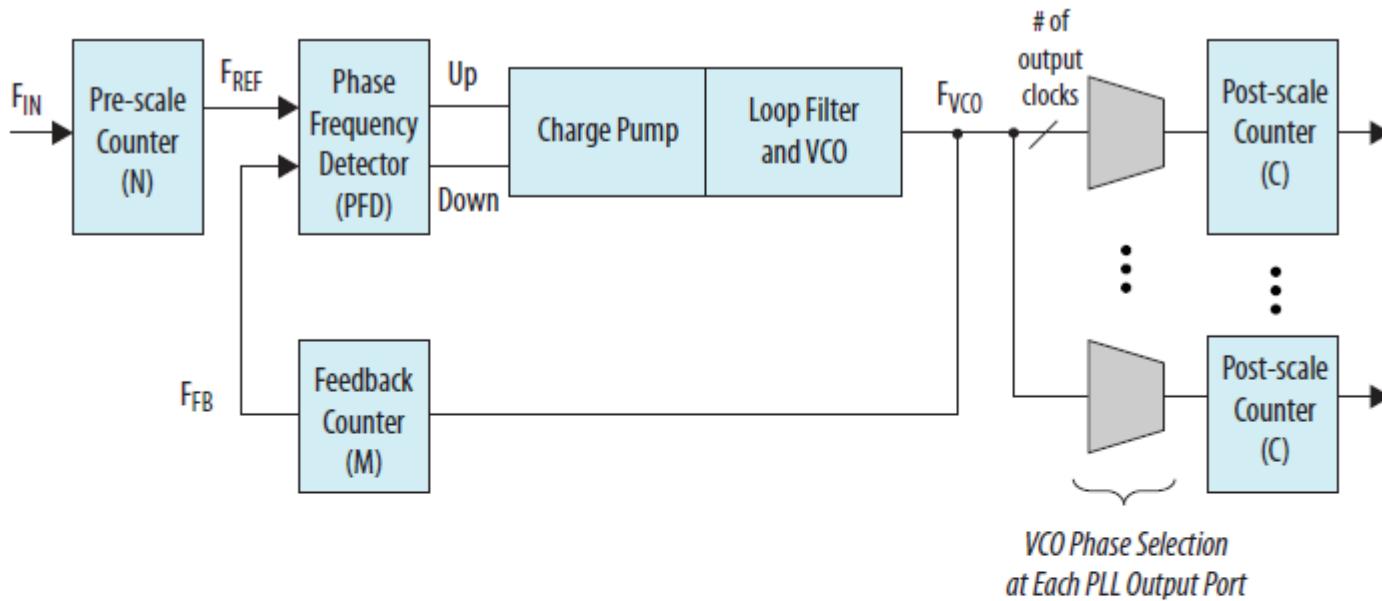
Cyclone IV E Family FPGA Layout



Basic PLL Operation



PLL in IntelFPGAs



The following terms are commonly used to describe the behavior of a PLL:

- PLL lock time—also known as the PLL acquisition time. PLL lock time is the time for the PLL to attain the target frequency and phase relationship after power-up, after a programmed output frequency change, or after a PLL reset.

Note: Simulation software does not model a realistic PLL lock time. Simulation shows an unrealistically fast lock time. For the actual lock time specification, refer to the device datasheet.

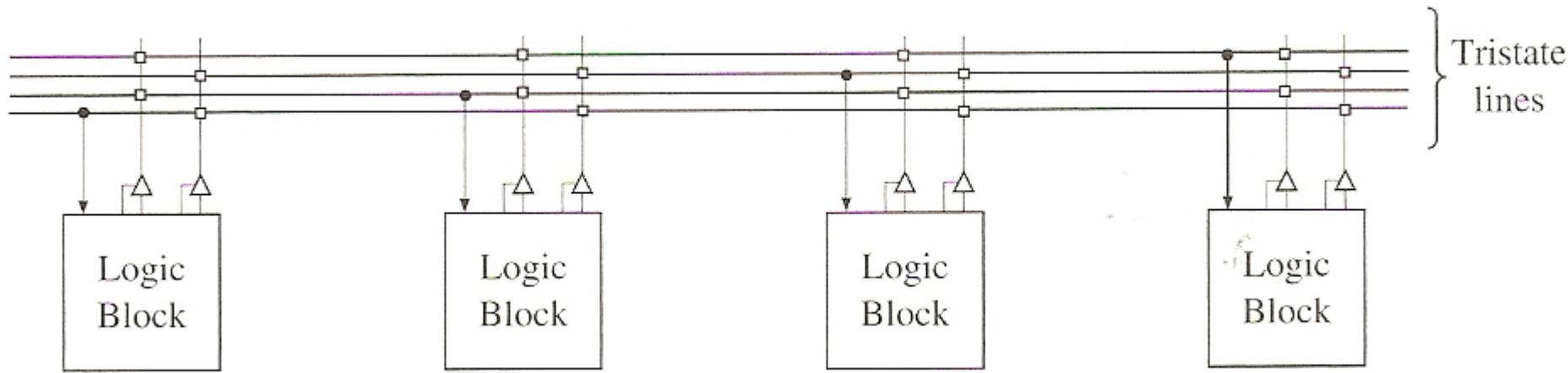
- PLL resolution—the minimum frequency increment value of a PLL VCO. The number of bits in the **M** and **N** counters determine the PLL resolution value.
- PLL sample rate—the F_{REF} sampling frequency required to perform the phase and frequency correction in the PLL. The PLL sample rate is f_{REF} / N .

Clock Skew and Clock Distribution

Clock Skew

There are several million gates in modern FPGA chips. When a clock is distributed to various parts of such a large chip, the delays in the wire carrying the clock can result in the clock edge arriving at different times at different parts. This difference in the actual edge of the clock as it arrives at different flip-flops or other devices is called clock skew. Clock skew is a problem in large systems, including modern microprocessors. Carefully planned clock distribution circuits are implemented in most systems in order to minimize the effect of clock skew. Modern FPGAs provide specialized clock distribution circuitry in order to create a clock of sufficient strength and low skew.

Global Lines



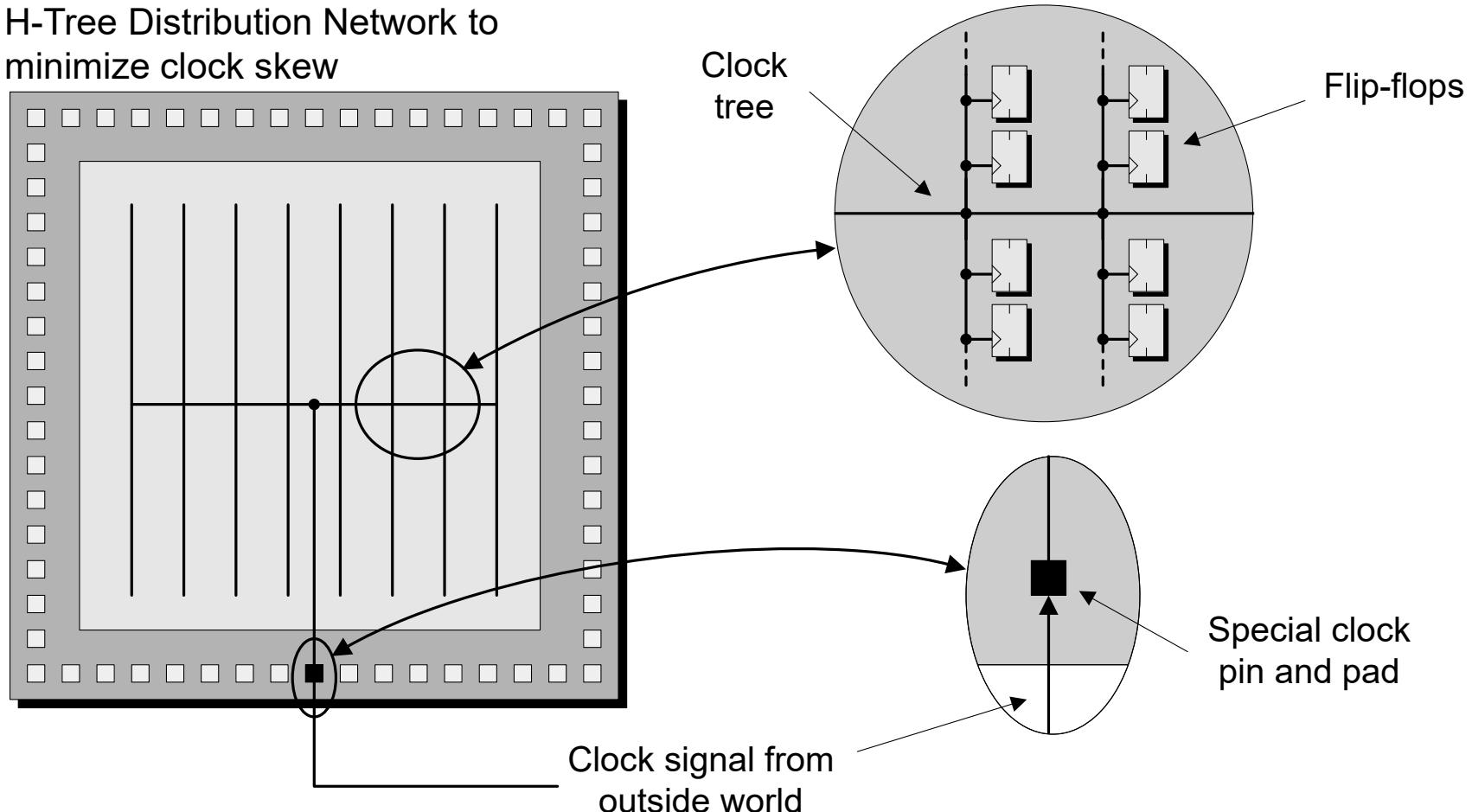
Lines that span the entire width/height of the device.
Supports High Fan-out and Low-skew clock distribution.

Internal Tri-state buffers often connect the logic blocks
to these lines.

A limited number of these lines are provided.

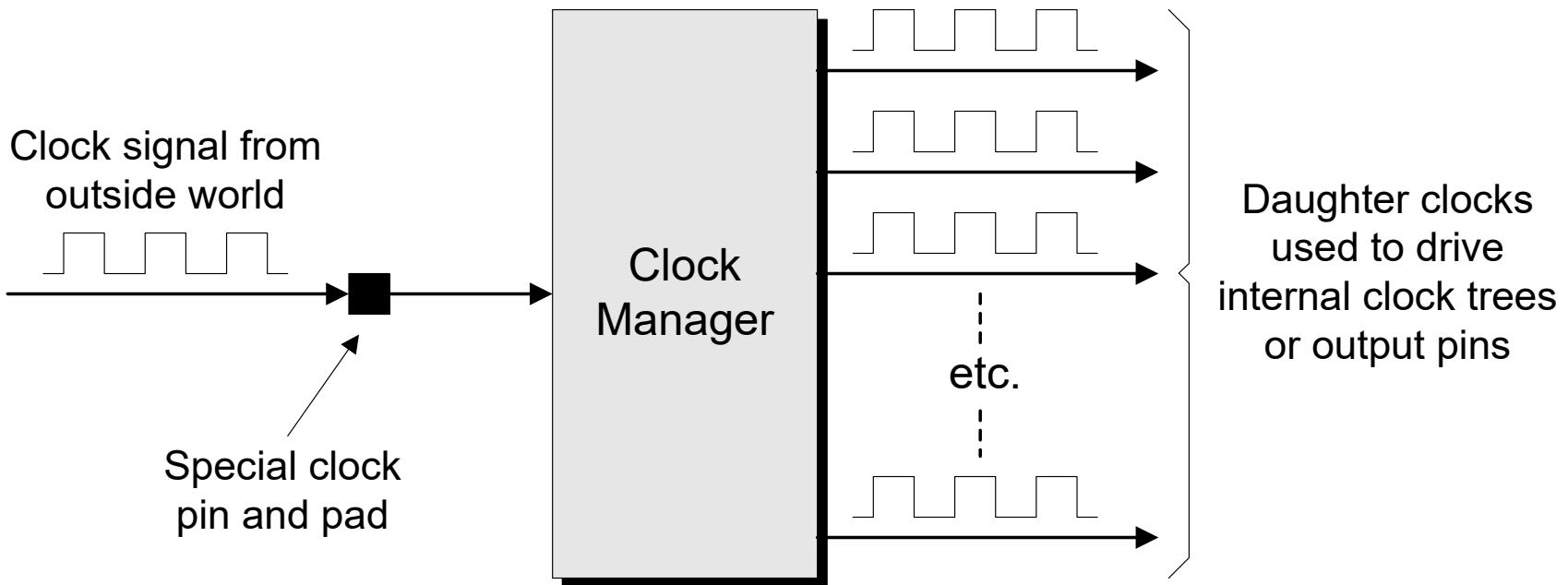
Clock Trees

- H-Tree Distribution Network to minimize clock skew



The Design Warrior's Guide to FPGAs
Devices, Tools, and Flows. ISBN 0750676043
Copyright © 2004 Mentor Graphics Corp. (www.mentor.com)

Digital Clock Manager

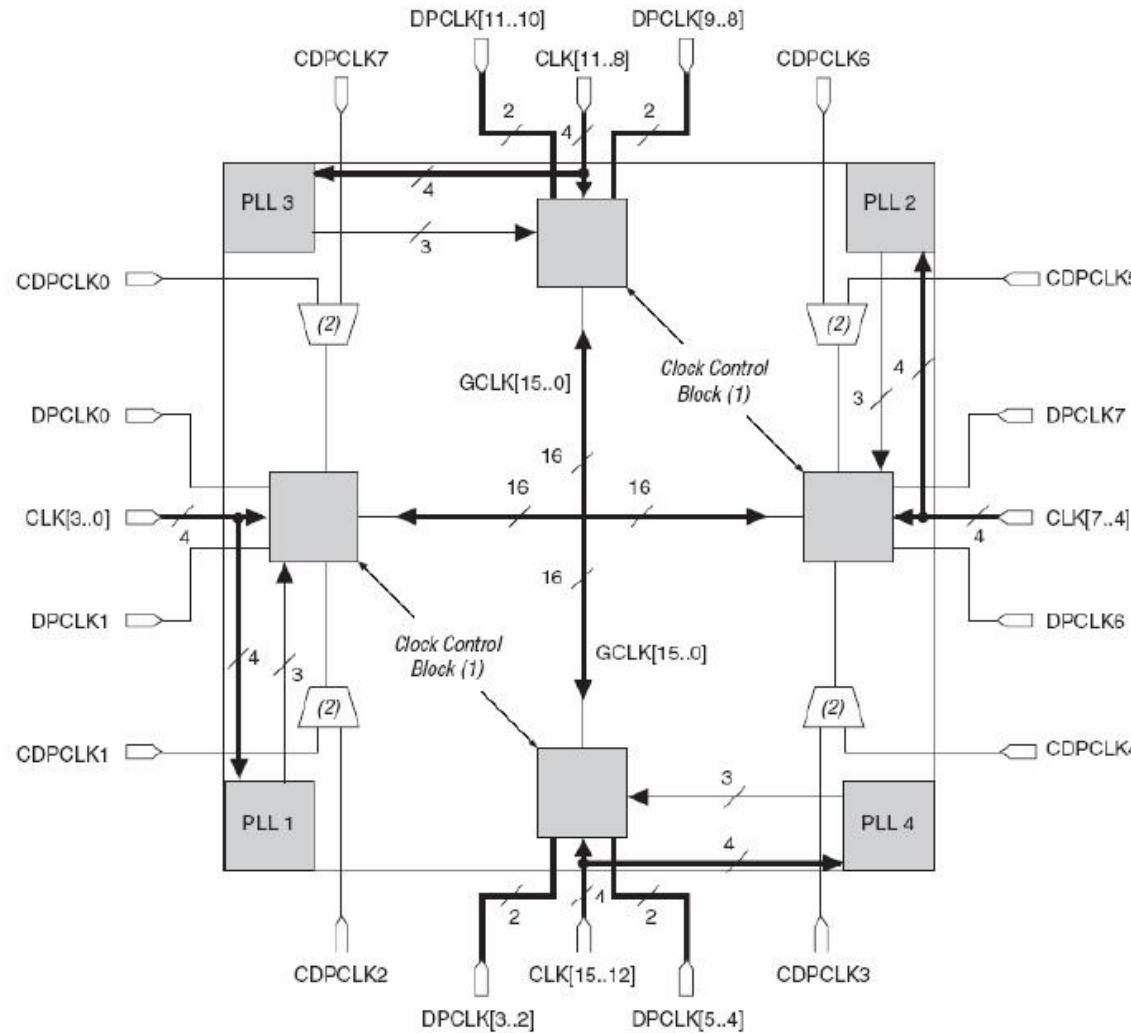


The Design Warrior's Guide to FPGAs
Devices, Tools, and Flows. ISBN 0750676043
Copyright © 2004 Mentor Graphics Corp. (www.mentor.com)

Global Clock Networks & PLLs

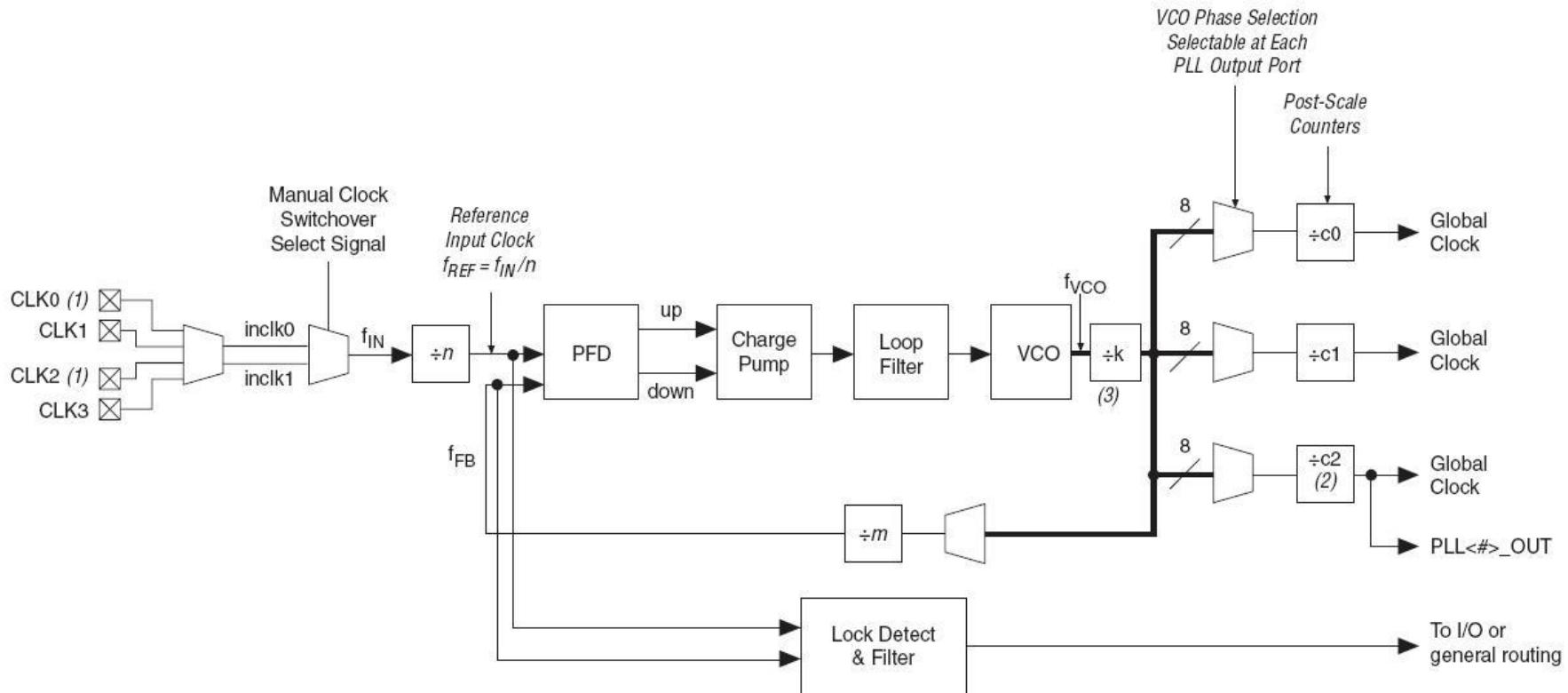
Altera Cyclone IV

Figure 2–12. EP2C20 & Larger PLL, CLK[], DPCLK[] & Clock Control Block Locations



Block Diagram Altera Cyclone II PLLs

Partitioning a Design in an FPGA



Cyclone IV PLL Features

Table 7–2. Cyclone II PLL Features

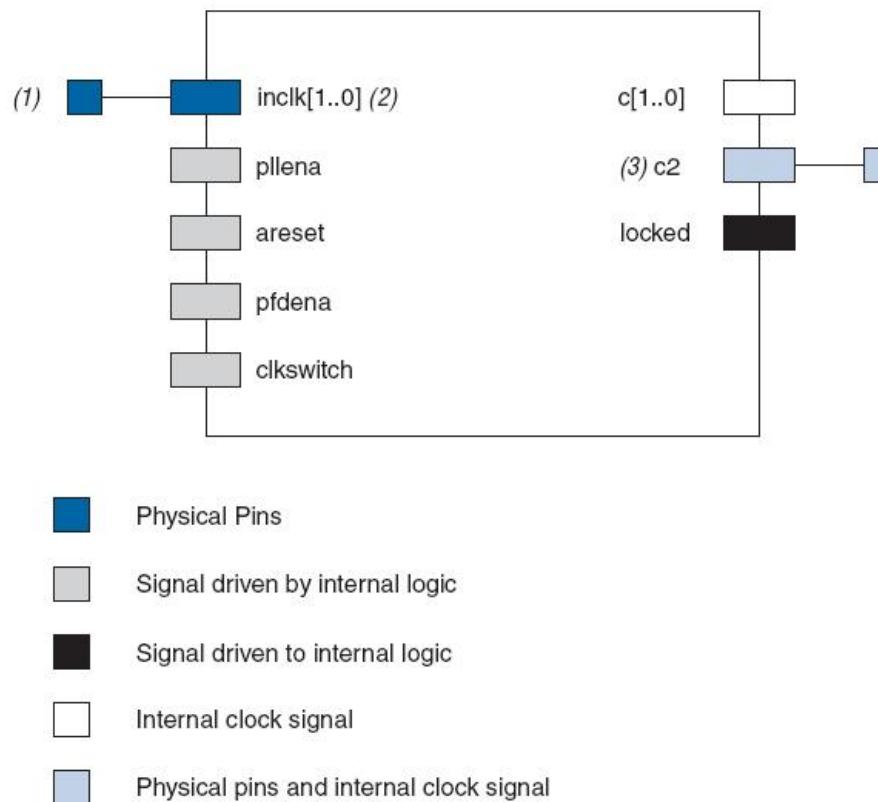
Feature	Description
Clock multiplication and division	$m / (n \times \text{post-scale counter})$ (1)
Phase shift	Down to 125-ps increments (2), (3)
Programmable duty cycle	✓
Number of internal clock outputs	Up to three per PLL (4)
Number of external clock outputs	One per PLL (4)
Locked port can feed logic array	✓
PLL clock outputs can feed logic array	✓
Manual clock switchover	✓
Gated lock	✓

Notes to Table 7–2:

- (1) m and post-scale counter values range from 1 to 32. n ranges from 1 to 4.
- (2) The smallest phase shift is determined by the voltage control oscillator (VCO) period divided by 8.
- (3) For degree increments, Cyclone II devices can shift output frequencies in increments of at least 45°. Smaller degree increments are possible depending on the VCO frequency.
- (4) The Cyclone II PLL has three output counters that drive the global clock network. One of these output counters (c2) can also drive a dedicated external I/O pin (single ended or differential). This counter output can also drive the external clock output ($\text{PLL}_{<\#>} \text{OUT}$) and internal global clock network at the same time.

Cyclone IV PLL Signals

Figure 7-3. Cyclone II PLL Signals



Notes to Figure 7-3:

- (1) These signals can be assigned to either a single-ended or differential I/O standard.
- (2) The **inclk** must be driven by one of two dedicated clock input pins.
- (3) This counter output can drive both a dedicated external clock output (**PLL<#>_OUT**) and the global clock network.

Cyclone IV Signals

Port Name	Type	Condition	Description
fbclk	Input	Optional	<p>The external feedback input port for the PLL.</p> <p>The Altera PLL IP core creates this port when the PLL is operating in external feedback mode or zero-delay buffer mode. To complete the feedback loop, a board-level connection must connect the <code>fbclk</code> port and the external clock output port of the PLL.</p>
fboutclk	Output	Optional	<p>The port that feeds the <code>fbclk</code> port through the mimic circuitry.</p> <p>The <code>fboutclk</code> port is available only if the PLL is in external feedback mode.</p>
locked	Output	Optional	<p>The Altera PLL IP core drives this port high when the PLL acquires lock. The port remains high as long as the PLL is locked.</p> <p>The PLL asserts the <code>locked</code> port when the phases and frequencies of the reference clock and feedback clock are the same or within the lock circuit tolerance. When the difference between the two clock signals exceeds the lock circuit tolerance, the PLL loses lock.</p>
outclk[]	Output	Required	The clock output of the PLL. The frequency of the output clock depends on the parameter settings.

Cyclone IV PLL Input Signals

Port Name	Type	Condition	Description
refclk	Input	Required	The reference clock that drives the clock network.
reset	Input	Required	The asynchronous reset port for the output clocks. Drive this port high to reset all output clocks to the initial value of 0.
zdbfbclk	Bidirectional	Optional	<p>The bidirectional port that connects to the mimic circuitry. This port must connect to a bidirectional pin that is placed on the positive feedback dedicated output pin of the PLL.</p> <p>The zdbfbclk port is available only if the PLL is in zero-delay buffer mode.</p>
refclk1	Input	Required	Second input clock signal that feeds into the PLL.

Cyclone IV PLL Signals

Port Name	Type	Condition	Description
extswitch	Input	Required	Assert this input signal high (1'b1) to manually switch the clock for at least 3 cycles.
activeclk	Output	Optional	Output signal to determine which input clock is in use by the PLL.
clkbad	Output	Optional	Output signal to determine which input clock is working.
cclk ^(s)	Input	Optional	c-Counter clock source from the fracturable fractional PLL output counter 4 or 13.
adjpll1in	Input	Optional	Adjacent fractional PLL clock source.
cascade_out	Output	Optional	Output signal to feed into other fractional PLLs. This port acts as a bus port when the upstream PLL has two or more output clocks.

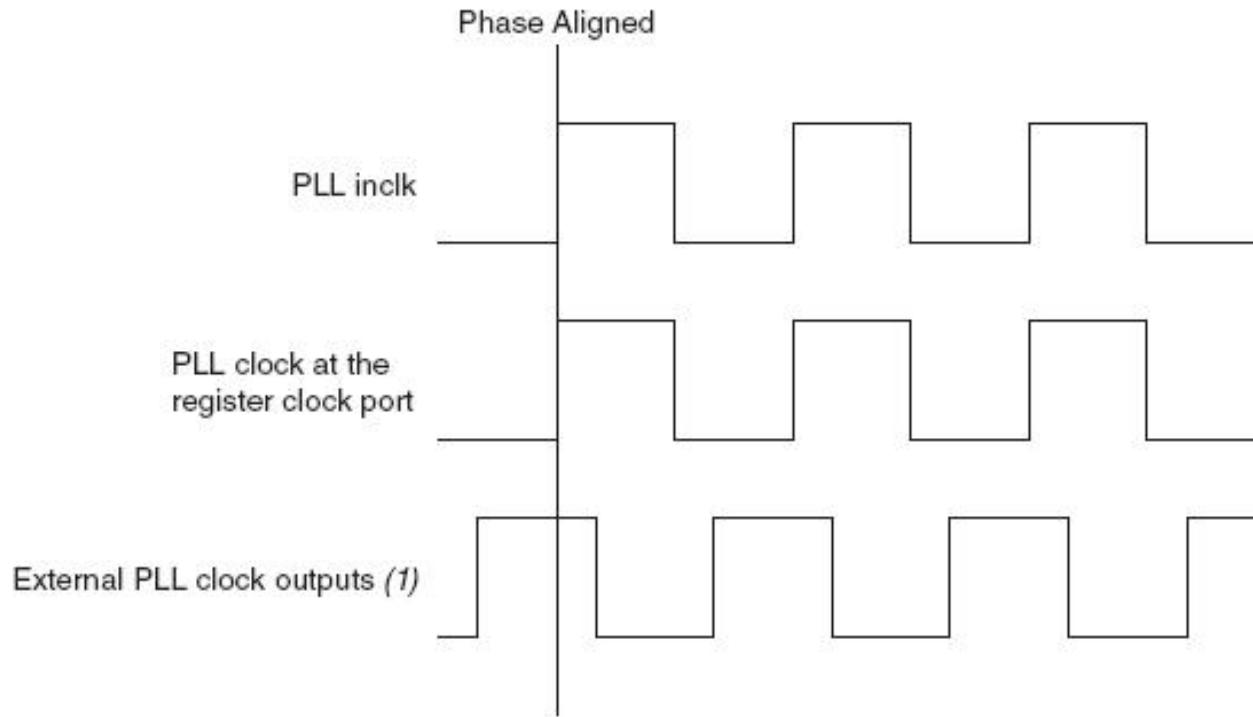
Global Clock Control Block Cyclone IV

Table 7–9. Clock Control Block Inputs

Input	Description
Dedicated clock inputs	Dedicated clock input pins can drive clocks or global signals, such as asynchronous clears, presets, or clock enables onto a given global clock network.
Dual-purpose clock (DPCLK and CDPCLK) I/O inputs	DPCLK and CDPCLK I/O pins are bidirectional dual function pins that can be used for high fan-out control signals, such as protocol signals, TRDY and IRDY signals for PCI, or DQS for DDR, via the global clock network.
PLL outputs	The PLL counter outputs can drive the global clock network.
Internal logic	The global clock network can also be driven through the logic array routing to enable internal logic (LEs) to drive a high fan-out, low skew signal path.

I/O Path Compensation in Cyclone PLLs

Figure 7–4. Phase Relationship between Cyclone II PLL Clocks in Normal Mode

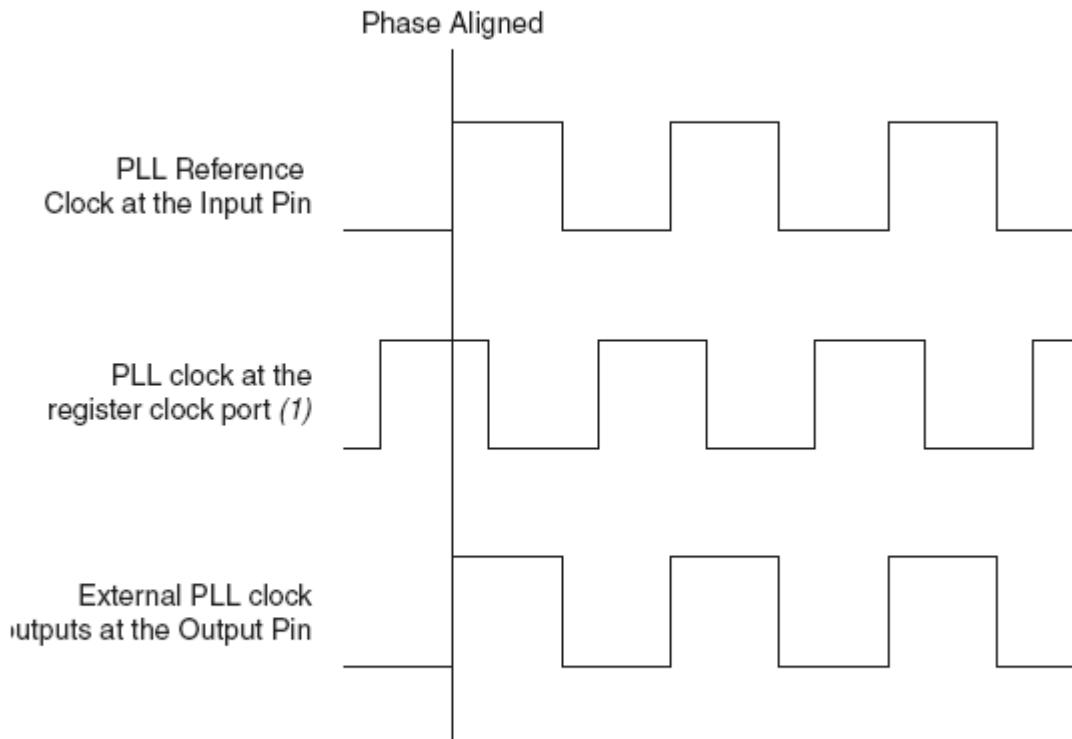


Note

- (1) The external clock output can lead or lag the PLL clock signals.

I/O Path Compensation in Cyclone II PLLs

Figure 7–5. Phase Relationship between Cyclone II PLL Clocks in Zero Delay Buffer Mode

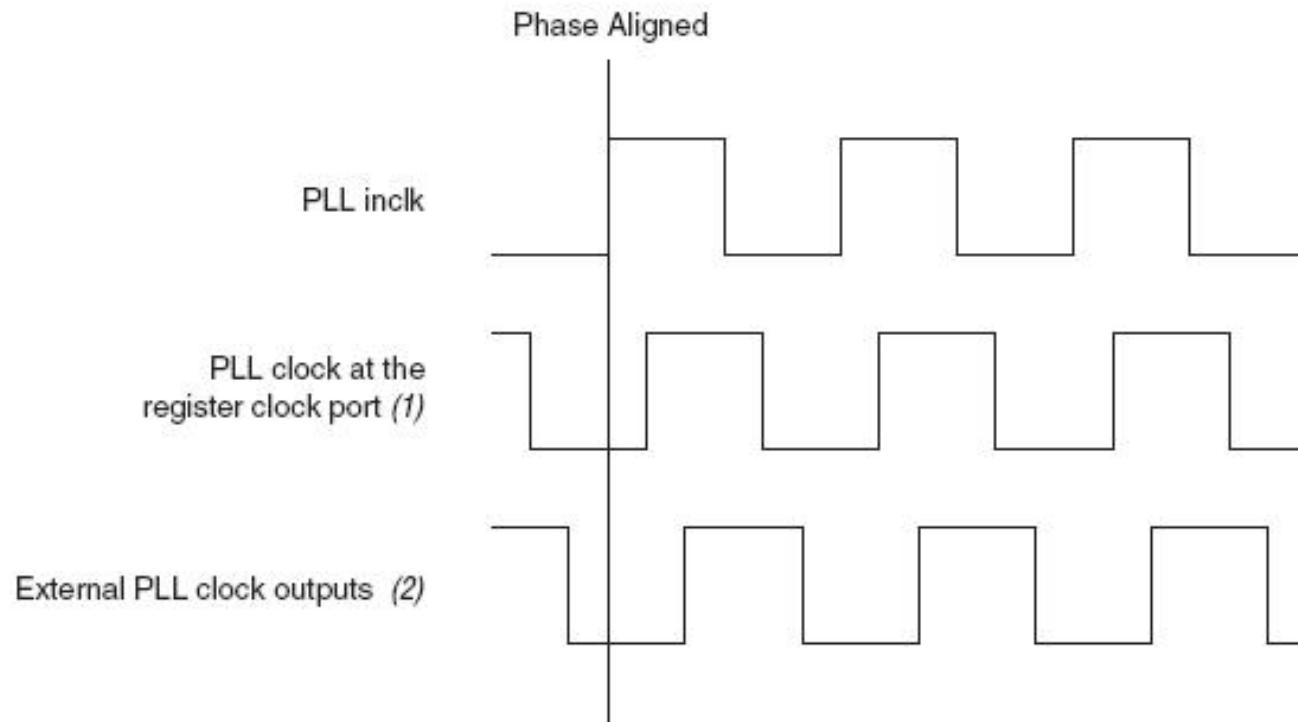


Note to Figure 7–5:

- (1) The internal clock output(s) can lead or lag the external PLL clock output (`PLL<#>_OUT`) signals.
-

I/O Path Compensation in Cyclone II PLLs

Figure 7–6. Phase Relationship between Cyclone II PLL Clocks in No Compensation Mode

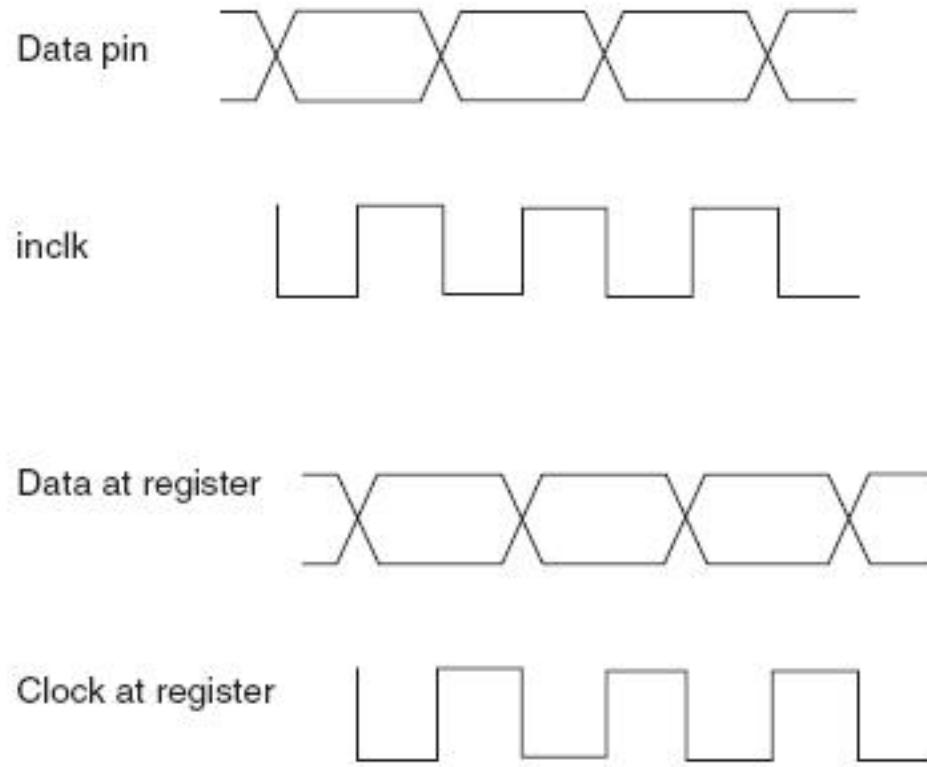


Notes to Figure 7–6:

- (1) Internal clocks fed by the PLL are in phase with each other.
 - (2) The external clock outputs can lead or lag the PLL internal clocks.
-

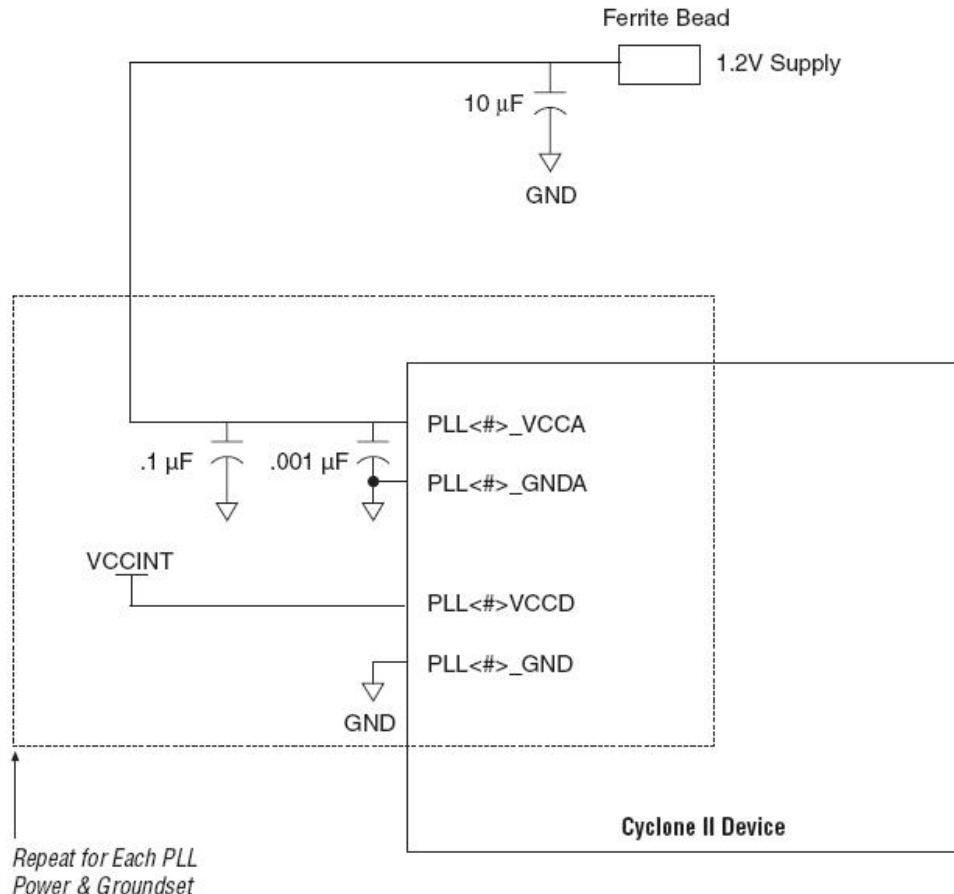
I/O Path Compensation in Cyclone II PLLs

Figure 7–7. Phase Relationship between Cyclone II PLL Clocks in Source-Synchronous Compensation Mode



PLL Power Management Issues

Figure 7-17. PLL Power Schematic for Cyclone II PLLs



Note to Figure 7-17:

- (1) Applies to PLLs 1 through 4.

“Gating” Clocks for Reduced Power Consumption in Cyclone II FPGAs

Figure 7–14. clkena Implementation

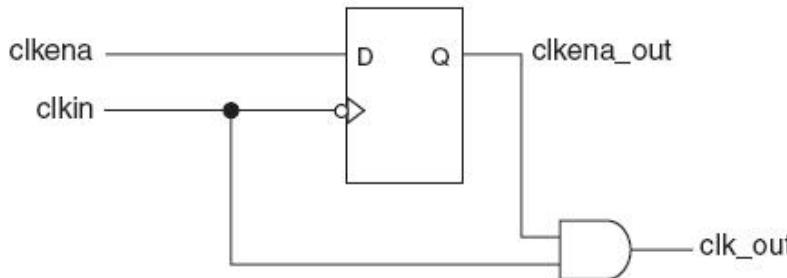
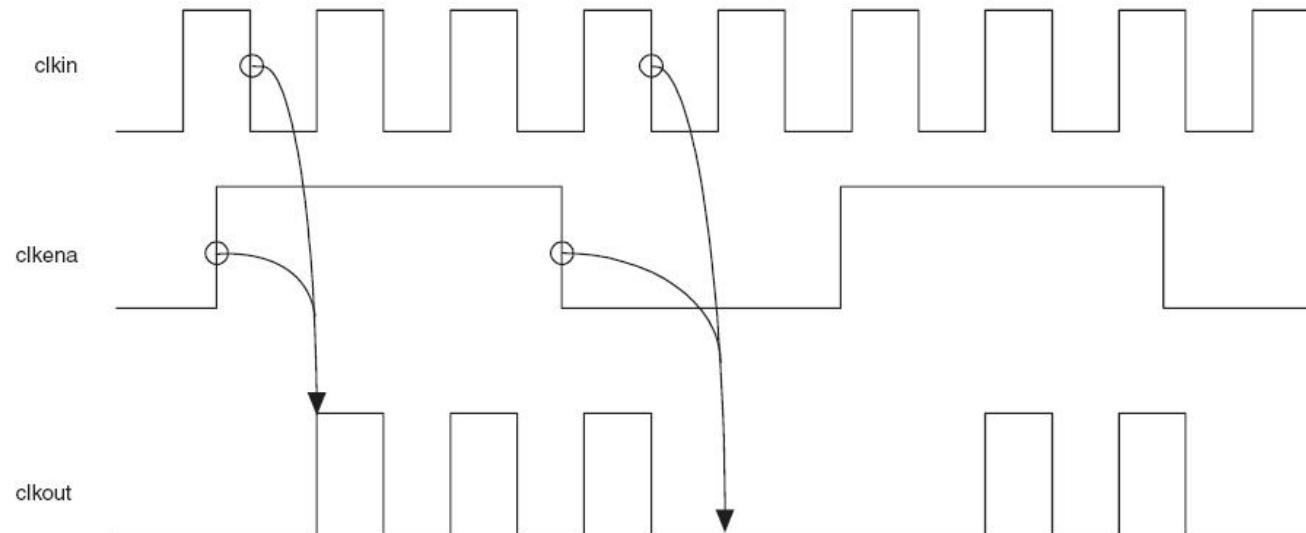
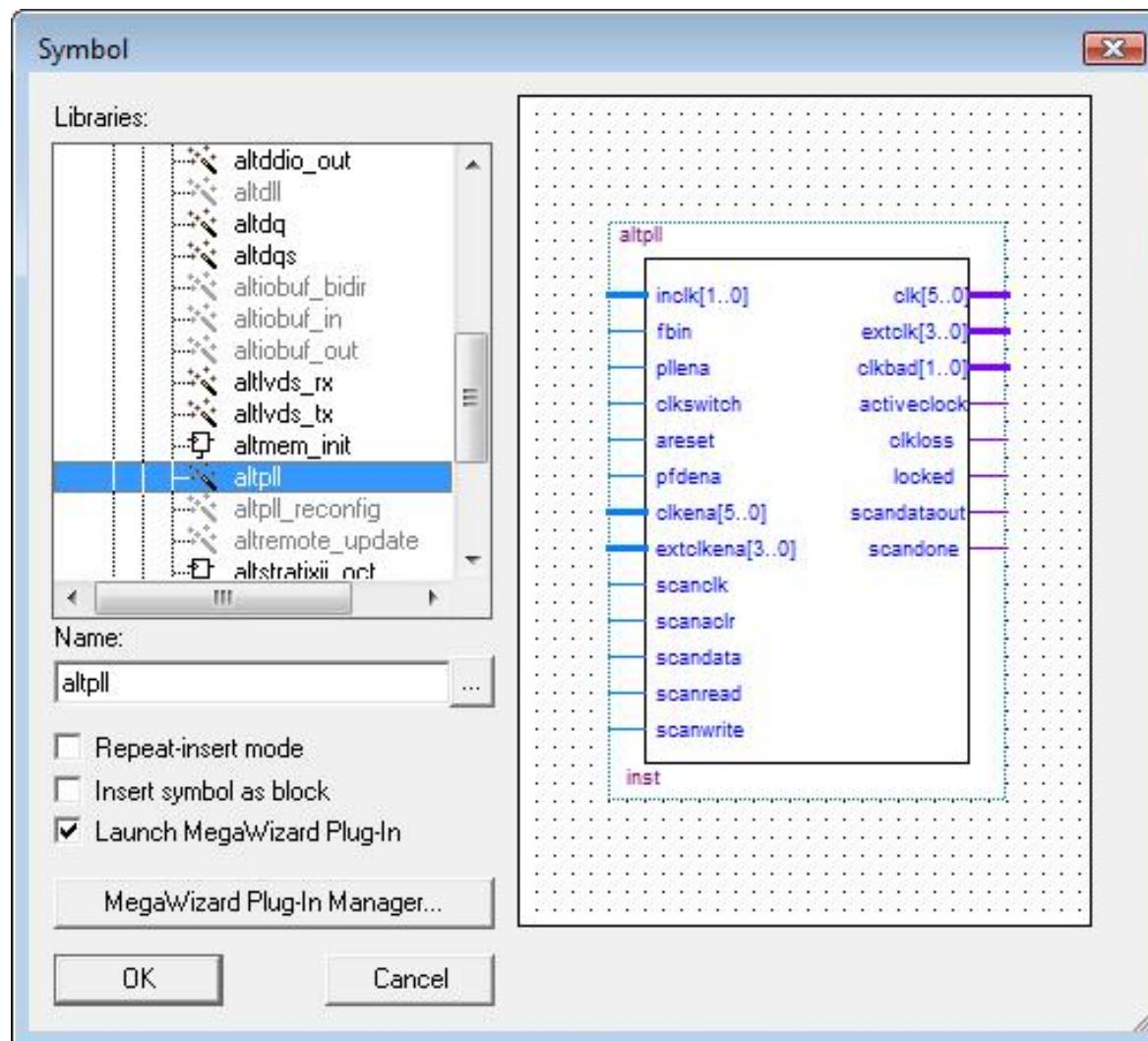


Figure 7–15. clkena Implementation



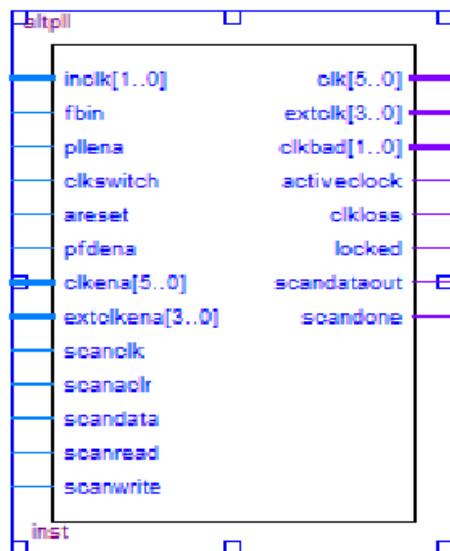
Configuring Cyclone II PLLs in Quartus II



PLL Instantiation

```
// This is the model of the clock converter device that utilizes one
// of the PLL elements in the Cyclone IV E to multiply the external
// clock by a factor of 4/5. This results in a 40 Mhz clock being
// produced whenever the module is driven by a 50Mhz signal. This is
// the frequency needed to obtain 800 x 600 60 Hz SVGA resolution.
module pll_clk_50_to_40(input clk_in,output clk_out);
    wire [4:0] clockgrp;
    assign clk_out = clockgrp[0:0];
```

```
altpll altpll_component (
    .inclk ({1'h0, clk_in}),
    .clk (clockgrp),
    .activeclock (),
    .areset (1'b0),
    .clkbad (),
    .clkena ({6{1'b1}}),
    .clkloss (),
    .clkswitch (1'b0),
    .configupdate (1'b0),
    .enable0 (),
    .enable1 (),
    .extclk (),
    .extclkena ({4{1'b1}}),
    .fbin (1'b1),
    .fbmimicbidir (),
    .fbout (),
    .ref (),
    .icdrclk (),
    .locked (),
    .pfdena (1'b1),
    .phasecounterselect ({4{1'b1}}),
    .phasedone (),
    .phasestep (1'b1),
    .phaseupdown (1'b1),
    .pllена (1'b1),
    .scanaclr (1'b0),
    .scanclk (1'b0),
    .scanklenka (1'b1),
    .scandata (1'b0),
    .scandataout (),
    .scandone (),
    .scanread (1'b0),
    .scanwrite (1'b0),
    .sclkout0 (),
    .sclkout1 (),
    .vcooverrange (),
    .vcounderrange () );
```



defparam

```
altpll_component.bandwidth_type = "AUTO",
altpll_component.clk0_divide_by = 5,
altpll_component.clk0_duty_cycle = 50,
altpll_component.clk0_multiply_by = 4,
altpll_component.clk0_phase_shift = "0",
altpll_component.compensate_clock = "CLK0",
altpll_component.inclk0_input_frequency = 20000,
altpll_component.intended_device_family = "Cyclone IV E",
altpll_component.lpm_hint = "CBX_MODULE_PREFIX=altpll0",
altpll_component.lpm_type = "altpll",
altpll_component.operation_mode = "NORMAL",
altpll_component pll_type = "AUTO",
altpll_component.port_activeclock = "PORT_UNUSED",
altpll_component.port_areset = "PORT_UNUSED",
altpll_component.port_clkbad0 = "PORT_UNUSED",
altpll_component.port_clkbad1 = "PORT_UNUSED",
altpll_component.port_clkloss = "PORT_UNUSED",
altpll_component.port_clkswitch = "PORT_UNUSED",
altpll_component.port_configupdate = "PORT_UNUSED",
altpll_component.port_fbin = "PORT_UNUSED",
altpll_component.port_inclk0 = "PORT_USED",
altpll_component.port_inclk1 = "PORT_UNUSED",
altpll_component.port_locked = "PORT_UNUSED",
altpll_component.port_pfdena = "PORT_UNUSED",
altpll_component.port_phasecounterselect = "PORT_UNUSED",
altpll_component.port_phasedone = "PORT_UNUSED",
altpll_component.port_phASESTEP = "PORT_UNUSED",
altpll_component.port_phaseupdown = "PORT_UNUSED",
altpll_component.port_pllена = "PORT_UNUSED",
altpll_component.port_scanaclr = "PORT_UNUSED",
altpll_component.port_scanclk = "PORT_UNUSED",
altpll_component.port_scanklenka = "PORT_UNUSED",
altpll_component.port_scandata = "PORT_UNUSED",
altpll_component.port_scandataout = "PORT_UNUSED",
altpll_component.port_scandone = "PORT_UNUSED",
altpll_component.port_scanread = "PORT_UNUSED",
altpll_component.port_scanwrite = "PORT_UNUSED",
altpll_component.port_clk0 = "PORT_USED",
altpll_component.port_clk1 = "PORT_UNUSED",
altpll_component.port_clk2 = "PORT_UNUSED",
altpll_component.port_clk3 = "PORT_UNUSED",
altpll_component.port_clk4 = "PORT_UNUSED",
altpll_component.port_clk5 = "PORT_UNUSED",
altpll_component.port_clkена0 = "PORT_UNUSED",
altpll_component.port_clkена1 = "PORT_UNUSED",
altpll_component.port_clkена2 = "PORT_UNUSED",
altpll_component.port_clkена3 = "PORT_UNUSED",
altpll_component.port_clkена4 = "PORT_UNUSED",
altpll_component.port_clkена5 = "PORT_UNUSED",
altpll_component.port_extclk0 = "PORT_UNUSED",
altpll_component.port_extclk1 = "PORT_UNUSED",
altpll_component.port_extclk2 = "PORT_UNUSED",
altpll_component.port_extclk3 = "PORT_UNUSED",
altpll_component.width_clock = 5;
```

CPE 322

Digital Hardware Design Fundamentals

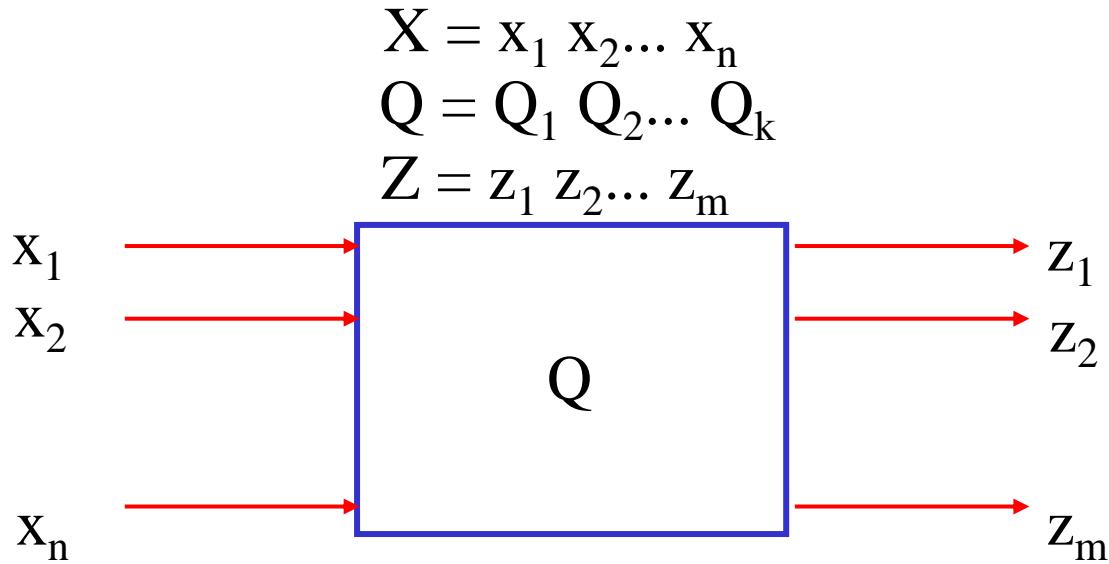
Electrical and Computer Engineering

Finite State Machine Representation &
Verilog



Sequential Networks

- Have memory (state)
 - Present state depends not only on the current input, but also on all previous inputs (history)
 - Future state depends on the current input and state



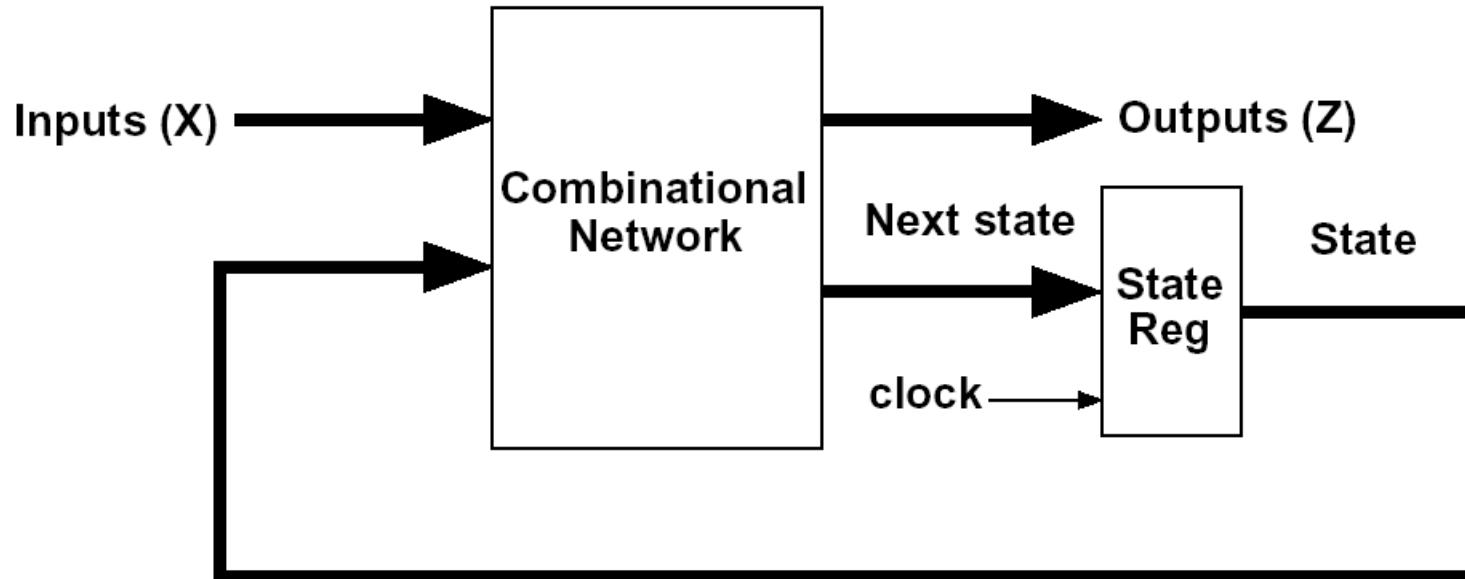
$$Z(t) = F(X(t), Q(t))$$

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

Flip-flops are commonly used as storage devices:
D-FF, JK-FF, T-FF

Mealy Sequential Networks

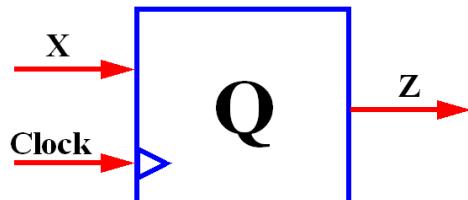
General model of Mealy Sequential Network



$$Q(t^+) = G(X(t), Q(t)) \quad Z(t) = F(X(t), Q(t))$$

- (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 (input of State Register)
- (3) The next state is clocked into the state register and the state changes

An Example: 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

State Graph

BCD to Exess3 Code Converter

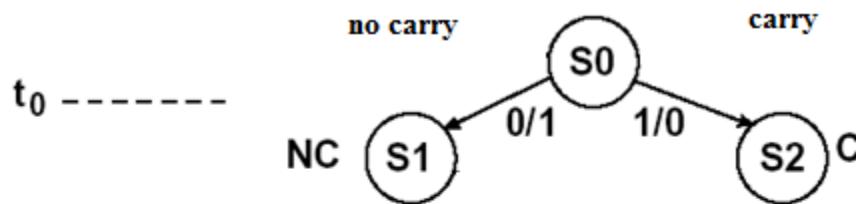
X/Z



State Graph

BCD to Exess3 Code Converter

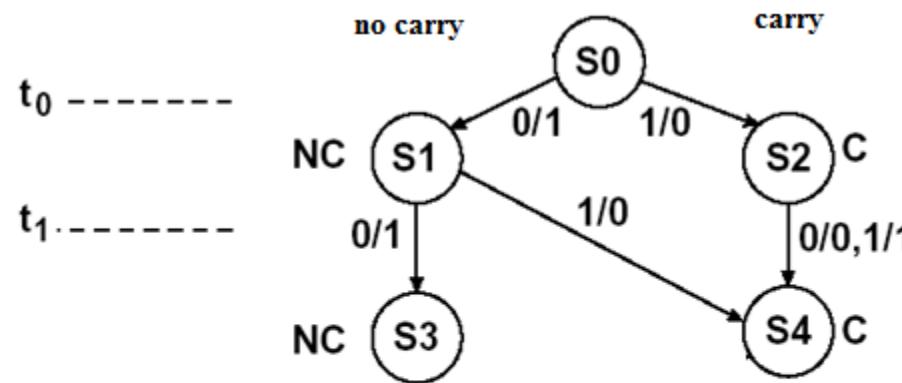
X/Z



State Graph

BCD to Exess3 Code Converter

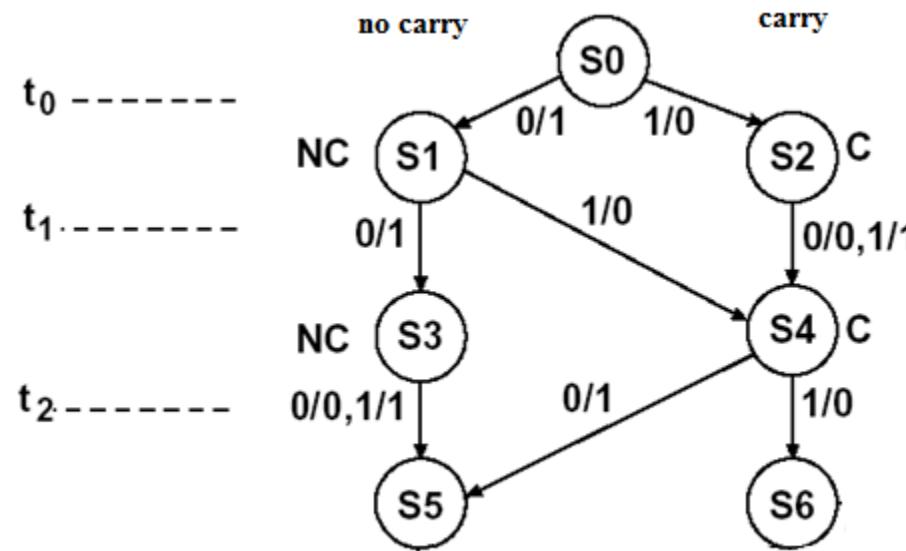
X/Z



State Graph

BCD to Exess3 Code Converter

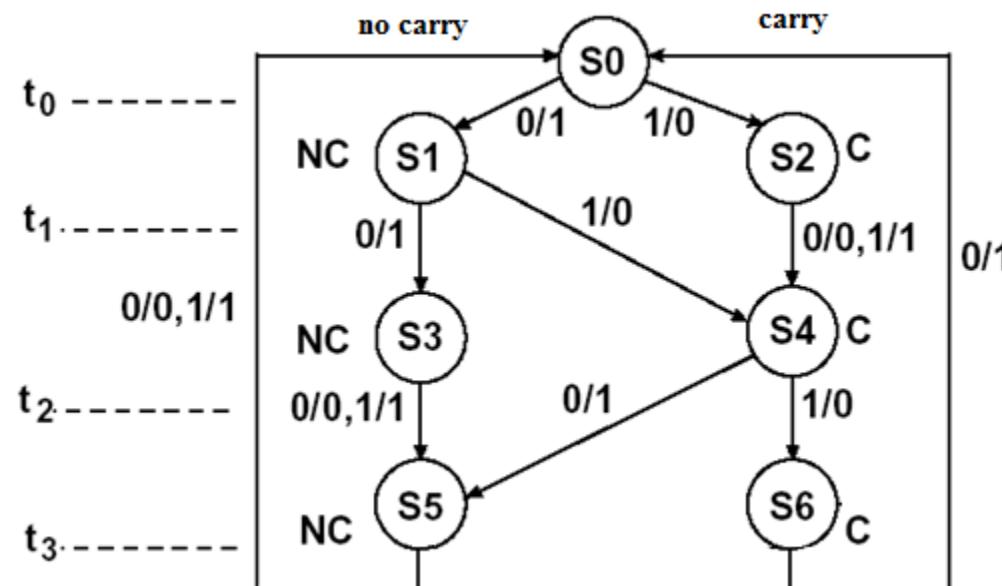
X/Z



State Graph

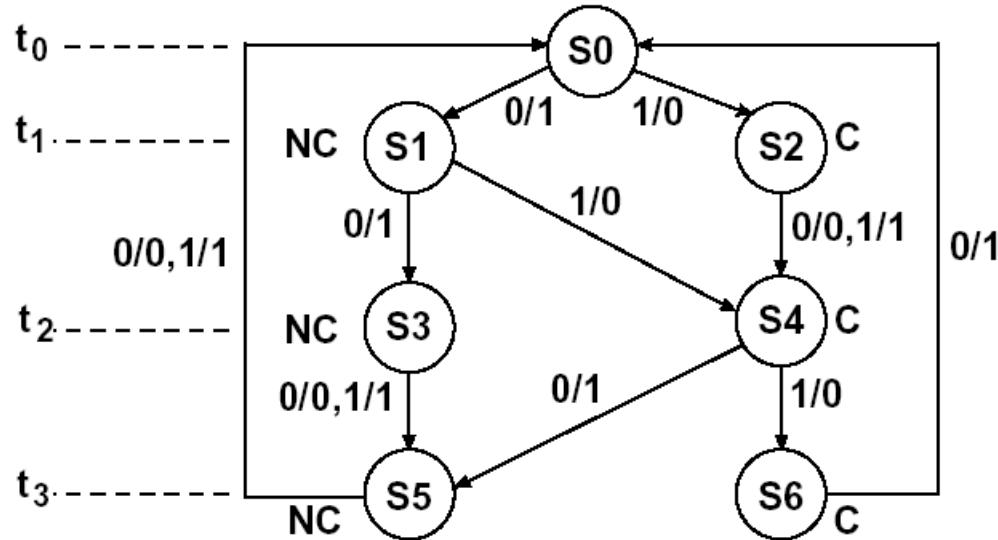
BCD to Exess3 Code Converter

X/Z



State Graph and Table for BCD to Exess3 Code Converter

X/Z



PS	NS		Z	
	X = 0	X = 1	X = 0	X = 1
S0	S1	S2	1	0
S1	S3	S4	1	0
S2	S4	S4	0	1
S3	S5	S5	0	1
S4	S5	S6	1	0
S5	S0	S0	0	1
S6	S0	-	1	-

Behavioral Verilog HDL Model

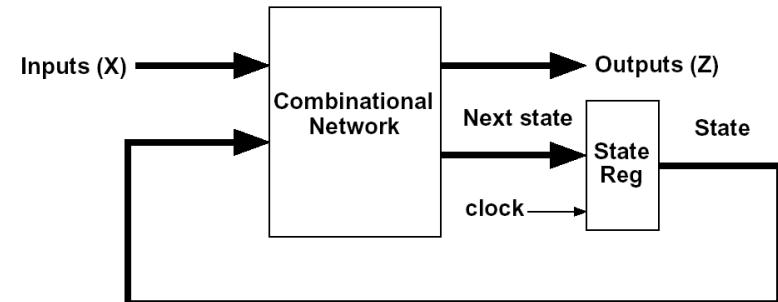
8421 BCD to Excess3 Code Converter (Mealy FSM)

```
// bcd to excess 3 converter
// Mealy Implementation
// Behavioral Model
module bcd_ex3_mealy(input X,CLK,output reg Z);
    reg [2:0] State=0,Nextstate=0;

    // Combinational Network
    always @ (State,X)
        case (State)
            0 : if (!X) begin Nextstate=1;Z=1; end
                else begin Nextstate=2; Z=0; end
            1 : if (!X) begin Nextstate=3; Z=1; end
                else begin Nextstate=4; Z=0; end
            2 : if (!X) begin Nextstate=4; Z=0; end
                else begin Nextstate=4; Z=1; end
            3 : if (!X) begin Nextstate=5; Z=0; end
                else begin Nextstate=5; Z=1; end
            4 : if (!X) begin Nextstate=5; Z=1; end
                else begin Nextstate=6; Z=0; end
            5 : if (!X) begin Nextstate=0; Z=0; end
                else begin Nextstate=0; Z=1; end
            6 : if (!X) begin Nextstate=0; Z=1; end
                else begin Nextstate=0; Z=1'b?; end
            default : begin Nextstate=0; Z=0;end
        endcase

    // State Reg Portion of Design
    always @ (posedge CLK)
        State=Nextstate;

endmodule
```



PS	NS		Z	
	X = 0	X = 1	X = 0	X = 1
S0	S1	S2	1	0
S1	S3	S4	1	0
S2	S4	S4	0	1
S3	S5	S5	0	1
S4	S5	S6	1	0
S5	S0	S0	0	1
S6	S0	-	1	-

Two always sections:

- the first represents the combinational network;
- the second represents the state register

State Assignment Rules

- 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.
-
- I. (1,2) (3,4) (5,6) (in the X=1 column, S₁ and S₂ both have NS S₄; in the X=0 column, S₃ & S₄ have NS S₅, and S₅ & S₆ have NS S₀)
 - II. (1,2) (3,4) (5,6) (S₁ & S₂ are NS of S₀; S₃ & S₄ are NS of S₁; and S₅ & S₆ are NS of S₄)
 - III. (0,1,4,6) (2,3,5)

PS	NS		Z	
	X = 0	X = 1	X = 0	X = 1
S ₀	S ₁	S ₂	1	0
S ₁	S ₃	S ₄	1	0
S ₂	S ₄	S ₄	0	1
S ₃	S ₅	S ₅	0	1
S ₄	S ₅	S ₆	1	0
S ₅	S ₀	S ₀	0	1
S ₆	S ₀	—	1	—

State Assignment Rules

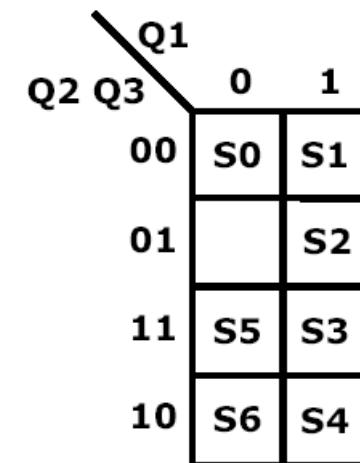
- I. States which have the same next state (NS) for a given input should be given adjacent assignments (look at the columns of the state table).
- II. States which are the next states of the same state should be given adjacent assignments (look at the rows).
- III. States which have the same output for a given input should be given adjacent assignments.

I. (1,2) (3,4) (5,6) (in the X=1 column, S₁ and S₂ both have NS S₄; in the X=0 column, S₃ & S₄ have NS S₅, and S₅ & S₆ have NS S₀)

II. (1,2) (3,4) (5,6) (S₁ & S₂ are NS of S₀; S₃ & S₄ are NS of S₁; and S₅ & S₆ are NS of S₄)

III. (0,1,4,6) (2,3,5)

PS	NS		Z	
	X = 0	X = 1	X = 0	X = 1
S ₀	S ₁	S ₂	1	0
S ₁	S ₃	S ₄	1	0
S ₂	S ₄	S ₄	0	1
S ₃	S ₅	S ₅	0	1
S ₄	S ₅	S ₆	1	0
S ₅	S ₀	S ₀	0	1
S ₆	S ₀	-	1	-



Transition Table

PS	NS		Z		Q1Q2Q3	Q1 [*] Q2 [*] Q3 [*]		Z	
	X=0	X=1	X=0	X=1		X=0	X=1	X=0	X=1
S0	S1	S2	1	0	000	100	101	1	0
S1	S3	S4	1	0	100	111	110	1	0
S2	S4	S4	0	1	101	110	110	0	1
S3	S5	S5	0	1	111	011	011	0	1
S4	S5	S6	1	0	110	011	010	1	0
S5	S0	S0	0	1	011	000	000	0	1
S6	S0	-	1	-	010	000	xxx	1	x
					001	xxx	xxx	x	x

$S_0 = 000, S_1 = 100, S_2 = 101, S_3 = 111, S_4 = 110, S_5 = 011, S_6 = 010$

K-maps

		XQ ₁	00	01	11	10
		Q ₂ Q ₃	00	01	11	10
Q ₂	Q ₃	00	1	1	1	1
		01	X	1	1	X
11	10	00	0	0	0	0
		01	0	0	0	X

$$D_1 = Q_1^+ = Q_2'$$

		XQ ₁	00	01	11	10
		Q ₂ Q ₃	00	01	11	10
Q ₂	Q ₃	00	0	1	1	0
		01	X	1	1	X
11	10	00	0	1	1	0
		01	0	1	1	X

$$D_2 = Q_2^+ = Q_1$$

		XQ ₁	00	01	11	10
		Q ₂ Q ₃	00	01	11	10
Q ₂	Q ₃	00	0	1	0	1
		01	X	0	0	X
11	10	00	0	1	1	0
		01	0	1	0	X

$$D_3 = Q_3^+ = Q_1 Q_2 Q_3 + X' Q_1 Q_3' + X Q_1' Q_2'$$

		XQ ₁	00	01	11	10
		Q ₂ Q ₃	00	01	11	10
Q ₂	Q ₃	00	1	1	0	0
		01	X	0	1	X
11	10	00	0	0	1	1
		01	1	1	0	X

$$Z = X' Q_3' + X Q_3$$

Data Flow Representation Verilog HDL

8421 BCD to Excess3 Code Converter (Mealy FSM)

```
// bcd to excess 3 converter
// Mealy Implementation
// Data Flow Model
module bcd_ex3_mealy(input X,CLK,output Z);

reg Q1=0,Q2=0,Q3=0;

// FF State Update Portion of design
// Active only on rising edge of clock
always @(posedge CLK)
begin
    Q1 <= ~Q2;
    Q2 <= Q1;
    Q3 <= (Q1 & Q2 & Q3) | (~X & Q1 & ~Q3) | (X & ~Q1 & ~Q2);
end

// Output Equation -- Continuous Assignment that is
// a function only of the state variables Q1,Q2,Q3
assign Z = (~X & ~Q3) | (X & Q3);

endmodule
```

$$\begin{aligned}Q_1^+ &= Q_2' \\Q_2^+ &= Q_1 \\Q_3^+ &= Q_1 Q_2 Q_3 + X' Q_1 Q_3' + X Q_1' Q_2' \\Z &= X' Q_3' + X Q_3\end{aligned}$$

Schematic Realization

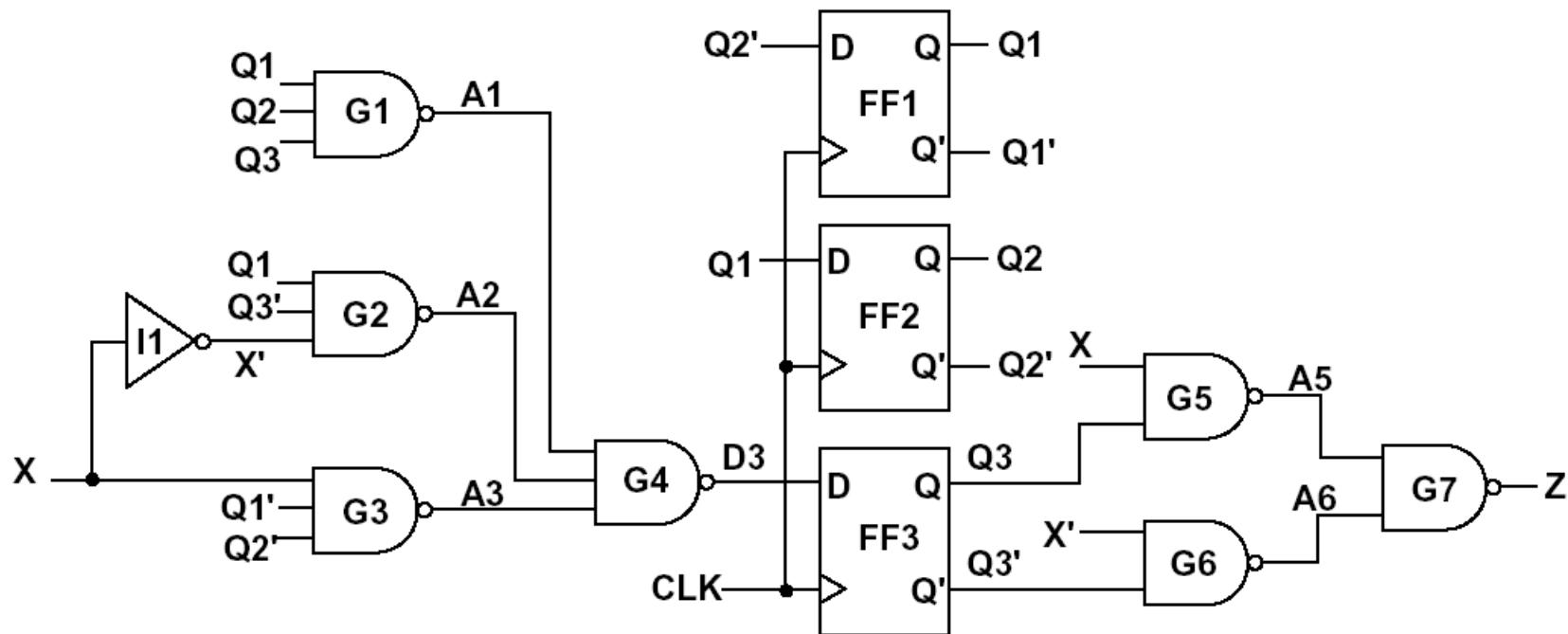
8421 BCD to Excess3 Code Converter (Mealy FSM)

$$Q_1^+ = Q_2'$$

$$Q_2^+ = Q_1$$

$$Q_3^+ = Q_1 Q_2 Q_3 + X' Q_1 Q_3' + X Q_1' Q_2'$$

$$Z = X' Q_3' + X Q_3$$



Structural Representation Verilog HDL

8421 BCD to Excess3 Code Converter (Mealy FSM)

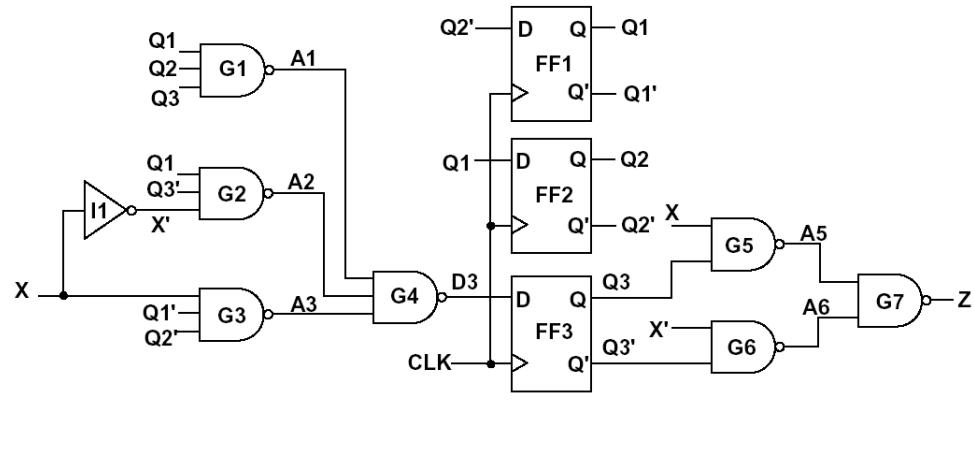
```
// bcd to excess 3 converter
// Mealy Implementation
// Structural Model
module bcd_ex3_mealy(input X,CLK,output Z);
    wire A1,A2,A3,A5,A6,D3,Q1,Q1_N,Q2,Q2_N,Q3

    not I1(X_N,X);
    nand G1(A1,Q1,Q2,Q3);
    nand G2(A2,Q1,Q3_N,X_N);
    nand G3(A3,X,Q1_N,Q2_N);
    nand G4(D3,A1,A2,A3);
    nand G5(A5,X,Q3);
    nand G6(A6,X_N,Q3_N);
    nand G7(Z,A5,A6);

    d_ff FF1(CLK,Q2_N,Q1,Q1_N);
    d_ff FF2(CLK,Q1,Q2,Q2_N);
    d_ff FF3(CLK,D3,Q3,Q3_N);

endmodule

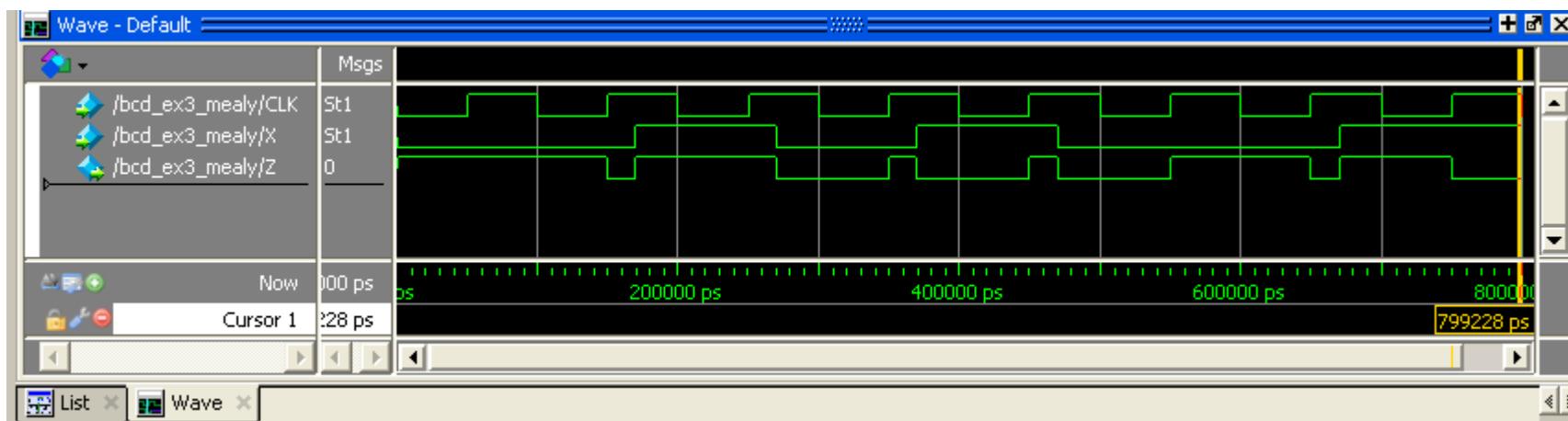
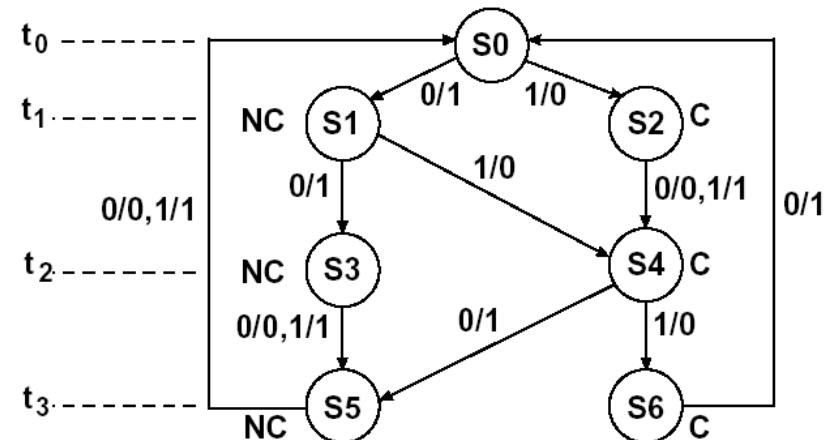
// basic rising edge D flip-flop module
module d_ff(input clk,d, output reg q, output q_n);
    initial q=0;
    always @ (posedge clk)
        q = d;
    assign q_n = ~q;
endmodule
```



Sequential Network Timing

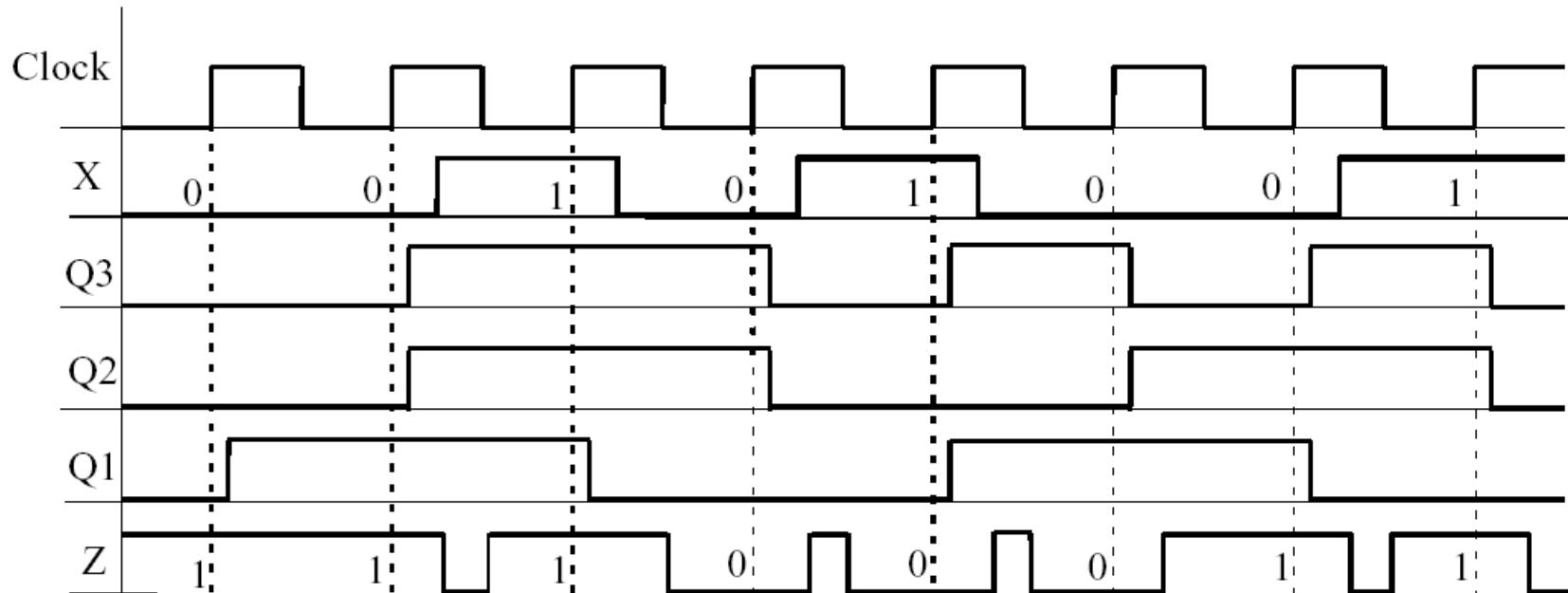
- Code converter
 - $X = 0010_1001 \Rightarrow Z = 1110_0011$

Changes in X are not synchronized with active clock edge => glitches (false output), e.g. at tb

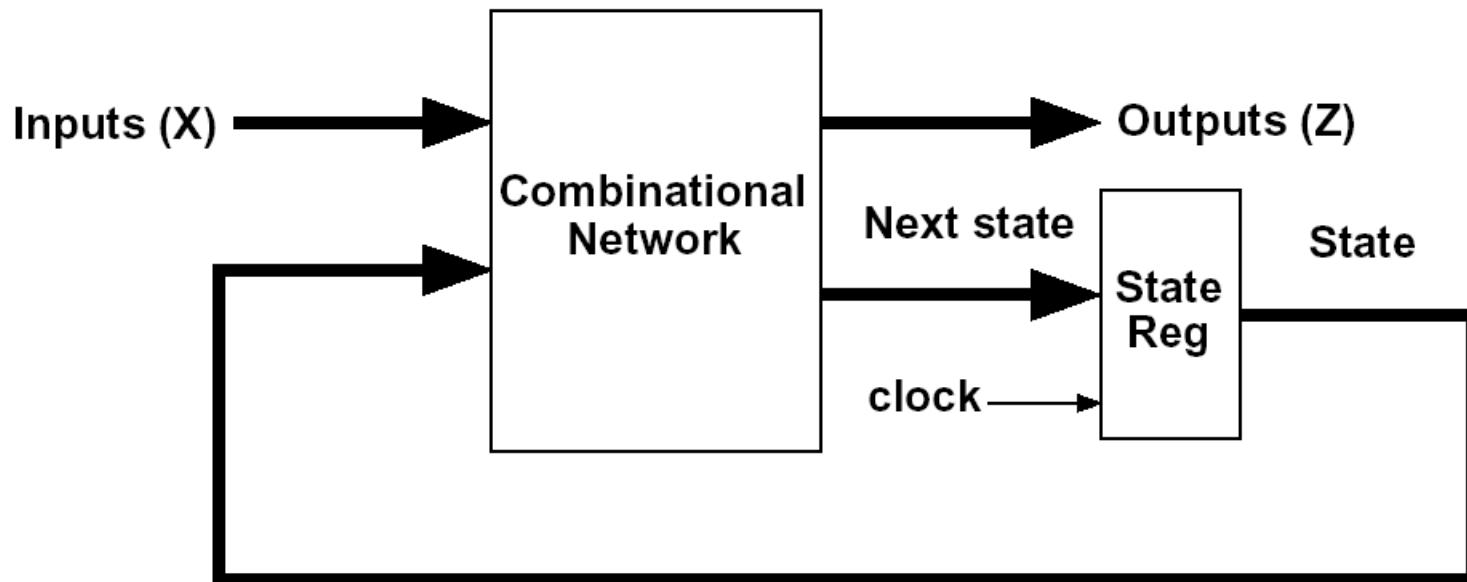


Sequential Network Timing (cont'd)

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)



Review: Mealy Sequential Networks

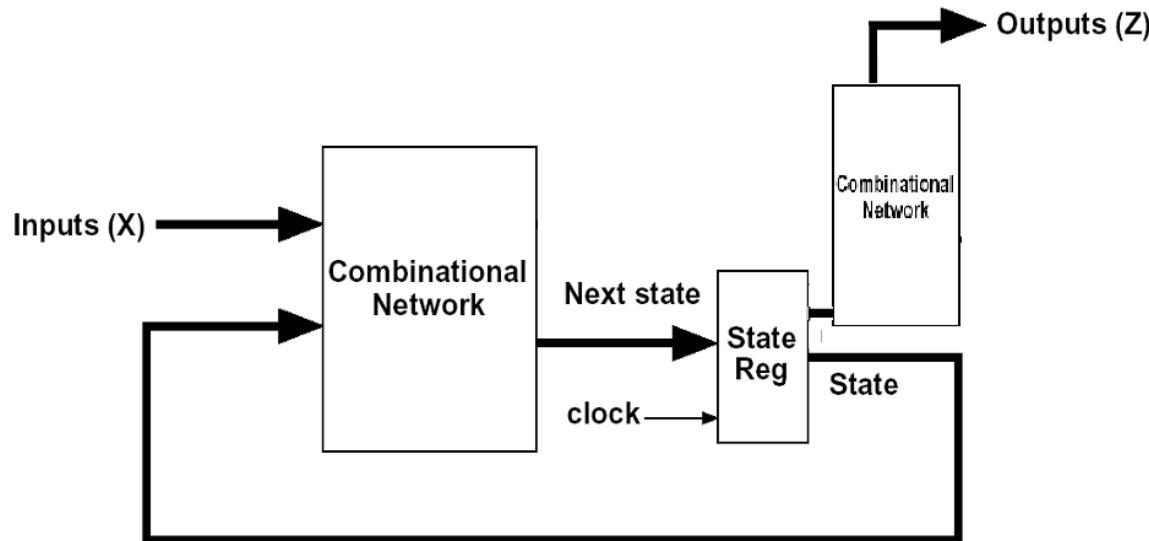


$$Q(t^+) = G(X(t), Q(t)) \quad Z(t) = F(X(t), Q(t))$$

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

Moore Sequential Networks

General model of Moore Sequential Network



- (1) X inputs are changed to a new value
- (2) The next state is clocked into the state registers
- (3) The outputs appear which are a function of the current state only (not the inputs)

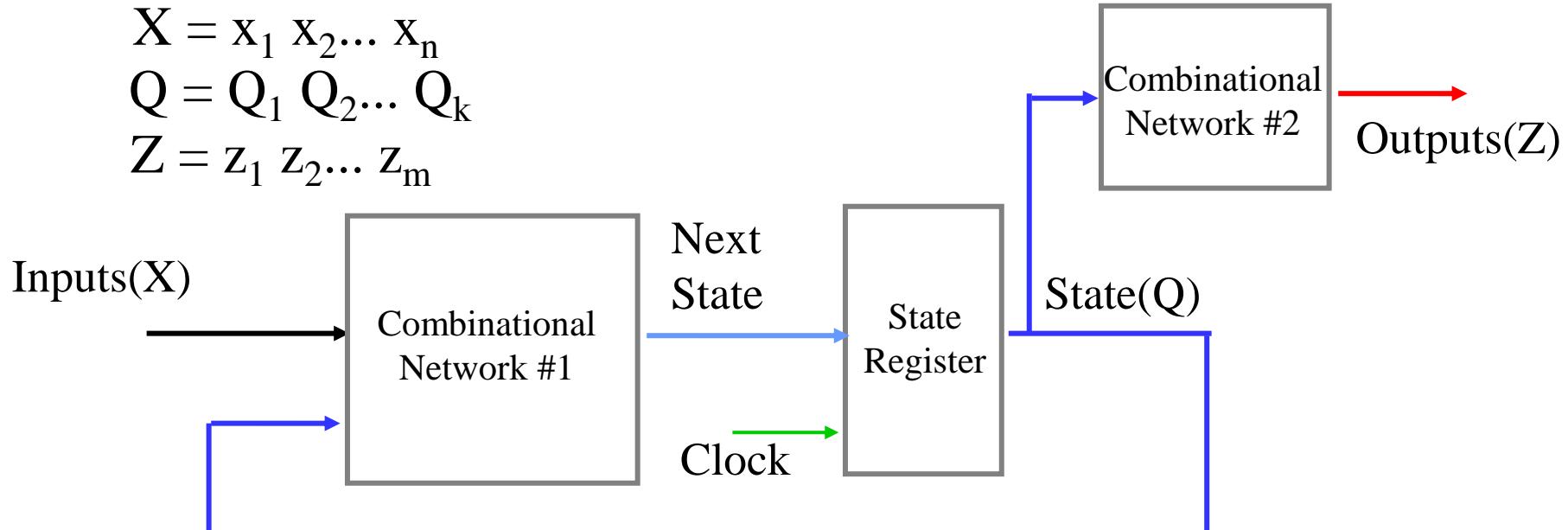
General Model of Moore Sequential Machine

Outputs depend only on present state!

$$X = X_1 \ X_2 \dots \ X_n$$

$$Q = Q_1 \ Q_2 \dots \ Q_k$$

$$Z = Z_1 \ Z_2 \dots \ Z_m$$

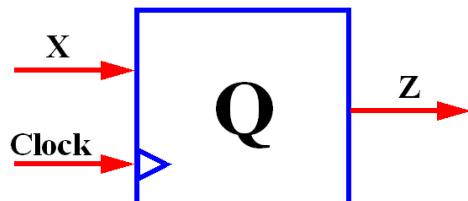


$$Q(t^+) = G(X(t), Q(t))$$

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

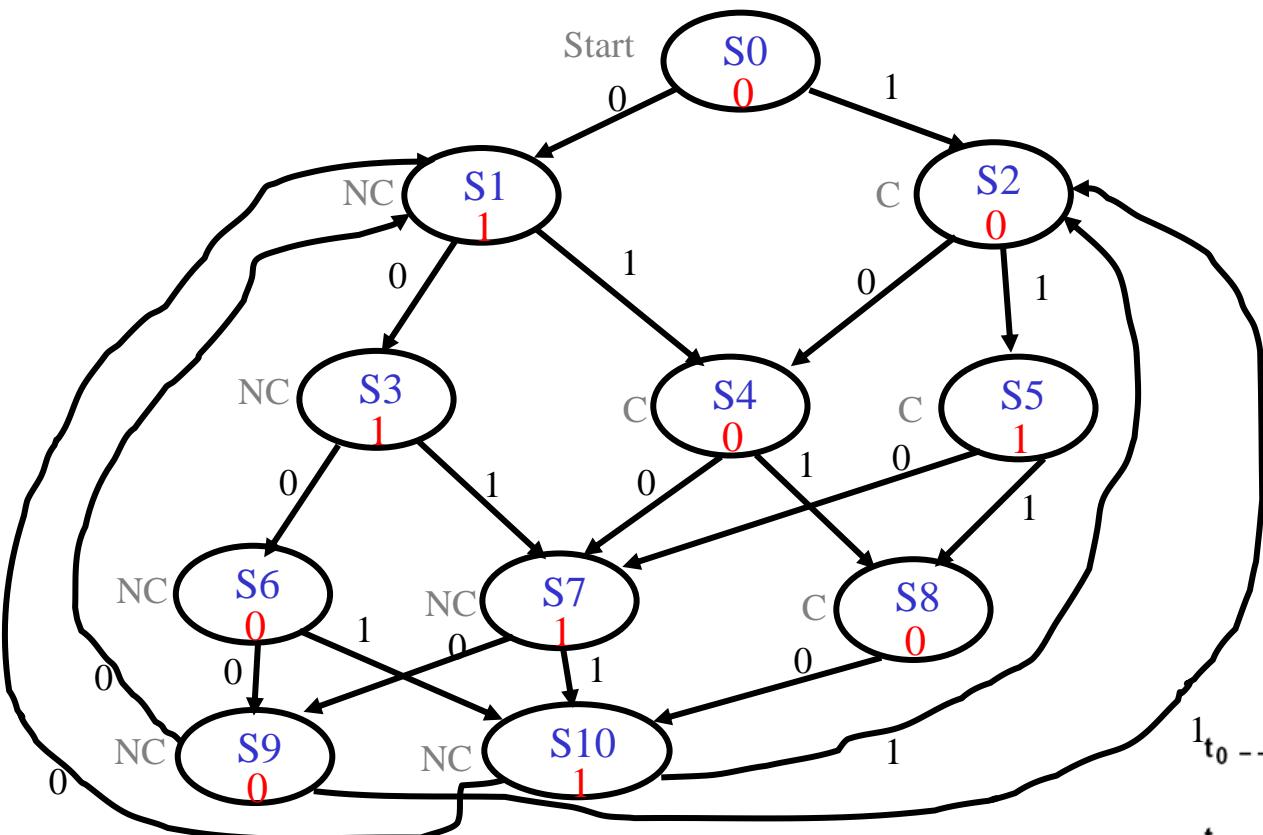
- (1) X inputs are changed to a new value
- (2) After a delay, the output of CN appears on the inputs of the state register
- (3) The next state is clocked into the state register resulting in a possible change in state, and possible new Z outputs appear after a CN delay

An Example: 8421 BCD to Excess3 Code Converter

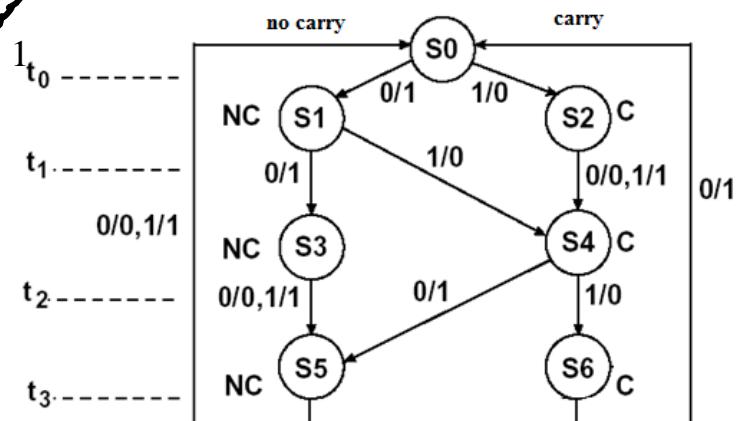


X (inputs)				Z (outputs)			
t3	t2	t1	t0	t3	t2	t1	t0
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

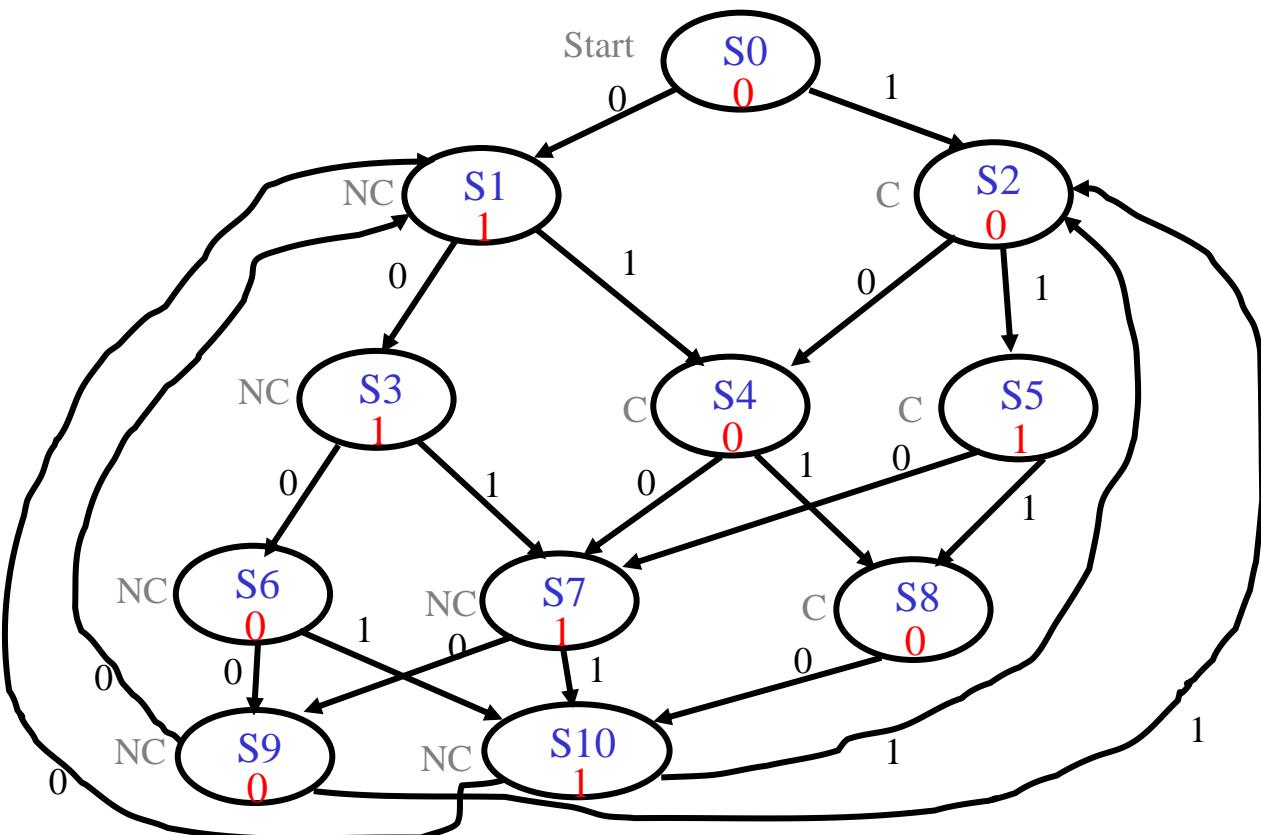
Moore Machine



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



Moore Machine



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

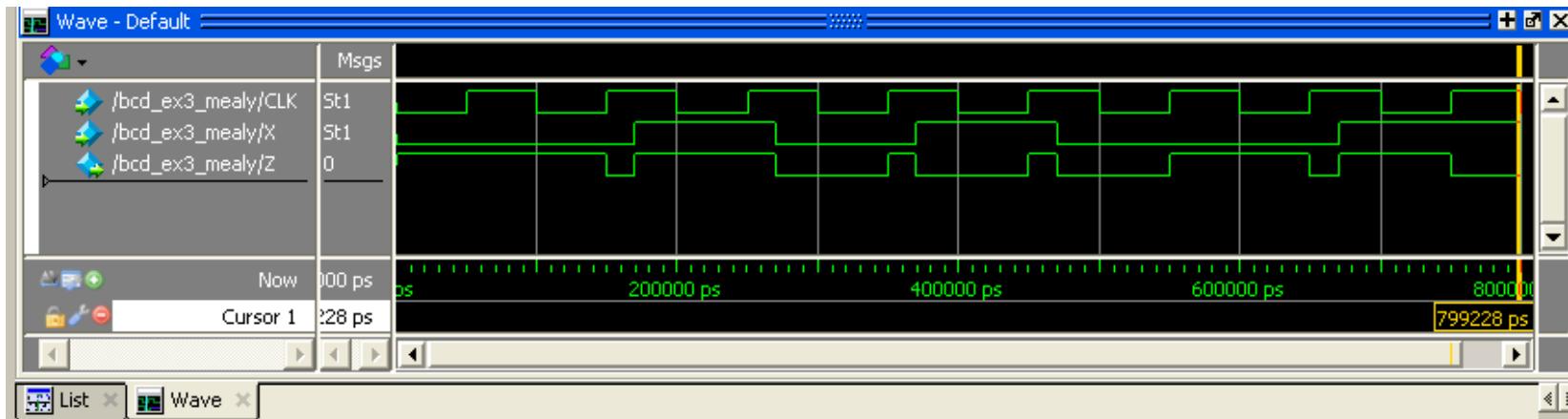
PS	NS		Z
	X=0	X=1	
S0	S1	S2	0
S1	S3	S4	1
S2	S4	S5	0
S3	S6	S7	1
S4	S7	S8	0
S5	S7	S8	1
S6	S9	S10	0
S7	S9	S10	1
S8	S10	-	0
S9	S1	S2	0
S10	S1	S2	1

FSM Timings

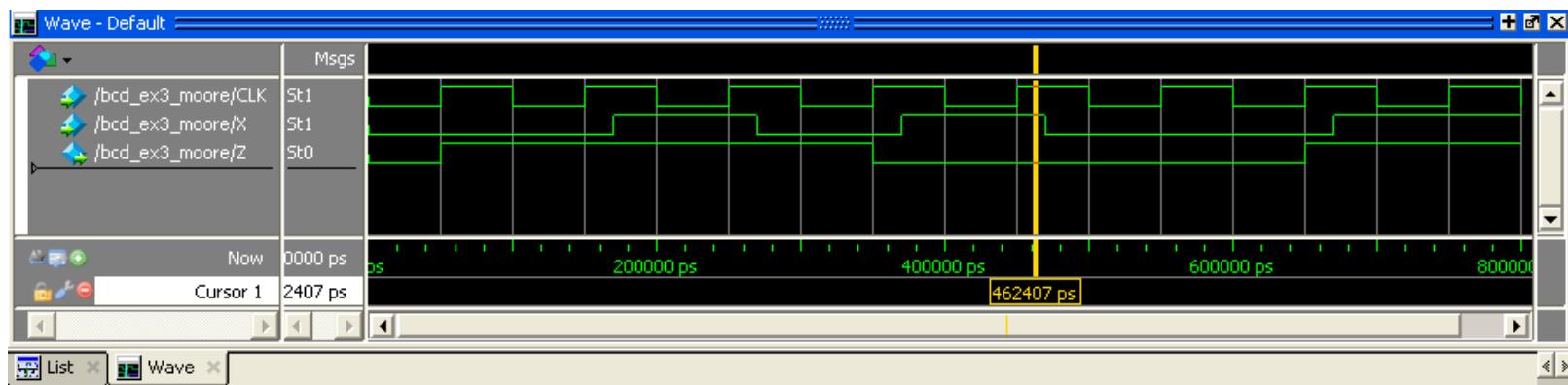
Differences between Mealy and Moore

- $X = 0010_1001 \Rightarrow Z = 1110_0011$

Mealy

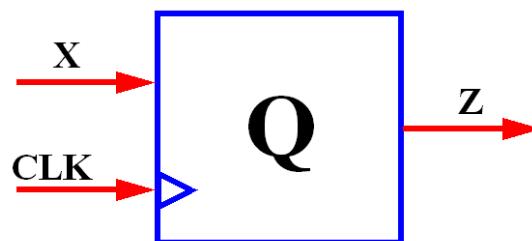
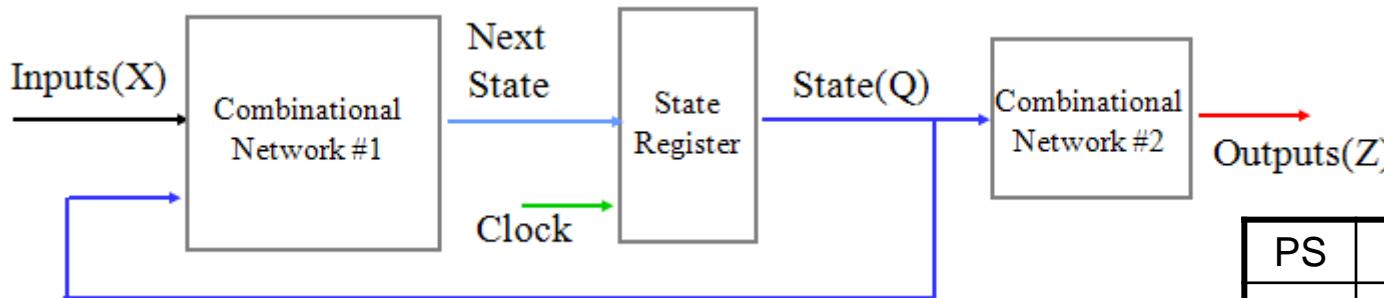


Moore



Behavioral Verilog HDL Model

8421 BCD to Excess3 Code Converter (Moore FSM)



```
// bcd to excess 3 converter
// Moore Implementation
// Behavioral Model
module bcd_ex3_moore(input X,CLK,output reg Z);

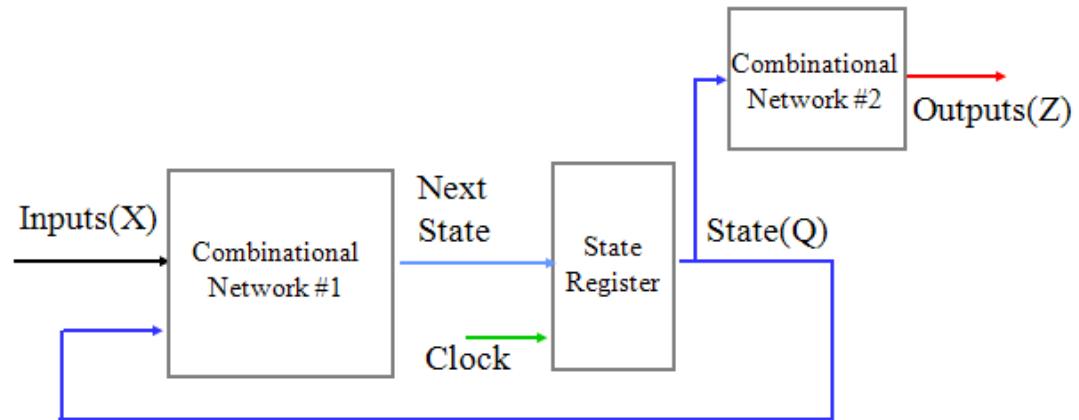
reg [3:0] state=0,Nextstate=0;
*
*
*
endmodule
```

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

Behavioral Verilog HDL Model

8421 BCD to Excess3 Code Converter (Moore FSM)

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



Behavioral Verilog HDL Model

8421 BCD to Excess3 Code Converter (Moore FSM)

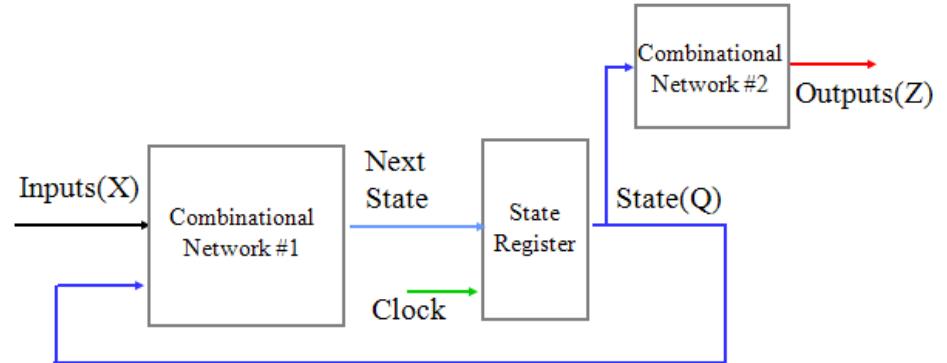
PS	NS		Z
	X=0	X=1	
S0	S1	S2	0
S1	S3	S4	1
S2	S4	S5	0
S3	S6	S7	1
S4	S7	S8	0
S5	S7	S8	1
S6	S9	S10	0
S7	S9	S10	1
S8	S10	-	0
S9	S1	S2	0
S10	S1	S2	1

```
// combinational Network #1
always @ (State,X)
  case (State)
    0 : if (!X) Nextstate=1;
         else Nextstate=2;
    1 : if (!X) Nextstate=3;
         else Nextstate=4;
    2 : if (!X) Nextstate=4;
         else Nextstate=5;
    3 : if (!X) Nextstate=6;
         else Nextstate=7;
    4 : if (!X) Nextstate=7;
         else Nextstate=8;
    5 : if (!X) Nextstate=7;
         else Nextstate=8;
    6 : if (!X) Nextstate=9;
         else Nextstate=10;
    7 : if (!X) Nextstate=9;
         else Nextstate=10;
    8 : Nextstate=10;
    9 : if (!X) Nextstate=1;
         else Nextstate=2;
   10: if (!X) Nextstate=1;
         else Nextstate=2;
  default : Nextstate=0;
endcase
```

Behavioral Verilog HDL Model

8421 BCD to Excess3 Code Converter (Moore FSM)

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



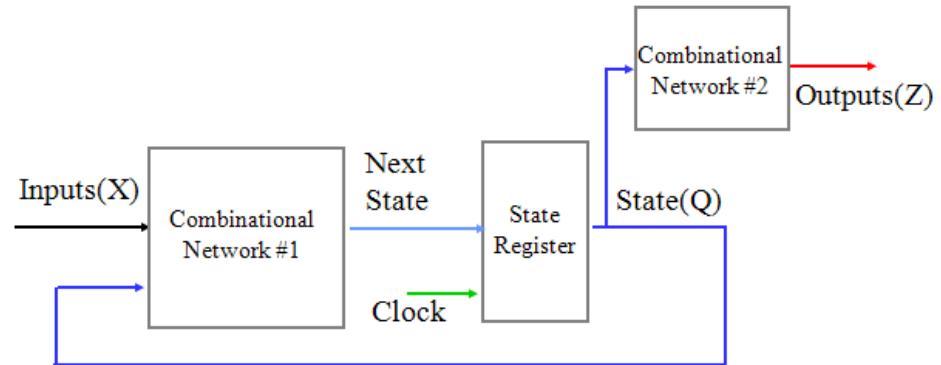
```

// combinational Network #2
always @ (State)
  case (State)
    0 : Z=0;
    1 : Z=1;
    2 : Z=0;
    3 : Z=1;
    4 : Z=0;
    5 : Z=1;
    6 : Z=0;
    7 : Z=1;
    8 : Z=0;
    9 : Z=0;
    10: Z=1;
  default : Z=0;
endcase
  
```

Behavioral Verilog HDL Model

8421 BCD to Excess3 Code Converter (Moore FSM)

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



```
// State Reg Portion of Design
always @ (posedge CLK)
  State=Nextstate;
```

ModelSim™ Waveform

Behavioral 8421 BCD to Excess3 Code Converter (Moore FSM)

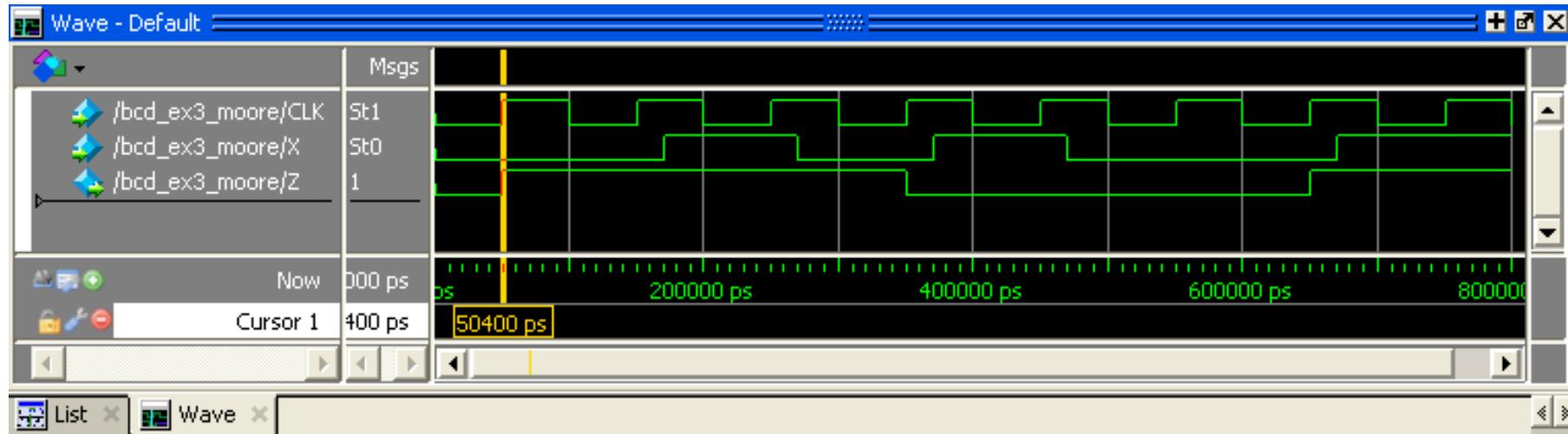
- $X = 0010_1001 \Rightarrow Z = 1110_0011$

[.add wave CLK X Z](#)

[.force CLK 0 0 ns, 1 50 ns -r 100 ns](#)

[.force X 0 0 ns, 1 170 ns, 0 270 ns, 1 370 ns, 0 470 ns, 1 670 ns](#)

[.run 800 ns](#)



One-hot State Encoding

```
// bcd to excess 3 converter
// Moore Implementation
// Data Flow Model -- one hot state assignment
module bcd_ex3_moore(input X,CLK,output Z);
    // flip flop outputs
    reg Q0=1,Q1=0,Q2=0,Q3=0,Q4=0,Q5=0,Q6=0,Q7=0,Q8=0,Q9=0,Q10=0;

    // FF State Update Portion of design
    // Active only on rising edge of clock
    always @(posedge CLK)
        begin
            Q0 <= 0;
            Q1 <= (Q0 & ~X) | (Q9 & ~X) | (Q10 & ~X);
            Q2 <= (Q0 & X) | (Q9 & X) | (Q10 & X);
            Q3 <= Q1 & ~X;
            Q4 <= (Q1 & X) | (Q2 & ~X);
            Q5 <= Q2 & X;
            Q6 <= Q3 & ~X;
            Q7 <= (Q3 & X) | (Q4 & ~X) | (Q5 & ~X);
            Q8 <= (Q4 & X) | (Q5 & X);
            Q9 <= (Q6 & ~X) | (Q7 & ~X);
            Q10 <= (Q6 & X) | (Q7 & X) | (Q8 & ~X);
        end

    // Output Equation -- Continuous Assignment that is
    // a function only of the state variables QA,QB,QC,QD
    assign Z = Q1 | Q3 | Q5 | Q7 | Q10;

endmodule
```

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

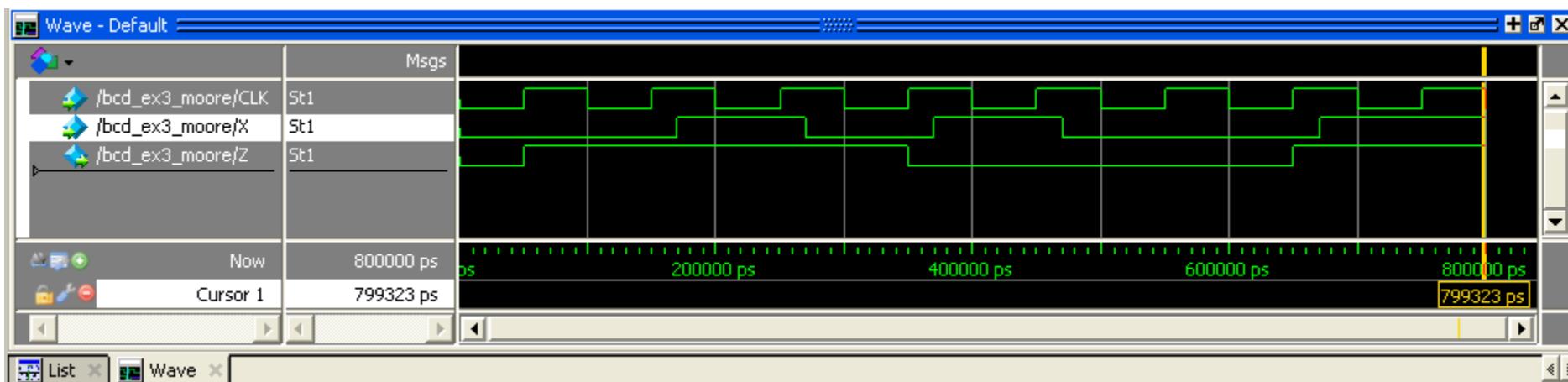
- $X = 0010_1001 \Rightarrow Z = 1110_0011$

- add wave CLK X Z

- force CLK 0 0 ns, 1 50 ns -r 100 ns

- force X 0 0 ns, 1 170 ns, 0 270 ns, 1 370 ns, 0 470 ns, 1 670 ns

- run 800 ns



State Assignment Rules

- 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.
-
- I. (1,2) (3,4) (5,6) (in the X=1 column, S₁ and S₂ both have NS S₄; in the X=0 column, S₃ & S₄ have NS S₅, and S₅ & S₆ have NS S₀)
 - II. (1,2) (3,4) (5,6) (S₁ & S₂ are NS of S₀; S₃ & S₄ are NS of S₁; and S₅ & S₆ are NS of S₄)
 - III. (0,1,4,6) (2,3,5)

Moore State Table

Current State	Next State		Output	
	$x=0$	$x=1$		
s_0	s_1	s_2	0	<u>Rule 1</u> $(0, 9, 10) (4, 5) (6, 7)$
s_1	s_3	s_4	1	
s_2	s_4	s_5	0	<u>Rule 2</u> $(1, 2) (3, 4) (4, 5)$
s_3	s_6	s_7	1.	$(6, 7) (7, 8) (9, 10)$
s_4	s_7	s_8	0	<u>Rule 3</u> $(1, 3, 5, 7, 10)$
s_5	s_7	s_8	1.	
s_6	s_9	s_{10}	0	$(0, 2, 4, 6, 8, 9)$
s_7	s_9	s_{10}	1.	
s_8	s_{10}	—	0	
s_9	s_1	s_2	0	
s_{10}	s_1	s_2	1	

State Table

State Assignments

		QA Q _B	00 01	11 10
		QC Q _D	00	S ₉
		01		S ₁₀
		11	S ₁	S ₃
		10	S ₀	S ₂
			S ₇	S ₆

Assignment Map

Assignments Q_A Q_B Q_C Q_D

$$S_0 = 0010 \quad S_9 = 0000$$

$$S_1 = 0111 \quad S_{10} = 0100$$

$$S_2 = 0110$$

$$S_3 = 1111$$

$$S_4 = 1011$$

$$S_5 = 1001$$

$$S_6 = 1010$$

$$S_7 = 1110$$

$$S_8 = 1100$$

Transition Diagram

Current State	Next State								Output
	x=0				x=1				
Q _A Q _B Q _C Q _D	Q _A ⁺	Q _B ⁺	Q _C ⁺	Q _D ⁺	Q _A ⁺	Q _B ⁺	Q _C ⁺	Q _D ⁺	
s ₉	0 0 0 0	0 1 1 1	- - - -	- - - -	0 1 1 0	- - - -	- - - -	- - - -	Z
	0 0 0 1	- - - -	- - - -	- - - -	- - - -	- - - -	- - - -	- - - -	
s ₈	0 0 1 0	0 1 1 1	- - - -	- - - -	0 1 1 0	- - - -	- - - -	- - - -	0
	0 0 1 1	- - - -	- - - -	- - - -	- - - -	- - - -	- - - -	- - - -	
s ₁₀	0 1 0 0	0 1 1 1	- - - -	- - - -	0 1 1 0	- - - -	- - - -	- - - -	1
	0 1 0 1	- - - -	- - - -	- - - -	- - - -	- - - -	- - - -	- - - -	
s ₂	0 1 1 0	1 0 1 1	- - - -	- - - -	1 0 0 1	- - - -	- - - -	- - - -	0
s ₁	0 1 1 1	1 1 1 1	- - - -	- - - -	1 0 1 1	- - - -	- - - -	- - - -	1
	1 0 0 0	- - - -	- - - -	- - - -	- - - -	- - - -	- - - -	- - - -	
s ₅	1 0 0 1	1 1 1 0	- - - -	- - - -	1 1 0 0	- - - -	- - - -	- - - -	1
s ₆	1 0 1 0	0 0 0 0	- - - -	- - - -	0 1 0 0	- - - -	- - - -	- - - -	0
s ₄	1 0 1 1	1 1 1 0	- - - -	- - - -	1 1 0 0	- - - -	- - - -	- - - -	0
s ₈	1 1 0 0	0 1 0 0	- - - -	- - - -	- - - -	- - - -	- - - -	- - - -	0
	1 1 0 1	- - - -	- - - -	- - - -	- - - -	- - - -	- - - -	- - - -	
s ₇	1 1 1 0	0 0 0 0	- - - -	- - - -	0 1 0 0	- - - -	- - - -	- - - -	1
	1 1 1 1	1 0 1 0	- - - -	- - - -	1 1 1 0	- - - -	- - - -	- - - -	

Resulting Boolean Equations

If we make Flip-Flop next state
Equations a function of Q_A, Q_B, Q_C, Q_D

$$Q_A^+ = m_6, m_7, m_9, m_{11}, m_{15}$$
$$Q_B^+ = m_0 + m_2 + m_4 + m_9 + m_{11} + m_{12} + m_7\bar{X} + m_{10}X + m_{14}X + m_{15}X$$
$$Q_C^+ = m_0 + m_2 + m_4 + m_7 + m_{15} + m_6\bar{X} + m_9\bar{X} + m_{11}\bar{X}$$
$$Q_D^+ = m_6 + m_7 + m_0\bar{X} + m_2\bar{X} + m_4\bar{X}$$

don't cares

$m_1, m_3, m_5, m_8, m_{13}$

We can use map Entered
variables in next state equations

Flip-flop Q_A

		Q_A^+			
		00	01	11	10
$Q_A Q_B$		00	-	-	-
X \ 1	0	-	-	1	1
	01	-	-	1	1
	11	-	-	1	-
	10	-	1	1	-

$Q_A^+ =$

Flip-flop Q_A

		Q_A^+			
		$Q_C Q_D$	01	11	10
$Q_A Q_B$		00	-	-	-
X	0	00	-	-	-
	01	01	-	1	1
	11	11	-	1	1
	10	10	1	1	1

$Q_A^+ = Q_D$

Flip-flop Q_A

		Q_A^+				
		$Q_C Q_D$	00	01	11	10
$Q_A Q_B$		00	-	-	-	-
X \ 1		01	-	-	1	1
0		11	-	-	1	-
1		10	-	1	1	-

$$Q_A^+ = Q_D + Q_A' Q_B Q_C$$

Flip-flop Q_A

		Q_A^+				
		$Q_C Q_D$	00	01	11	10
$Q_A Q_B$		00	-	-	-	-
X \ 1		01	-	-	1	1
0		11	-	-	1	-
1		10	-	1	1	-

$$Q_A^+ = Q_D + Q_A' Q_B Q_C$$

Flip-flop Q_B

$$Q_B^+ = \begin{array}{c} Q_C Q_D \\ Q_A Q_B \\ \diagdown X \end{array}$$

		Q_B^+				
		00	01	11	10	
		00	$\begin{matrix} 1 \\ 1 \end{matrix}$	$\begin{matrix} - \\ - \end{matrix}$	$\begin{matrix} - \\ - \end{matrix}$	$\begin{matrix} 1 \\ - \end{matrix}$
		01	$\begin{matrix} 1 \\ 1 \end{matrix}$	$\begin{matrix} - \\ - \end{matrix}$	$\begin{matrix} 1 \\ - \end{matrix}$	$\begin{matrix} - \\ - \end{matrix}$
		11	$\begin{matrix} - \\ 1 \end{matrix}$	$\begin{matrix} - \\ - \end{matrix}$	$\begin{matrix} 1 \\ - \end{matrix}$	$\begin{matrix} 1 \\ - \end{matrix}$
		10	$\begin{matrix} - \\ - \end{matrix}$	$\begin{matrix} 1 \\ 1 \end{matrix}$	$\begin{matrix} 1 \\ 1 \end{matrix}$	$\begin{matrix} 1 \\ - \end{matrix}$

Flip-flop Q_B

		Q_B^+				
		$Q_C Q_D$	00	01	11	10
$Q_A Q_B$		00	1 1	-	-	1
X	1	01	1 1	-	1	-
0	11	-	-	1	1	1
10	10	-	1 1	1	1	1

$Q_B^+ = Q_C'$

Flip-flop Q_B

		Q_B^+				
		$Q_C Q_D$	00	01	11	10
$Q_A Q_B$		00	1	-	-	1
X	1	01	1	-	1	-
0	1	11	-	-	1	1
1	0	10	-	1	1	1

$Q_B^+ = Q_C' + Q_A' Q_B'$

Flip-flop Q_B

		Q_B^+				
		$Q_C Q_D$	00	01	11	10
$Q_A Q_B$		00	1	-	-	1
X	1	01	1	-	-	1
0	11	1	-	-	1	1
10	10	-	1	1	1	1

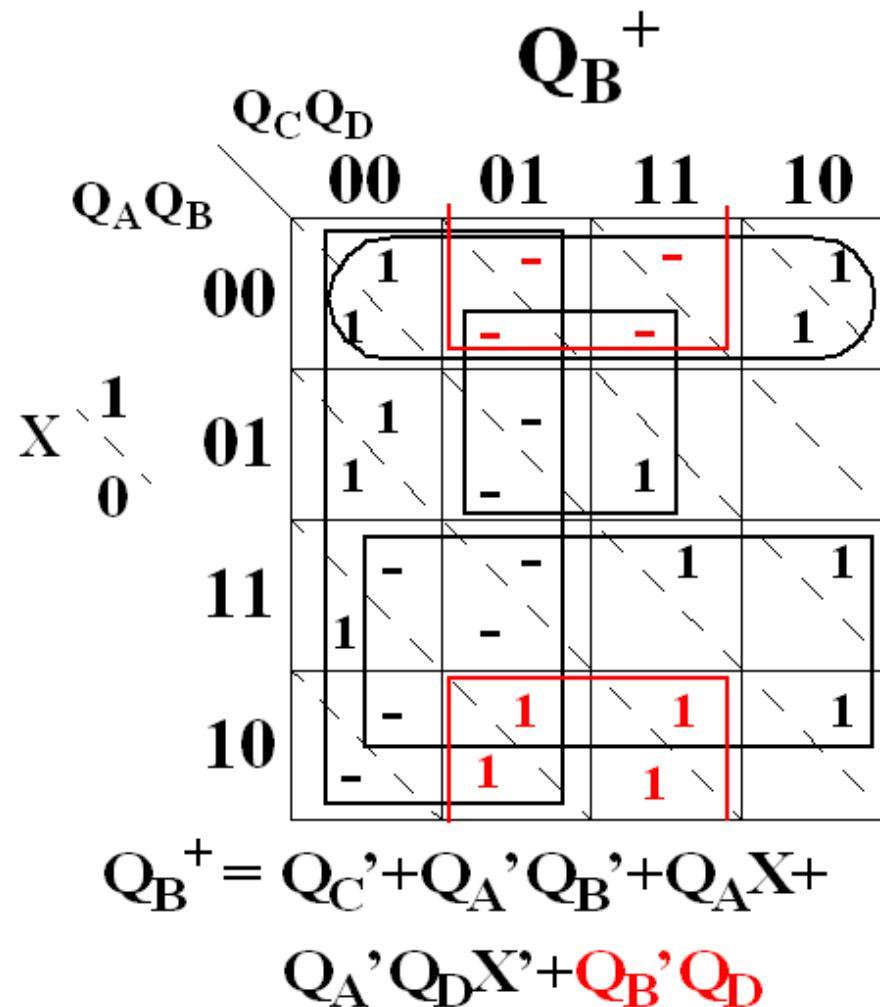
$$Q_B^+ = Q_C' + Q_A' Q_B' + Q_A X$$

Flip-flop Q_B

		Q_B^+				
		$Q_C Q_D$	00	01	11	10
$Q_A Q_B$		00	1	-	-	1
X		01	1	-	1	
0		11	-	-	1	1
10		10	-	1	1	1

$$Q_B^+ = Q_C' + Q_A' Q_B' + Q_A X + Q_A' Q_D X'$$

Flip-flop Q_B



Flip-flop Q_B

		Q_B^+			
		$Q_C Q_D$	$Q_A Q_B$	$Q_C Q_D$	Q_B^+
		00	01	11	10
X		00	1	-	-
1		01	-	-	1
0		11	1	-	-
-		10	-	1	1

$$\begin{aligned} Q_B^+ = & Q_C' + Q_A' Q_B' + Q_A X + \\ & Q_A' Q_D X' + Q_B' Q_D \end{aligned}$$

Flip-flop Q_C

		Q_C^+			
		00	01	11	10
$Q_A Q_B$		00	-	-	-
X \ 1	0	1 1	-	-	1 1
	01	1 1	-	1 1	1 1
	11	-	-	1 1	-
	10	-	1 1	1 1	-

$Q_C^+ =$

Flip-flop Q_C

		Q_C^+				
		$Q_C Q_D$	00	01	11	10
$Q_A Q_B$		00	1 1	-	-	1 1
X	1 0	01	1 1	-	1 1	1
		11	-	-	1 1	-
		10	-	1 1	1 1	-

$Q_C^+ = Q_D X'$

Flip-flop Q_C

		Q_C^+				
		00	01	11	10	
		00	1	-	-	1
X		01	1	1	1	1
Q _A Q _B		11	-	-	1	-
Q _C Q _D		10	-	1	1	-

$$Q_C^+ = Q_D X + Q_A' Q_B'$$

Flip-flop Q_C

		Q_C^+			
		00	01	11	10
		$Q_C Q_D$	$Q_A Q_B$	Q_C^+	Q_C^+
X	0	1	-	-	1
0	1	1	-	1	1
1	1	-	-	1	1
1	0	-	-	1	-
0	1	-	-	1	-
1	0	-	-	1	-
0	0	1	1	1	1

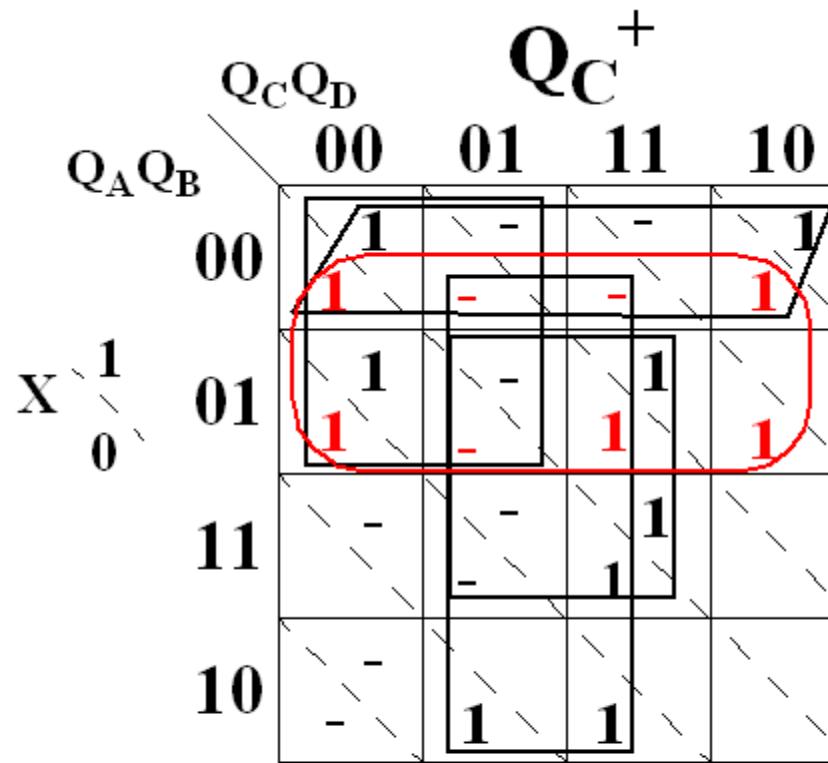
$$Q_C^+ = Q_D X' + Q_A' Q_B' + \textcolor{red}{Q_A' Q_C'}$$

Flip-flop Q_C

		Q_C^+				
		00	01	11	10	
		00	1	-	-	1
$X \backslash Q_A Q_B$		01	1	-	1	1
0		11	-	-	1	1
1		10	-	1	1	-

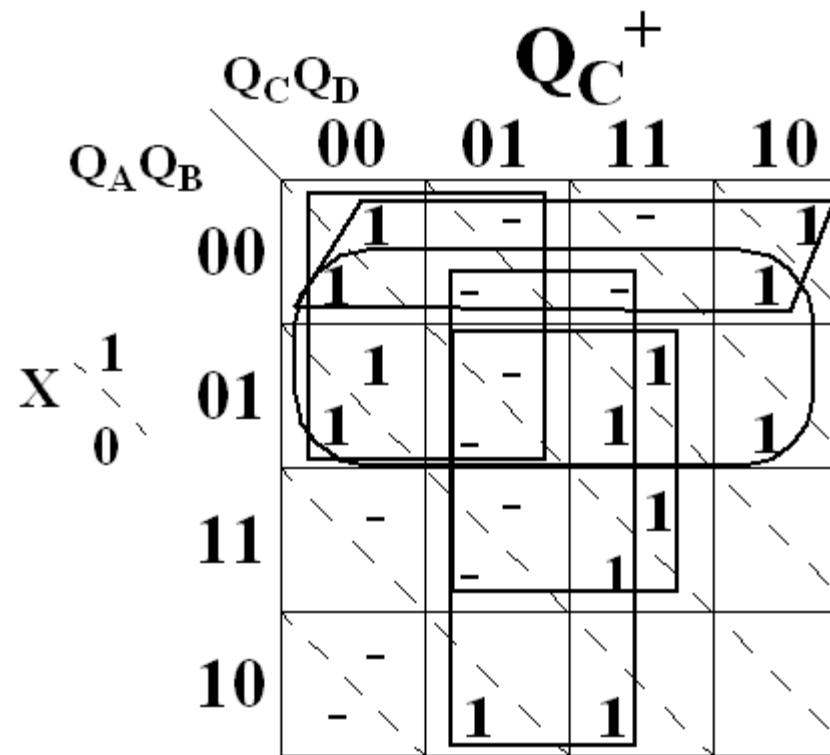
$$\begin{aligned} Q_C^+ = & Q_D X' + Q_A' Q_B' + Q_A' Q_C' + \\ & \textcolor{red}{Q_B Q_D} \end{aligned}$$

Flip-flop Q_C



$$Q_C^+ = Q_D X' + Q_A' Q_B' + Q_A' Q_C' + \\ Q_B Q_D + \textcolor{red}{Q_A' X'}$$

Flip-flop Q_C



$$\begin{aligned} Q_C^+ = & Q_D X' + Q_A' Q_B' + Q_A' Q_C' + \\ & Q_B Q_D + Q_A' X' \end{aligned}$$

Flip-flop Q_D

		Q_D^+				
		$Q_C Q_D$	00	01	11	10
$Q_A Q_B$		00	1	-	-	1
X		01	1	-	1	1
0		11	-	-	-	-
1		10	-	-	-	-
		$Q_D^+ =$				

Flip-flop Q_D

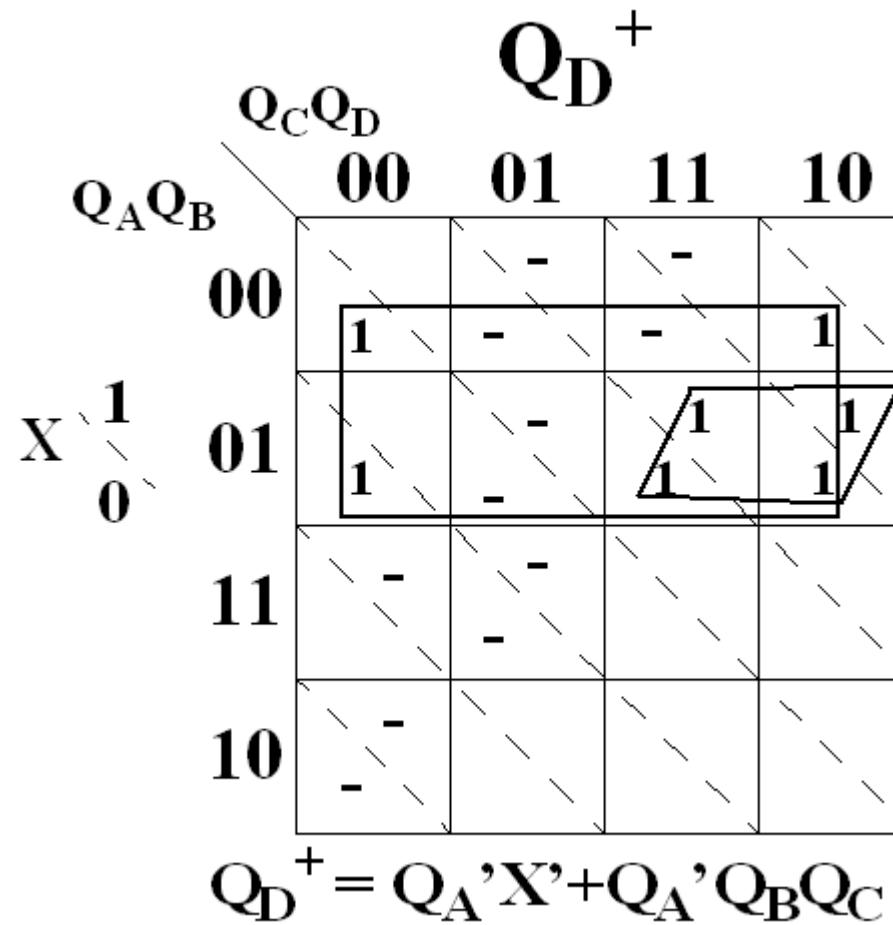
		Q_D^+				
		$Q_C Q_D$	00	01	11	10
$Q_A Q_B$		00	1	-	-	1
X		01	1	-	1	1
0		11	-	-	-	-
1		10	-	-	-	-
		$Q_D^+ = Q_A' X'$				

Flip-flop Q_D

		Q_D^+				
		$Q_C Q_D$	00	01	11	10
$Q_A Q_B$		00	1	-	-	1
X \ 1		01	1	-	1	1
0		11	-	-	-	-
10		-	-	-	-	-

$Q_D^+ = Q_A' X' + Q_A' Q_B Q_C$

Flip-flop Q_D



Output Z

		Z				
		Q _C Q _D	00	01	11	10
Q _A Q _B		00	-	-		
01		01	1	-	1	
11		11	-	1	1	
10		10	-	1		

Z=

Output Z

		Z				
		$Q_C Q_D$	00	01	11	10
$Q_A Q_B$		00	-	-	-	-
01		1	-	1		
11			-	1	1	
10		-	1			

$$Z = Q_C' Q_D$$

Output Z

		Z				
		$Q_C Q_D$	00	01	11	10
$Q_A Q_B$		00	-	-	-	-
01		1	-	1		
11			-	1	1	
10		-	1			

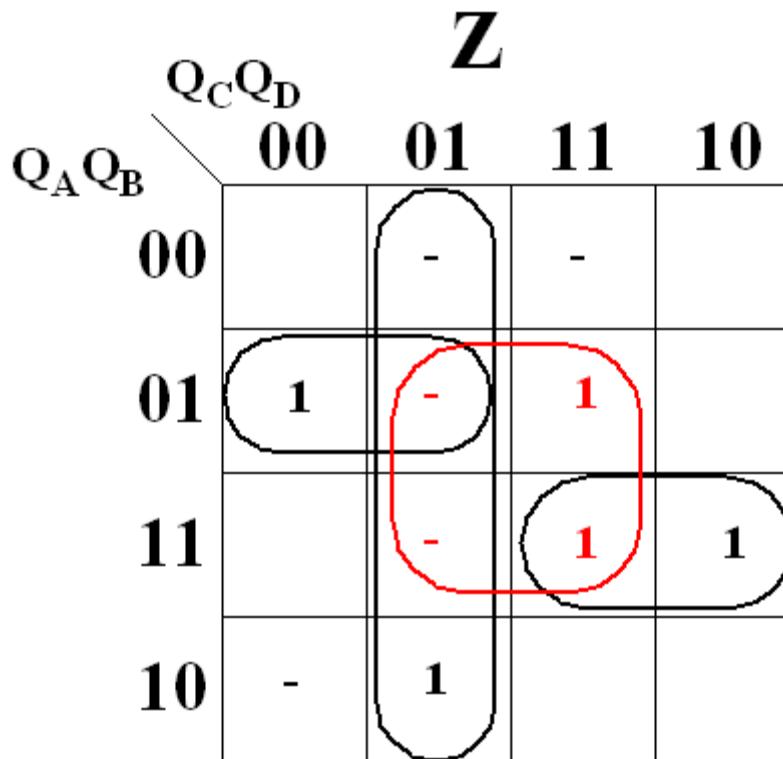
$$Z = Q_C' Q_D + \textcolor{red}{Q_A Q_B Q_C}$$

Output Z

		Z				
		$Q_C Q_D$	00	01	11	10
$Q_A Q_B$		00	-	-	-	-
01		1	-	-	1	-
11		-	-	1	1	-
10		-	1	-	-	-

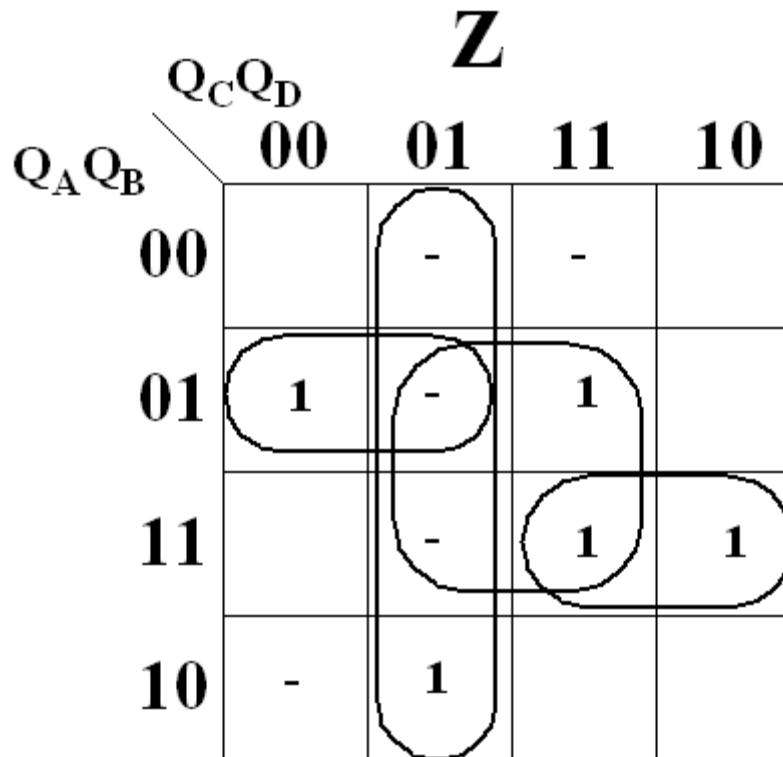
$$Z = Q_C' Q_D + Q_A Q_B Q_C + \\ Q_A' Q_B Q_C'$$

Output Z



$$\begin{aligned} Z = & Q_C'Q_D + Q_AQ_BQ_C + \\ & Q_A'Q_BQ_C' + \color{red}{Q_BQ_D} \end{aligned}$$

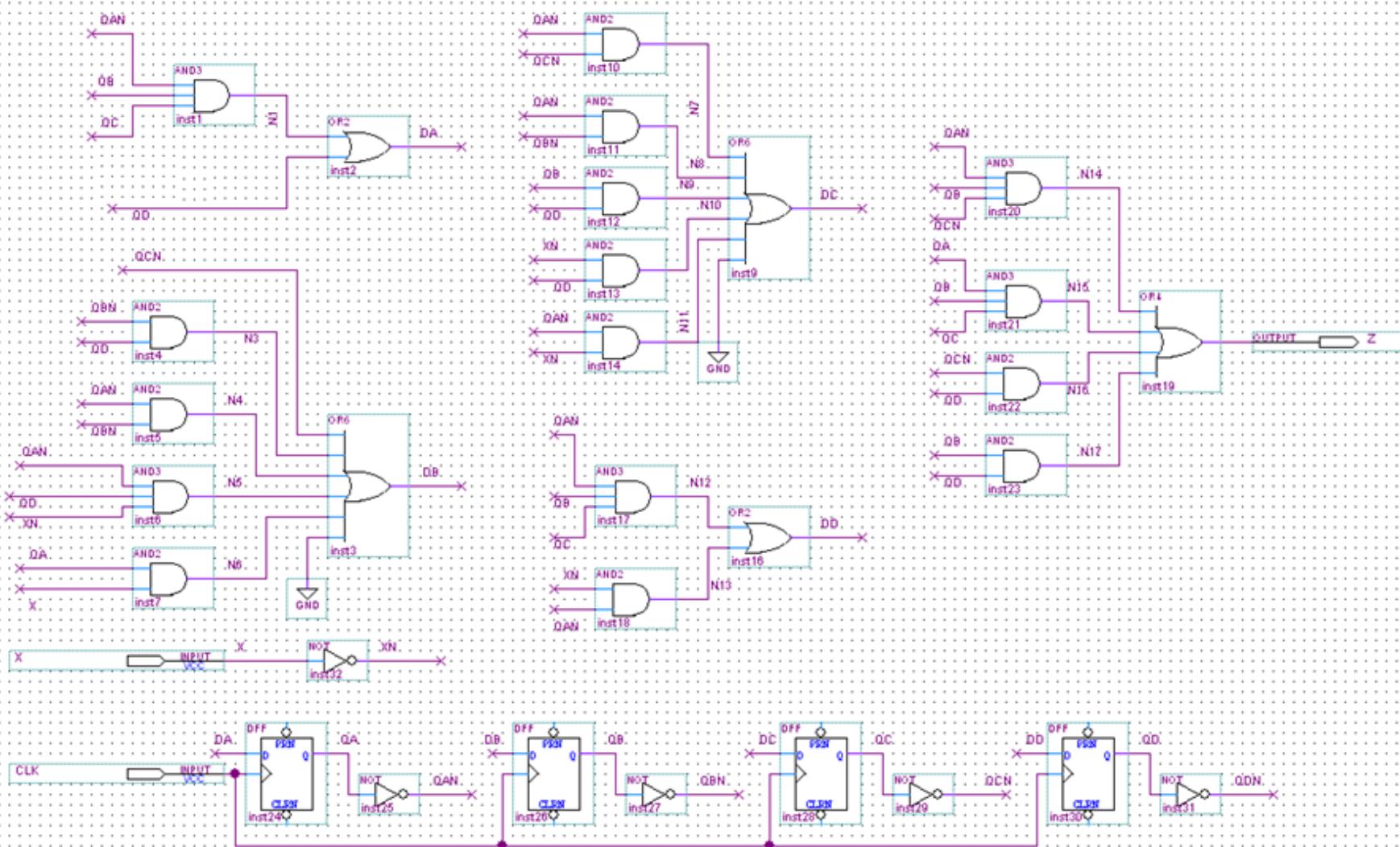
Output Z



$$\begin{aligned}Z &= Q_C' Q_D + Q_A Q_B Q_C + \\&Q_A' Q_B Q_C' + Q_B Q_D\end{aligned}$$

Quartus II Schematic Realization

8421 BCD to Excess3 Code Converter (Moore FSM)



Data Flow Representation Verilog HDL

8421 BCD to Excess3 Code Converter (Moore FSM)

```
// bcd to excess 3 converter
// Moore Implementation
// Data Flow Model
module bcd_ex3_moore(input X,CLK,output Z);

reg QA=0,QB=0,QC=0,QD=0;

// FF State Update Portion of design
// Active only on rising edge of clock
always @(posedge CLK)
begin
    QA <= QD | (~QA & QB & QC);
    QB <= ~QC | (~QA & ~QB) | (QA & X) | (~QA & QD & ~X) | (~QB & QD);
    QC <= (QD & ~X) | (~QA & ~QB) | (~QA & ~QC) | (QB & QD) | (~QA & ~X);
    QD <= (~QA & ~X) | (~QA & QB & QC);
end

// Output Equation -- Continuous Assignment that is
// a function only of the state variables QA,QB,QC,QD
assign Z = (~QC & QD) | (QA & QB & QC) | (~QA & QB & ~QC) | (QB & QD);

endmodule
```

ModelSim™ Waveform

Data Flow 8421 BCD to Excess3 Code Converter (Moore FSM)

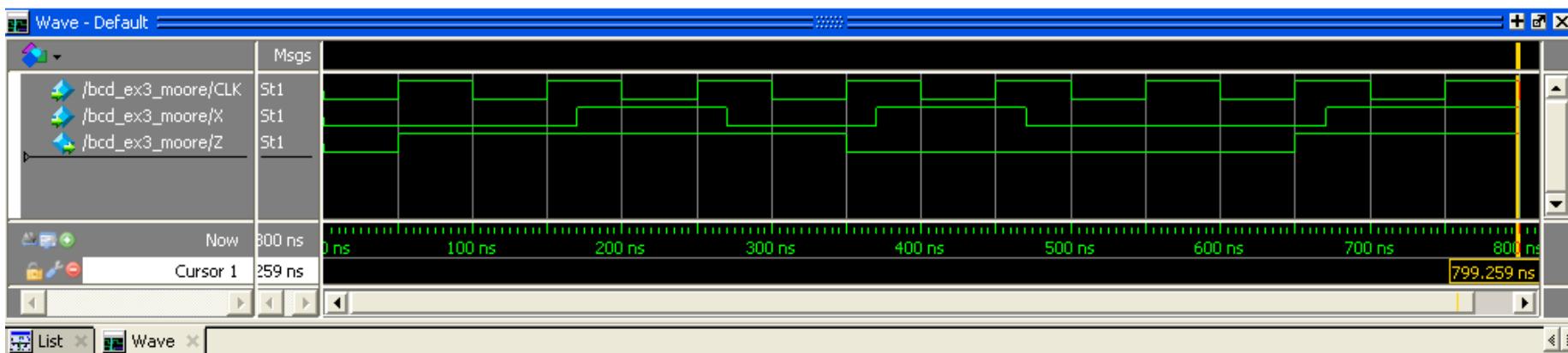
- $X = 0010_1001 \Rightarrow Z = 1110_0011$

[.add wave CLK X Z](#)

[.force CLK 0 0 ns, 1 50 ns -r 100 ns](#)

[.force X 0 0 ns, 1 170 ns, 0 270 ns, 1 370 ns, 0 470 ns, 1 670 ns](#)

[.run 800 ns](#)

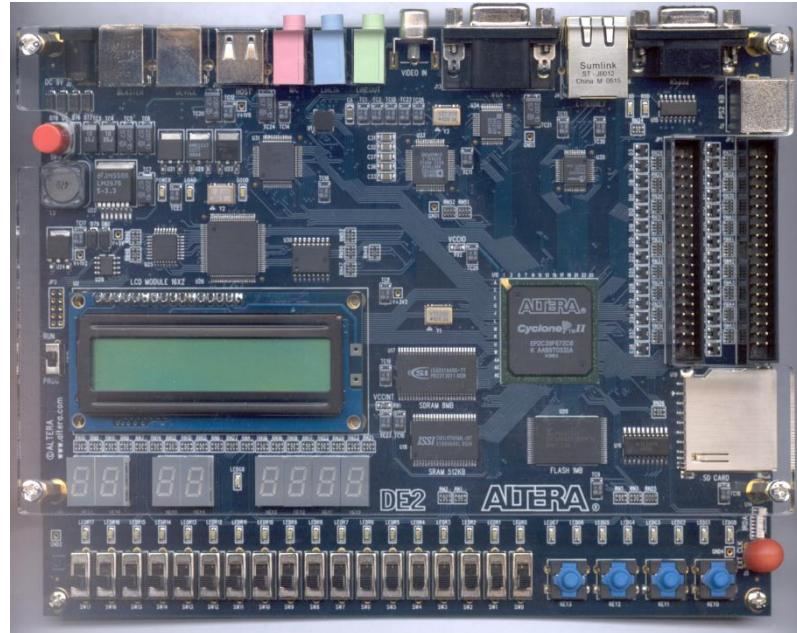


CPE 322

Digital Hardware Design Fundamentals

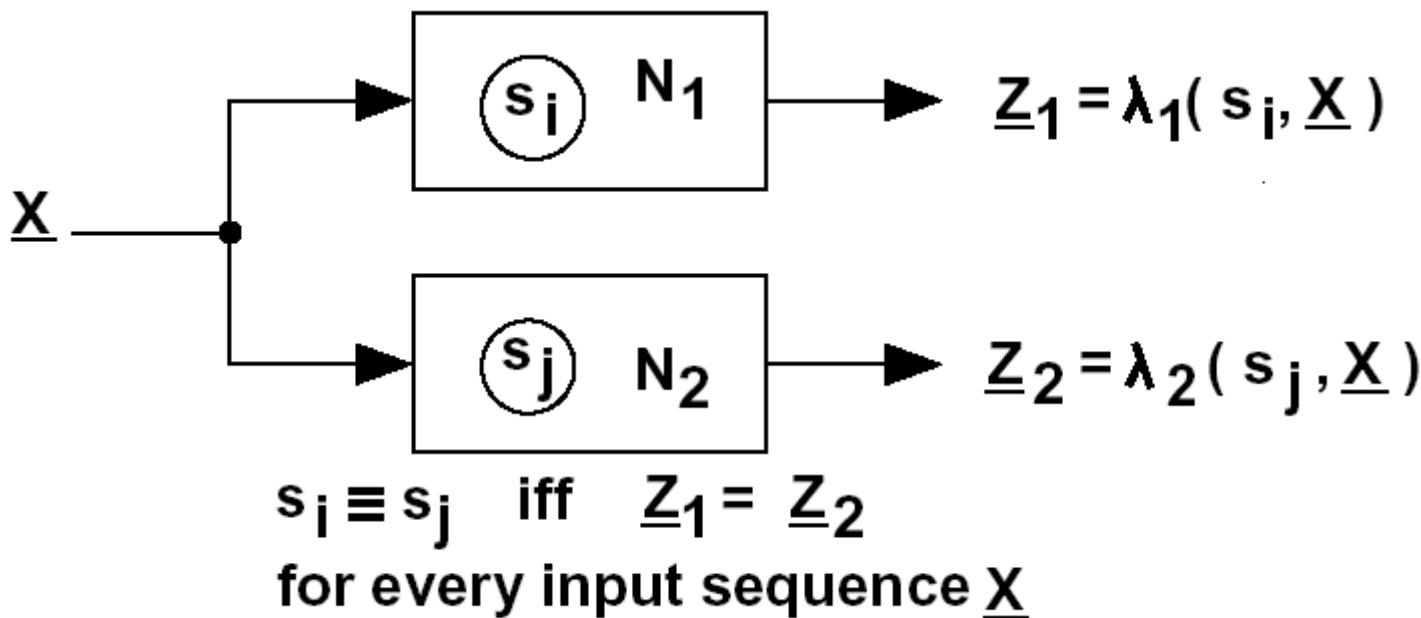
Electrical and Computer Engineering

State Reduction in Mealy and Moore Finite State Machines



Equivalent States

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



Definition: Two states are equivalent $s_i == s_j$ only and only if, for every input sequence \underline{X} , the output sequences \underline{Z}_1 and \underline{Z}_2 are the same.

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

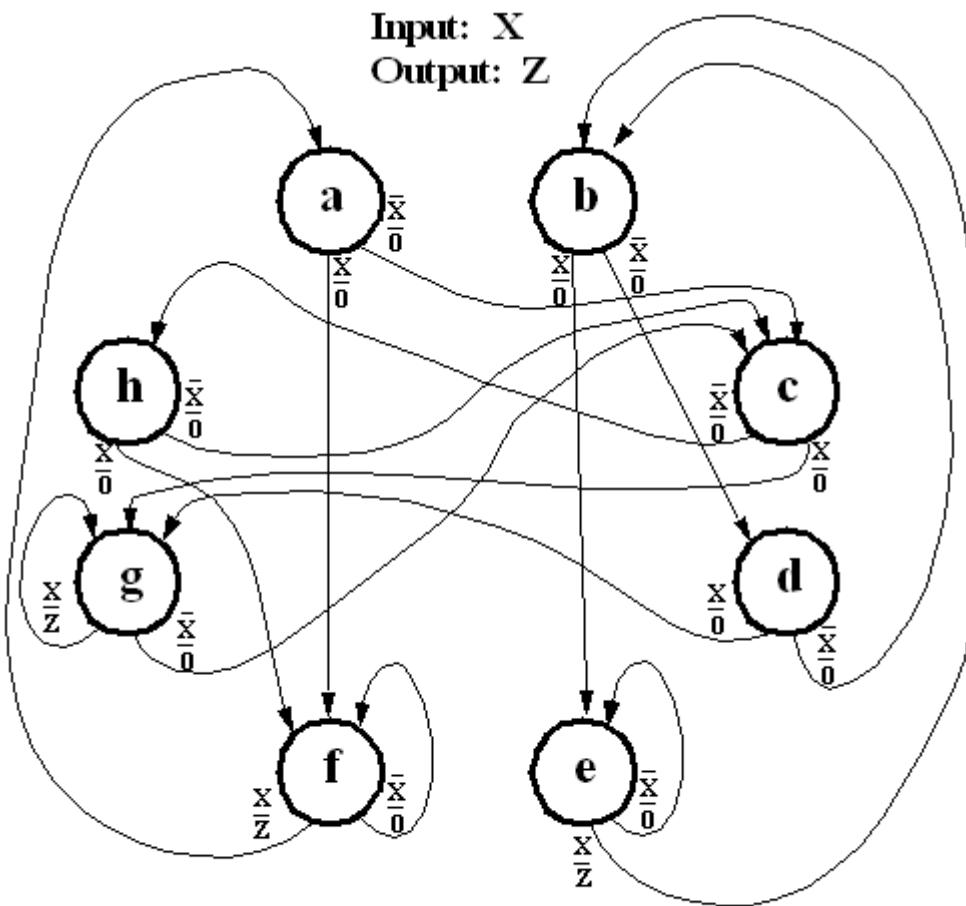
State Equivalence Theorem

- Two states, S_i and S_j are *equivalent* (i.e. $S_i == S_j$) if and only if for every single input X , the outputs are the same and the next states are found to be equivalent.
- Only one representation of the state is needed for each set of equivalent states found.
 - The others can be removed.

Equivalent State Determination Methodology

- If two states are the same state then they are equivalent.
- Other equivalent states are found by systematically examining each two state combination present in the FSM to determine if they are equivalent based upon the application of the State Equivalence Theorem.
- This is a multiphase operation with state pairs that are found to be not equivalent being removed from further consideration in the next phase.
- States whose equivalence cannot be determined are passed on to the next phase.
- When there is no further non-equivalent states found in a phase then the process has found all distinct states.
 - Any remaining two-state combinations must be equivalent.

State Table Reduction Example

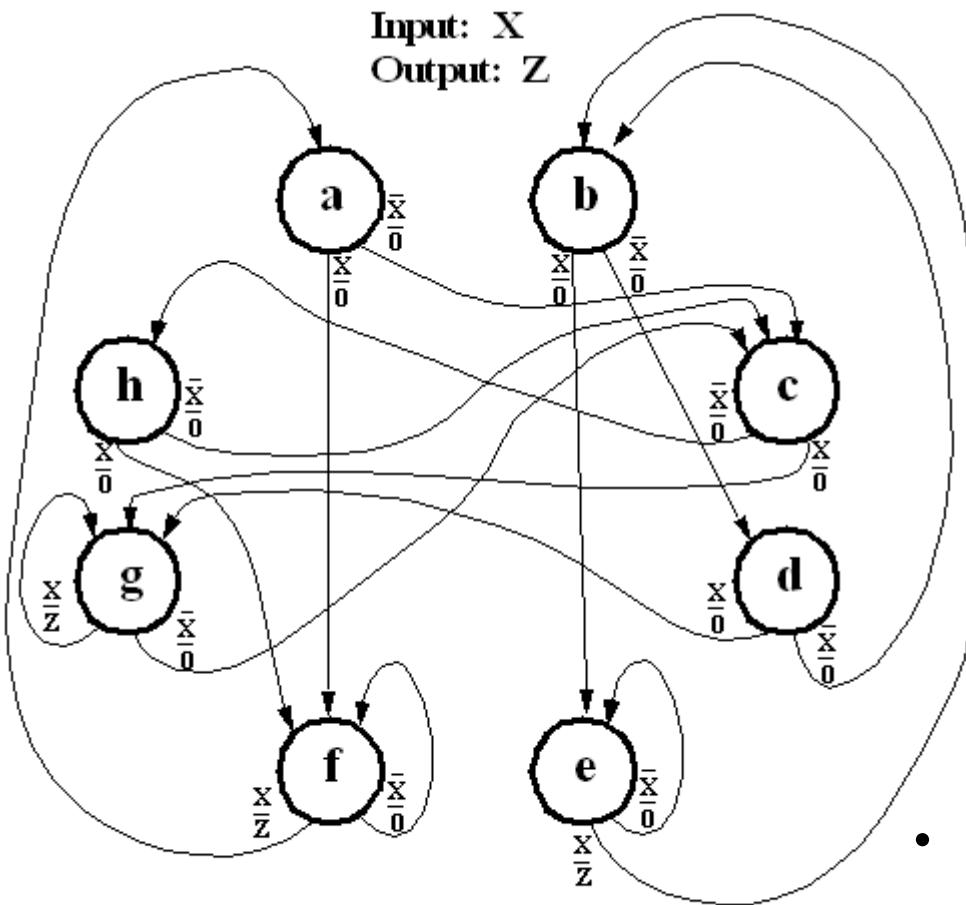


Extended State Transition Graph

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

State Table

State Table Reduction Example



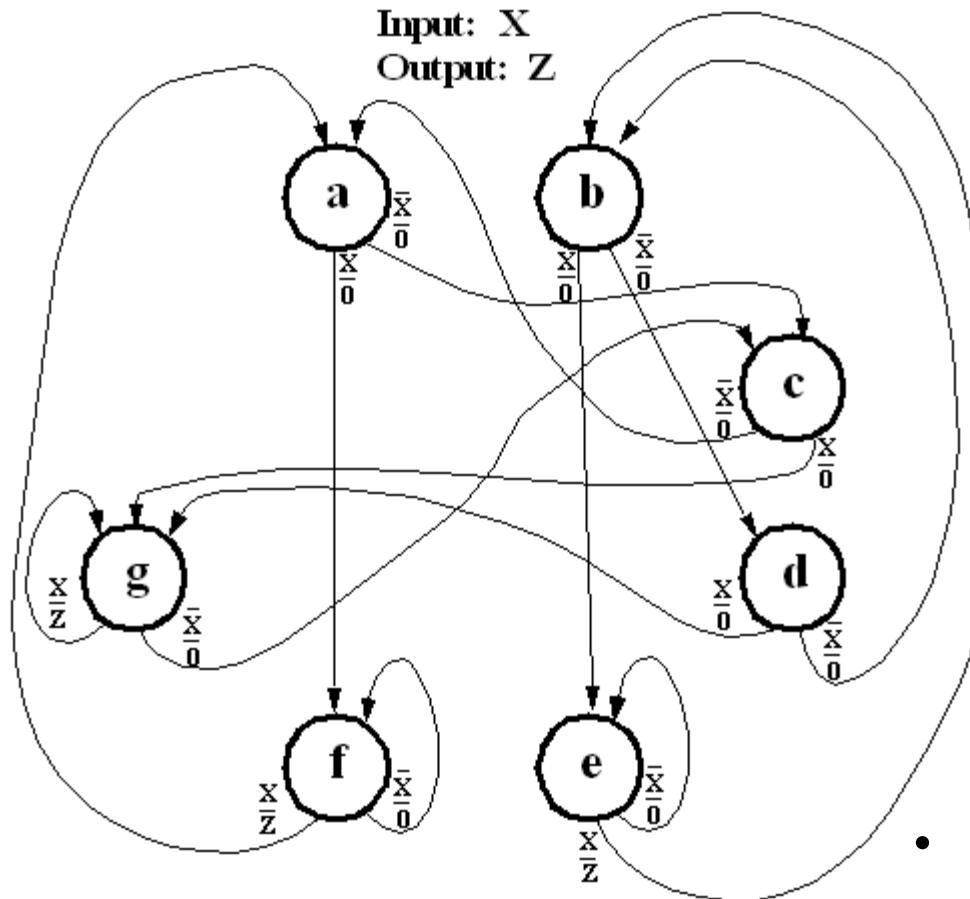
Extended State Transition Graph

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

State Table

- By direct application of the Equivalent State Theorem for states **a** and **h**.

State Table Reduction Example



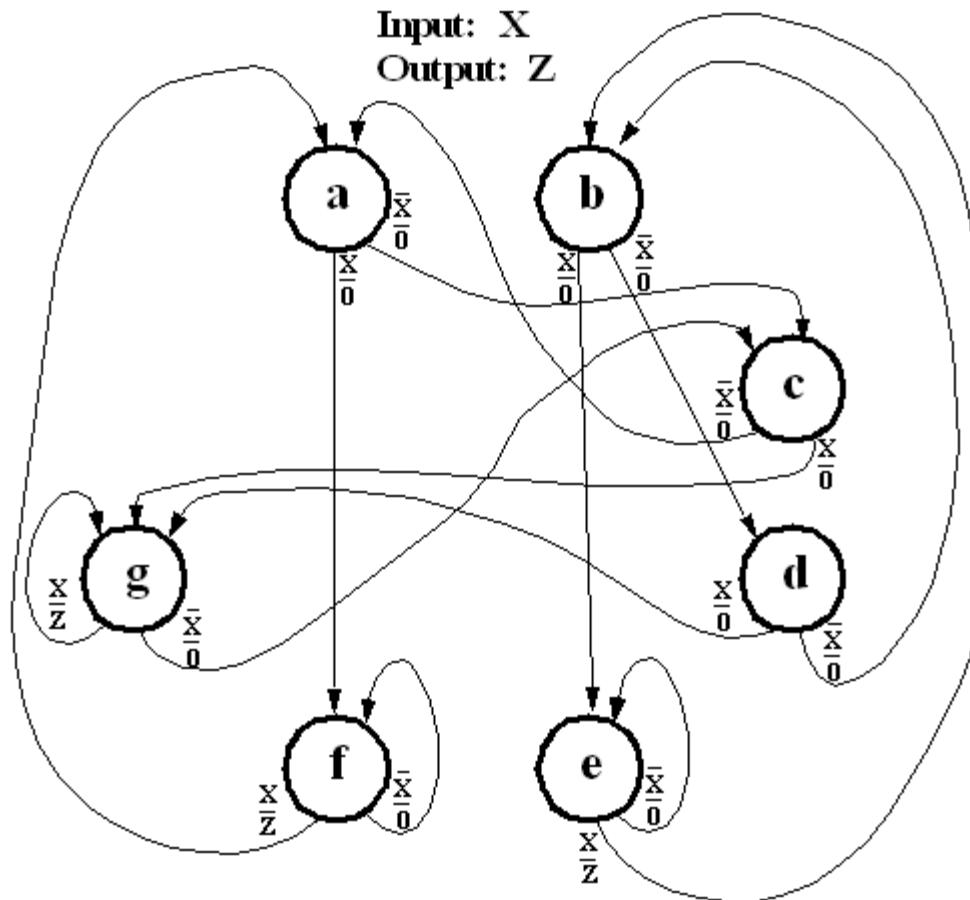
Extended State Transition Graph

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

State Table

- By direct application of the Equivalent State Theorem for states **a** and **h**.

State Table Reduction Example



Extended State Transition Graph

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

State Table

State Table Reduction Example

- This reduction was done by inspection but further state reduction requires an iterative evaluation of the states.

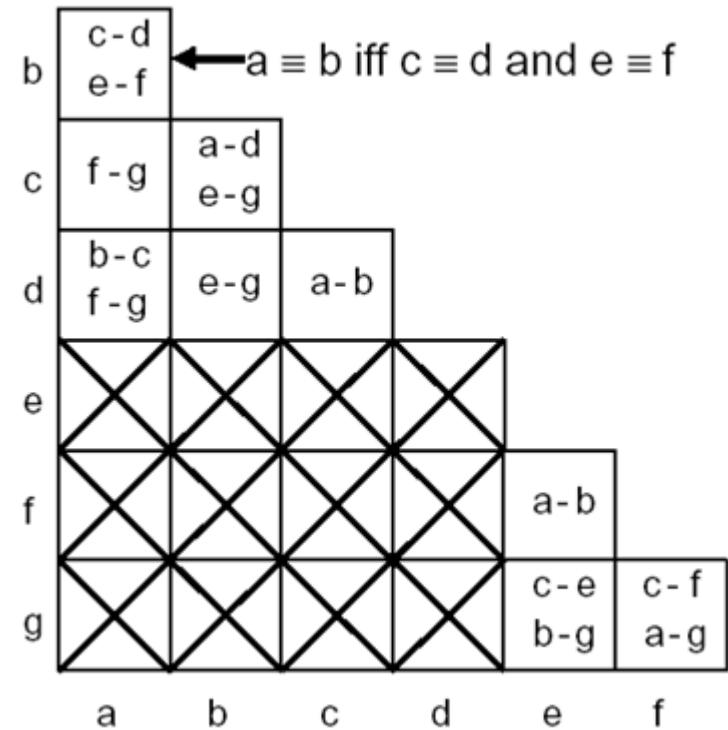
Implication Table Construction

- Evaluating the two state equivalence comparison process is aided by the use of an implication table
 - This is in effect a lower triangular portion of a square matrix where both dimensions represent the number of states in the FSM.
 - On an N state FSM representation:
 - The y axis proceeds from State 2 to State N.
 - The x axis goes from State 1 to State N-1.
 - Each Square of the Implication table should be labeled with the conditions necessary for state equivalence for the two states associated with the (row,column) pair.
 - This is obtained from the State Table or STG.
 - State pairs that cannot be equivalent are marked by placing an X in the (row/column) square on the implication table.

State Table Reduction

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

State Table



- State combinations whose outputs differ are not equivalent so the corresponding square is marked with an X
- Other Squares contain the next state requirements for equivalency.
 - For example States a and b have the same output => they are same iff $c=d$ and $e=f$. We say $c-d$ and $e-f$ are *implied pairs* for a-b. They may or may not be equivalent – can not tell in this phase

State Table Reduction

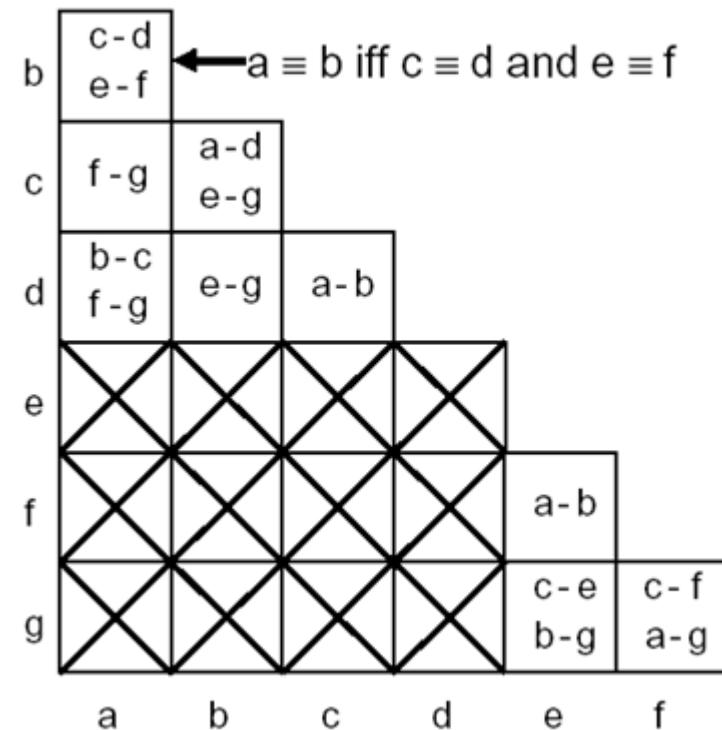
- Consider square (b,a) to be equivalent $c == d \& e == f$. Can't determine remain implied pairs.
- Consider square (c,a) to be equivalent $f == g$. Can't determine remain implied pairs.
- Consider square (d,a) to be equivalent $b == c \& f == g$. Can't determine remain implied pairs.
- Consider square (c,b) to be equivalent $a == d \& e == g$. Can't determine remain implied pairs.
- Consider square (d,b) to be equivalent $e == g$. Can't determine remain implied pairs.
- Consider square (d,c) to be equivalent $a == b$ Can't determine remain implied pairs.

b	c - d e - f						
c	f - g	a - d e - g					
d	b - c f - g	e - g	a - b				
e							
f						a - b	
g						c - e b - g	c - f a - g
	a	b	c	d	e	f	

- Consider square (f,e) to be equivalent $a == b$ Can't determine remain implied pairs.

State Table Reduction

- Consider square (g,e) to be equivalent
 $c == e \&\& b == g$. This is not true.
 $c \neq e$ since it has an X in the
(e,c) square [same is true for
(b,g) square but only one is
needed to declare f not equivalent
to g].



a ≡ b iff c ≡ d and e ≡ f

b	c-d e-f					
c	f-g	a-d e-g				
d	b-c f-g	e-g	a-b			
e						
f				a-b		
g				c-e b-g	c-f	a-g
a						f

State Table Reduction

- Consider square (g,e) to be equivalent $c == e \&& b == g$. This is not true.
 $c \neq e$ since it has an X in the (e,c) square [same is true for (b,g) square but only one is needed to declare f not equivalent to g].
- X is placed in (g,e) location.

$a \equiv b \text{ iff } c \equiv d \text{ and } e \equiv f$

b	c-d e-f					
c	f-g	a-d e-g				
d	b-c f-g	e-g	a-b			
e						
f					a-b	
g					c-e b-g	c-f a-g
	a	b	c	d	e	f

State Table Reduction

- Consider square (g,f) to be equivalent
 $e == f \&& a == g$. This is not true.
 $a != g$ since it has an X in the
(g,a) square.

$\leftarrow a \equiv b \text{ iff } c \equiv d \text{ and } e \equiv f$

b	c-d e-f					
c	f-g	a-d e-g				
d	b-c f-g	e-g	a-b			
e						
f					a-b	
g					c-e b-g	c-f a-g
	a	b	c	d	e	f

State Table Reduction

- Consider square (g,f) to be equivalent
 $e == f \&& a == g$. This is not true.
 $a != g$ since it has an X in the
(g,a) square.
- Place an X in the (g,f) location.

$\leftarrow a \equiv b \text{ iff } c \equiv d \text{ and } e \equiv f$

c-d	e-f					
b						
c	f-g	a-d	e-g			
d	b-c			a-b		
e	f-g	e-g				
f					a-b	
g					c-e	c-f
					b-g	a-g
						f
	a	b	c	d	e	f

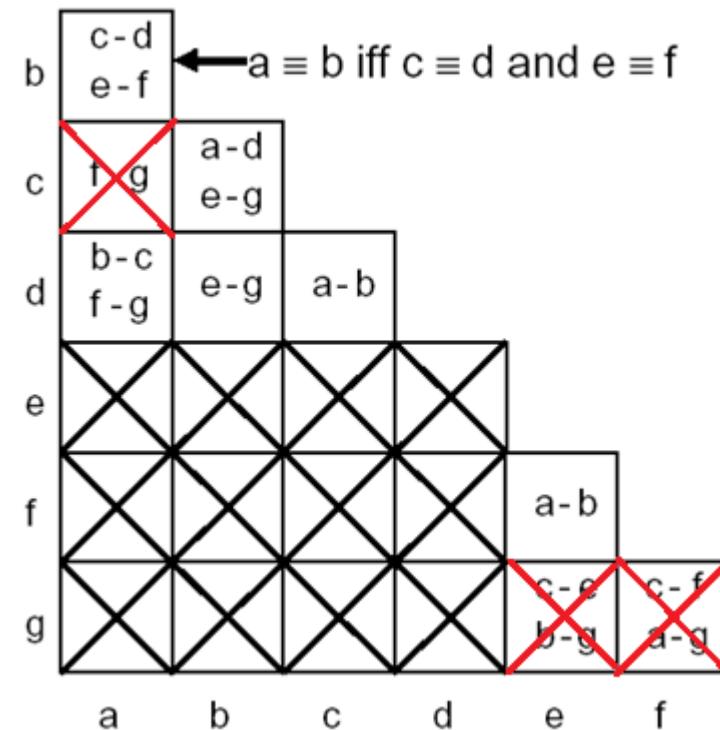
State Table Reduction

- Consider square (b,a) to be equivalent
 $c == d \ \&\& e == f$. Can't determine
remain implied pairs.
- Consider square (c,a) to be equivalent
 $f == g$. This is not true.
 $f \neq g$ since it now has an X in the
(f,g) square.

	a	b	c	d	e	f
a	X					
b		X				
c			X			
d				X		
e					X	
f						X

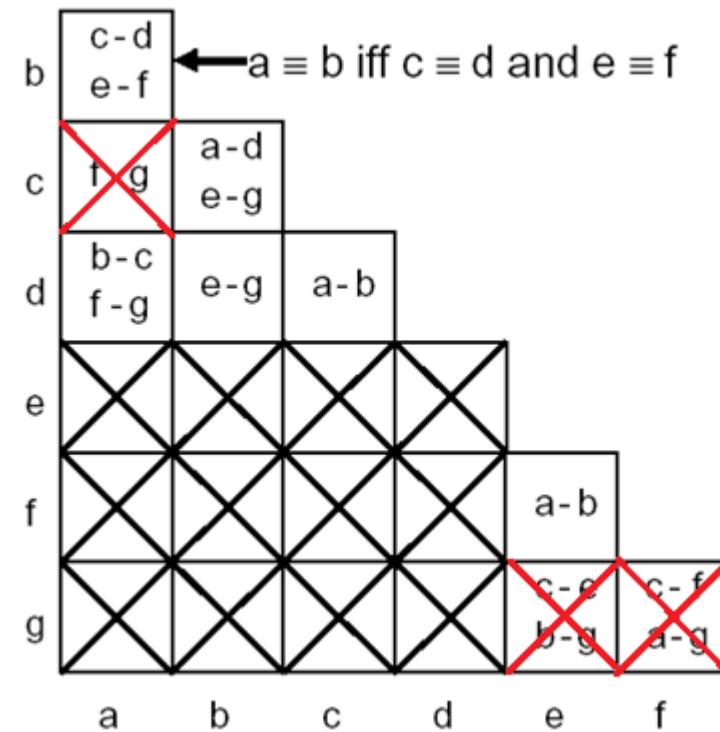
State Table Reduction

- Consider square (b,a) to be equivalent $c == d \ \&\& e == f$. Can't determine remain implied pairs.
- Consider square (c,a) to be equivalent $f == g$. This is not true.
 $f \neq g$ since it now has an X in the (f,g) square.
- Place an X in the (c,a) square.



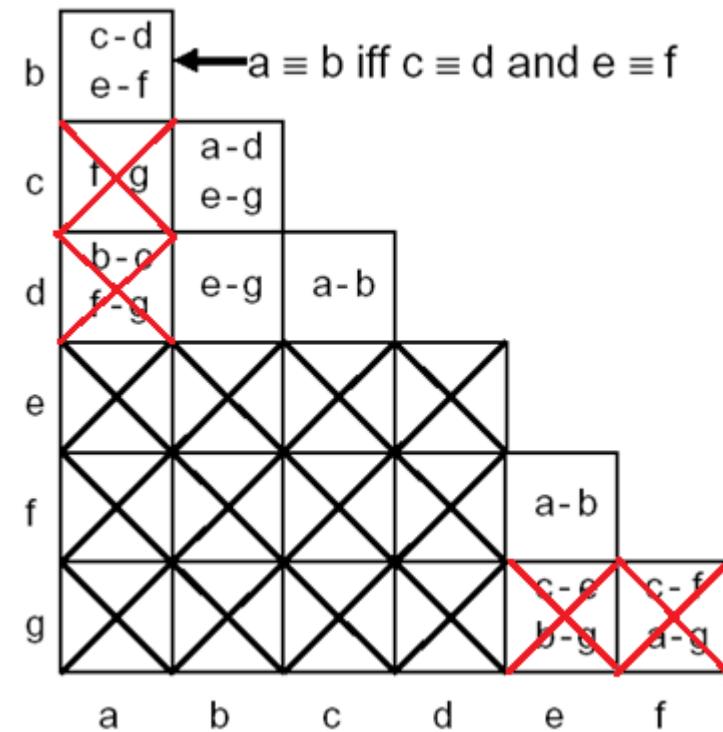
State Table Reduction

- Consider square (d,a) to be equivalent
 $b == c \& f == g$. This is not true.
 $f \neq g$ since it now has an X in the
(f,g) square.



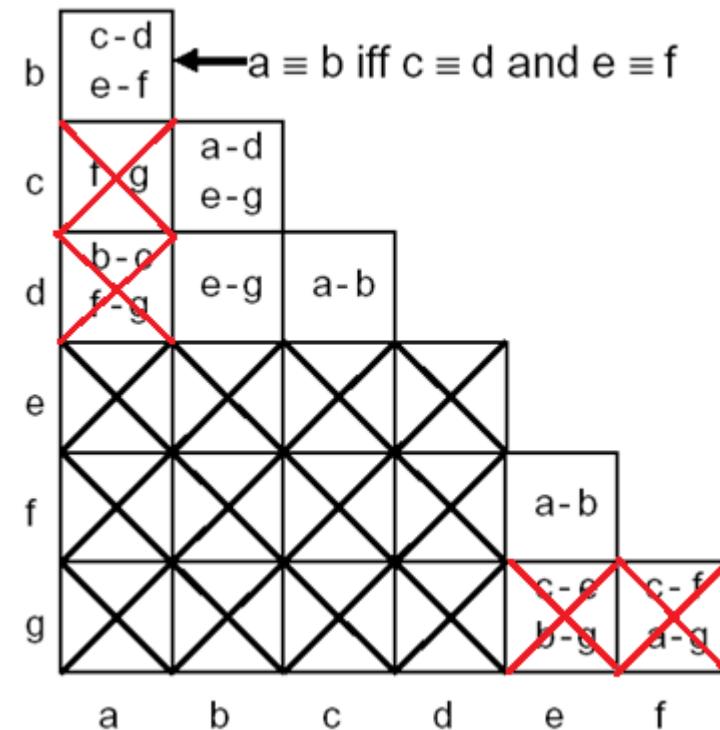
State Table Reduction

- Consider square (d,a) to be equivalent
 $b == c \& f == g$. This is not true.
 $f \neq g$ since it now has an X in the
(f,g) square.
- Place an X in the (d,a) square.



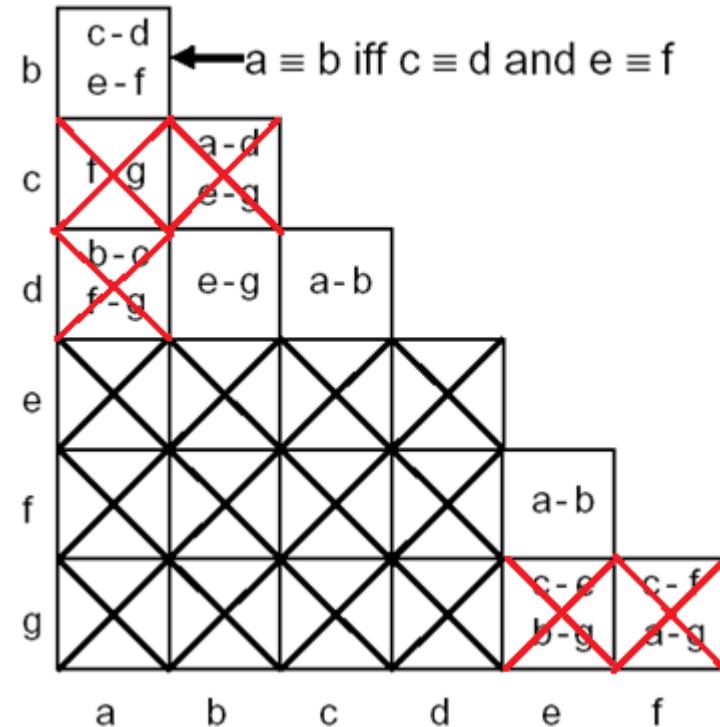
State Table Reduction

- Consider square (c,b) to be equivalent
 $a==d \&& e == g$. This is not true.
 $a != d$ and $e != g$ since they now
both have an X in the (d,a) and
(g,e) squares.



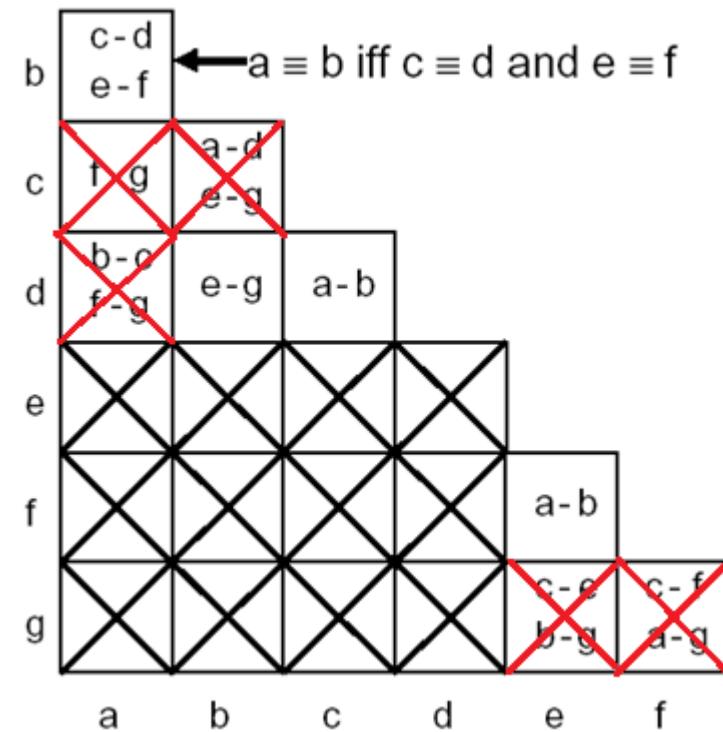
State Table Reduction

- Consider square (c,b) to be equivalent $a==d \&& e == g$. This is not true.
 $a \neq d$ and $e \neq g$ since they now both have an X in the (d,a) and (g,e) squares.
- Place an X in the (c,b) square.



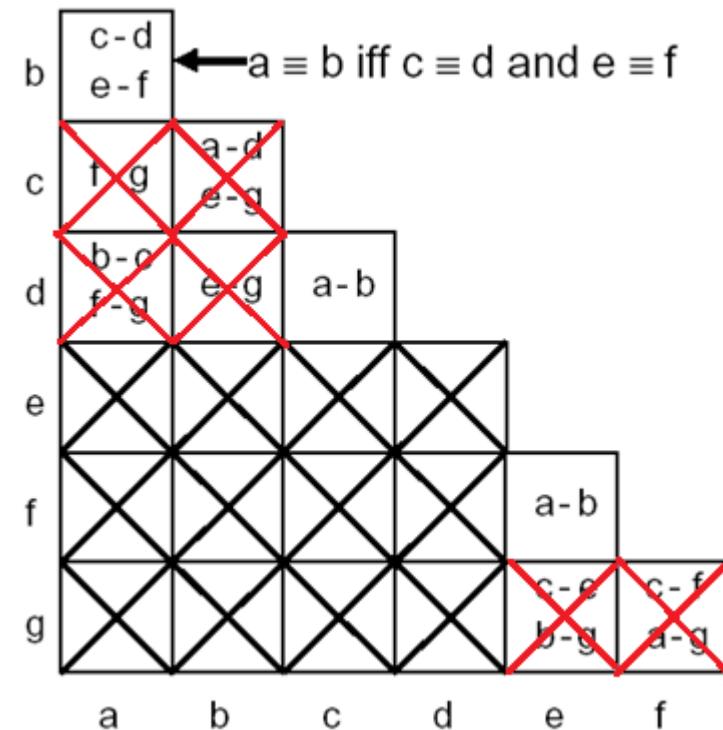
State Table Reduction

- Consider square (d,b) to be equivalent
 $e == g$. This is not true.
 $e \neq g$ since it now has
an X in the (g,e) square.



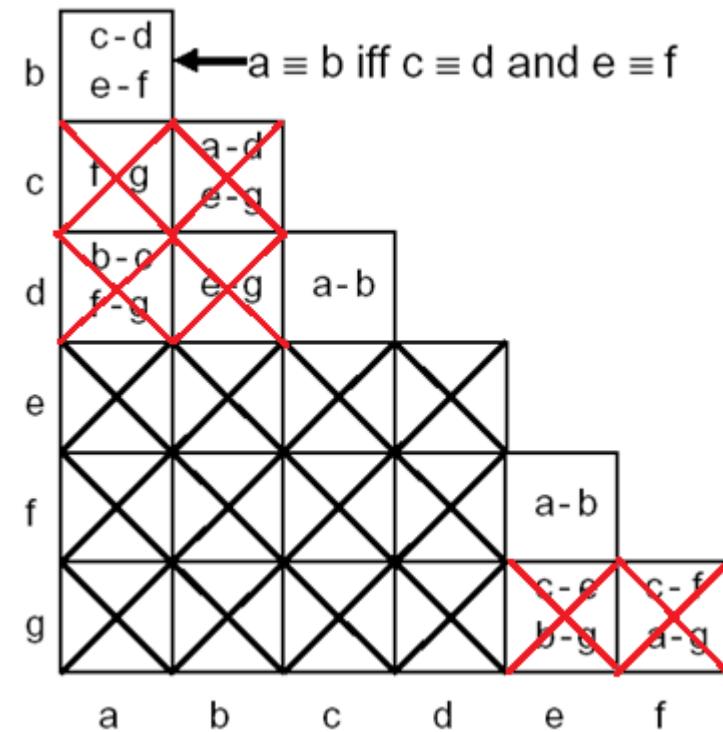
State Table Reduction

- Consider square (d,b) to be equivalent
 $e == g$. This is not true.
 $e \neq g$ since it now has
an X in the (g,e) square.
- Place an X in the (d,b) square.



State Table Reduction

- Consider square (d,c) to be equivalent
 $a == b$ Can't determine
remain implied pairs.
- Consider square (f,e) to be equivalent
 $a == b$ Can't determine
remain implied pairs.
- Since at least one non-equivalence
was found need to re-evaluate all other
non-resolved implied pairs.



- Considering all the implied pair squares (b,a), (d,c) and (f,e) results in no new non-equivalences. Thus all non-equivalences have been found and the process stops.
- Remaining implied pairs are equivalent states.
 - (i.e. $a==b$, $c==d$, and $e==f$)

State Table Reduction

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

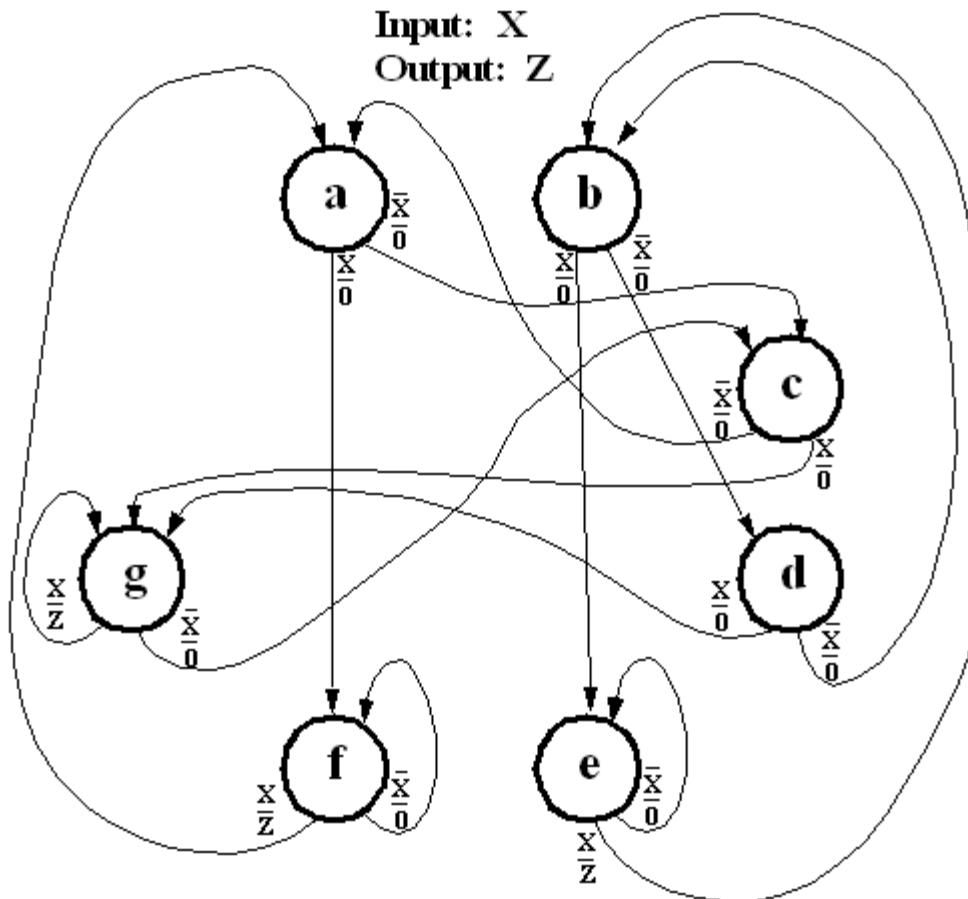
State Table

$$a \equiv b, c \equiv d, e \equiv f$$

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

Final Reduced Table

State Reduction Example

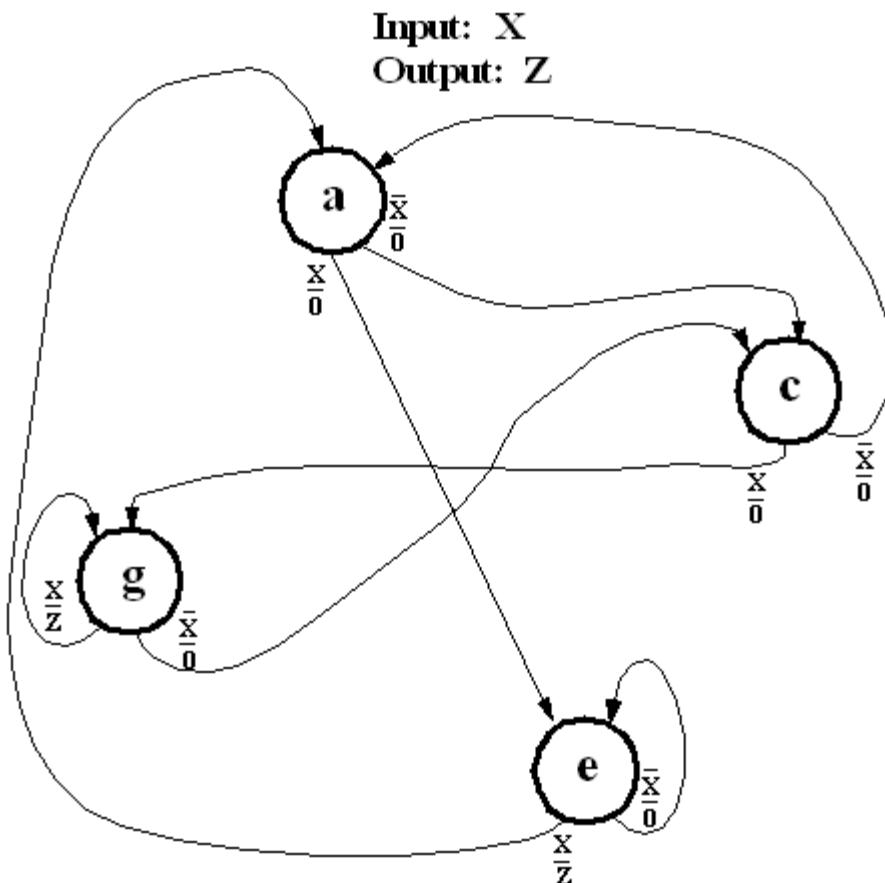


Extended State Transition Graph

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

State Table

State Reduction Example



$a \equiv b, c \equiv d, e \equiv f$

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

Final Reduced Table

Extended State Transition Graph
(Reduced Representation)

Implication Table Method

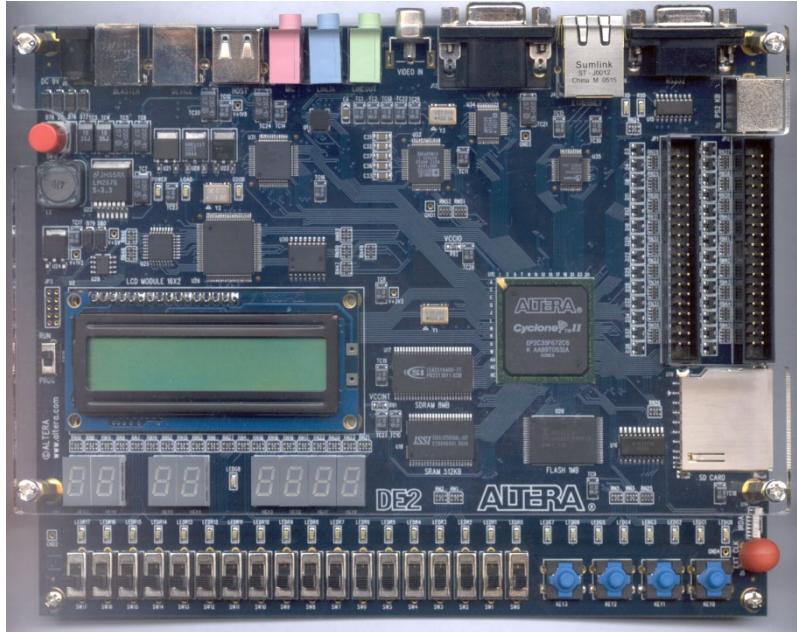
- 1. Construct a chart that contains a square for each pair of states.
- 2. Compare each pair in the state table. If the outputs associated with states i and j are different, place an X in square $i-j$ to indicate that $i \neq j$.
If outputs are the same, place the implied pairs in square $i-j$.
If outputs 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 $i \neq j$, and place X in square $i-j$.
- 4. If any X s were added in step 3, repeat step 3 until no more X s are added.
- 5. For each square $i-j$ that does not contain an X , $i == j$.

CPE 322 Fundamentals of Hardware Design

Electrical and Computer Engineering
University of Alabama in Huntsville

Hardware Testing and Design For Testability Issues

Hardware/Software Co-Design and General Trade-offs



Hardware Testing and Design for Testability

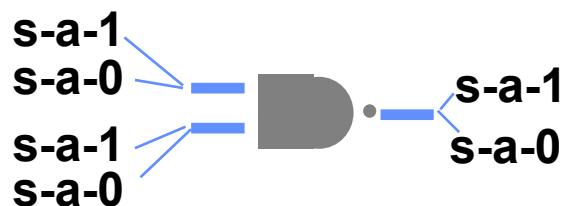
- Testing during design process
 - use Verilog HDL test benches to verify that the overall design and algorithms used are correct
 - verify timing and logic after the synthesis
- Post-fabrication testing
 - when a digital system is manufactured, test to verify that it is free from manufacturing defects
 - today, cost of testing is major component of the manufacturing cost
 - efficient techniques are needed to test and design digital systems so that they are easy to test

Testing Combinational Logic

- Common types of errors
 - short circuit
 - open circuit
- If the input to a gate is shorted to ground, the input acts as if it is stuck at logic 0
 - s-a-0 (stuck-at-0) faults
- If the input to a gate is shorted to positive supply voltage, the input acts as if it is stuck at logic 1
 - s-a-1 (stuck-at-1) faults

Stuck-at Faults

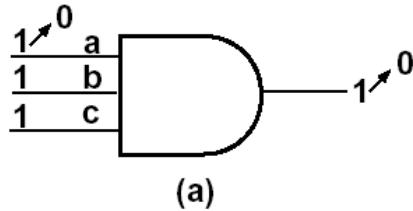
- How many single stuck-at faults —
 - $2(n + 1)$ — where n is the number of inputs



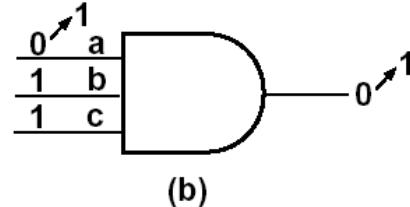
- We will assume
 - that there is only one stuck-at-fault active at a time in the whole circuit
 - “SSF” — single stuck-at fault

Stuck-at Faults for AND and OR gates

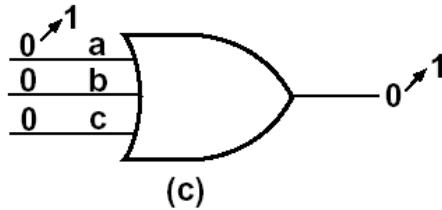
Test a
for s-a-0



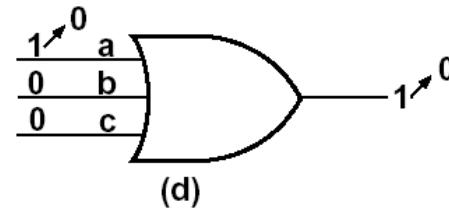
Test a
for s-a-1



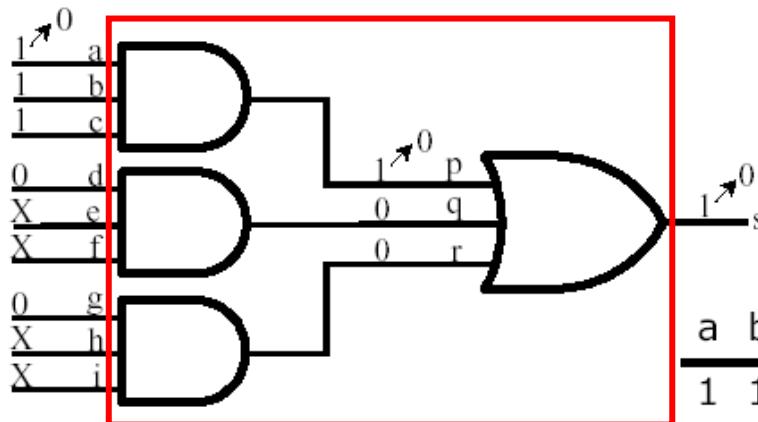
Test a
for s-a-1



Test a
for s-a-0



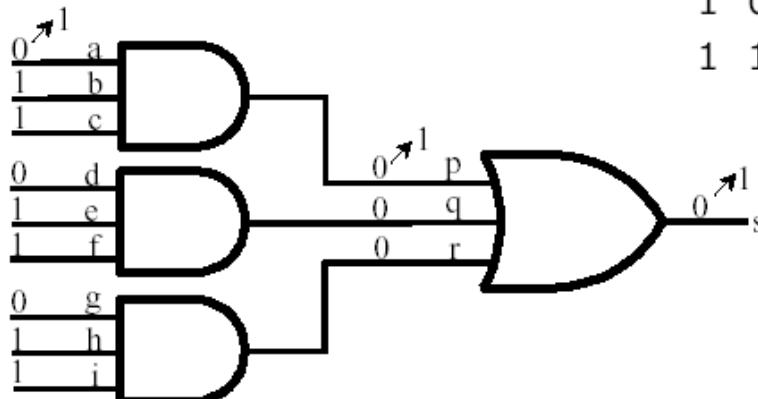
Testing an AND-OR Network



(a) stuck-at-0 test

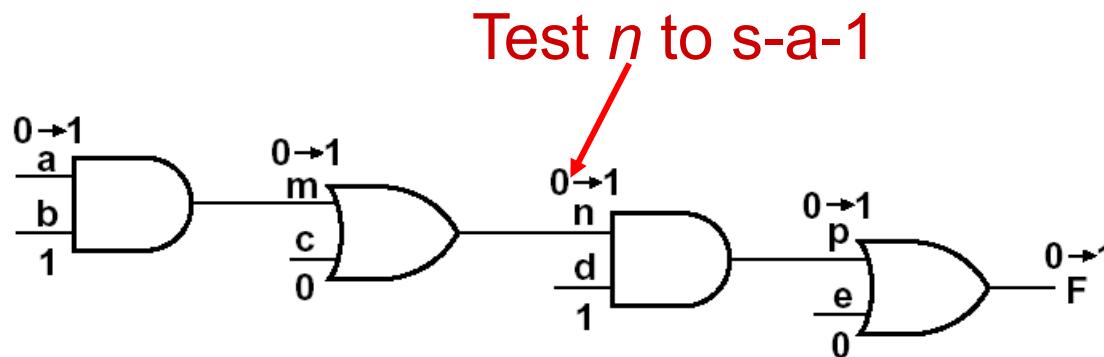
BRUTE-FORCE testing:
apply $2^9=512$ different input
combinations and check the output

a	b	c	d	e	f	g	h	i	Faults Tested
1	1	1	0	X	X	0	X	X	a0, b0, c0, p0
0	X	X	1	1	1	0	X	X	d0, e0, f0, q0
0	X	X	0	X	X	1	1	1	g0, h0, i0, r0
0	1	1	0	1	1	0	1	1	a1, d1, g1, p1, q1, r1
1	0	1	1	0	1	1	0	1	b1, e1, h1, p1, q1, r1
1	1	0	1	1	0	1	1	0	c1, f1, i1, p1, q1, r1



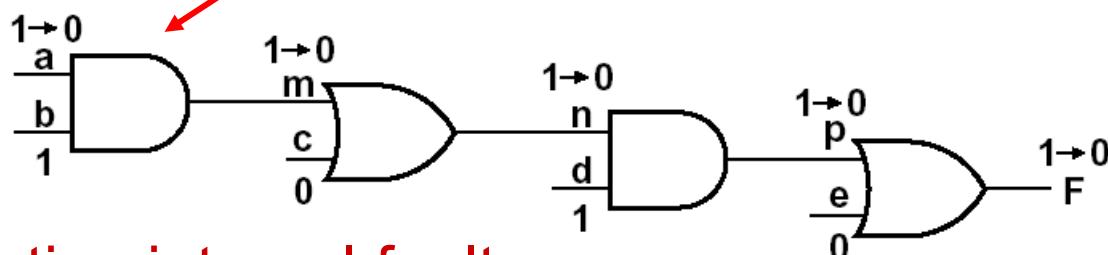
(b) stuck-at-1 test

Path Detection & Sensitization: Small Example



$n=0 \Rightarrow$
 $m=0, c = 0 \Rightarrow$
 $a=0, b=1, c=0$
 $d=1, e=0$

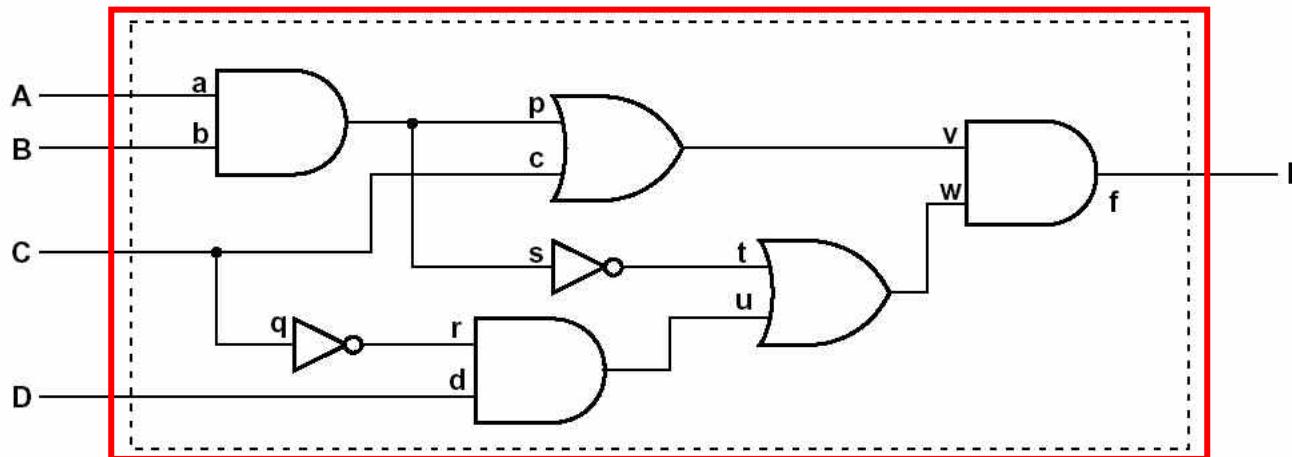
Change a to 1 =>
We can test a, m, n , or p to s-a-0



Testing internal faults:
choose a set of inputs that will excite the fault and
then propagate the fault to the network output

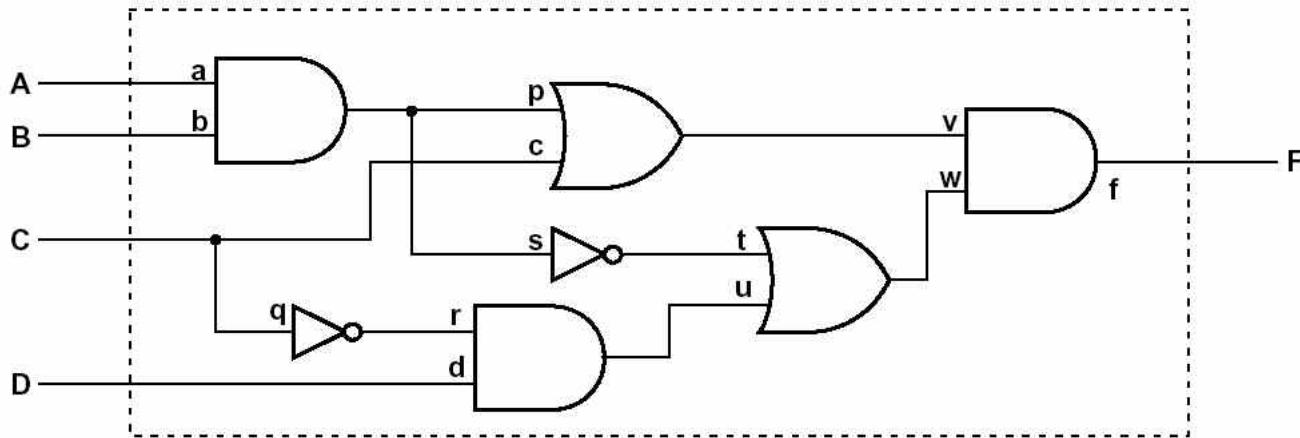
An Example

- What is a minimum set of test vectors to test the network below for all stuck-at-1 and stuck-at-0 faults?



- E.g., start with A-a-p-v-f-F path, determine the test vector to test s-a-0
- determine the list of faults covered
- select an untested fault, determine the required ABCD inputs
- determine the additional faults tested
- repeat the process until all faults are covered

An Example (cont'd)



- Step 1: A-a-p-v-f-F, s-a-0
 - ABCD: 1101 (+)
- Step 2: s-a-0 for c
 - C=1, p=0, w=1 => ABCD=1011 (*)
- Step 3: s-a-0 for q
 - C=1, D=1, t=0, s=1 => ABCD=1111 (#)
- Step 4: s-a-1 for a
 - A=0, B=1, C=0, D=1 => ABCD=0101 (&)
- Step 5: s-a-1 for d (%)
 - D=0, C =0, t=1 => ABCD = 1100

	0	1
a	+	&
b	+	*
c	*	&
d	+	%
p	+	*
q	#	+
r	+	#
s	#	*
t	*	#
u	+	#
v	+	&
w	+	#
f	+	#

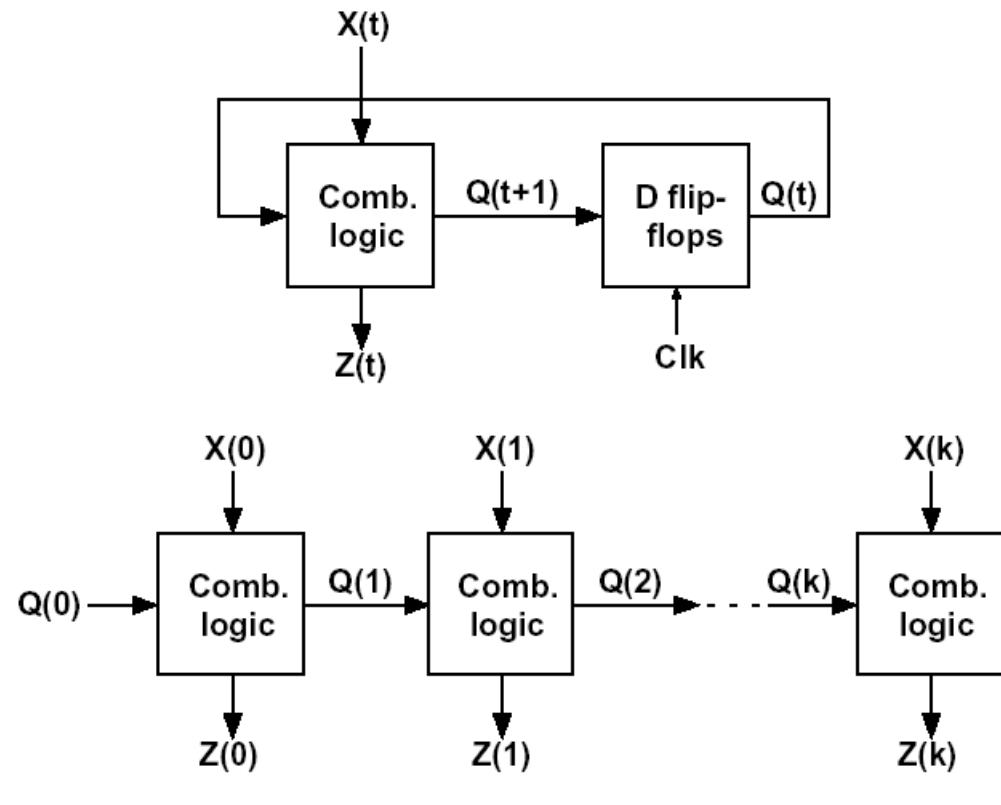
Testing Sequential Logic

- In general, much more difficult than testing combinational logic since we must use sequences of inputs
 - typically we can observe inputs and outputs, not the state of flip-flops
 - assume the reset input, so we can reset the network to the initial state
- Test procedure
 - reset the network to the initial state
 - apply a test sequence and observe the output sequence
 - if the output is correct, repeat the test for another sequence
- How many test sequences do we have?
 - how do we test that the initial state of the network under test is equivalent to the initial state of the correct network?
 - what is the sequence length?

Testing Sequential Logic (cont'd)

- In practice, if the network has N or fewer states, then apply only input sequences of length less than or equal $2N-1$

- Example
 - consider a network which includes 5 inputs, 1 output, and 4 states
 - total number of test sequences: $(2^5)^7 = 2^{35} \Rightarrow$ infeasible (!)
 - derive a small set of test sequences that will adequately test a SN



Testing Sequential Logic (cont'd)

- Consider input sequence

- $X = 0\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ 1$

- Output sequence

- $Z = 0\ 0\ 1\ 0\ 1\ 1\ 1\ 1\ 0$

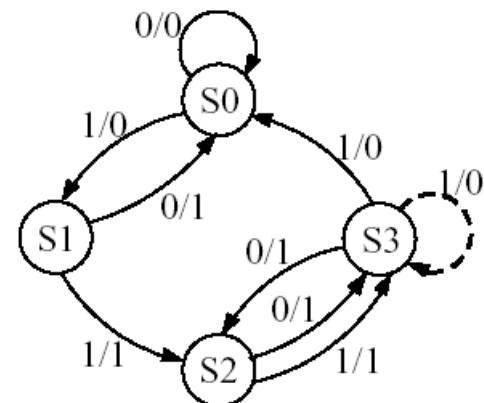
- If we change the network

- $S3 \rightarrow S0 \Rightarrow S3 \rightarrow S3$,

- the output sequence
will be the same

- Find distinguishing sequence

- an input sequence that will
distinguish each state from the
other states

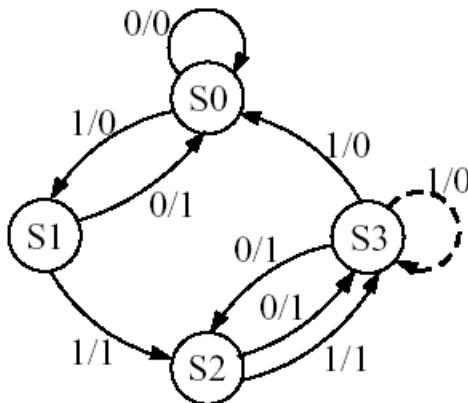


Q1Q2	State	Next State		Output	
		X=0	1	X=0	1
00	S0	S0	S1	0	0
10	S1	S0	S2	1	1
01	S2	S3	S3	1	1
11	S3	S2	S0	1	0

Input sequence: $X=11$

- $S0: Z = 01$
- $S1: Z = 11$
- $S2: Z = 10$
- $S3: Z = 00$

Testing Sequential Logic (cont'd)



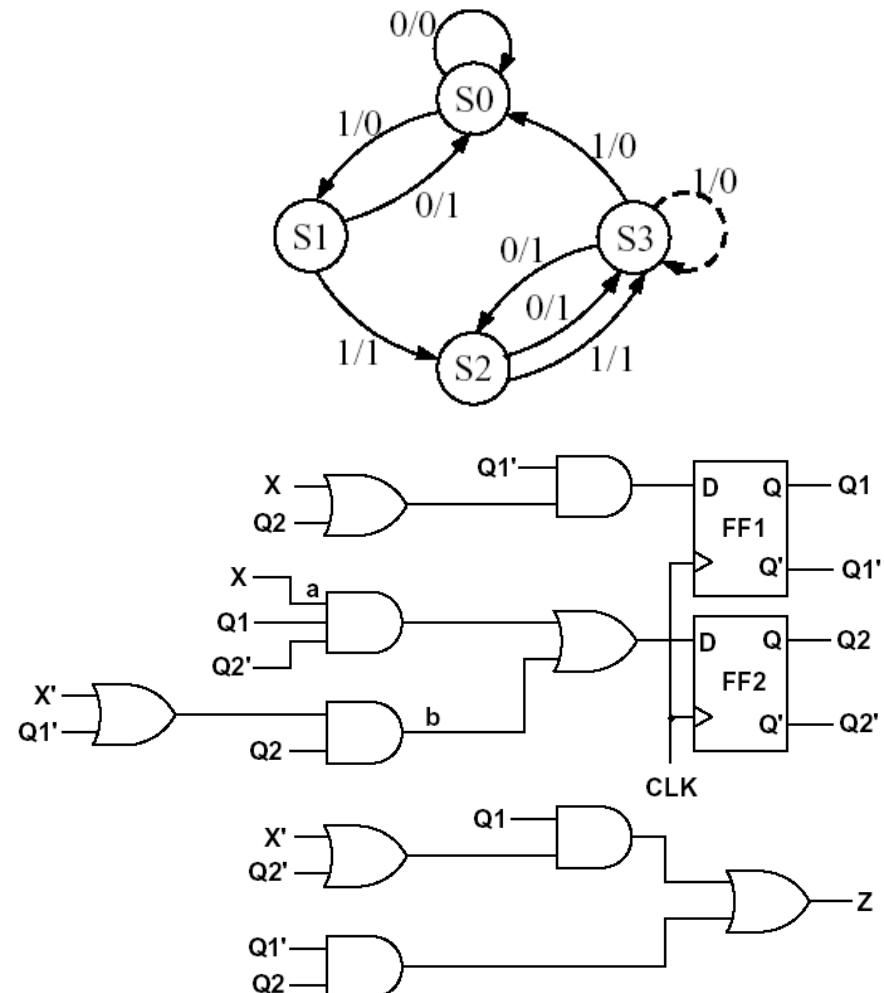
Q1Q2	State	Next State		Output	
		X=0	1	X=0	1
00	S0	S0	S1	0	0
10	S1	S0	S2	1	1
01	S2	S3	S3	1	1
11	S3	S2	S0	1	0

Verify each entry in the table using the following sequences:

Input	Output	Transition Verified
R 0 1 1	0 0 1	(S0 to S0)
R 1 1 1	0 1 1	(S0 to S1)
R 1 0 1 1	0 1 0 1	(S1 to S0)
R 1 1 1 1	0 1 1 0	(S1 to S2)
R 1 1 0 1 1	0 1 1 0 0	(S2 to S3)
R 1 1 1 1 1	0 1 1 0 0	(S2 to S3)
R 1 1 0 0 1 1	0 1 1 1 1 0	(S3 to S2)
R 1 1 0 1 1 1	0 1 1 0 1 0	(S3 to S0)

Testing Sequential Logic (cont'd)

- Implementation of the FSM
 - $S_0=00$, $S_1=10$, $S_2=01$, $S_3=11$
- Test a for s-a-1
 - to do this Q_1Q_2 must be 10
=> go to the state S_1 and then set X to 0 (R10)
 - in normal operation,
the next state will be S_0 ;
if a is s-a-1 then next state is S_2
 - distinguish the state (S_0 or S_2);
apply sequence 11
 - Final sequence: R1011
Normal output: 0101
Faulty output: 0110

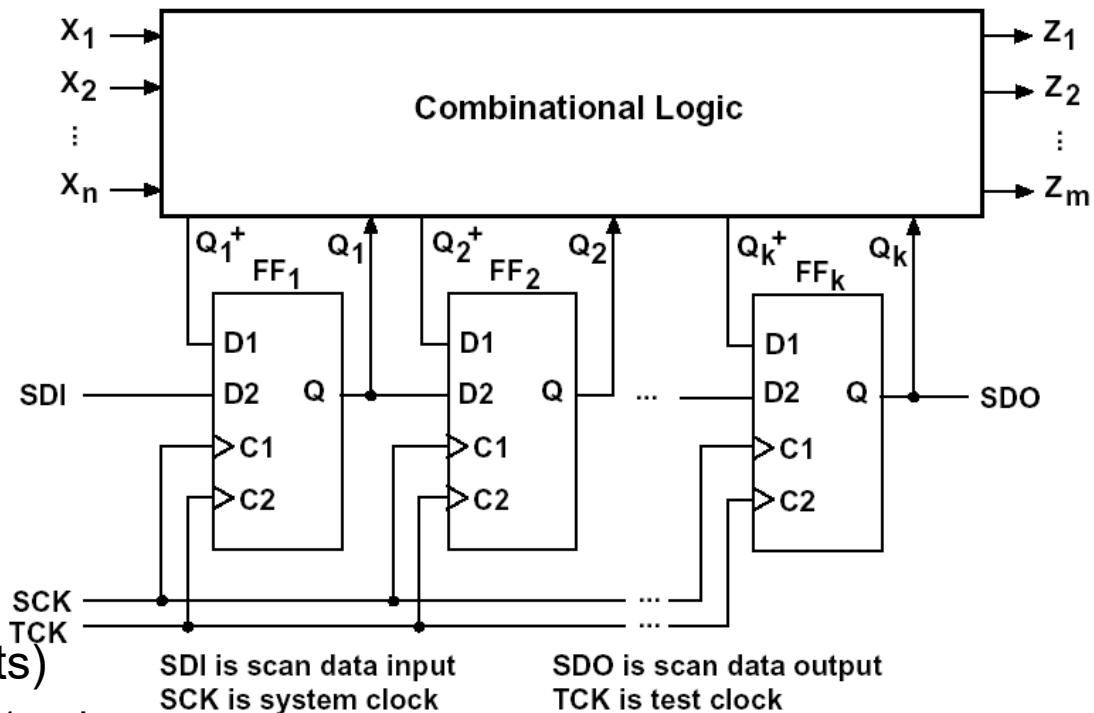


Scan Testing

- Testing of sequential networks is greatly simplified if we can observe the state of all the flip-flops instead of just observing the network outputs
 - Connect the output of each flip-flop to one of the IC pins?
 - Arrange flip-flops to form a shift register => shift out the state of flip-flops bit by bit using a single serial output pin => Scan path testing

Scan Path Testing

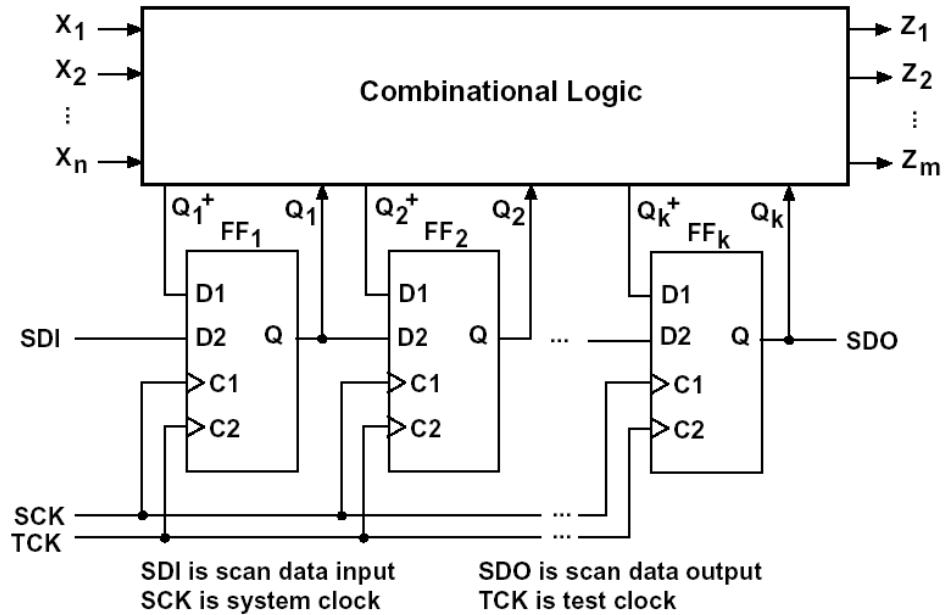
- Sequential network is separated into a combinational logic part and a state register composed of flip-flops



- Two ports FFs
(2 D inputs and 2 clock inputs)
 - D_1 is stored in the FF on $C1$ pulse
 - D_2 is stored in the FF on $C2$ pulse
 - Q of each FF is connected to D_2 of the next FF to form a shift register

Scan Path Testing

- Normal operation
 - system clock SCK = C1
 - inputs: $X_1 X_2 \dots X_N$
 - outputs: $Z_1 Z_2 \dots Z_N$
- Testing
 - FFs are set to a specified state using the SDI and TCK
 - test vector is applied $X_1 X_2 \dots X_N$
 - outputs $Z_1 Z_2 \dots Z_N$ are verified
 - SCK is pulsed to take the network to the next state
 - next state is verified by pulsing the TCK to shift the state code out of the scan register via SDO

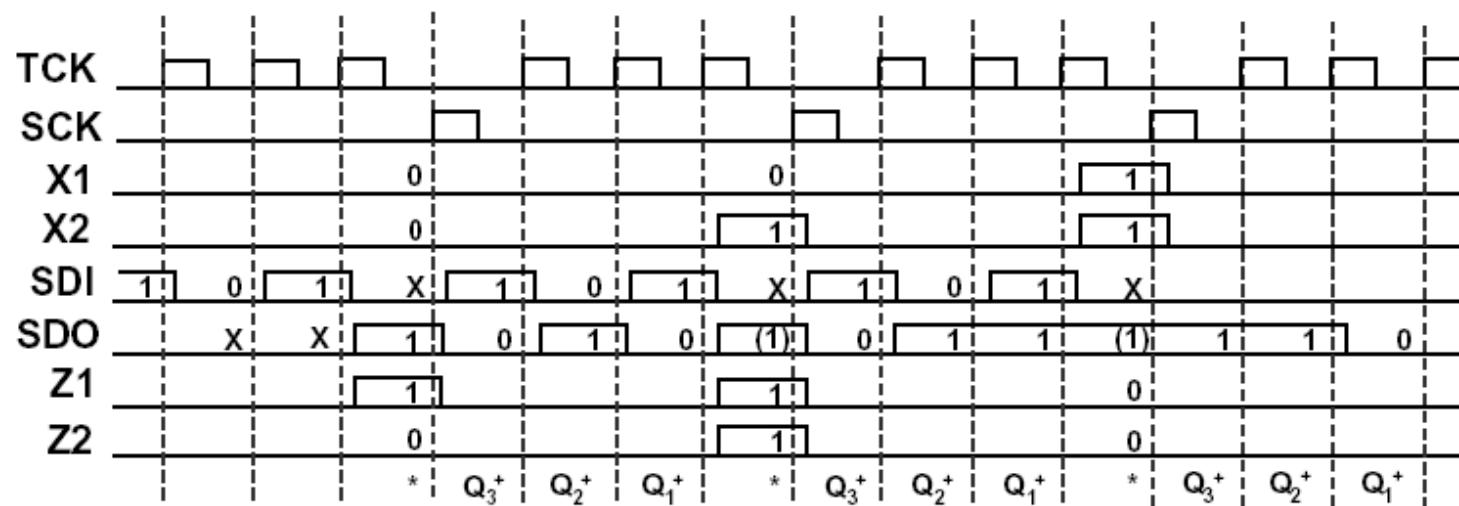


Scan Path Testing: An Example

- SQ: X_1X_2 , $Q_1Q_2Q_3$, Z_1Z_2

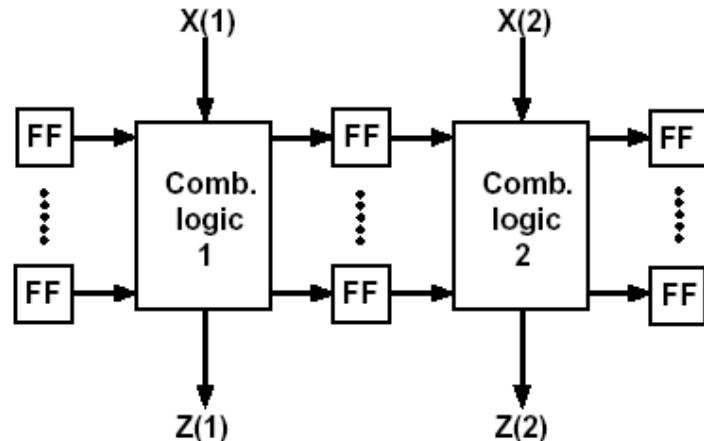
One row of the state transition table:

$Q_1Q_2Q_3$	$Q_1^+Q_2^+Q_3^+$				Z_1Z_2				
	$X_1X_2 =$	00	01	11	10	00	01	11	10
101						010	110	011	111

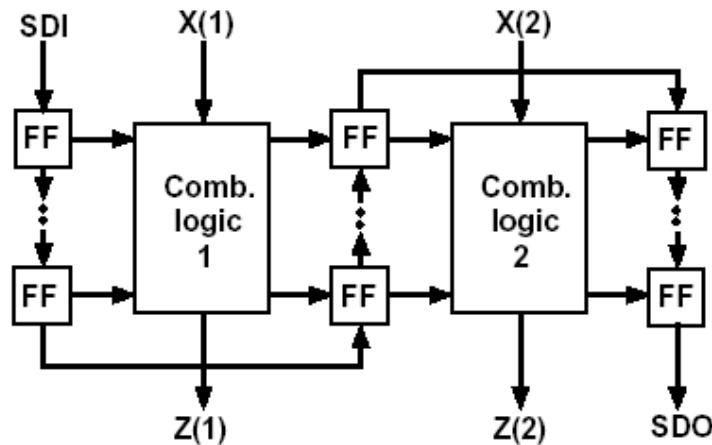


* Read output (output at other times not shown)

Scan Chain

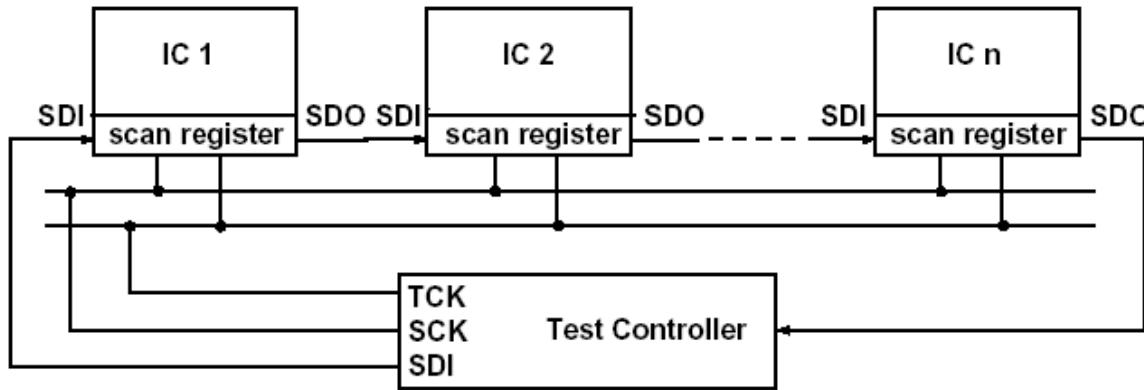


(a) Without scan chain



(b) With scan chain added

Scan Test with Multiple ICs

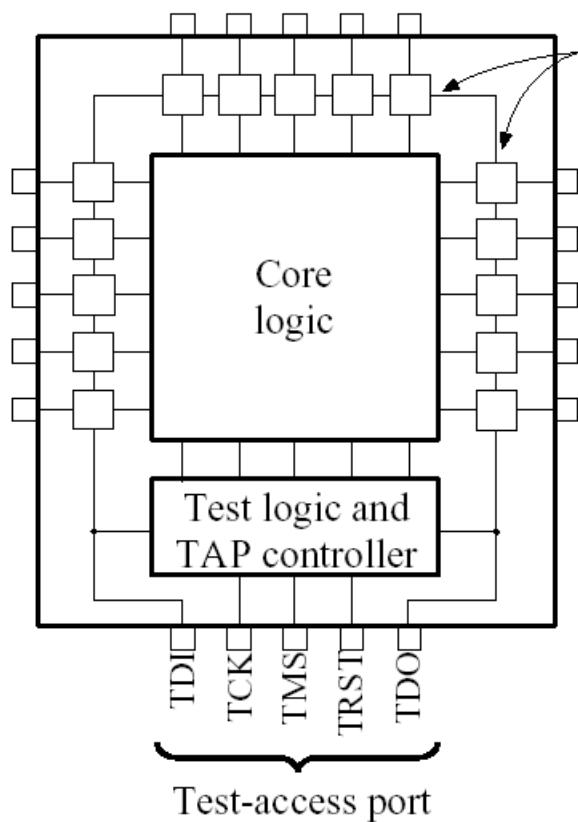


Boundary Scan

- PCB testing has become more difficult
 - ICs have become more complex, with more and more pins
 - PCBs have become more denser with multiple layers and fine traces
 - Bed-of-nails testing
 - use sharp probes to contact the traces on the board
 - test data are applied to and read from various ICs
 - => not practical for high-density PCBs with fine traces and complex ICs
- Boundary scan test methodology:
introduced to facilitate the testing of complex PC boards
 - developed by JTAG (Joint Task Action Group)
 - adopted as ANSI/IEEE Standard 1149.1 –
“Standard Test Access Port and Boundary Scan Architecture”
 - IC manufacturers make ICs that conform the standard
 - ICs can be linked together on a PCB, so that they can be tested using only a few pins on the PCB edge connector

Boundary Scan Register

- Boundary Scan Register (BSR) – cells of the BSR are placed between input or output pins and the internal core logic
- Four or five pins of the IC are devoted to the test-access-port (TAP)

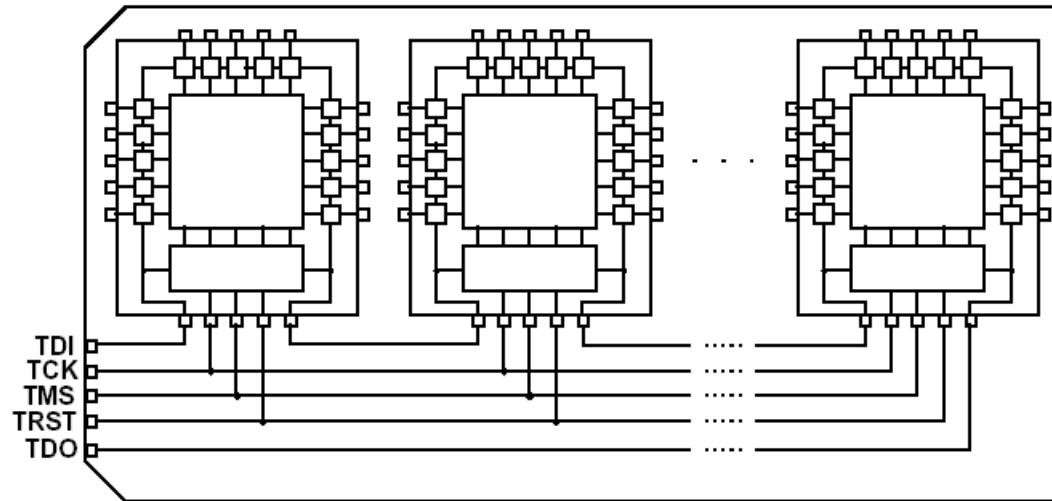


Boundary
scan cells

TAP pins

- TDI – Test data input
(data are shifted serially into the BSR)
- TCK – Test clock
- TMS – Test mode select
- TDO – Test data output (serial output from BSR)
- TRST – Test reset
(resets the TAP controller and test logic – optional pin)

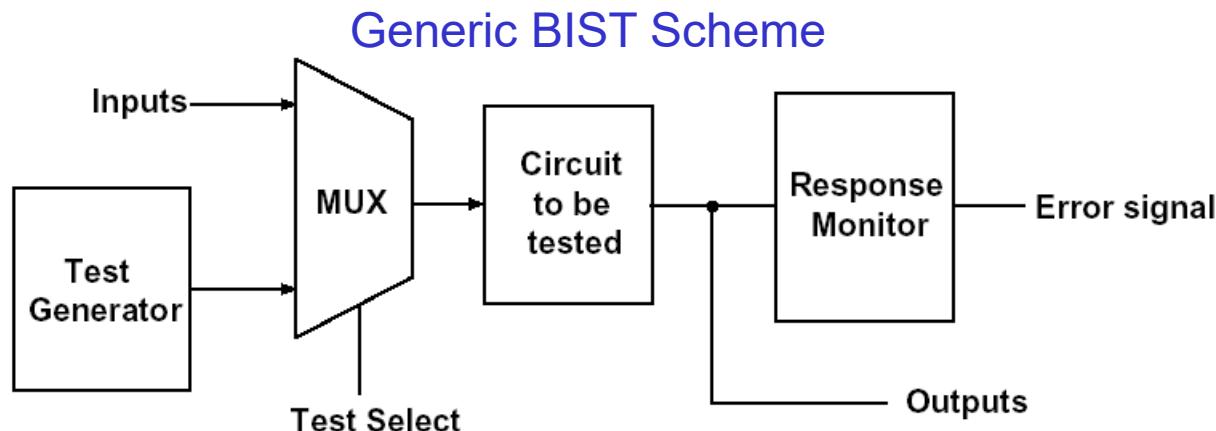
PCB with Boundary Scan ICs



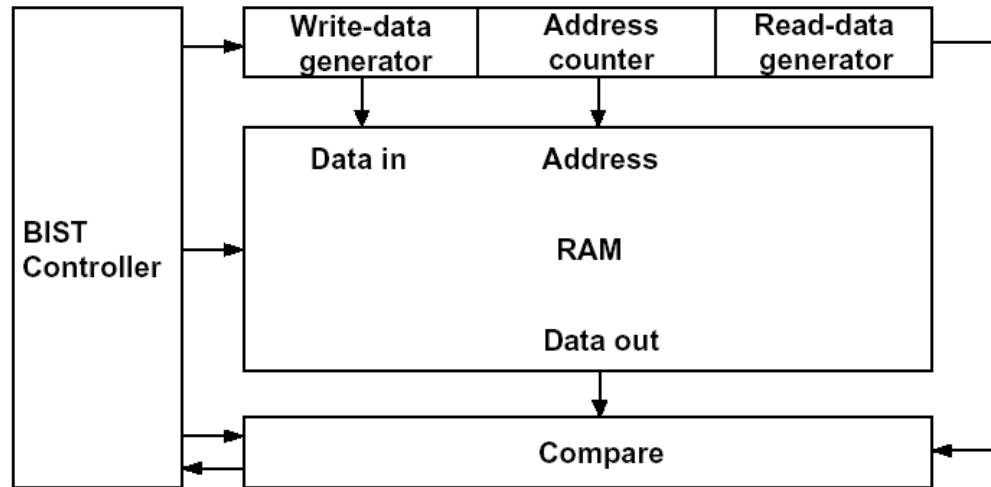
- BSRs in the ICs are linked together serially in a single chain with input TDI and output TDO.
- TCK, TMS, TRST are connected in parallel to all of the ICs.

Built-In Self-Test

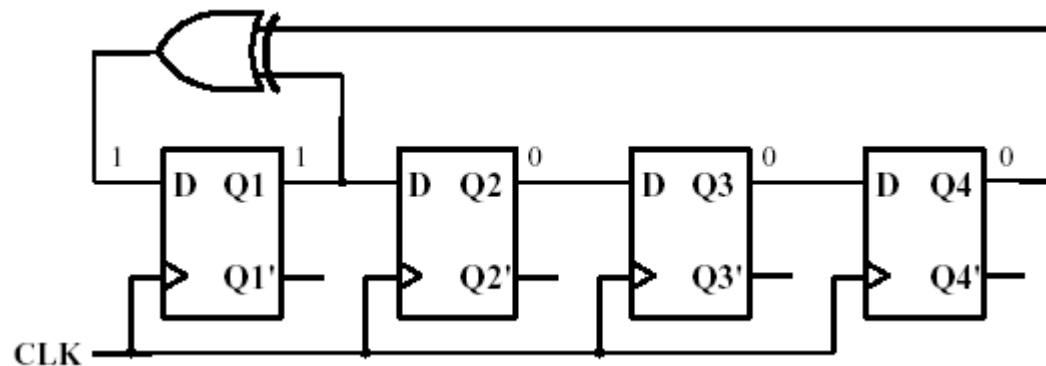
- Add logic to the IC so that it can test itself
 - Built-In Self-Test – BIST
- Using BIST
 - when test mode is selected by the test-select signal, an on-chip test generator applies test patterns to the circuit under test
 - the resulting outputs are observed by the response monitor, which produces an error signal if an incorrect output is detected



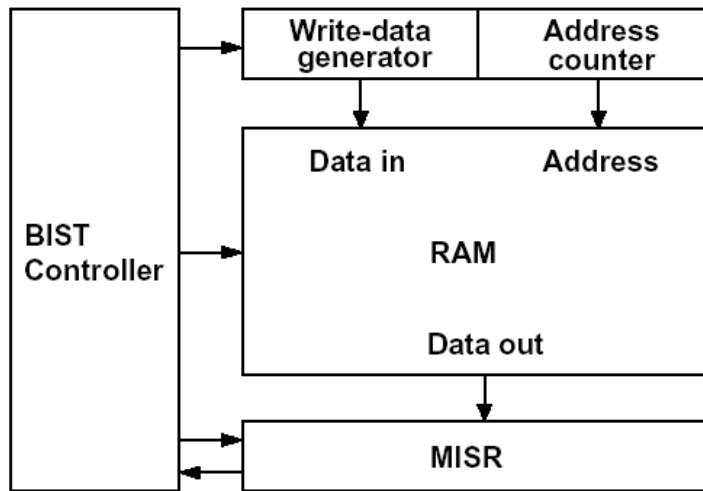
Self-Test Circuit for RAM



Linear Feedback Shift Registers (LFSR)



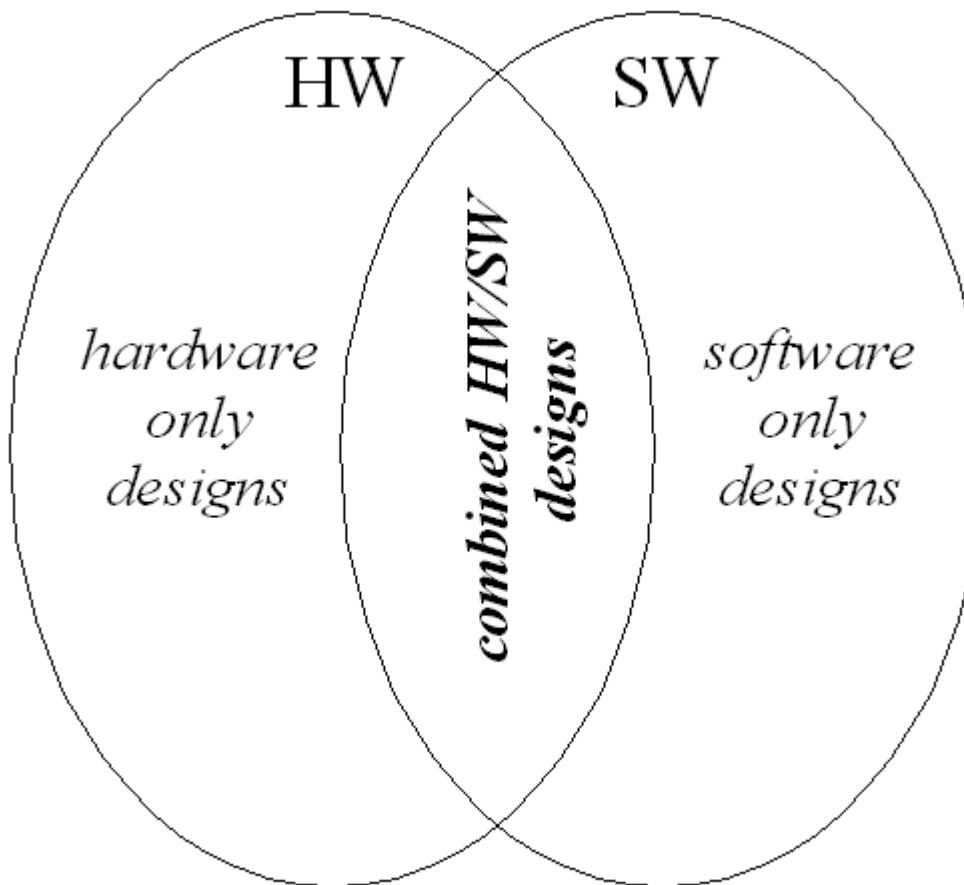
Self-Test Circuit for RAM with Signature Regs



MISR – Multiple Input
Signature Register

E.g. for MISR –form a check-sum by adding up all data bytes stored in the RAM

Hardware/Software Design Space Continuum



Hardware/Software Trade-offs

- Performance of hardware for a given function can be much faster than a software implementation because
 - There is little or no instruction fetch overhead like there is in a traditional instruction set processor
 - It can make use of the available fine grain concurrency and execute many operations in parallel
- Software is better suited for irregularly-structured control structures
- Traditionally software was considered to be more flexible and field upgradable when compared to hardware
 - FPGA's blur this distinction though

Hardware/Software Trade-offs

- Cost of implementing a function in hardware is usually much greater than implementing the same function in software
 - Requires additional chip area in an Application Specific Integrated Circuit or board area in a PC board or reconfigurable logic area in an FPGA.
 - Usually takes more time to create a hardware design than a software one
- Both hardware and software resources are limited but hardware resources tend to have tighter limits and should be reserved for elements that have the greatest effect on the performance (or possibly other important constraints such as reliability)

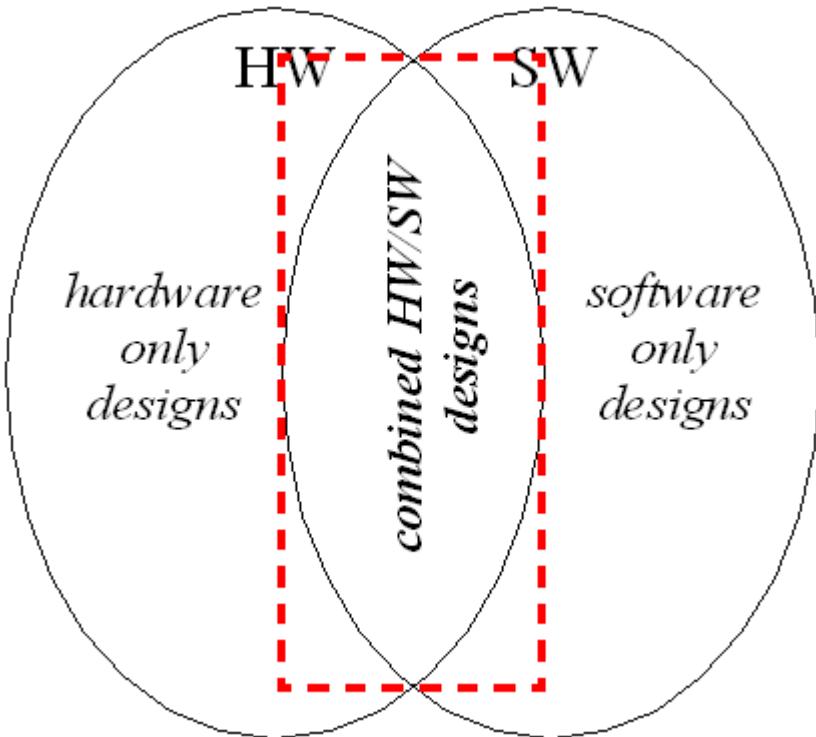
Hardware/Software Trade-offs

- Both hardware and software solutions allow for reduced energy consumption
 - Dynamic and Static Voltage Frequency Scaling, DVFS
 - Dynamic power management
 - Power gating with retention modes
 - Clock Gating (single and multi-level)
- Specific hardware energy consumption strategies can be applied at a finer grain which can result in extremely low power consumption for custom hardware.
- These features are also often dynamically controllable by software at a coarser grain using commercially available hardware for much less cost.

Differences between ISP Programming and FPGA Reconfiguration

- Instruction Set Processors, ISPs, such as microcontrollers, allow you to change the program or programs that execute in one or more thread control units.
- FPGAs and other reconfigurable hardware devices allow you to reconfigure both the control units and the data paths of the device.
- In both cases this allows one to increase the usability of the devices involved.
- The allowable number of configurations is much smaller for an ISP when compared to an FPGA but the size of the program (reconfiguration information) is also much smaller!

Embedded Systems



- Application-specific systems which contain hardware and software tailored for a particular task
- Generally part of a larger system
- Responds to external inputs and drives external outputs
- Often must respond in real-time
- Must adhere to strict area (footprint), performance, and power consumption constraints

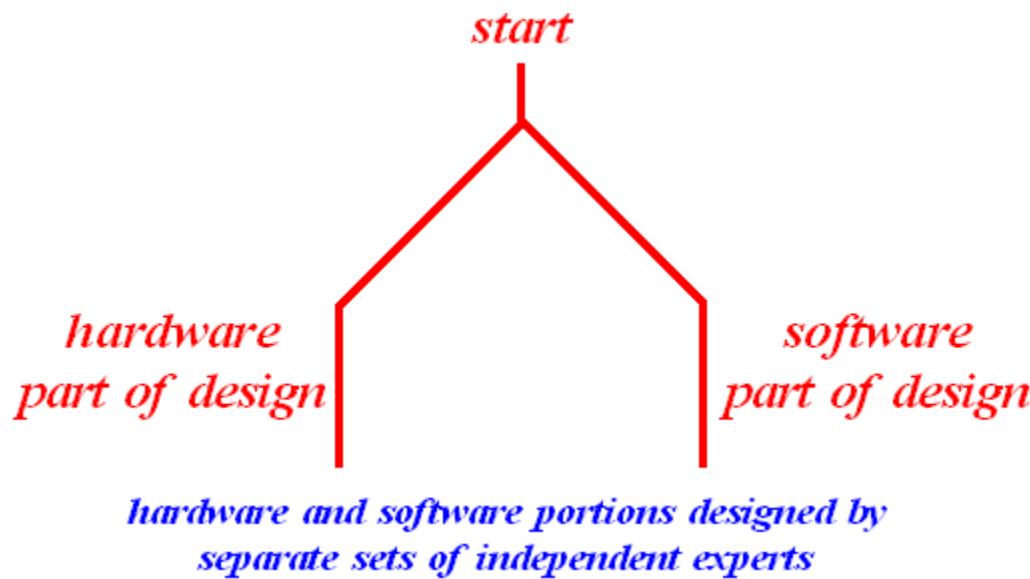
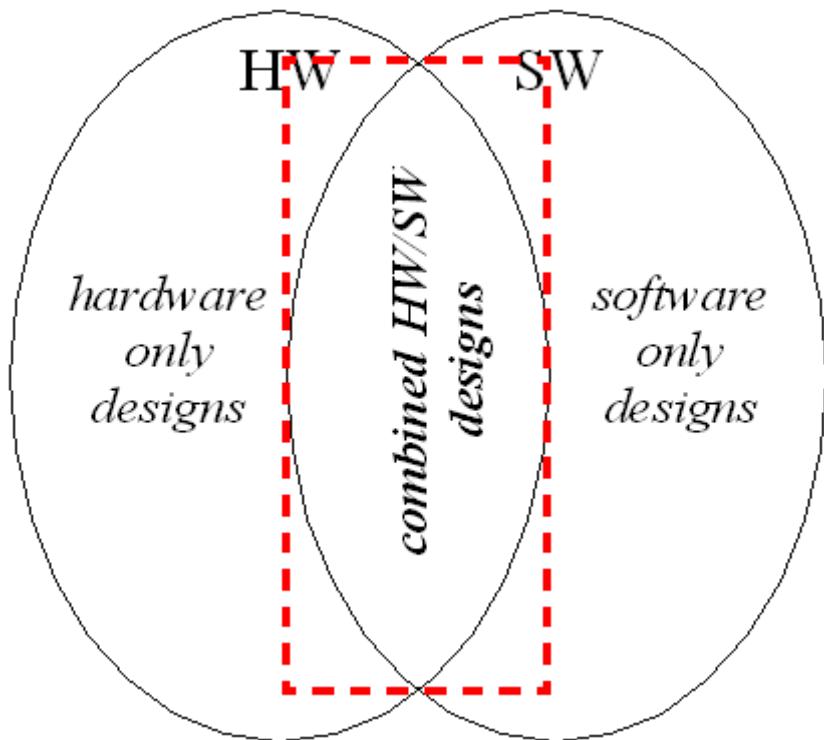
Embedded Systems Examples

- Automotive: Anti-lock breaks, engine control, dynamic ride control, engine diagnostics
- Consumer Electronics: cameras, DVD Controllers, Microwave Ovens, Toasters, Refrigerators, Washers, Smart Card Controllers, RFID systems, Stand Alone GPS systems, TV remote controller, landline telephones, smart scales,
- Industrial Process Controllers: motor control systems , electronic data acquisition and supervisory control, automated laboratory instrument control
- Computer Peripherals: printers, external disk controllers, FAX controllers
- Cell Phones and Peripherals: wireless headsets, wifi bridging devices
- Medical Applications: ECG display and diagnostics, blood cell recorder/ analyzer, patient monitor system
- Network Systems: Routers, network switches, external firewalls

Traditional HW/SW Design Process

- System requirements are developed and then analyzed to determine the “level” of technology required to fulfill these needs.
- Hardware and software teams then independently develop their designs
 - combining of design efforts occur late in the development cycle during the first prototype testing.
- Often leads to a generalized hardware platform being selected and all specialization occurring in the software.

Traditional HW/SW Design Process

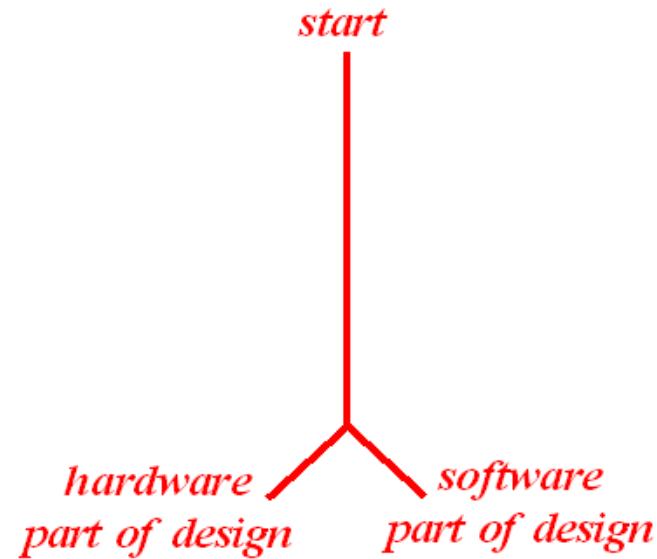
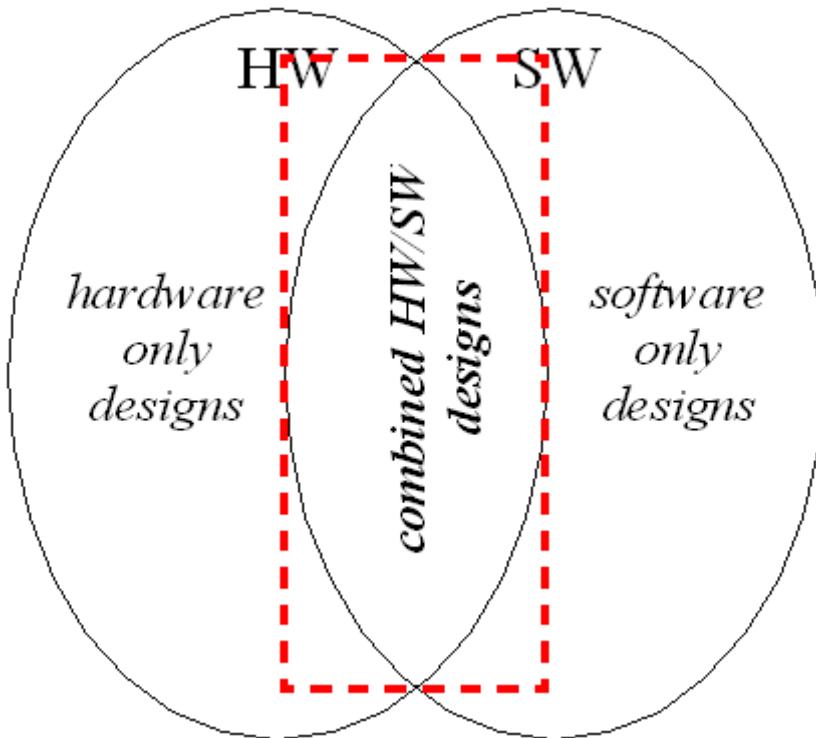


Hardware/Software Co-Design

Meeting system level objectives by exploiting the synergism of hardware and software through their concurrent design.

"Hardware/Software Co-Design", Giovanni De Micheli, and Rajesh K. Gupta, Readings in Hardware/Software Codesign Academic Press, 2002

Hardware/Software Codesign



*Hardware and software portions designed by the same group of experts with cooperation with one another.
Placement of functionality can occur later in the design cycle*

Hardware/Software Co-Design

Meeting system level objectives by exploiting the synergism of hardware and software through their concurrent design.

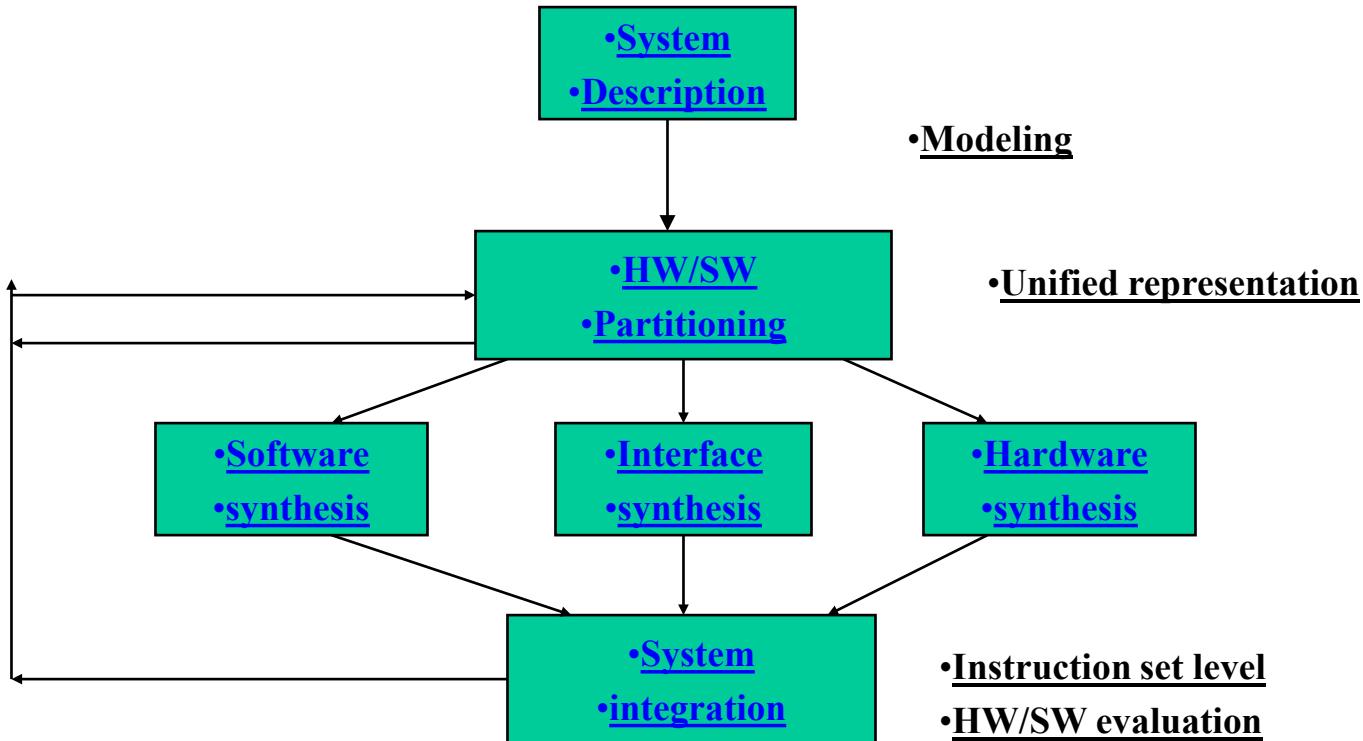
"Hardware/Software Co-Design", Giovanni De Micheli, and Rajesh K. Gupta, Readings in Hardware/Software Codesign Academic Press, 2002

Hardware/Software Co-Design

Design Flow for HW/SW Co-Design

- specification
- modeling
- design space exploration and partitioning
- synthesis and optimization
- validation
- implementation

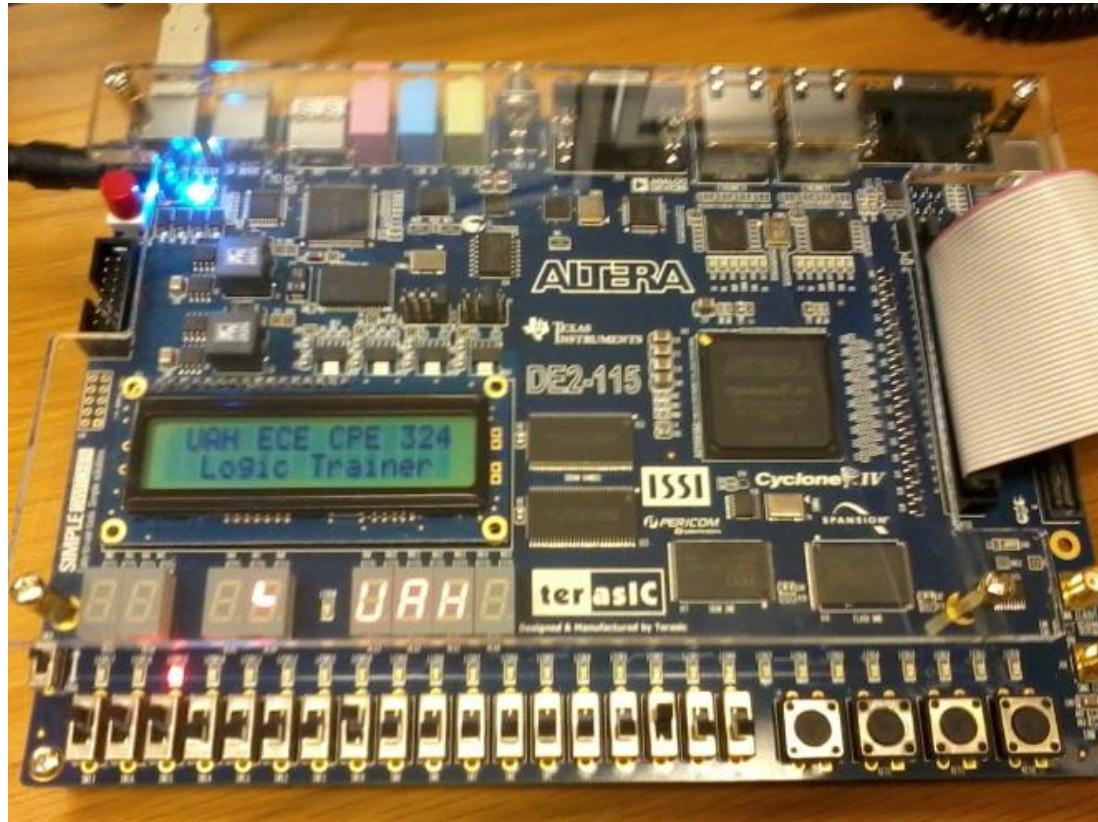
Hardware/Software Co-Design



CPE 322

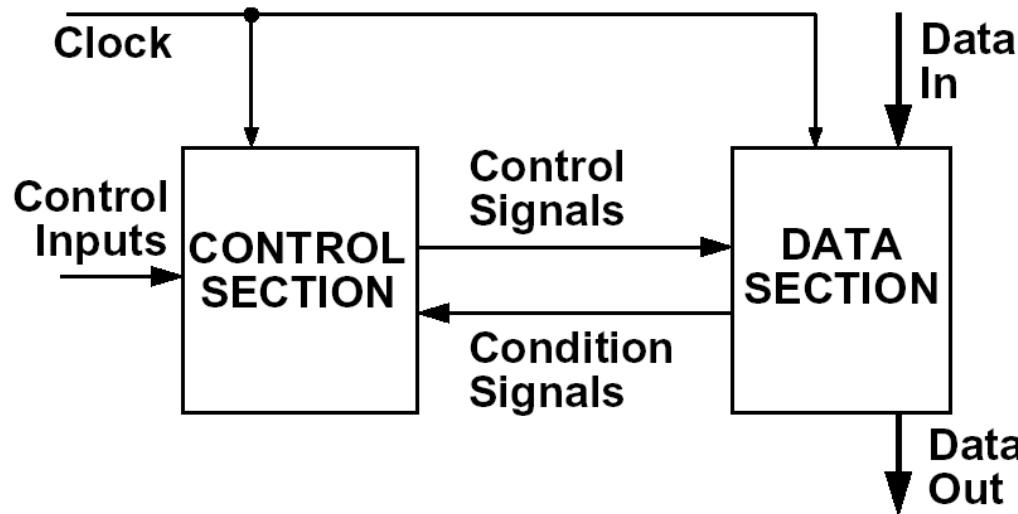
Digital Hardware Design Fundamentals

Timing Considerations in Synchronous Sequential Design

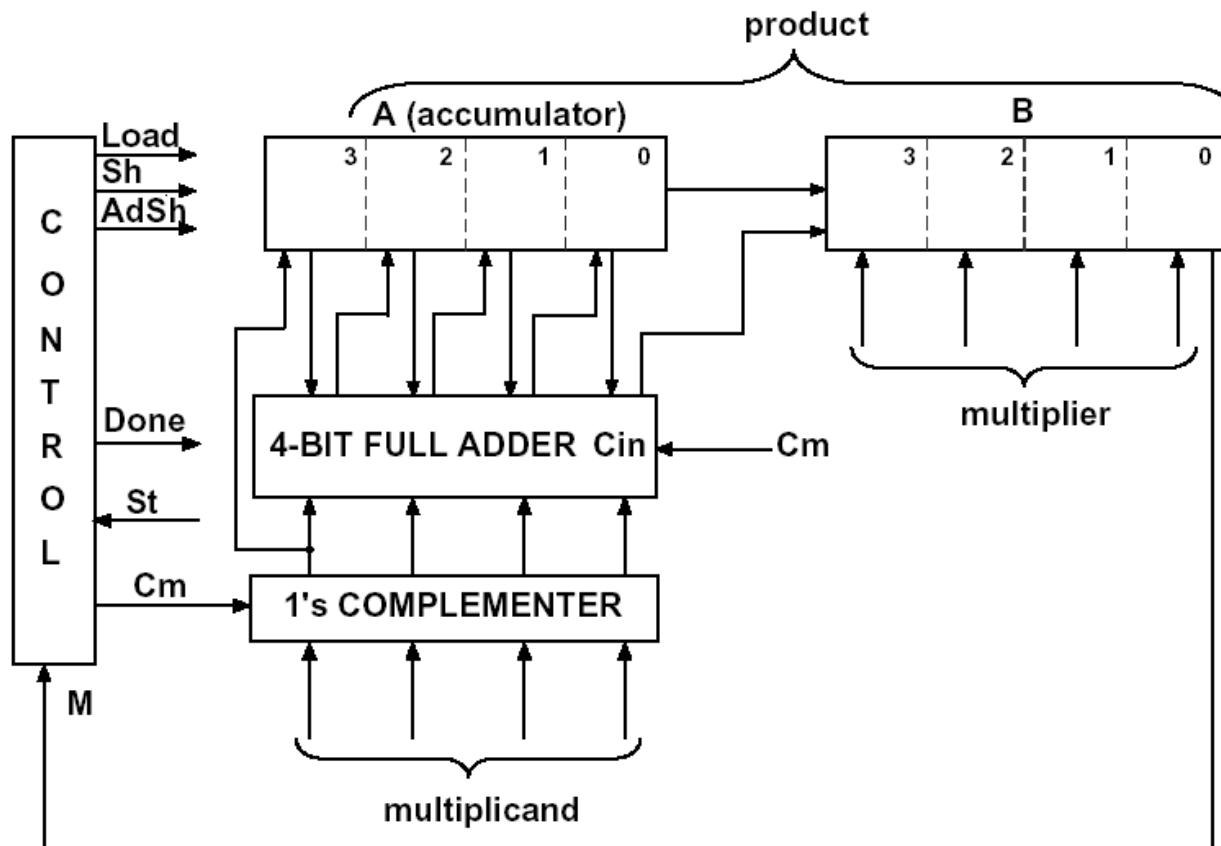


Synchronous Design

- 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



Example: Faster Multiplier

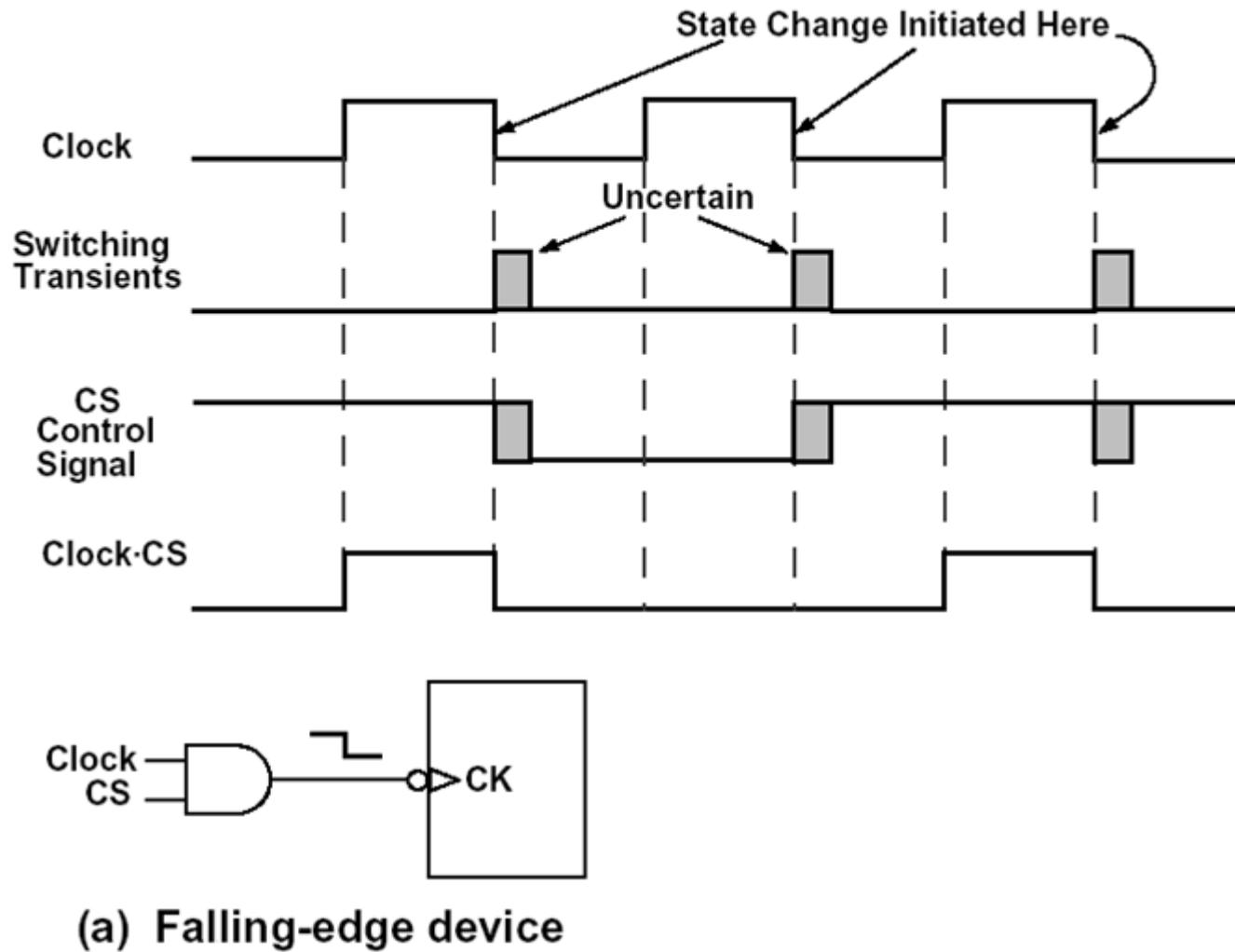


- Move wires from the adder outputs one position to the right => add and shift can occur at the same clock cycle

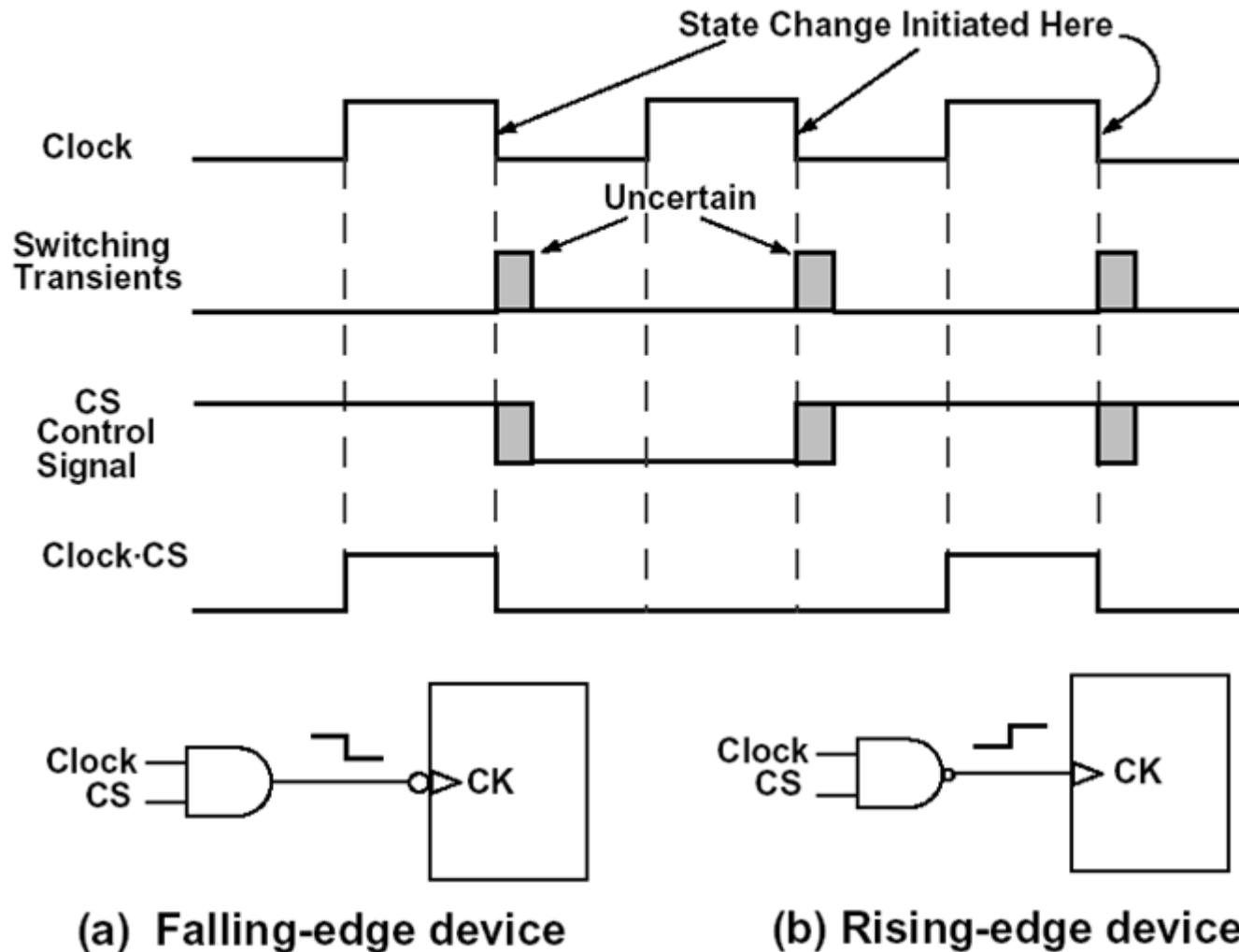
Control Signal Timing Issues

- Change in state of the flip-flops in control section determined by the propagation delay
- Time control signals change depend upon this FF propagation delay and combination network delay
- Glitches and spikes may occur in the control signals due to hazards in the network
- Noise may be introduced on the control signals by changing signals in another part of the circuit
- **THIS MEANS THAT THERE IS A TIME INTERVAL AFTER THE ACTIVE EDGE OF THE CLOCK WHERE THE STATE OF THE CONTROL SIGNAL IS NOT KNOWN AND MAY NOT BE STABLE**

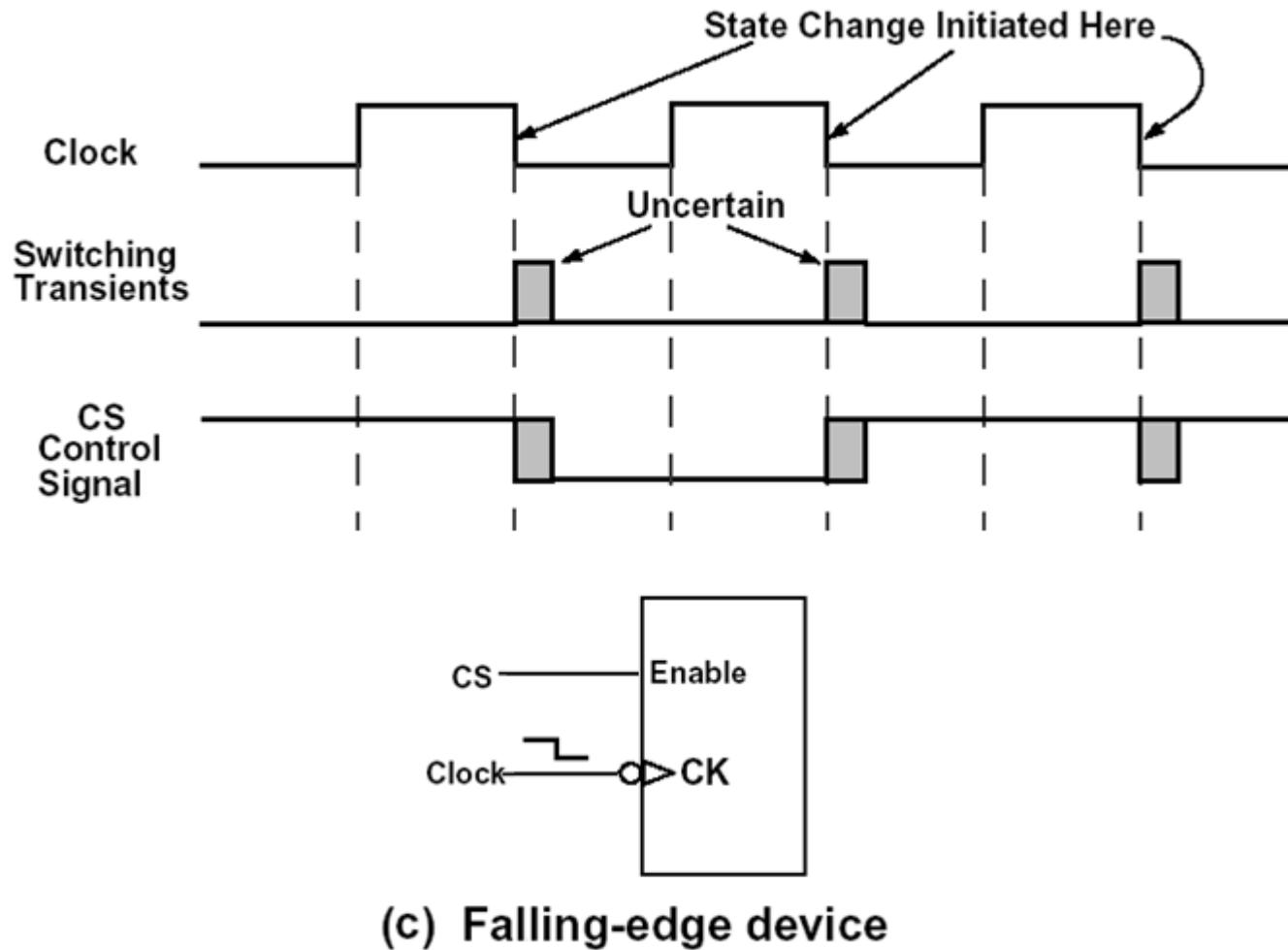
Control Signal Clock Gating when Active Edge is the Falling-Edge



Control Signal Gating when Active Clock Edge is the Falling-Edge



Use of an Enable when Active Clock Edge is the Falling-Edge



Control Signal Gating when Active Clock Edge is the Rising-Edge

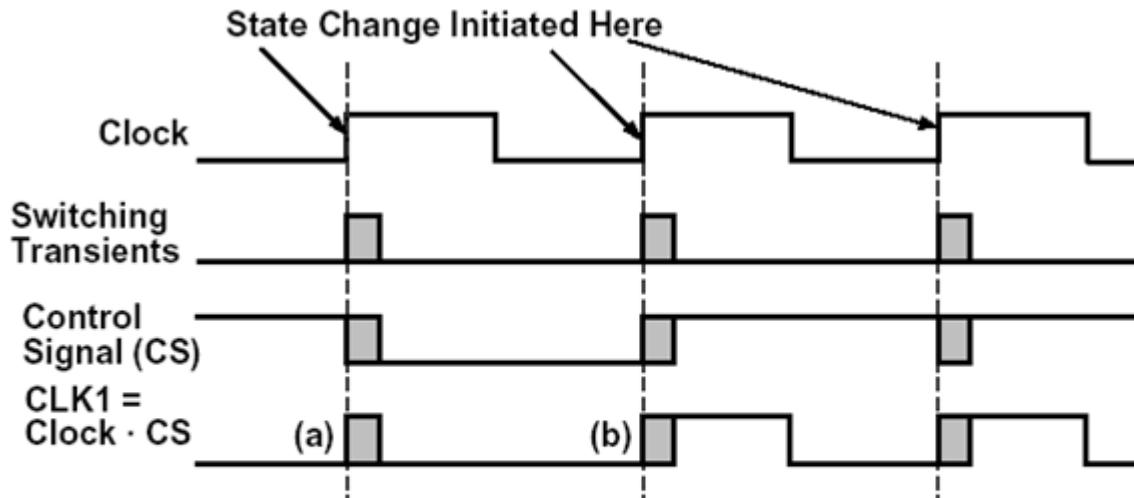
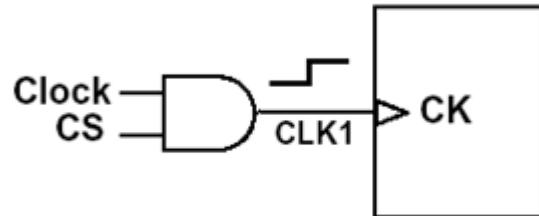


Figure 1-35 Incorrect Design



(a) Rising-edge device

Control Signal Gating when Active Clock Edge is the Rising-Edge

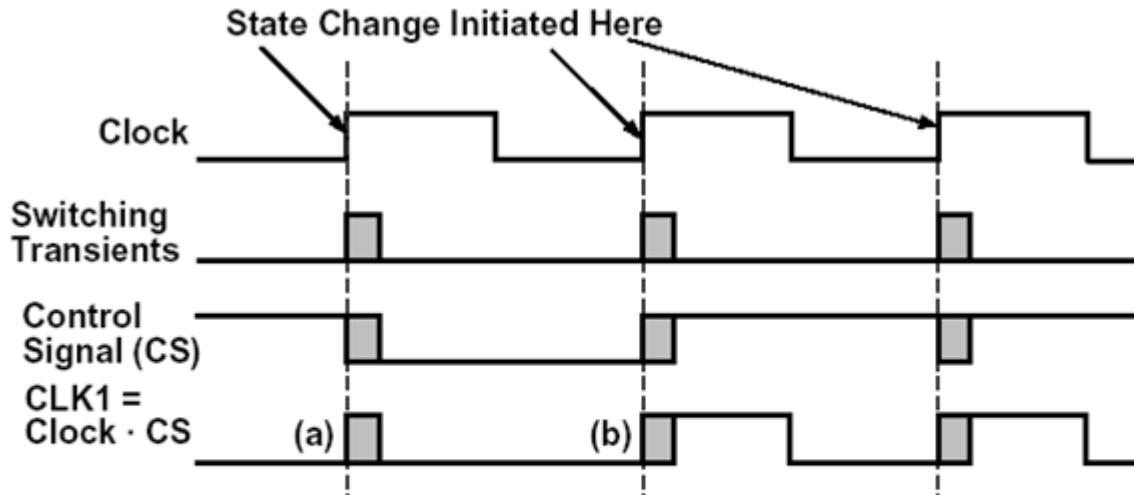
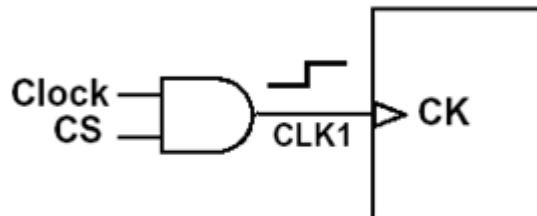
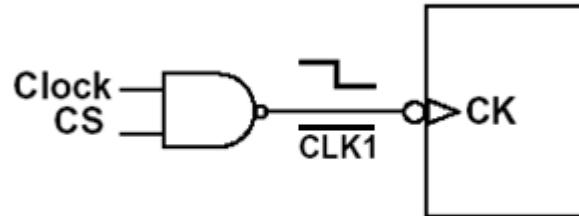


Figure 1-35 Incorrect Design



(a) Rising-edge device

Figure 1-35 Incorrect Design



(b) Falling-edge device

Control Signal Gating when Active Clock Edge is the Rising-Edge

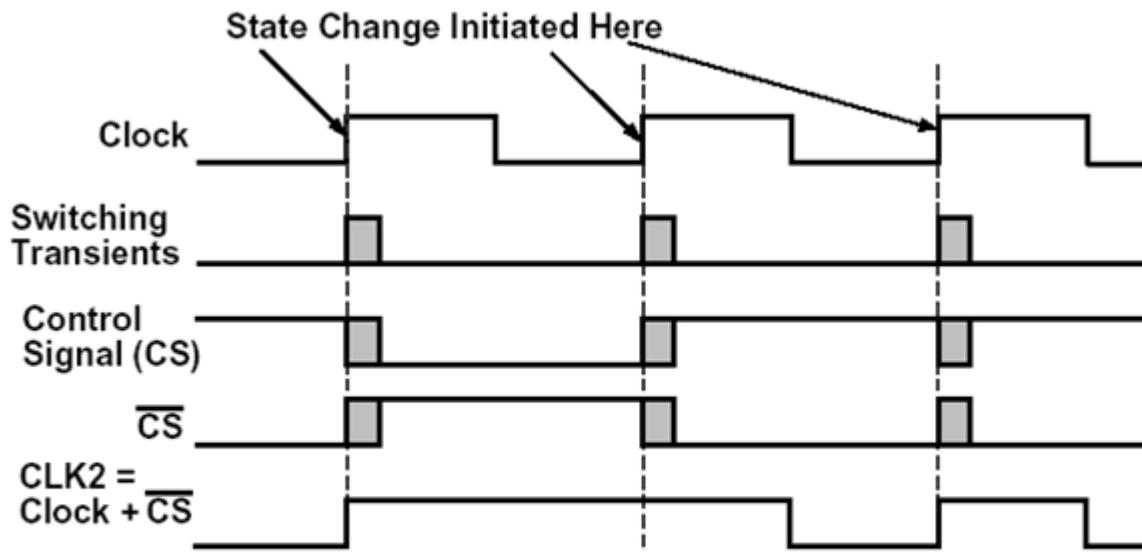
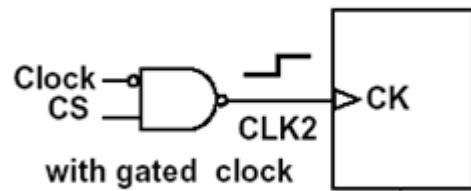


Figure 1-36 Correct Design



(a) Rising-edge device

Control Signal Gating when Active Clock Edge is the Rising-Edge

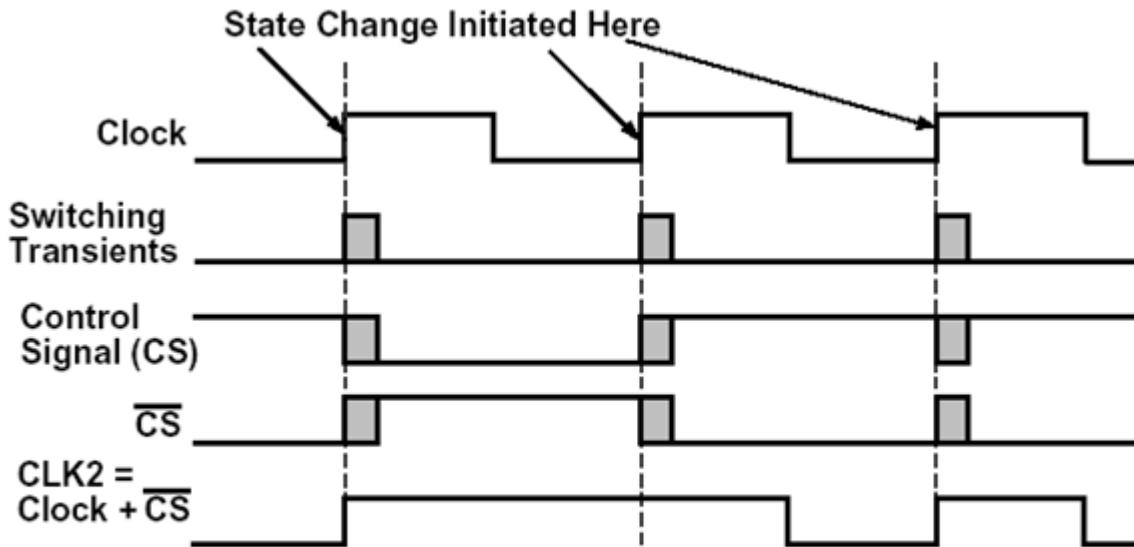
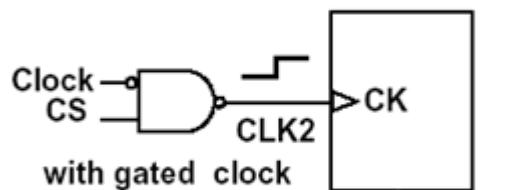
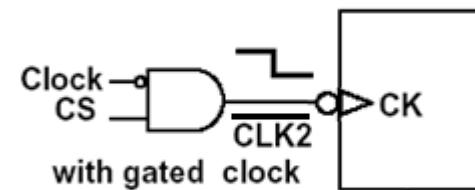


Figure 1-36 Correct Design



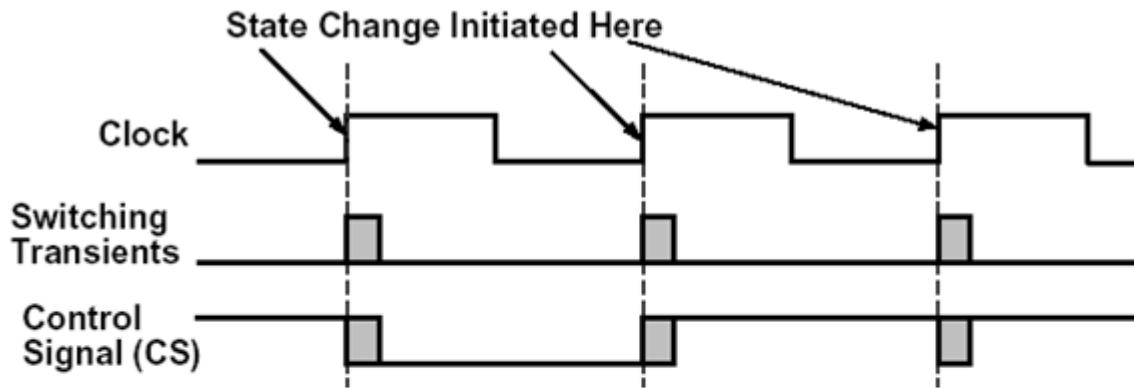
(a) Rising-edge device

Figure 1-36 Correct Design



(b) Falling-edge device

Control Signal Gating when Active Clock Edge is the Rising-Edge



Control Signal Gating when Active Clock Edge is the Rising-Edge

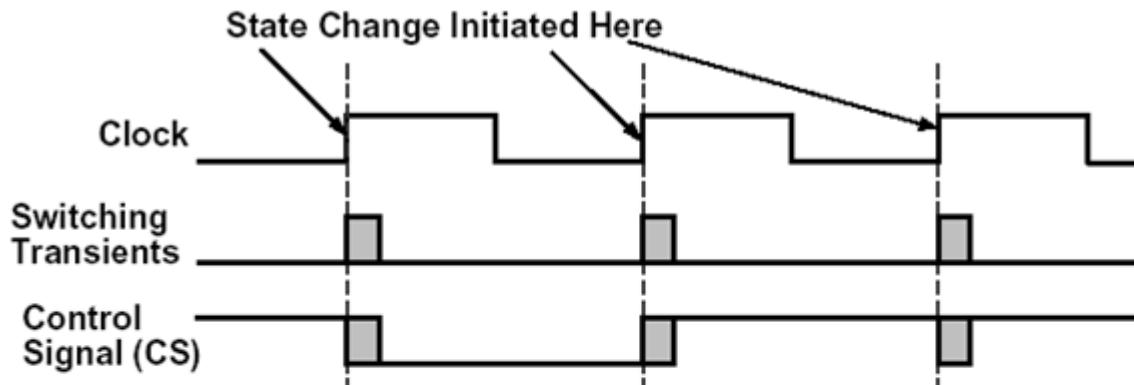
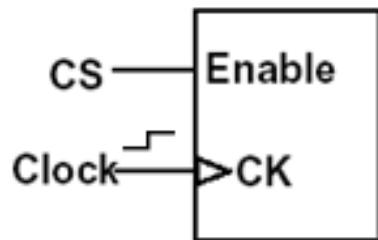
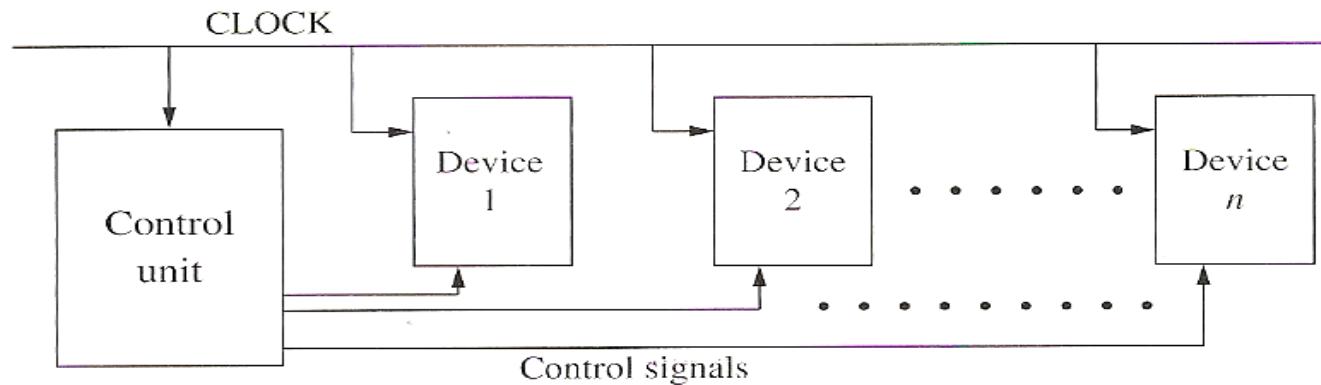


Figure 1-36 Correct Design



(c) Rising-edge device

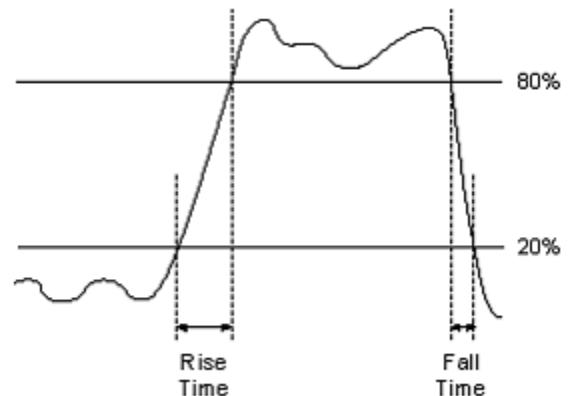
Synchronous Digital Systems



- 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
- Control section + Data Section
- Mealy and Moore Machines typically form the control sections of synchronous designs

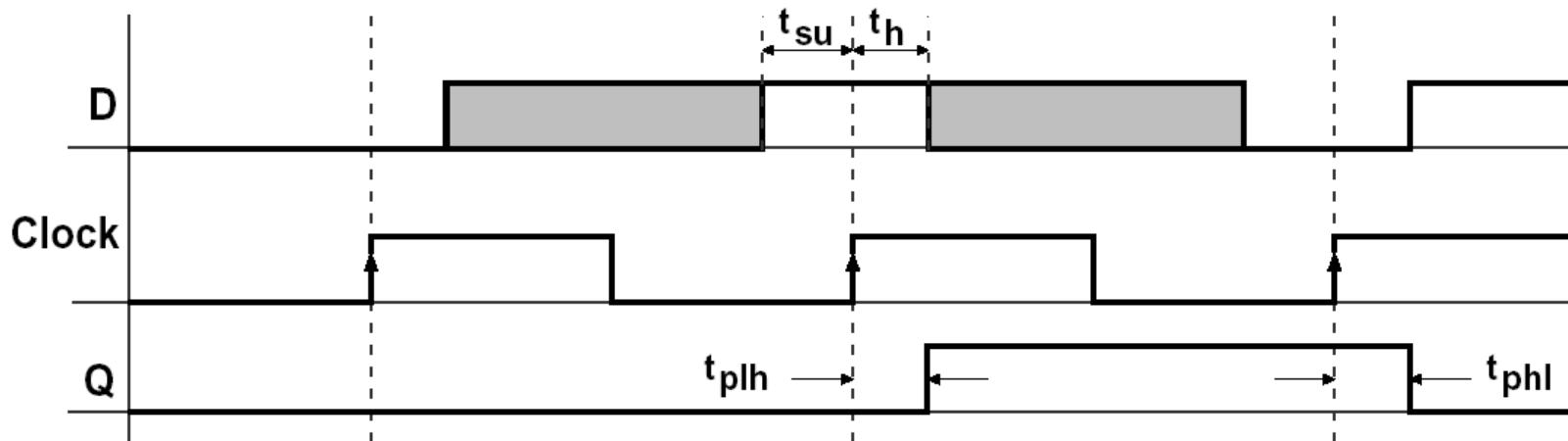
Rise/Fall Times & Pulse Width

- The rise time or fall time of a signal is usually defined as the time it takes the signal to change between the 10% to 90% values of the transition.
- An ideal timing diagram does not show rise times and fall times and each gate is assumed to have a 0 gate delay.
- The pulse width of a positive pulse is usually defined as the time it takes the signal to go from its 50% value on the rising edge of the pulse to its 50% value on the falling edge of the pulse.



Setup and Hold Times

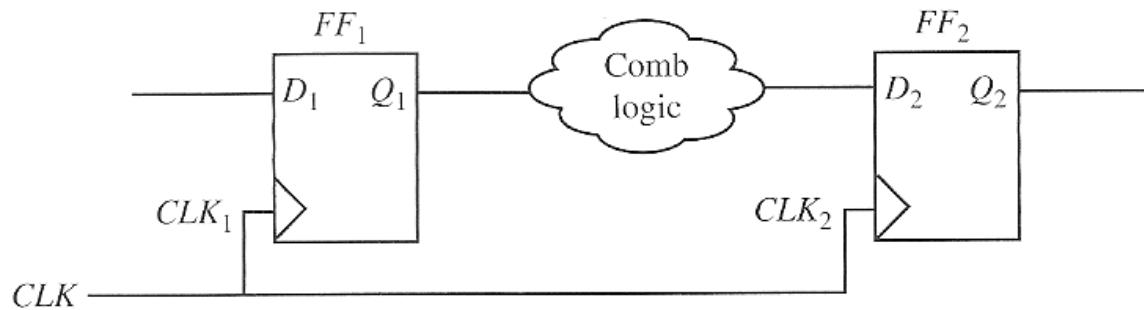
- For a real D-FF
 - D input must be stable for a certain amount of time before the active edge of clock cycle => *Setup time*
 - D input must be stable for a certain amount of time after the active edge of the clock => *Hold time*
- Propagation time: from the time the clock changes to the time the output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



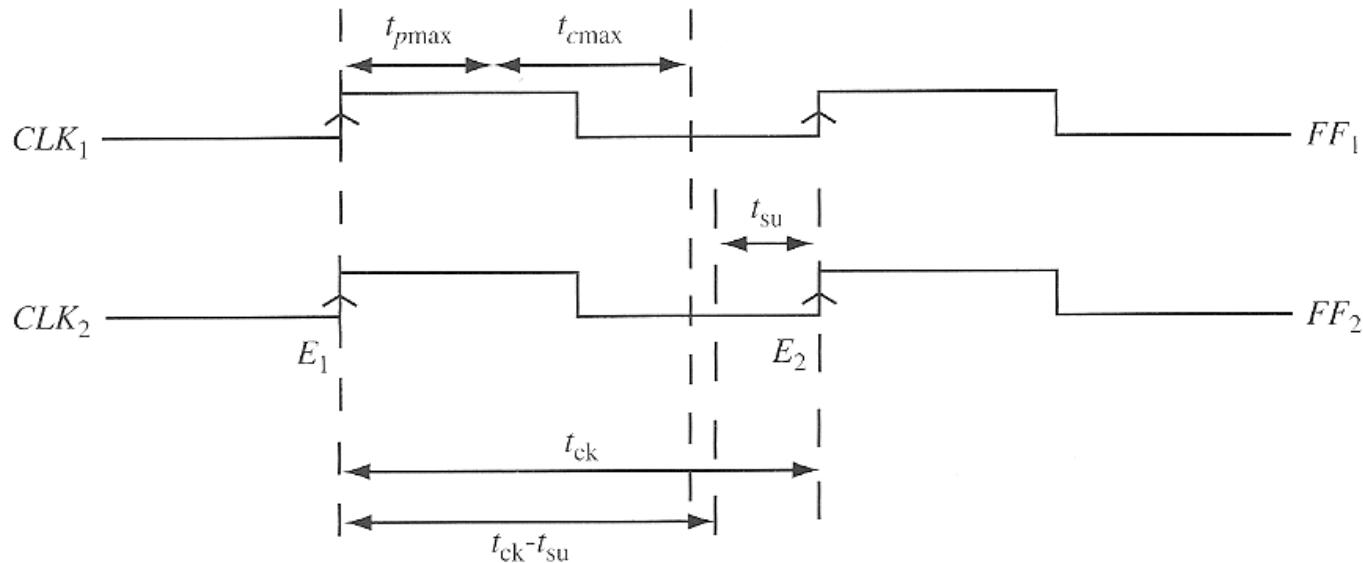
Manufacturers provide minimum t_{su} , t_h , and maximum t_{plh} , t_{phl}

FF to FF Path via Combinational Logic

(when setup time is met)



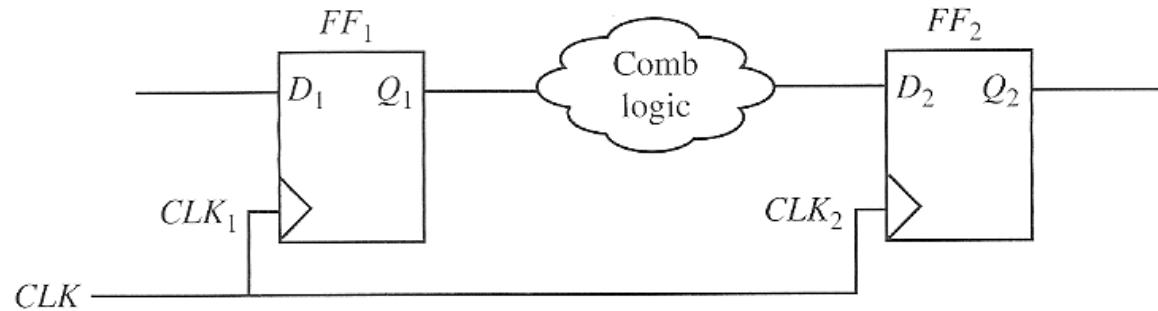
(a) Circuit



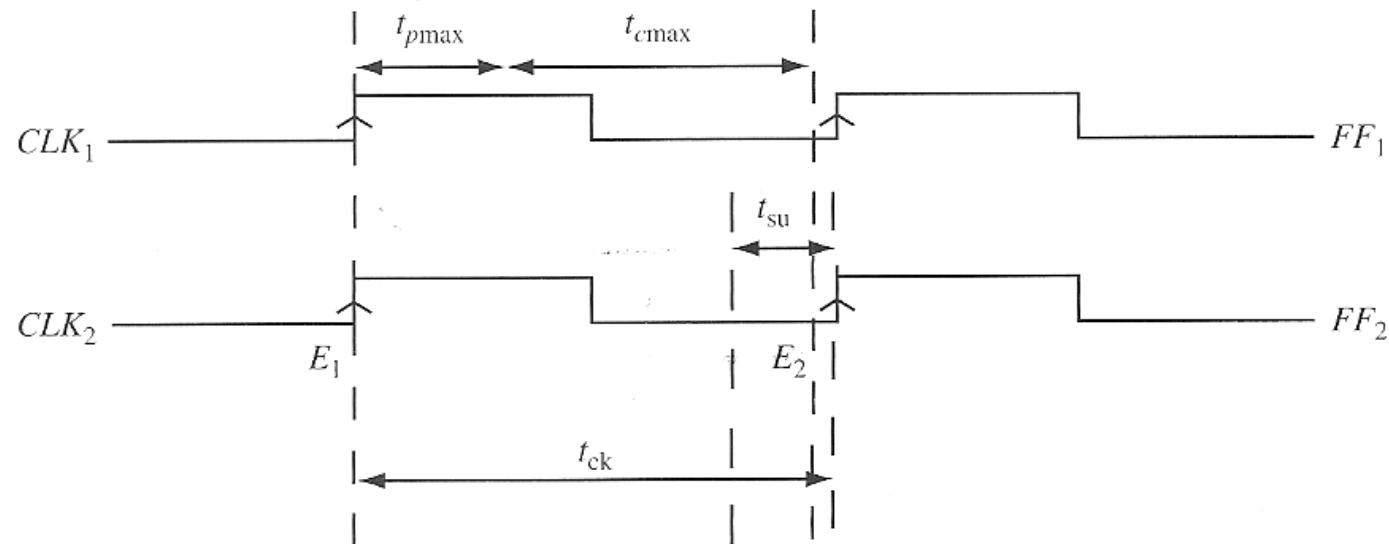
(b) Timing diagram when setup time is met

FF to FF Path via Combinational Logic

(when setup time is not met)



(a) Circuit



(c) Timing diagram when setup time is violated

Maximum Clock Frequency

$t_{c\max}$ -Max propagation delay through the combinational network
(path from register outputs back to register inputs – ignoring inputs)

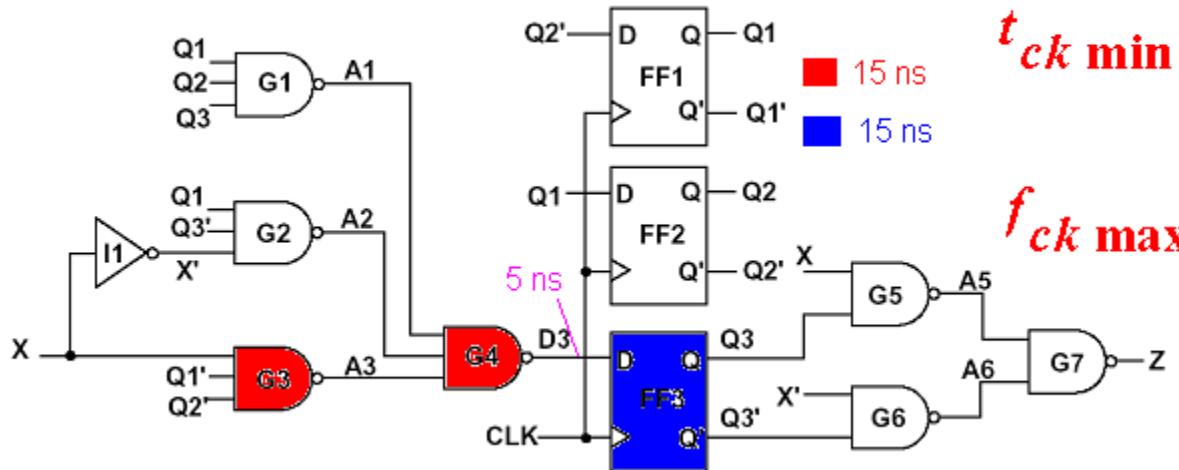
$t_{p\max}$ - Max propagation delay from the time the clock changes to the flip-flop output changes { = max(tplh, tphl)}

t_{ck} - Clock period

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

Example:

$$\begin{aligned} t_{p\max} &= 15 \text{ ns}, t_{su} = 5 \text{ ns}, \\ t_{gate} &= 15 \text{ ns} \end{aligned}$$

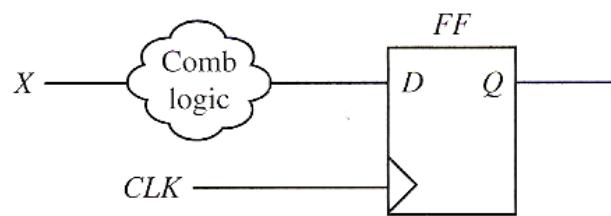


$$\begin{aligned} t_{ck\min} &= 2 \cdot 15\text{ns} + 15\text{ns} + 5\text{ns} \\ &= 50\text{ns} \end{aligned}$$

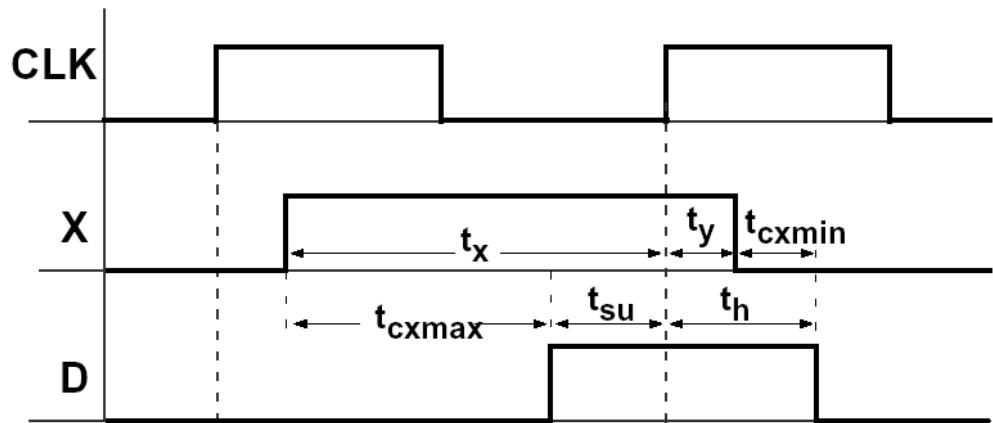
$$\begin{aligned} f_{ck\max} &= \frac{1}{t_{ck\min}} = \frac{1}{50\text{ns}} \\ &= 20\text{Mhz} \end{aligned}$$

Setup Time Violations, Input X point of view

- Occurs if the change in X that is fed forward through the combinational network causes input D to change too late before the clock edge



What about X?



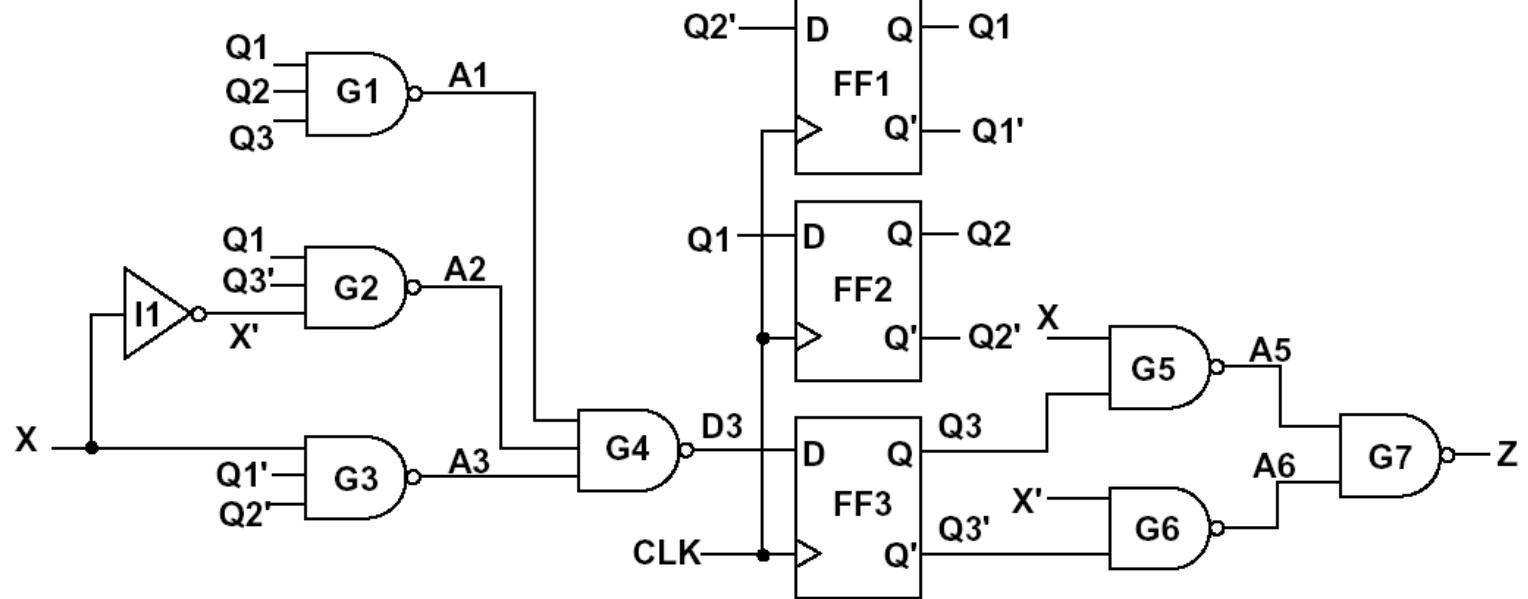
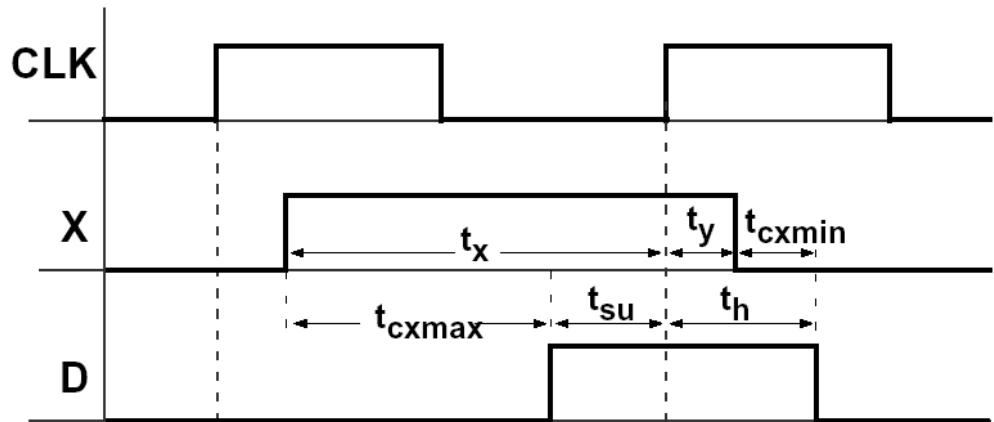
Make sure that input changes propagate to the flip-flops inputs such that setup time is satisfied.

$$t_x \geq t_{cxmax} + t_{su}$$

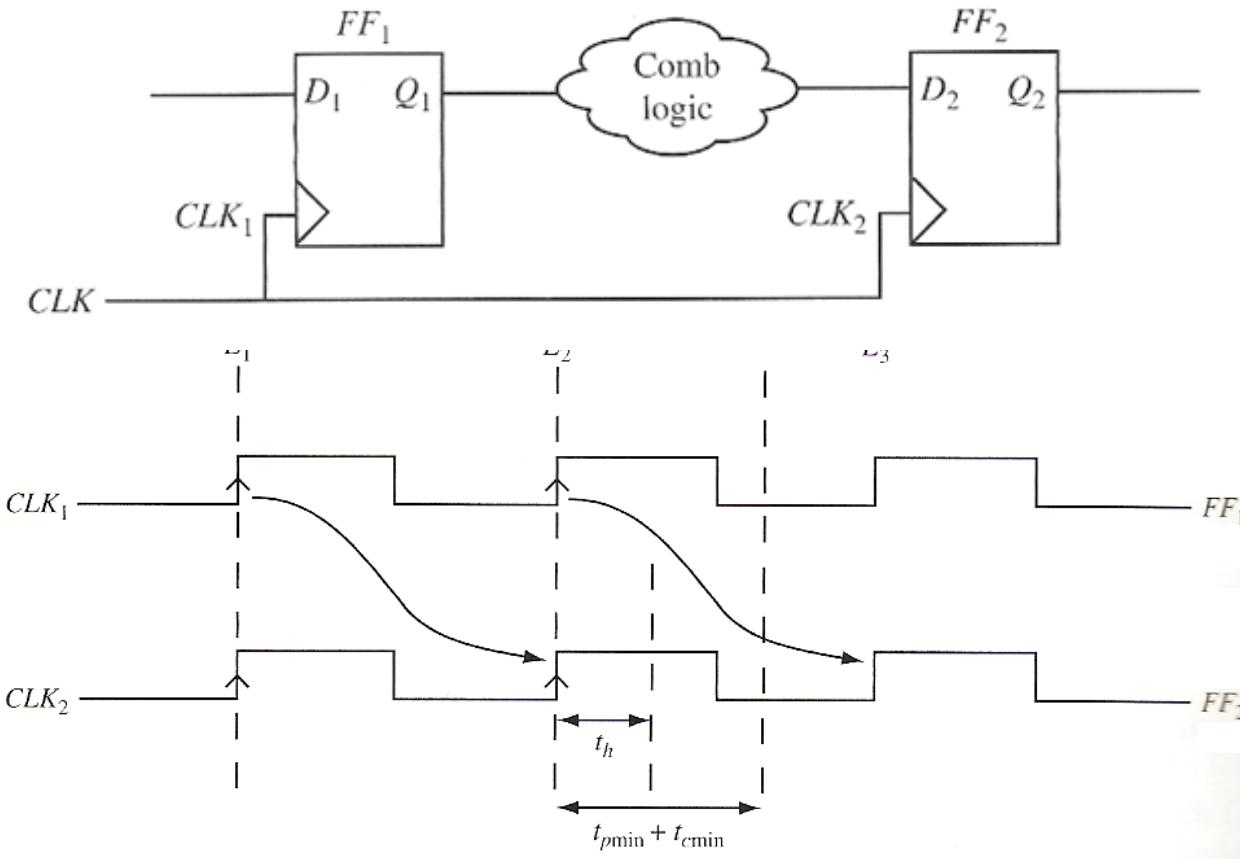
$t_{cx\ max}$ - Max propagation delay through the combinational network from input X (or any other input) to the flip-flop input D

Setup Time Violations, Input X point of view

$$t_x \geq t_{cxmax} + t_{su}$$



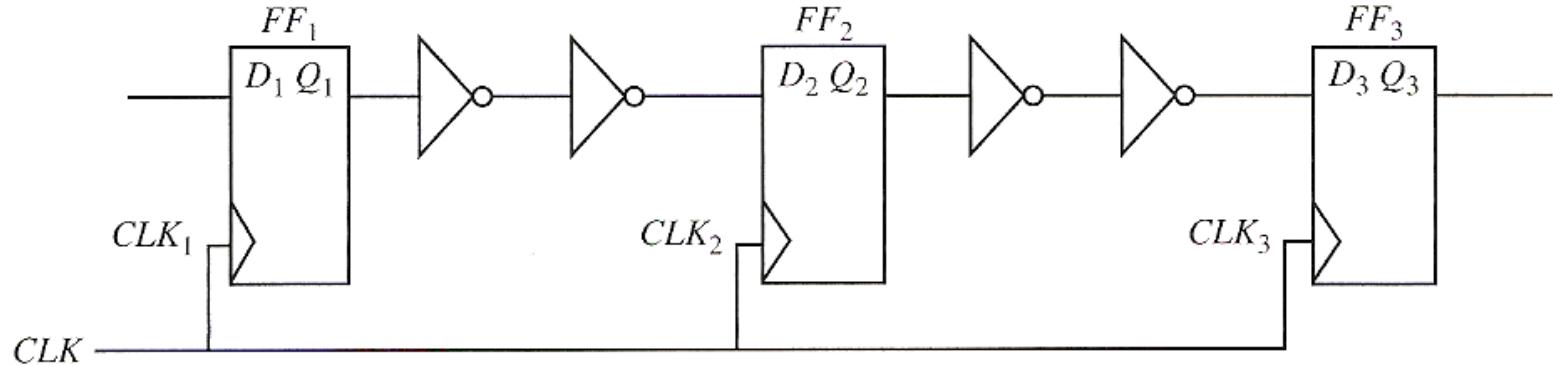
Hold Time Rule for FF



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

Note: if a circuit has a hold-time violation, it cannot be corrected by changing the clock frequency – must be redesigned by adding combinational delays!

Fixing Hold Time Problems in a Shift Register Design using Buffers

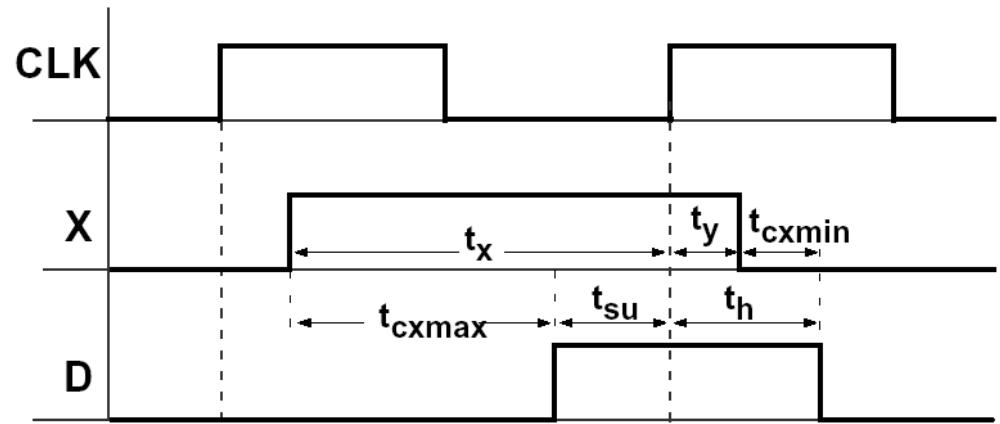


Hold Time Violations (not considering the Input X)

- Occurs if the change in Q that is fed back through the combinational network causes input D to change too soon after the clock edge

Hold time is satisfied if:

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

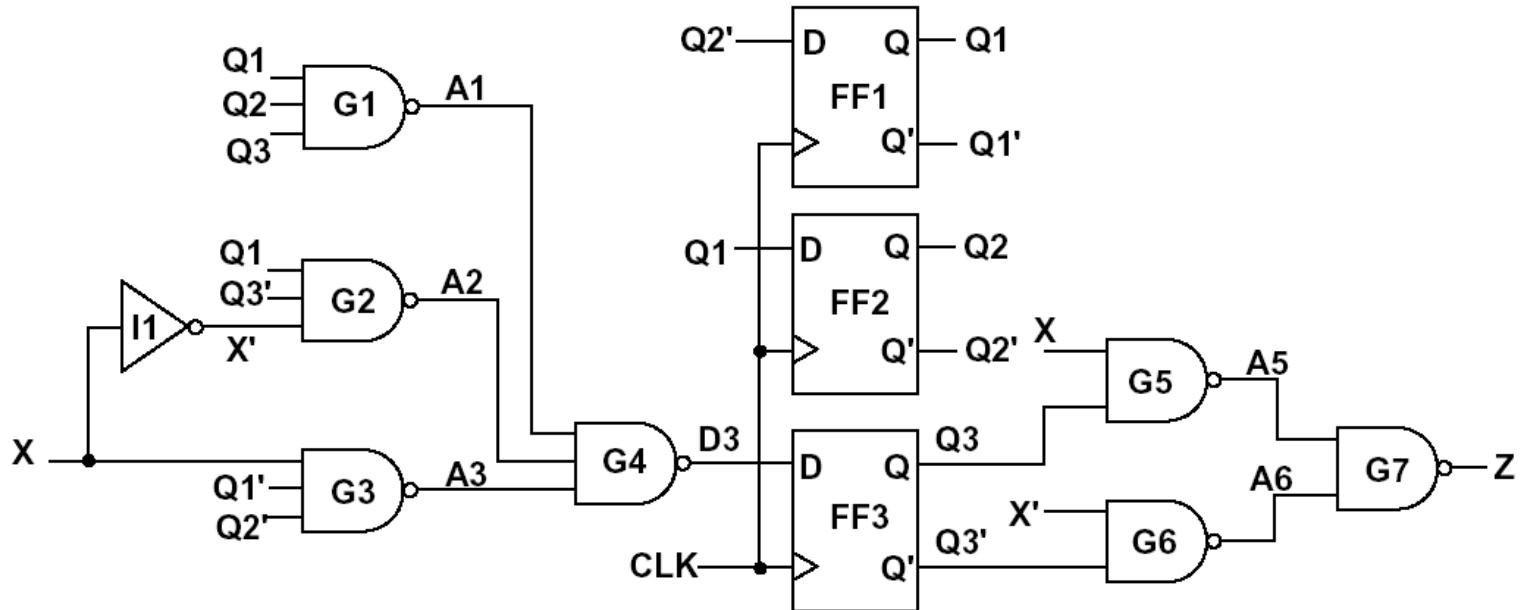
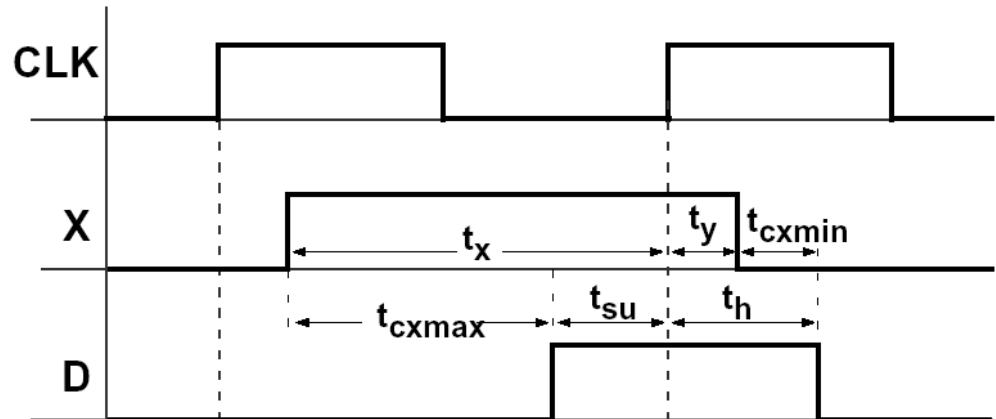


$t_{c\min}$ - Min propagation delay through the combinational network

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

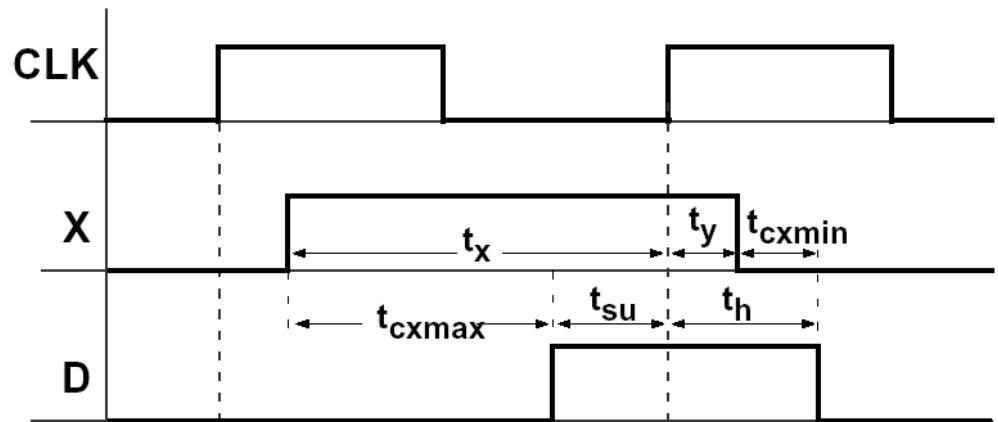
Hold Time Violations (not considering the Input X)

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



Hold Time Violations, Input X point of view

- Occurs if the change in X that is fed back through the combinational network causes input D to change too soon after the clock edge



What about X?

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

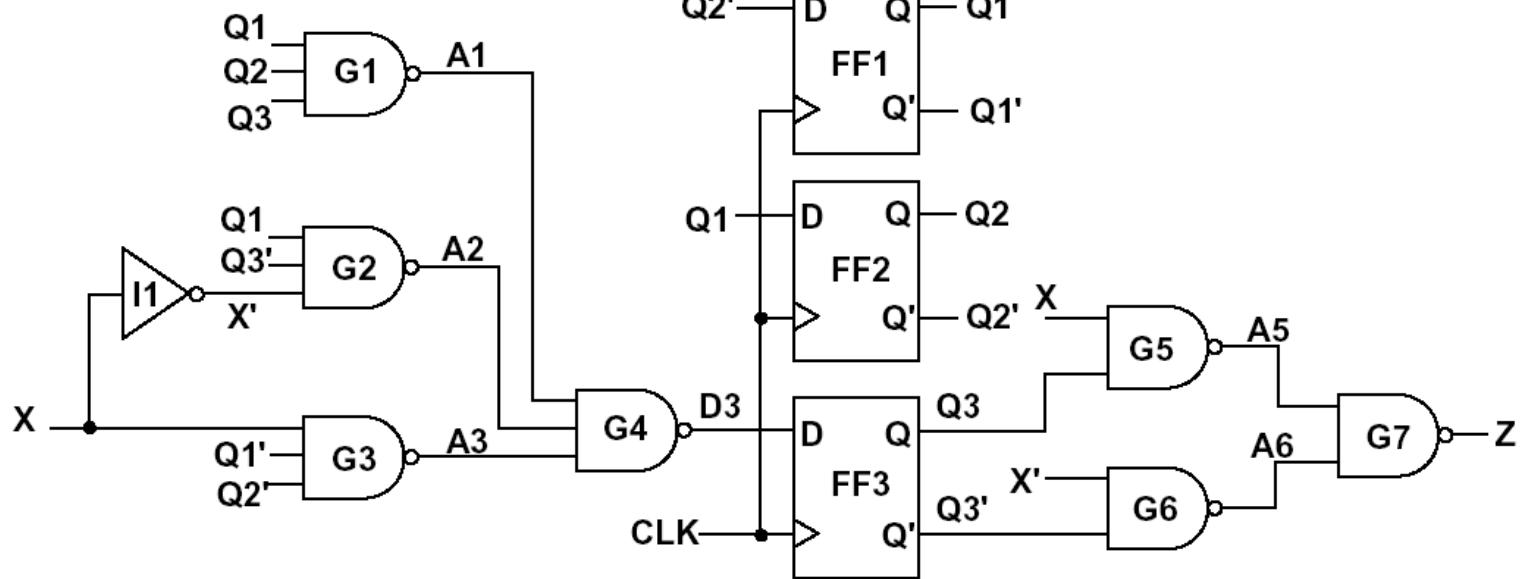
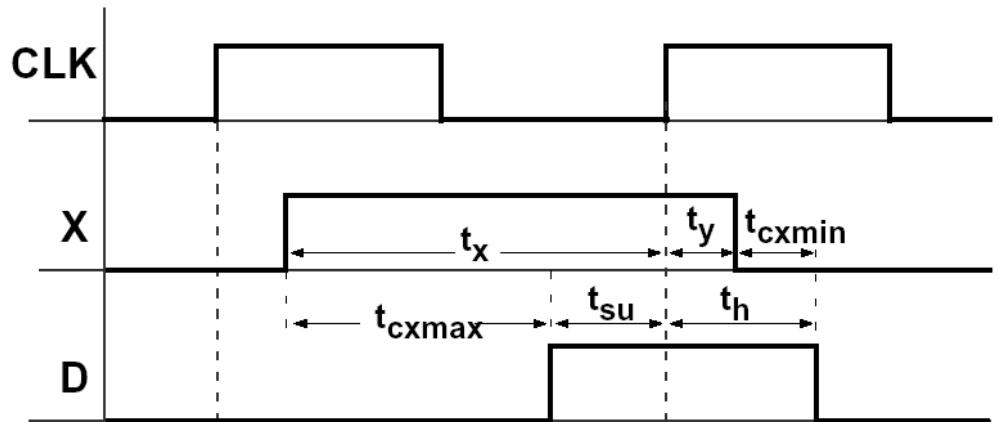
If X changes at time t_y after the active edge, hold time is satisfied if

$$t_y \geq t_h - t_{cxmin}$$

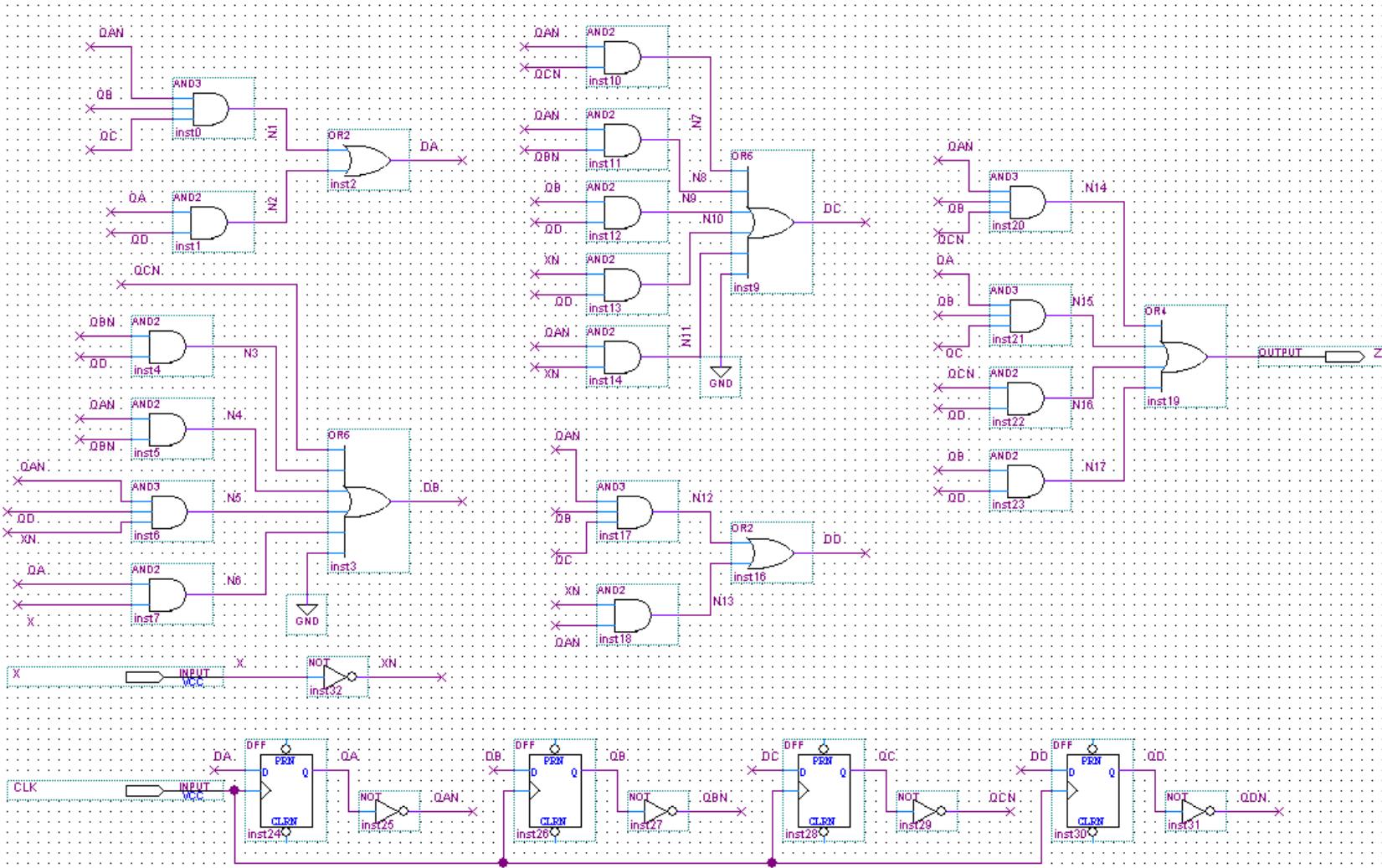
t_{cxmin} - Min propagation delay through the combinational network from input X (or any other input) to the flip-flop input D

Hold Time Violations, Input X point of view

$$t_y \geq t_h - t_{cxmin}$$



Quartus II Moore Implementation of BCD to Excess 3

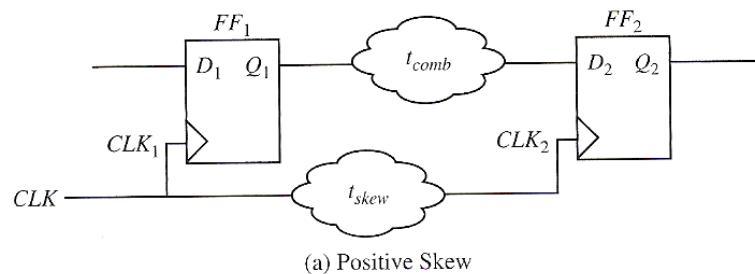


Clock Skew in Synchronous Designs

- **Clock Skew** - absolute time difference in clock signal arrival between two points in the clock network.
 - Often caused by delays in the interconnect within the clock distribution network.
 - Can also be caused by combinational logic used to selectively gate the clock of certain devices
- **Positive Skew** –capturing flip-flop gets the clock delayed with reference to the launching flip-flop
- **Negative Skew** – launching filip-flop get the clock delayed with reference to the capturing flip-flop

Setup & Hold Equations with Clock Skew

Positive Clock Skew



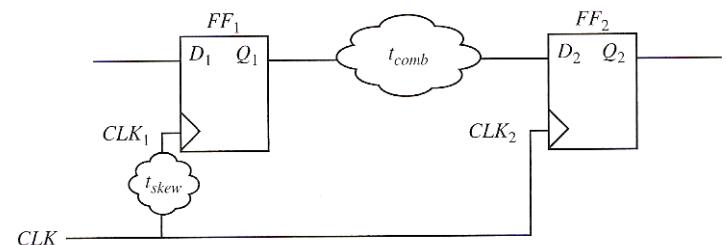
(a) Positive Skew

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

$$t_{p\min} + t_{c\min} \geq t_h + t_{skew}$$

Positive skew is good for setup time,
but it is bad for hold time.

Negative Clock Skew

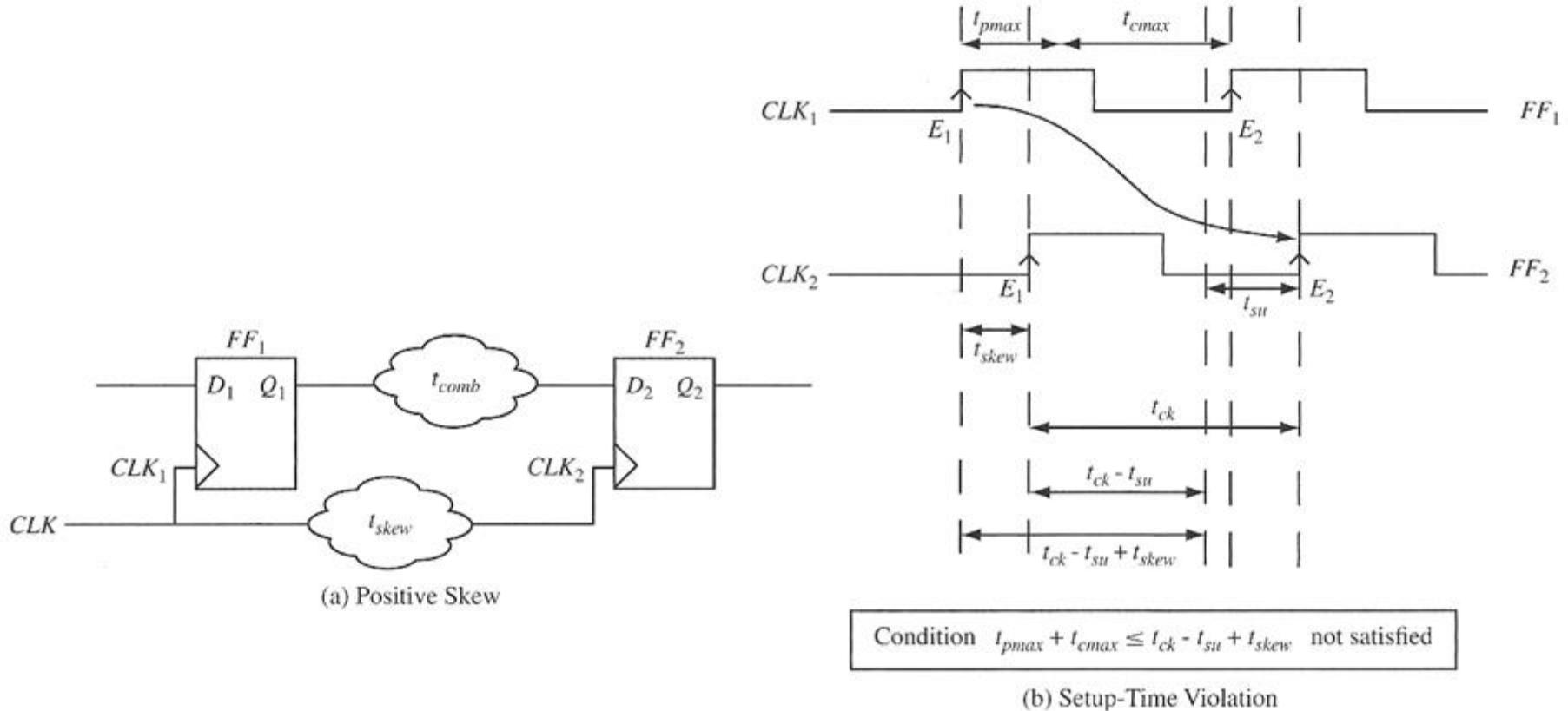


$$t_{ck} \geq t_{p\max} + t_{c\max} + t_{skew} + t_{su}$$

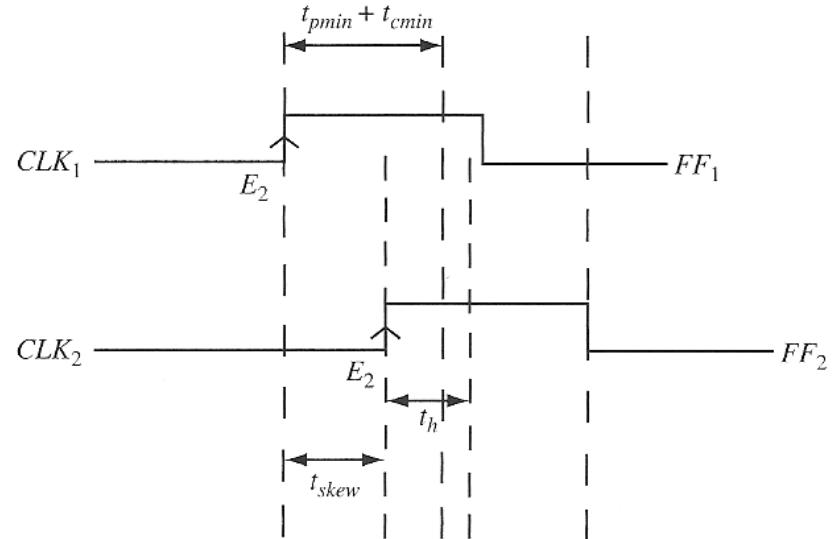
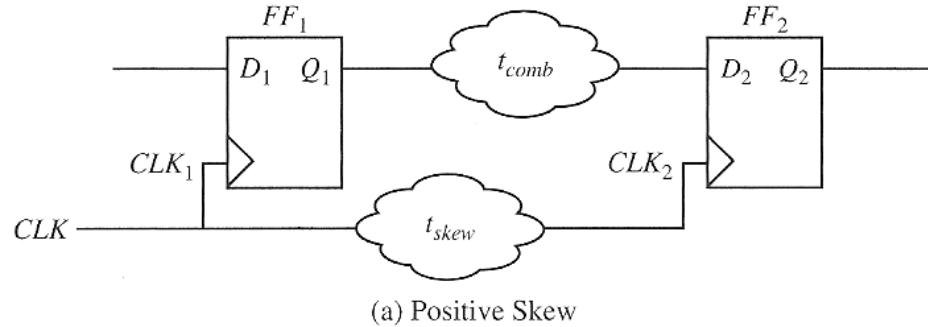
$$t_{p\min} + t_{c\min} \geq t_h - t_{skew}$$

Negative skew is good for hold time,
but it is bad for setup time.

Positive Clock Skew – Setup Time



Positive Clock Skew – Hold Time



(c) Hold-Time Violation

Timing Analysis Example

```
+-----+  
; Timing Analyzer Summary  
+-----+  
; Type ; Slack ; Required Time ; Actual Time ; From ; To ;  
+-----+  
; Worst-case tsu ; N/A ; None ; 11.900 ns ; X ; inst26 ;  
; Worst-case tco ; N/A ; None ; 19.100 ns ; inst24 ; Z ;  
; Worst-case th ; N/A ; None ; -1.700 ns ; X ; inst28 ;  
; Clock Setup: 'CLK' ; N/A ; None ; 93.46 MHz ( period = 10.700 ns ) ; inst28 ; inst26 ;  
; Total number of failed paths ; ; ; ; ; ;  
+-----+  
  
+-----+  
; Timing Analyzer Settings ;  
+-----+  
; Option ; Setting ; From ; To ; Entity Name ;  
+-----+  
; Device Name ; EPF10K70RC240-4 ; ; ; ;  
; Timing Models ; Final ; ; ; ;  
; Default hold multicycle ; Same as Multicycle ; ; ; ;  
; Cut paths between unrelated clock domains ; On ; ; ; ;  
; Cut off read during write signal paths ; On ; ; ; ;  
; Cut off feedback from I/O pins ; On ; ; ; ;  
; Report Combined Fast/Slow Timing ; Off ; ; ; ;  
; Ignore Clock Settings ; Off ; ; ; ;  
; Analyze latches as synchronous elements ; On ; ; ; ;  
; Enable Recovery/Removal analysis ; Off ; ; ; ;  
; Enable Clock Latency ; Off ; ; ; ;  
; Number of source nodes to report per destination node ; 10 ; ; ; ;  
; Number of destination nodes to report ; 10 ; ; ; ;  
; Number of paths to report ; 200 ; ; ; ;  
; Report Minimum Timing Checks ; Off ; ; ; ;  
; Use Fast Timing Models ; Off ; ; ; ;  
; Report IO Paths Separately ; Off ; ; ; ;  
+-----+
```

Setup/Combinational/Hold Time Report

```
+-----+  
; tsu ;  
+-----+-----+-----+-----+-----+  
; Slack ; Required tsu ; Actual tsu ; From ; To ; To Clock ;  
+-----+-----+-----+-----+-----+  
; N/A ; None ; 11.900 ns ; X ; inst26 ; CLK ;  
; N/A ; None ; 10.900 ns ; X ; inst28 ; CLK ;  
; N/A ; None ; 7.700 ns ; X ; inst30 ; CLK ;  
+-----+-----+-----+-----+  
  
+-----+  
; tco ;  
+-----+-----+-----+-----+  
; Slack ; Required tco ; Actual tco ; From ; To ; From Clock ;  
+-----+-----+-----+-----+  
; N/A ; None ; 19.100 ns ; inst26 ; Z ; CLK ;  
; N/A ; None ; 19.100 ns ; inst30 ; Z ; CLK ;  
; N/A ; None ; 19.100 ns ; inst24 ; Z ; CLK ;  
; N/A ; None ; 18.800 ns ; inst28 ; Z ; CLK ;  
+-----+-----+-----+  
  
+-----+  
; th ;  
+-----+-----+-----+-----+  
; Minimum Slack ; Required th ; Actual th ; From ; To ; To Clock ;  
+-----+-----+-----+-----+  
; N/A ; None ; -1.700 ns ; X ; inst28 ; CLK ;  
; N/A ; None ; -2.000 ns ; X ; inst30 ; CLK ;  
; N/A ; None ; -4.600 ns ; X ; inst26 ; CLK ;  
+-----+-----+-----+-----+
```

```

Warning: Found pins functioning as undefined clocks and/or memory enables
Info: Assuming node "CLK" is an undefined clock
Info: Clock "CLK" has Internal fmax of 93.46 MHz between source register "inst24" and destination register "inst26" (period= 10.7 ns)
Info: + Longest register to register delay is 6.700 ns
    Info: 1: + IC(0.000 ns) + CELL(0.000 ns) = 0.000 ns; Loc. = LC6_D30; Fanout = 6; REG Node = 'inst24'
    Info: 2: + IC(0.500 ns) + CELL(2.000 ns) = 2.500 ns; Loc. = LC4_D30; Fanout = 1; COMB Node = 'inst3~74'
    Info: 3: + IC(0.000 ns) + CELL(2.000 ns) = 4.500 ns; Loc. = LC5_D30; Fanout = 1; COMB Node = 'inst3~71'
    Info: 4: + IC(0.500 ns) + CELL(1.700 ns) = 6.700 ns; Loc. = LC1_D30; Fanout = 6; REG Node = 'inst26'
    Info: Total cell delay = 5.700 ns ( 85.07 % )
    Info: Total interconnect delay = 1.000 ns ( 14.93 % )
Info: - Smallest clock skew is 0.000 ns
Info: + Shortest clock path from clock "CLK" to destination register is 7.000 ns
    Info: 1: + IC(0.000 ns) + CELL(2.900 ns) = 2.900 ns; Loc. = PIN_91; Fanout = 4; CLK Node = 'CLK'
    Info: 2: + IC(4.100 ns) + CELL(0.000 ns) = 7.000 ns; Loc. = LC1_D30; Fanout = 6; REG Node = 'inst26'
    Info: Total cell delay = 2.900 ns ( 41.43 % )
    Info: Total interconnect delay = 4.100 ns ( 58.57 % )
Info: - Longest clock path from clock "CLK" to source register is 7.000 ns
    Info: 1: + IC(0.000 ns) + CELL(2.900 ns) = 2.900 ns; Loc. = PIN_91; Fanout = 4; CLK Node = 'CLK'
    Info: 2: + IC(4.100 ns) + CELL(0.000 ns) = 7.000 ns; Loc. = LC6_D30; Fanout = 6; REG Node = 'inst24'
    Info: Total cell delay = 2.900 ns ( 41.43 % )
    Info: Total interconnect delay = 4.100 ns ( 58.57 % )
Info: + Micro clock to output delay of source is 1.400 ns
Info: + Micro setup delay of destination is 2.600 ns
Info: tsu for register "inst26" (data pin = "X", clock pin = "CLK") is 11.900 ns
Info: + Longest pin to register delay is 16.300 ns
    Info: 1: + IC(0.000 ns) + CELL(2.900 ns) = 2.900 ns; Loc. = PIN_212; Fanout = 5; PIN Node = 'X'
    Info: 2: + IC(7.200 ns) + CELL(2.000 ns) = 12.100 ns; Loc. = LC4_D30; Fanout = 1; COMB Node = 'inst3~74'
    Info: 3: + IC(0.000 ns) + CELL(2.000 ns) = 14.100 ns; Loc. = LC5_D30; Fanout = 1; COMB Node = 'inst3~71'
    Info: 4: + IC(0.500 ns) + CELL(1.700 ns) = 16.300 ns; Loc. = LC1_D30; Fanout = 6; REG Node = 'inst26'
    Info: Total cell delay = 8.600 ns ( 52.76 % )
    Info: Total interconnect delay = 7.700 ns ( 47.24 % )
Info: + Micro setup delay of destination is 2.600 ns
Info: - Shortest clock path from clock "CLK" to destination register is 7.000 ns
    Info: 1: + IC(0.000 ns) + CELL(2.900 ns) = 2.900 ns; Loc. = PIN_91; Fanout = 4; CLK Node = 'CLK'
    Info: 2: + IC(4.100 ns) + CELL(0.000 ns) = 7.000 ns; Loc. = LC1_D30; Fanout = 6; REG Node = 'inst26'
    Info: Total cell delay = 2.900 ns ( 41.43 % )
    Info: Total interconnect delay = 4.100 ns ( 58.57 % )

```

```
Info: tco from clock "CLK" to destination pin "Z" through register "inst26" is 19.100 ns
Info: + Longest clock path from clock "CLK" to source register is 7.000 ns
Info: 1: + IC(0.000 ns) + CELL(2.900 ns) = 2.900 ns; Loc. = PIN_91; Fanout = 4; CLK Node = 'CLK'
Info: 2: + IC(4.100 ns) + CELL(0.000 ns) = 7.000 ns; Loc. = LC1_D30; Fanout = 6; REG Node = 'inst26'
Info: Total cell delay = 2.900 ns ( 41.43 % )
Info: Total interconnect delay = 4.100 ns ( 58.57 % )
Info: + Micro clock to output delay of source is 1.400 ns
Info: + Longest register to pin delay is 10.700 ns
Info: 1: + IC(0.000 ns) + CELL(0.000 ns) = 0.000 ns; Loc. = LC1_D30; Fanout = 6; REG Node = 'inst26'
Info: 2: + IC(0.500 ns) + CELL(2.700 ns) = 3.200 ns; Loc. = LC3_D30; Fanout = 1; COMB Node = 'inst19~72'
Info: 3: + IC(2.500 ns) + CELL(5.000 ns) = 10.700 ns; Loc. = PIN_86; Fanout = 0; PIN Node = 'Z'
Info: Total cell delay = 7.700 ns ( 71.96 % )
Info: Total interconnect delay = 3.000 ns ( 28.04 % )
Info: th for register "inst28" (data pin = "X", clock pin = "CLK") is -1.700 ns
Info: + Longest clock path from clock "CLK" to destination register is 7.000 ns
Info: 1: + IC(0.000 ns) + CELL(2.900 ns) = 2.900 ns; Loc. = PIN_91; Fanout = 4; CLK Node = 'CLK'
Info: 2: + IC(4.100 ns) + CELL(0.000 ns) = 7.000 ns; Loc. = LC7_D30; Fanout = 5; REG Node = 'inst28'
Info: Total cell delay = 2.900 ns ( 41.43 % )
Info: Total interconnect delay = 4.100 ns ( 58.57 % )
Info: + Micro hold delay of destination is 3.100 ns
Info: - Shortest pin to register delay is 11.800 ns
Info: 1: + IC(0.000 ns) + CELL(2.900 ns) = 2.900 ns; Loc. = PIN_212; Fanout = 5; PIN Node = 'X'
Info: 2: + IC(7.200 ns) + CELL(1.700 ns) = 11.800 ns; Loc. = LC7_D30; Fanout = 5; REG Node = 'inst28'
Info: Total cell delay = 4.600 ns ( 38.98 % )
Info: Total interconnect delay = 7.200 ns ( 61.02 % )
Info: Quartus II Classic Timing Analyzer was successful. 0 errors, 1 warning
Info: Allocated 100 megabytes of memory during processing
Info: Processing ended: Tue Feb 20 08:57:44 2007
Info: Elapsed time: 00:00:00
```

Metastability Issues

- Metastability in digital systems can occur when the data input to a flip-flop is asynchronous to the clock, which can lead to setup or hold time violations.
 - It can cause flip-flops to switch late or not at all.
 - It can present a brief pulse at a flip-flop output (called a runt pulse) or cause flip-flop output oscillations.
 - This can cause system failures
- Source: Xilinx Application notes: XAPP077 January, 1997

Metastability Issues

- To ensure reliable operation, the input to a register must be stable for a minimum time before the clock edge (register setup time or t_{su}) and for a minimum time after the clock edge (register hold time or t_H).
- The register output then is available after a specified clock-to-output delay (t_{co}).
- If a data signal transition violates a register's t_{su} or t_H requirements, the output of the register may go into a metastable state.
- In a metastable state, the register output hovers at a value between the high and low states for some arbitrary period of time
- This means the output transition to a defined high or low state is delayed beyond the specified t_{co} .
- Source: Altera White paper, "Understanding Metastability FPGAs"
<http://www.altera.com/literature/wp/wp-01082-quartus-ii-metastability.pdf>, 2009

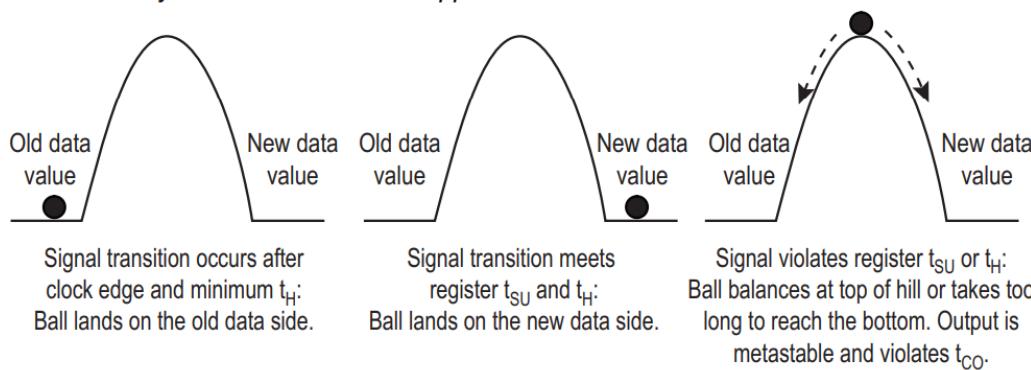
Metastability Issues

- Not every signal transition that violates a register's t_{SU} or t_H results in a metastable output.
- More often than not, these timing violations result in nondeterministic but stable output from the register.
- The likelihood that a register enters a metastable state and the time required to return to a stable state vary depending on the process technology used to manufacture the device and on the operating conditions.

Metastability Issues

- Metastability does not always cause a problem.
 - If the data output signal resolves to a valid state before the next register captures the data, then the metastable signal does not negatively impact the system operation.
 - But if the metastable signal does not resolve to a low or high state before it reaches the next design register, it can cause the system to fail.

Figure 1. Metastability Illustrated as a Ball Dropped on a Hill



Source: Intel (Altera) WP 01082

Metastability Issues

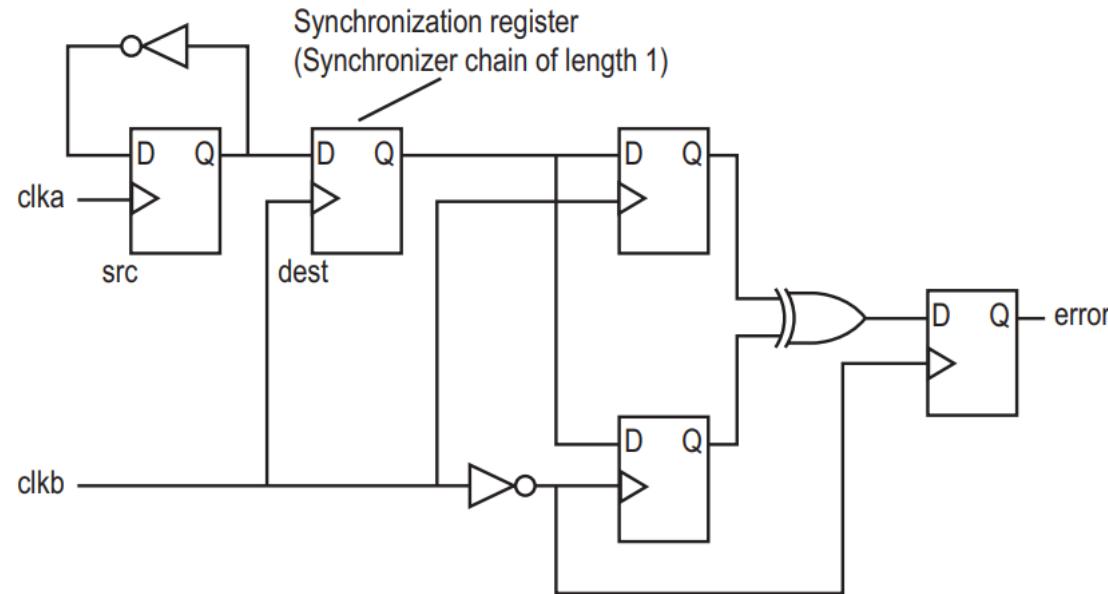
- In properly designed synchronous systems, the input signals must always meet the register timing requirements, so metastability does not occur.
- Metastability can occur when
 - When the input signal is an asynchronous signal.
 - When the clock skew is too large (rise and fall time are more than the tolerable values).
 - When interfacing two domains operating at two different frequencies or at the same frequency but with different phase.
 - When the combinational delay is such that flip-flop data input changes in the critical window (setup or hold time window)

Source: ASIC World, “What is Metastability”

Metastability Issues

- In safety critical applications it may be necessary to quantify the risks involved.
 - Mean Time Between Failures is the accepted metric

Figure 4. Test Circuit Structure for Metastability Characterization



Source: Intel (Altera) WP 01082

Determining if Metastability is a problem

MTBF Calculation

The metastability MTBF for a specific signal transfer, or all the transfers in a design, can be calculated using information about the design and the device characteristics. The MTBF of a synchronizer chain is calculated with the following formula and parameters:

$$MTBF = \frac{e^{t_{MET}/C_2}}{C_1 \cdot f_{CLK} \cdot f_{DATA}}$$

The C_1 and C_2 constants depend on the device process and operating conditions.

The f_{CLK} and f_{DATA} parameters depend on the design specifications: f_{CLK} is the clock frequency of the clock domain receiving the asynchronous signal and f_{DATA} is the toggling frequency of the asynchronous input data signal. Faster clock frequencies and faster-toggling data reduce (or worsen) the MTBF.

The t_{MET} parameter is the available metastability settling time, or the timing slack available beyond the register's t_{CO} , for a potentially metastable signal to resolve to a known value. The t_{MET} for a synchronization chain is the sum of the output timing slacks for each register in the chain.

The overall design MTBF can be determined by the MTBF of each synchronizer chain in the design. The failure rate for a synchronizer is $1/MTBF$, and the failure rate for the entire design is calculated by adding the failure rates for each synchronizer chain, as follows:

$$\text{failure_rate}_{\text{design}} = \frac{1}{MTBF_{\text{design}}} = \sum_{i=1}^{\text{number of chains}} \frac{1}{MTBF_i}$$

Source: Intel (Altera) WP 01082

Metastability Measurements

The circuit in [Figure 1](#) was implemented in an XC2VP4 device using 0.13 micron, 9-layer-metal technology. Two different implementations put QA, the flip-flop under test, into a CLB and into an IOB.

$$\text{MTBF} = \frac{e^{K2 * \tau}}{F1 * F2 * K1}$$

Table 1: Virtex-II Pro Metastability Measurements and Calculations (1997 data for XC4005E added for comparison)

Device	XC2VP4			XC2VP4			XC4005E -3
V _{CC} (V)	1.5	1.35	1.65	1.5	1.35	1.65	5.0
Flip-Flop Type	CLB	CLB	CLB	IOB	IOB	IOB	CLB
Low Frequency (MHz)	300	310	390	310	300	420	109
Half Period (ps)	1667	1613	1283	1613	1667	1190	4587
MTBF1 (ms)	60,000	20,000	60,000	30,000	30,000	30,000	1,000
High Frequency (MHz)	390	420	490	420	430	500	124.4
Half Period (ps)	1282	1190	1020	1190	1163	1000	4019
MTBF2 (ms)	1.69	1.046	5.16	0.987	1.84	6.96	0.016
Half Period Difference (ps)	385	423	262	423	504	190	568
Ln (MTBF1 / MTBF2)	10.478	9.86	9.36	10.322	9.70	8.37	11.09
K2 (per ns)	27.2	23.3	35.7	24.4	19.24	44.05	19.52
1 / K2 = tau (ps)	36.8	42.9	28.0	41.0	52.0	22.7	51.2
MTBF multiply / 100 (ps)	15.2	10.3	35.6	11.5	6.85	81.8	7.04

[Table 1](#) lists the experimental results from which the exponential factor K2 was derived. The clock frequency was adjusted manually, while counting errors. Measurements were taken at room temperature, but testing at V_{CC} extremes gives an indication of performance at higher and lower temperature.

Tolerating Metastability

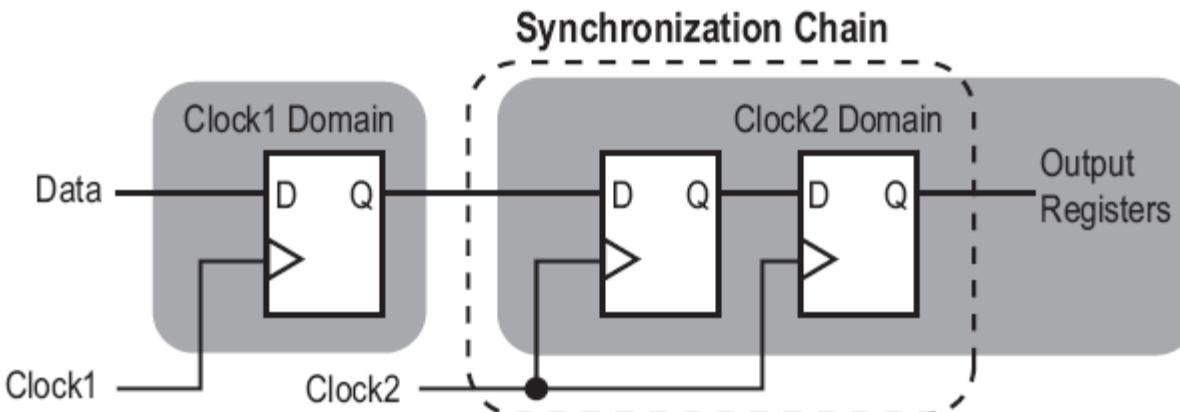
- Method 1: Make sure the clock period is long enough to allow for the resolution of quasi-stable states and for the delay of whatever logic may be in the path to the next flip-flop.
- Method 2: Add one or more successive synchronizing flip-flops to the synchronizer.
 - This approach allows for an entire clock period (except for the setup time of the second flip-flop) for metastable events in the first synchronizing flip-flop to resolve themselves.
 - Disadvantage added latency

Metastability Solutions

- Neither of these approaches can guarantee that metastability cannot pass through the synchronizer; they simply reduce the probability to practical levels.

Cascaded Registers to Tolerate Metastability (from Altera Literature)

Sample Synchronization Register Chain



- The registers in the chain are all clocked by the same or phase-related clocks
- The first register in the chain is driven from an unrelated clock domain, or asynchronously
- Each register fans out to only one register, except the last register in the chain

Source: Intel (Altera) WP 01082

Review: 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

Review: Asynchronous Design

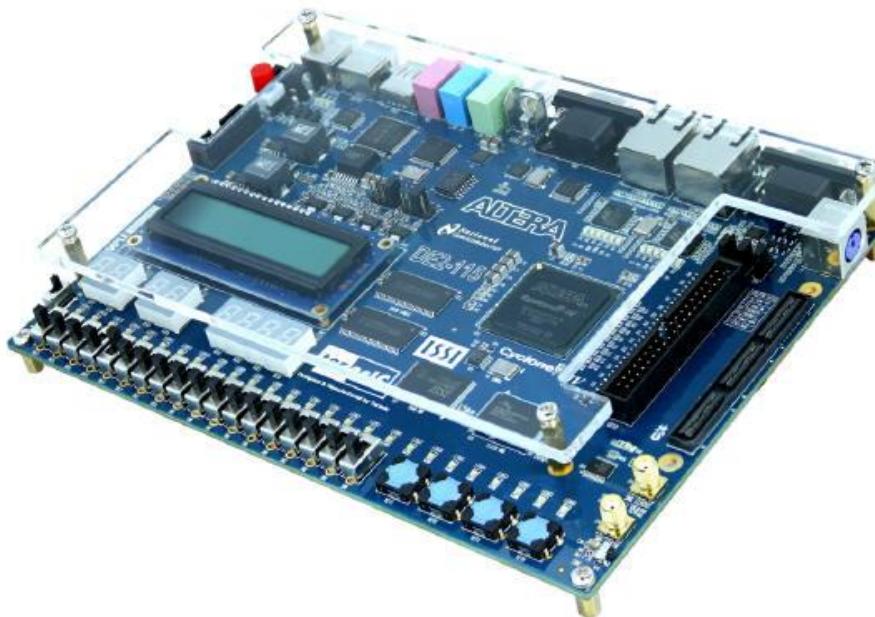
- Disadvantage - More difficult
 - Problems
 - Race conditions: final state depends on the order in which variables change
 - Hazards
 - Special design techniques are needed to cope with races and hazards
- Advantages = Disadvantages of Synchronous Design
 - In high-speed synchronous design propagation 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 – no need for synchronizers
 - Clock cycle is determined by the worst-case delay

CPE 322

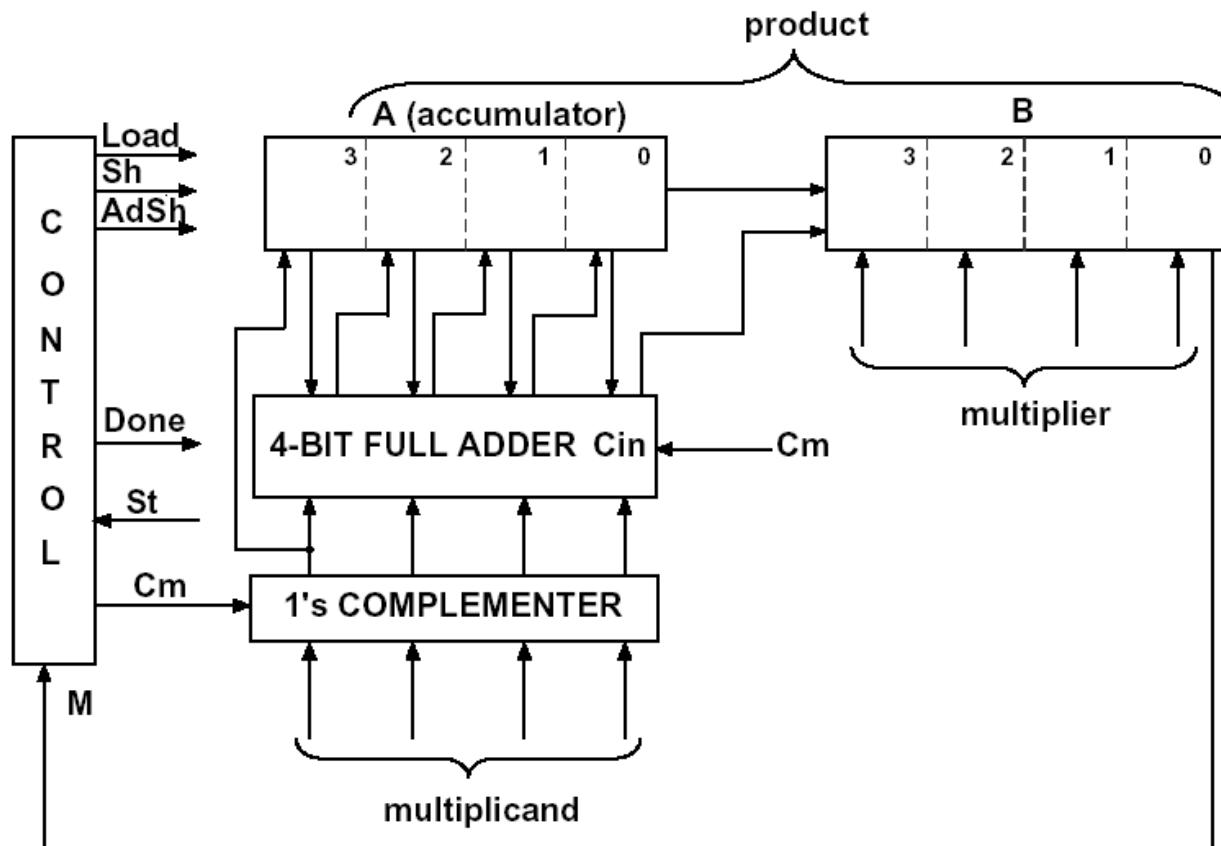
Digital Hardware Design Fundamentals

Electrical and Computer Engineering
UAH

Algorithmic State Machine Notation



Example: Faster Multiplier

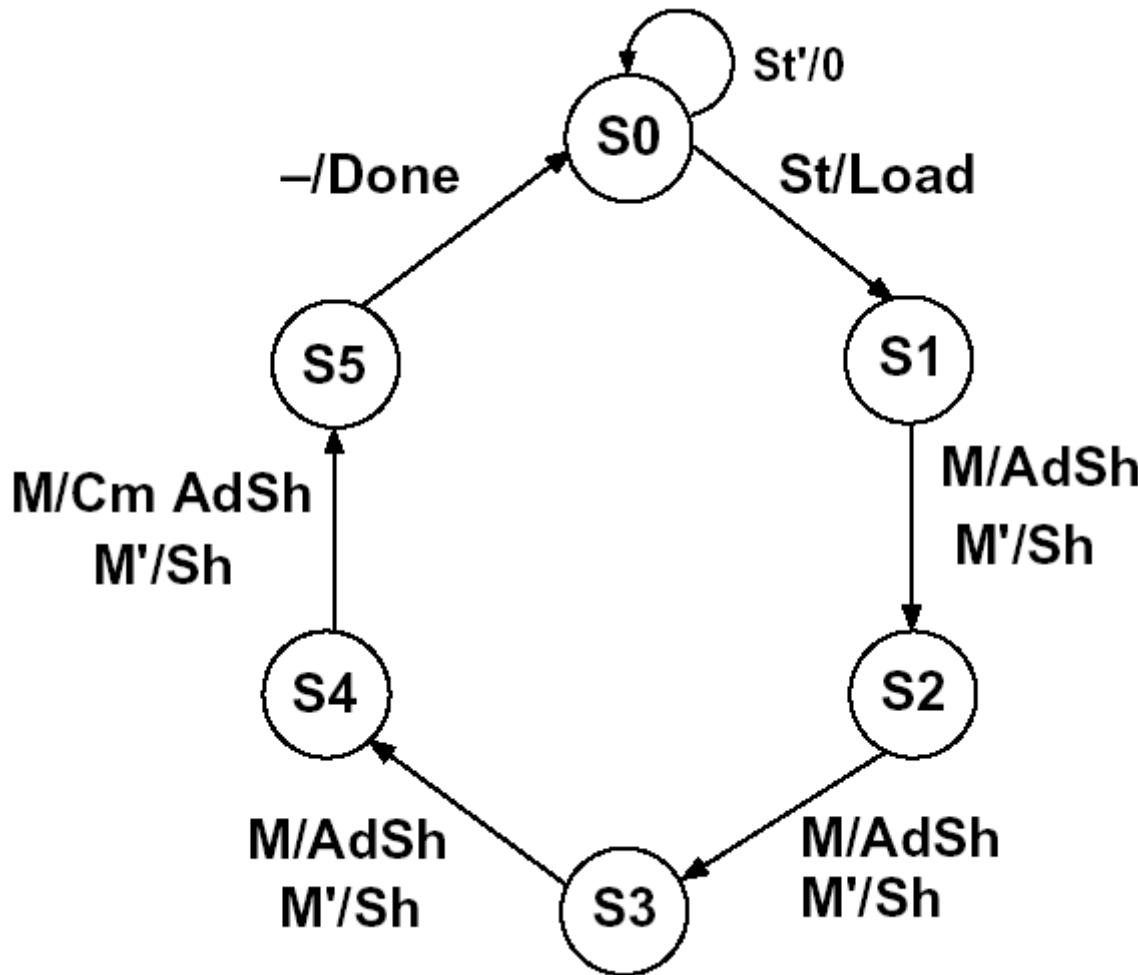


- Move wires from the adder outputs one position to the right => add and shift can occur at the same clock cycle

Extended State Transition Graph

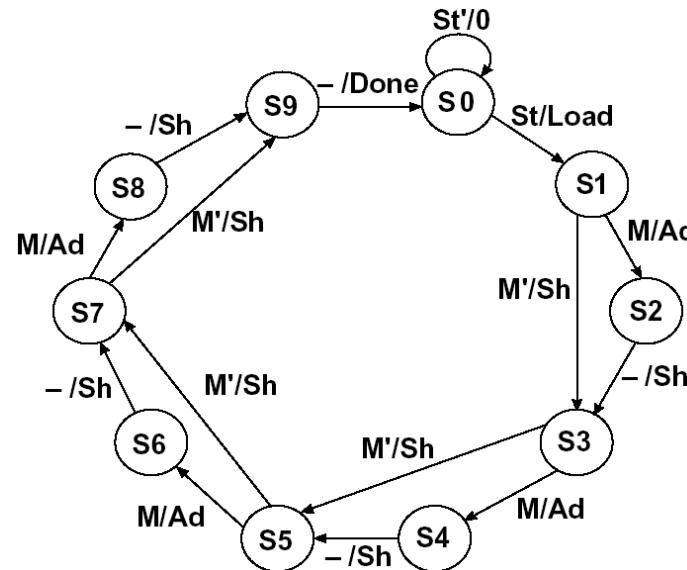
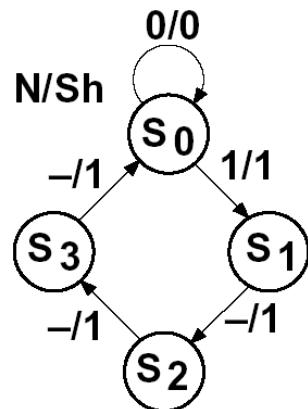
- Similarities to basic State Transition Graph
 - Nodes represent states
 - Arcs (or edges) represent transition between states
 - Labelling on arcs are Inputs/Outputs
- Differences with basic State Transition Graph
 - To reduce Clutter
 - Only Inputs that impact a transition from one state to another are present.
 - Only Outputs that are TRUE for a given transition are listed.

Extended State Transition Graph for Multiplier



Digital Design with ASM Charts

- State Transition Graphs are used to describe state machines controlling a digital system

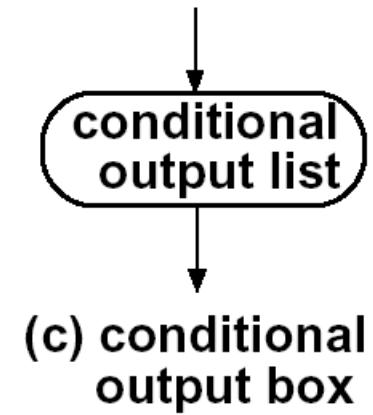
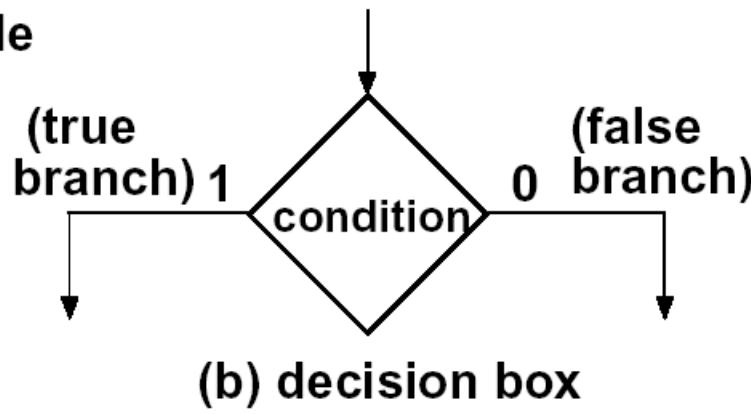
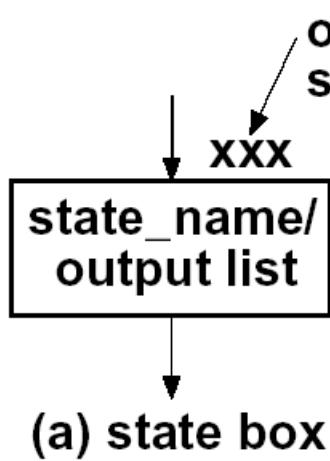


- Alternative: use algorithmic state machine flowchart

State Machine Charts

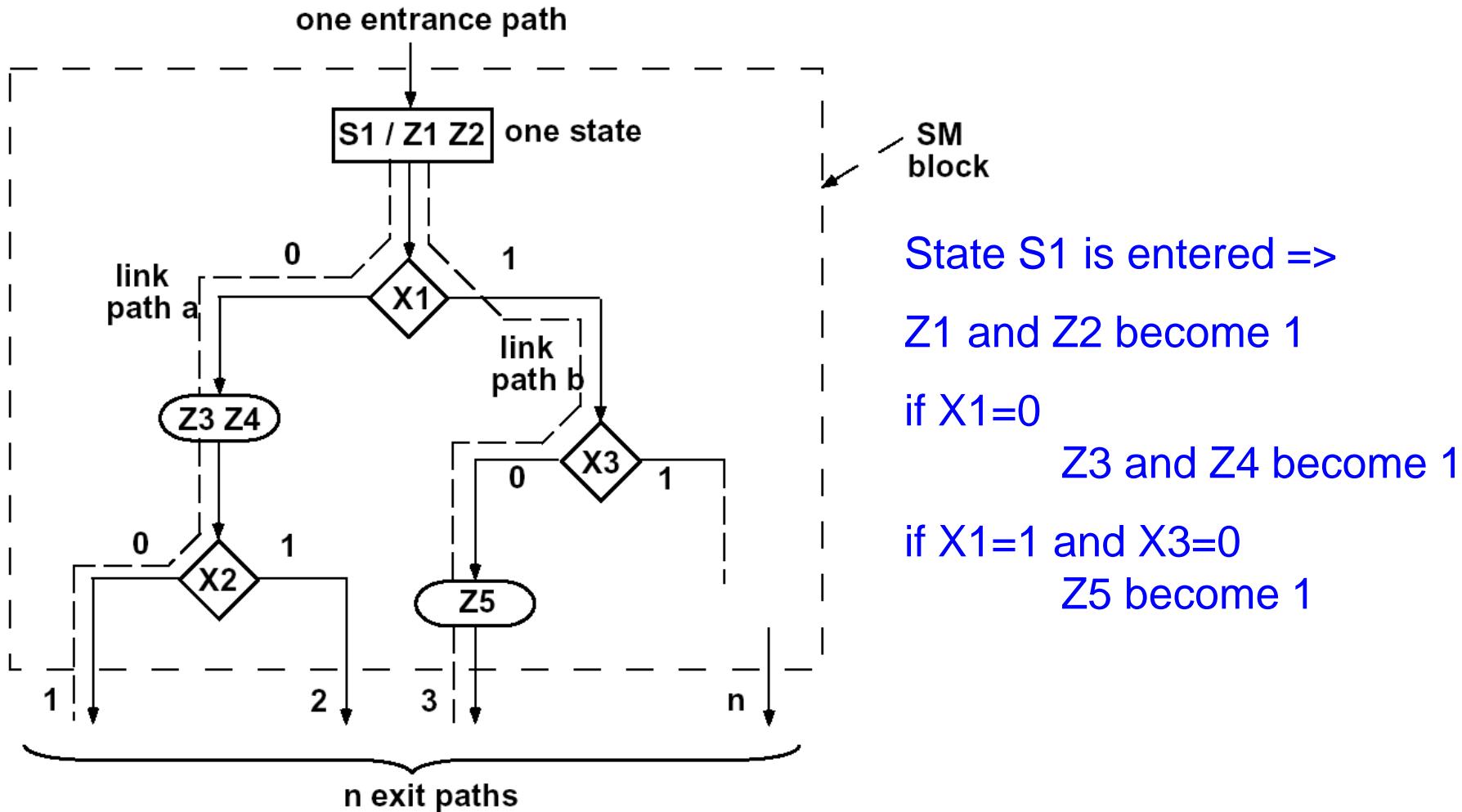
- ASM (*Algorithmic State Machine*) or simply a SM chart
 - Easier to understand the operation of digital system by examining of the ASM chart instead of equivalent state transition graph
 - ASM chart leads directly to hardware realization

Components of ASM charts

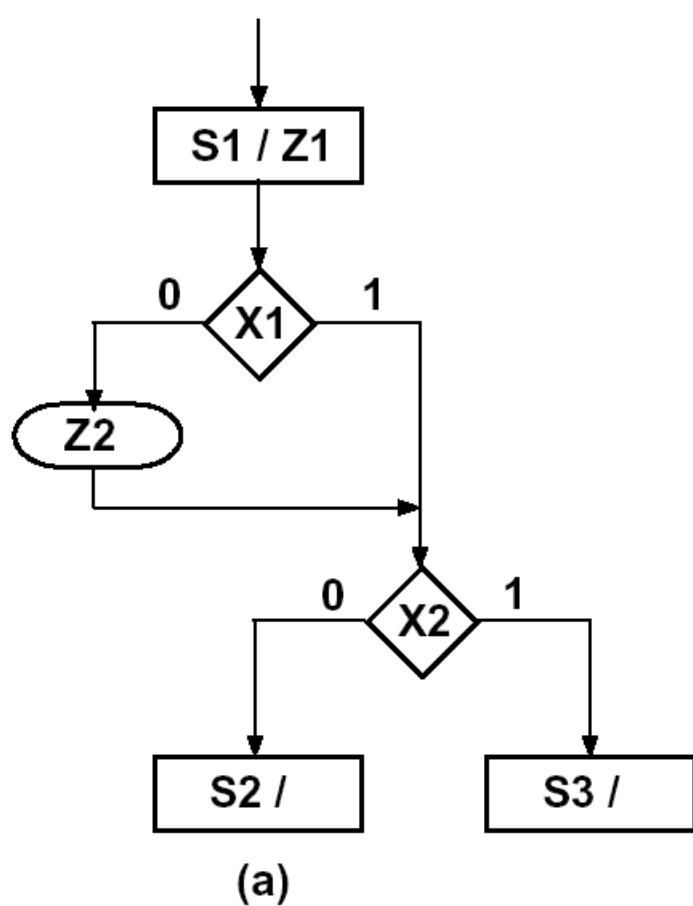


ASM Blocks

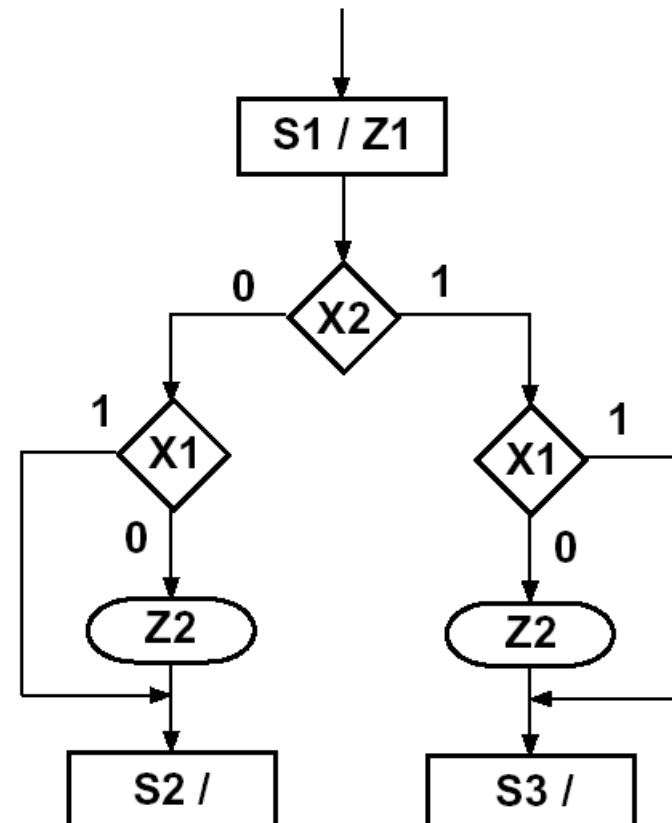
ASM chart is constructed from SM blocks



Equivalent SM Blocks

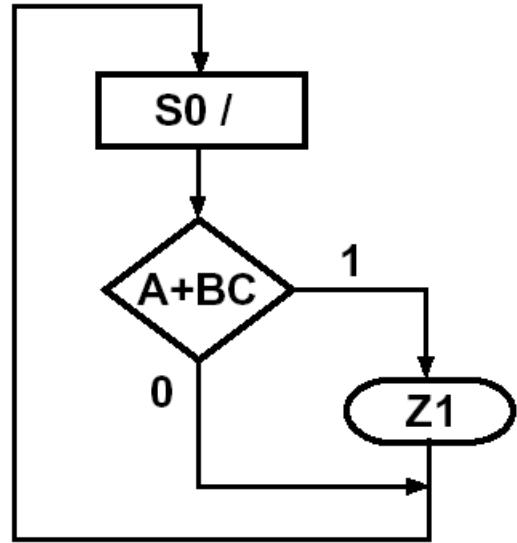


(a)

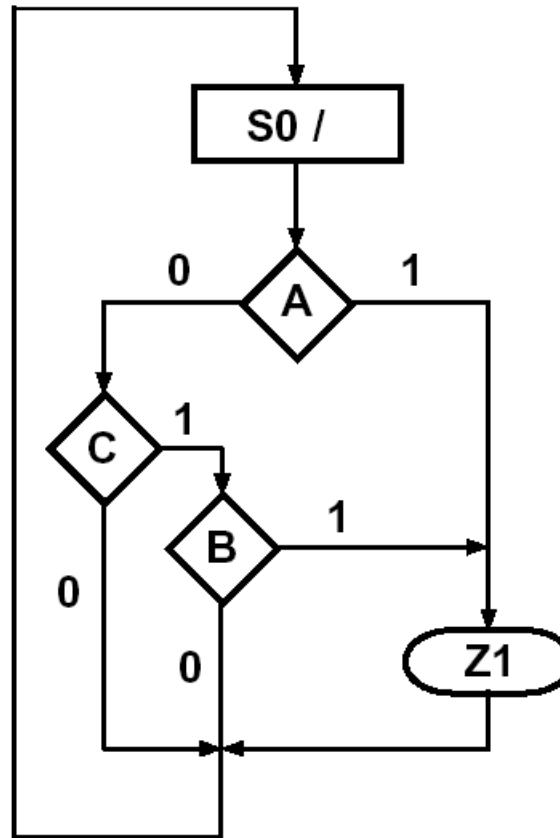


(b)

Equivalent ASM Charts for Comb Networks

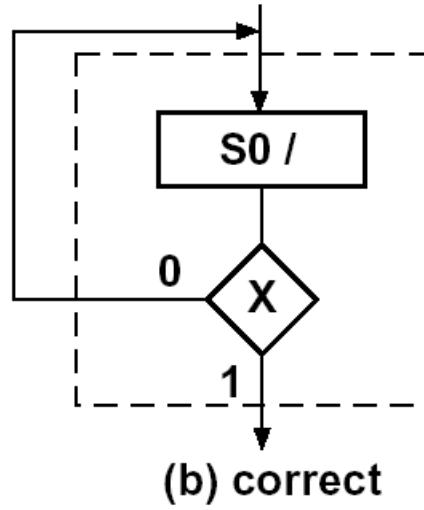
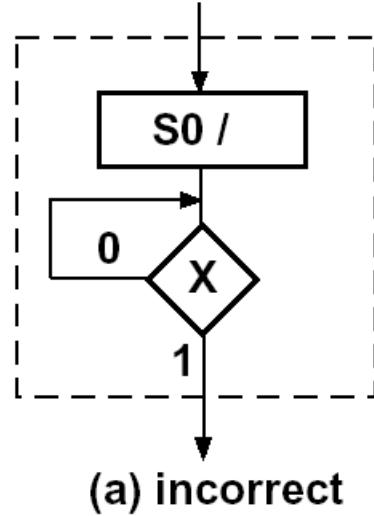


(a)

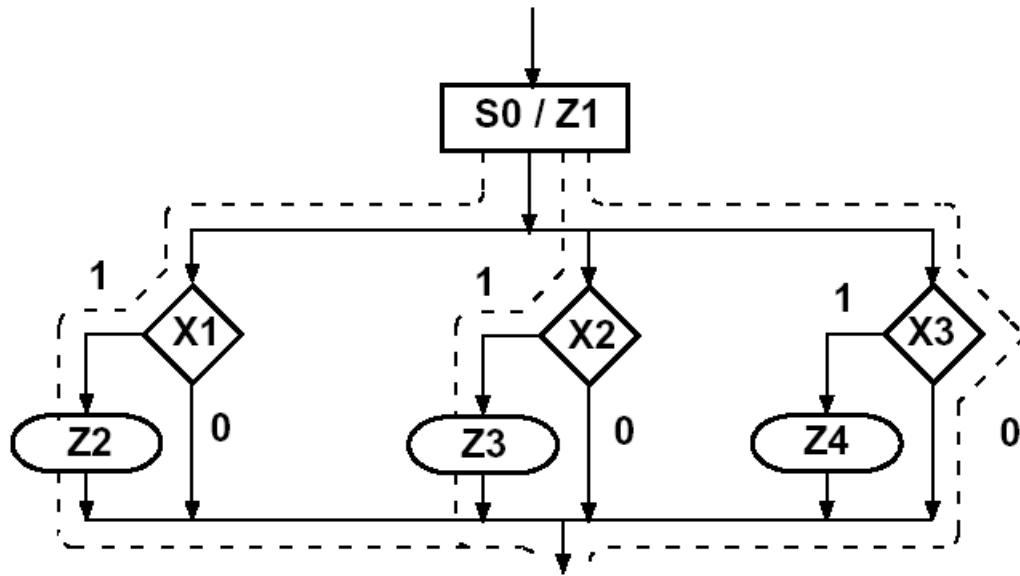


(b)

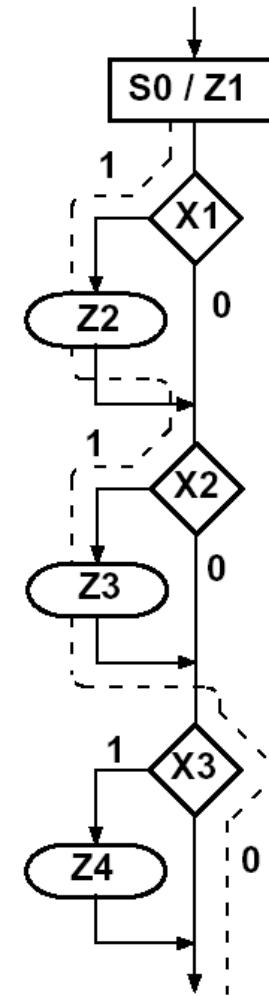
Block with Feedback



Equivalent ASM Blocks

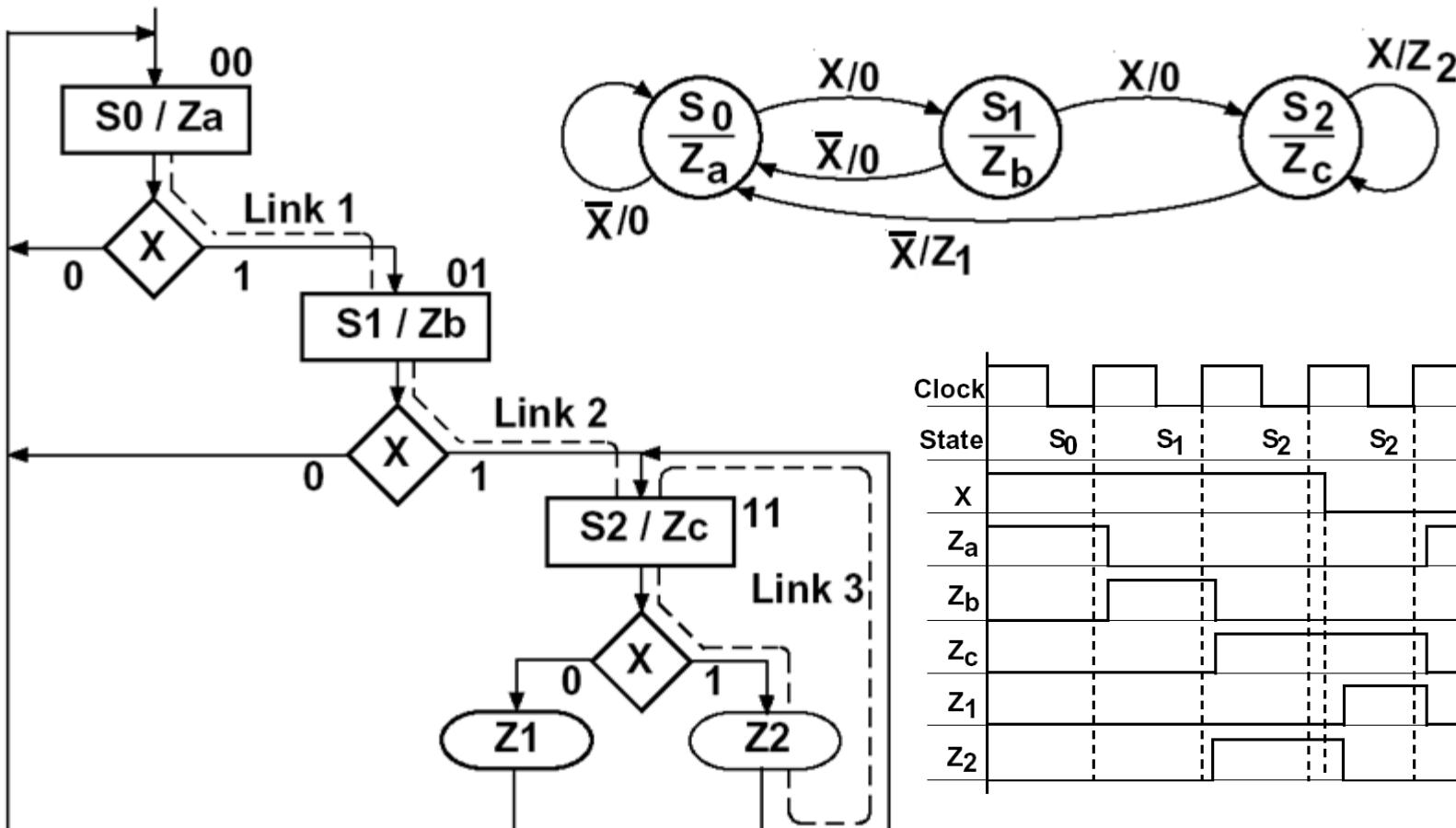


(a) Parallel form

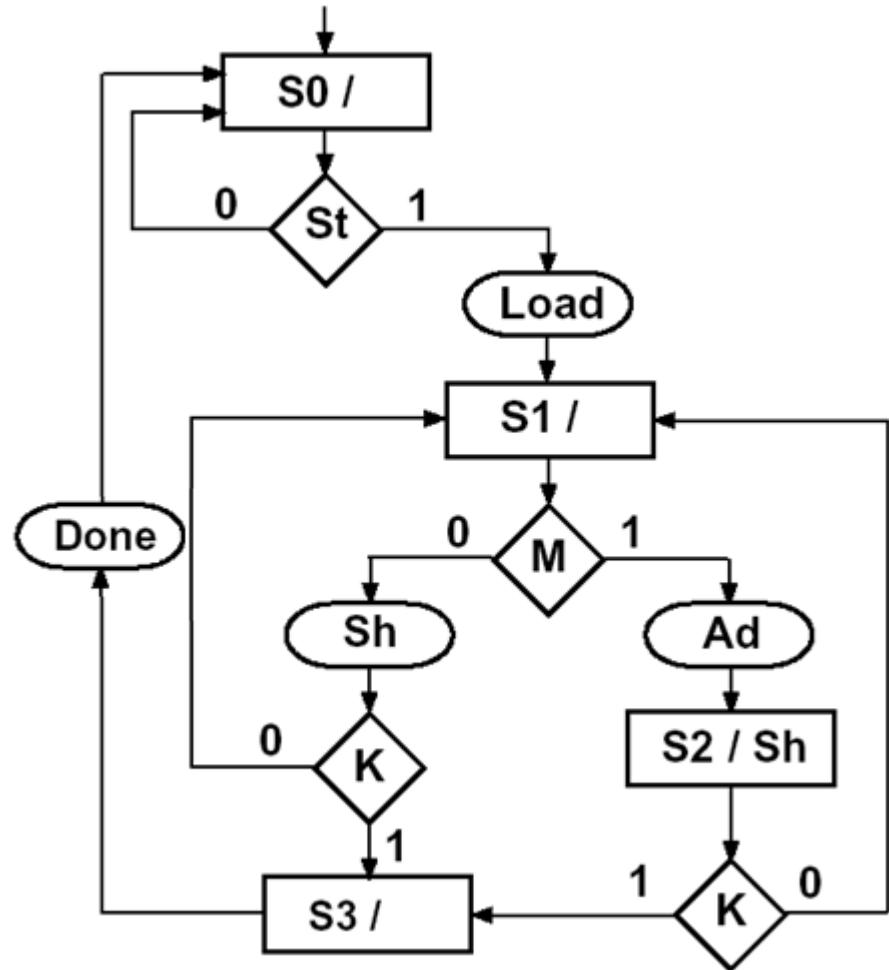
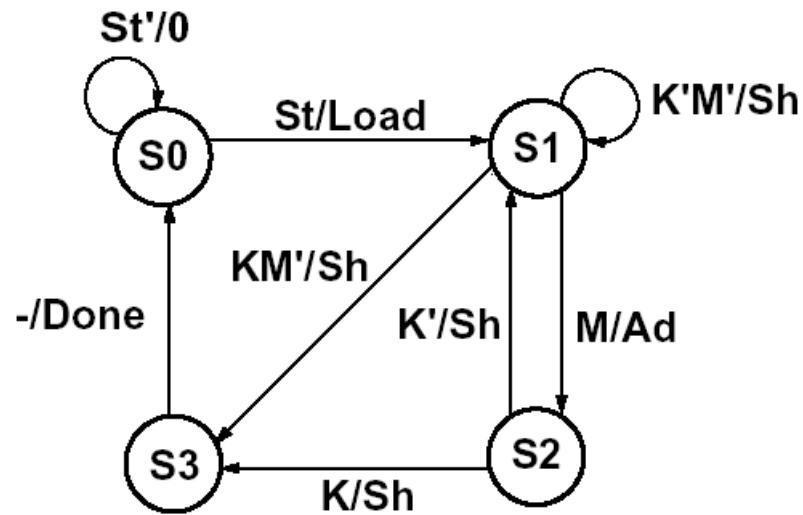


(b) Serial form

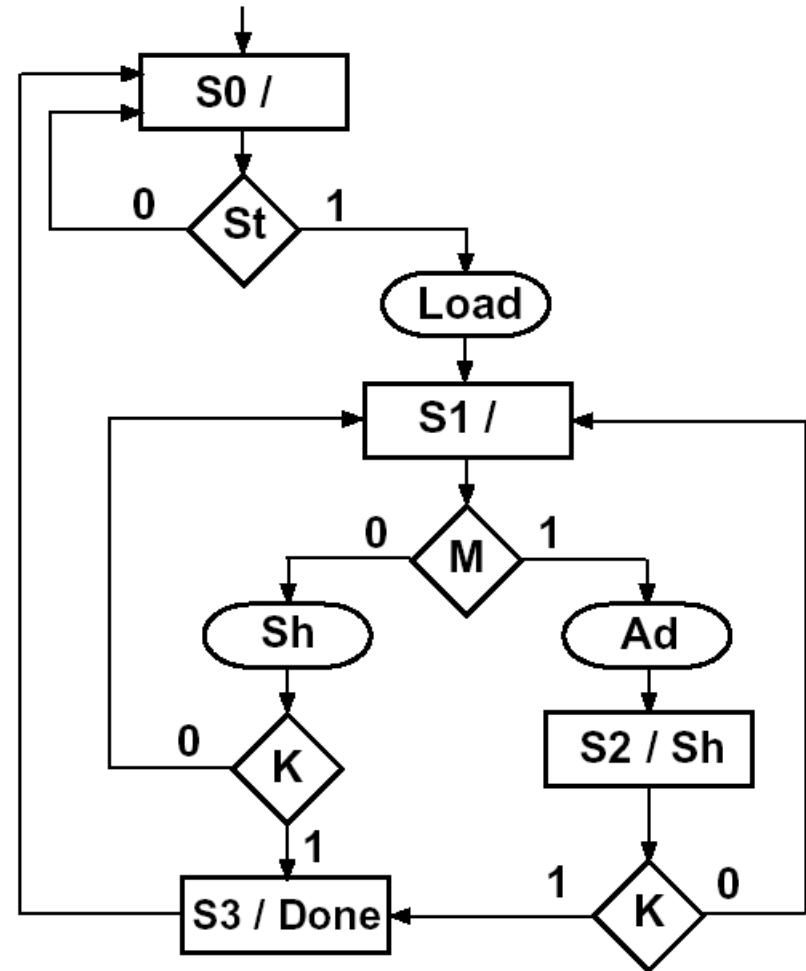
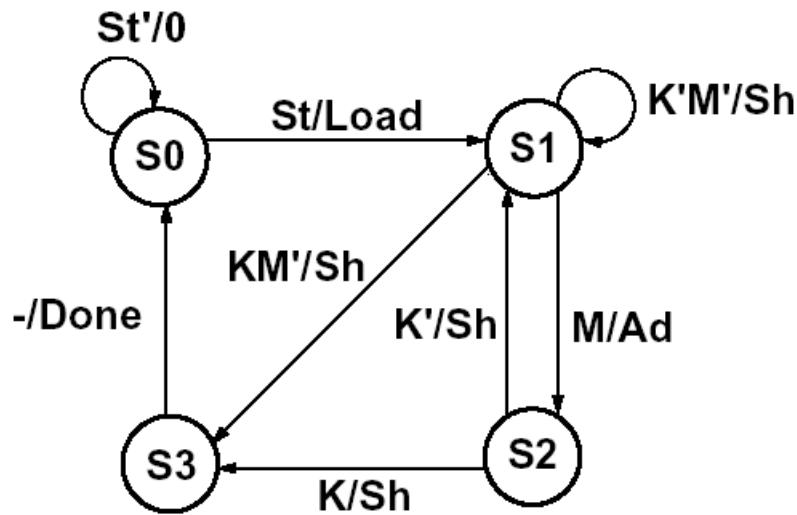
Converting an Extended STG to an ASM Chart



ASM Chart for Binary Multiplier



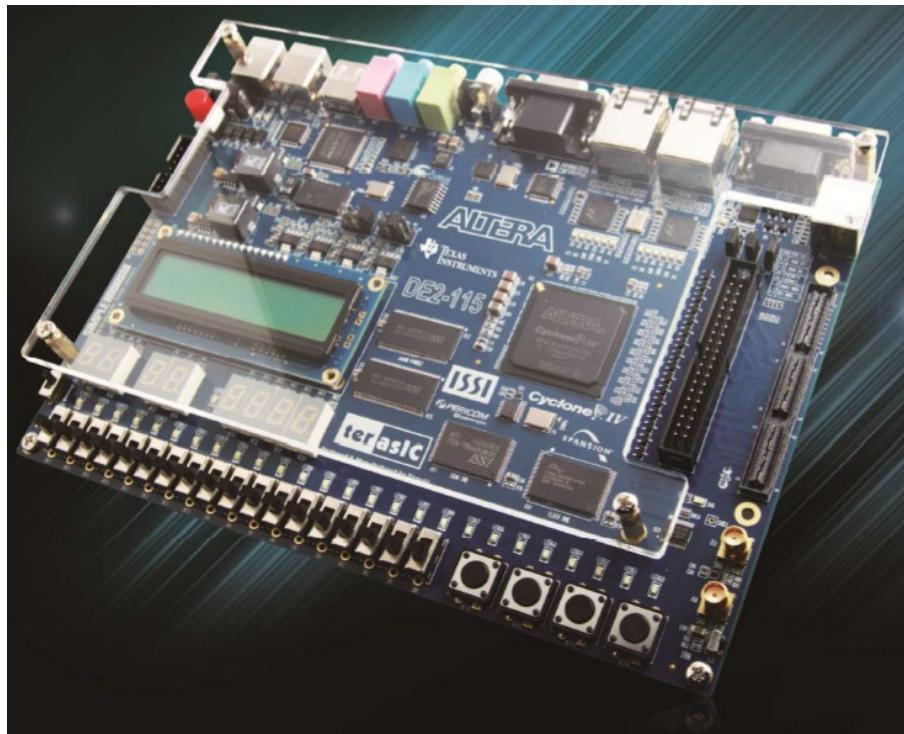
ASM Chart for Binary Multiplier



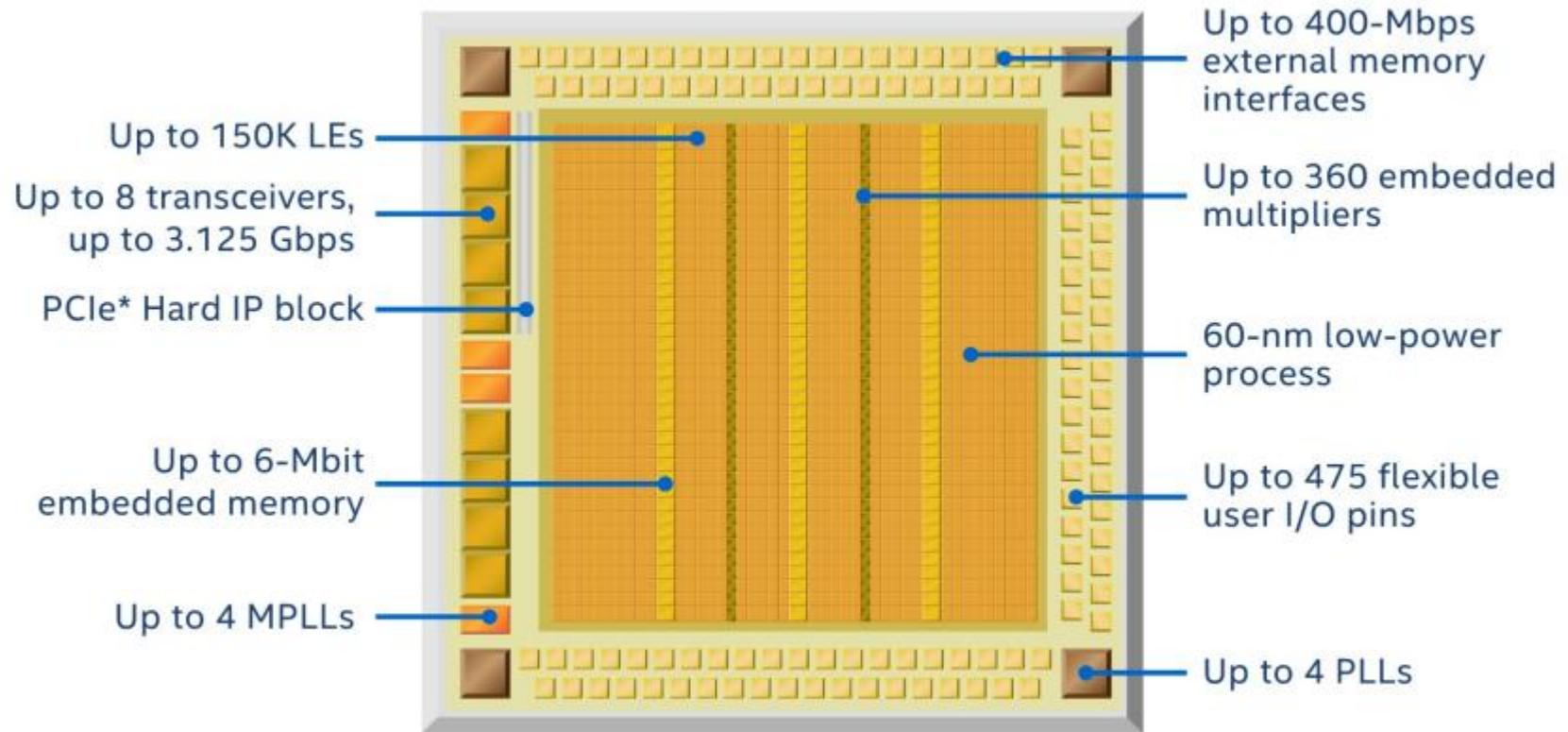
CPE 322

Digital Hardware Design Fundamentals

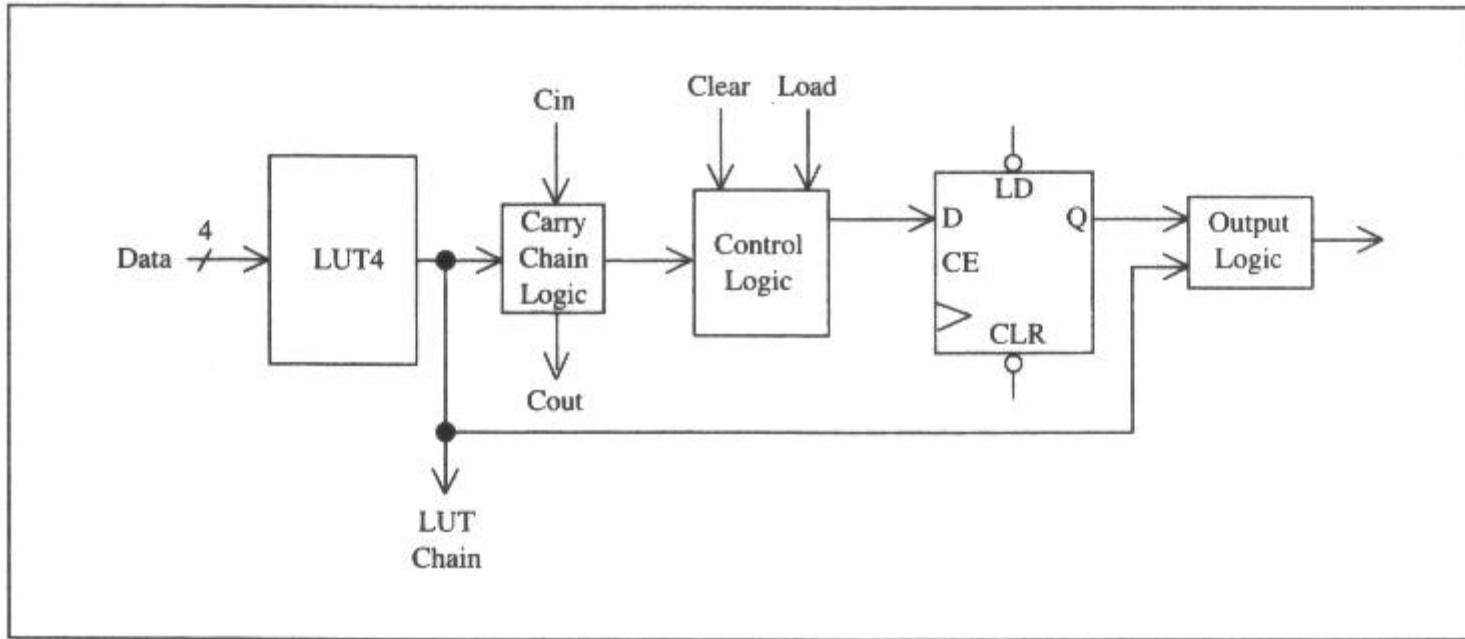
Non CLB FPGA Building Blocks:
Dedicated Memory Blocks &
Dedicated Multipliers



Cyclone IV E Family FPGA Layout



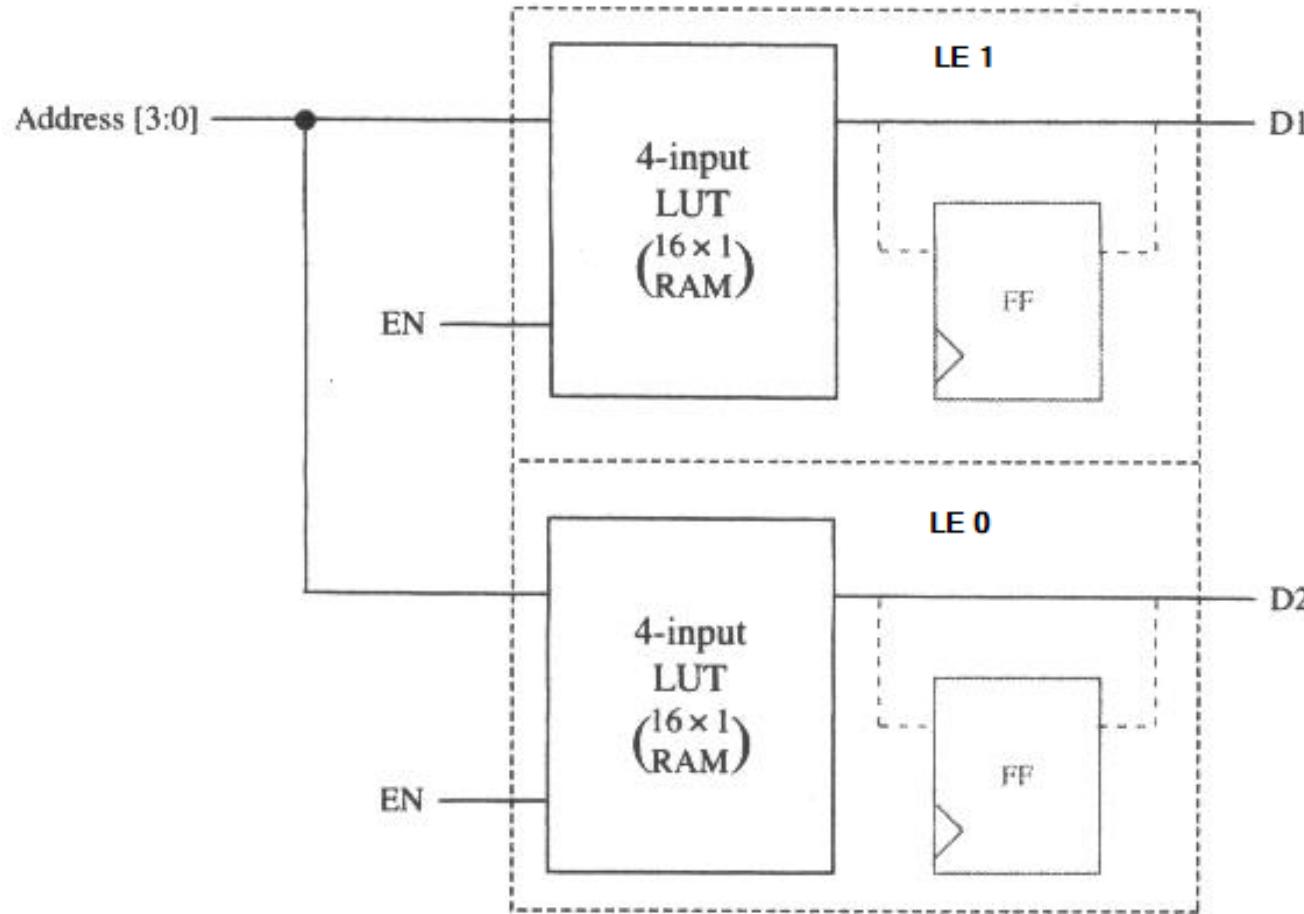
IntelFPGA's Configurable Logic Block (Cyclone IV E Logic Element -- LE)



114,480 LEs on the IntelFPGA EP4CE115 FPGA in the DE2-115

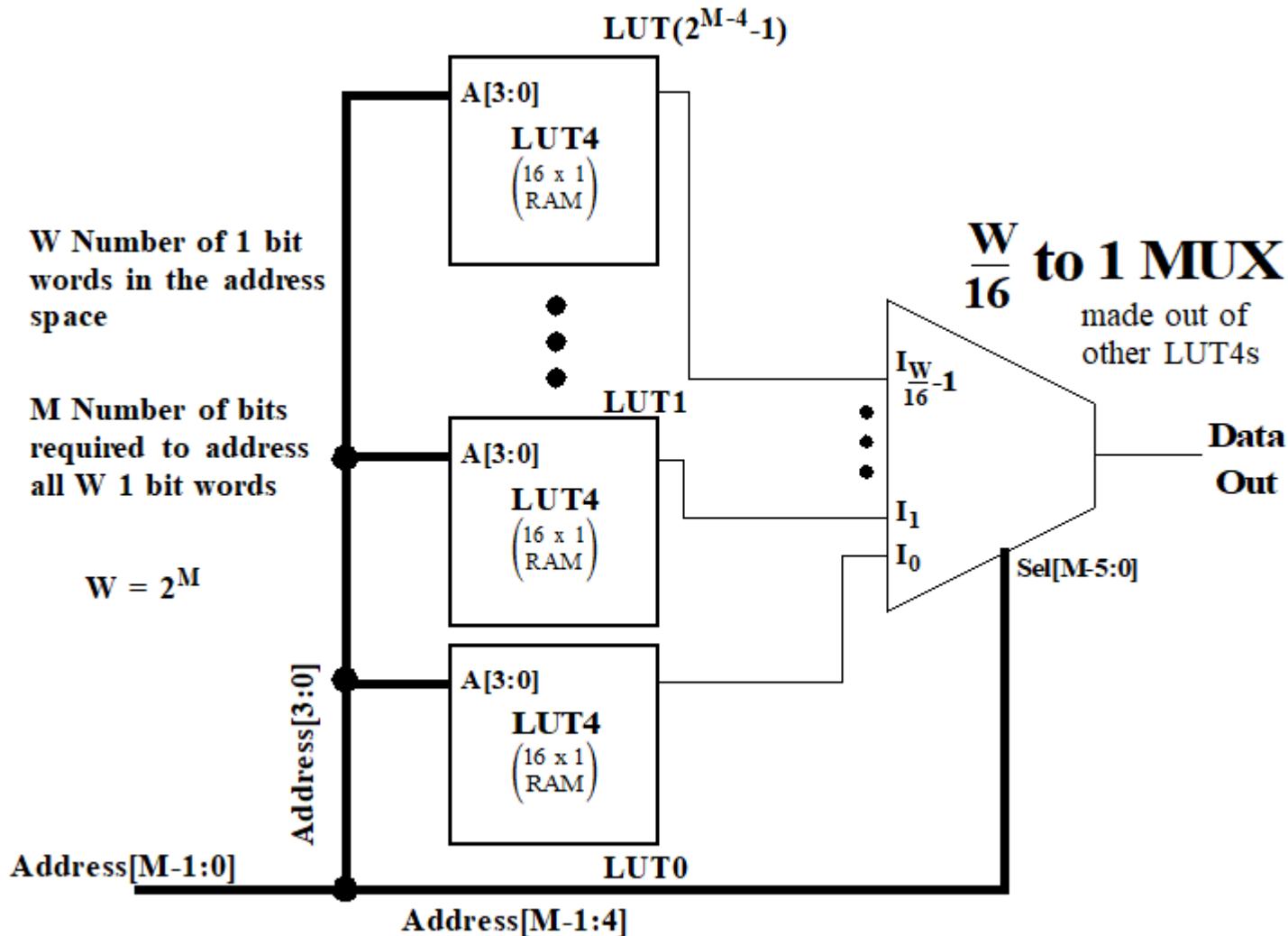
Creating Memory from LUTs

(16 x 2 memory)

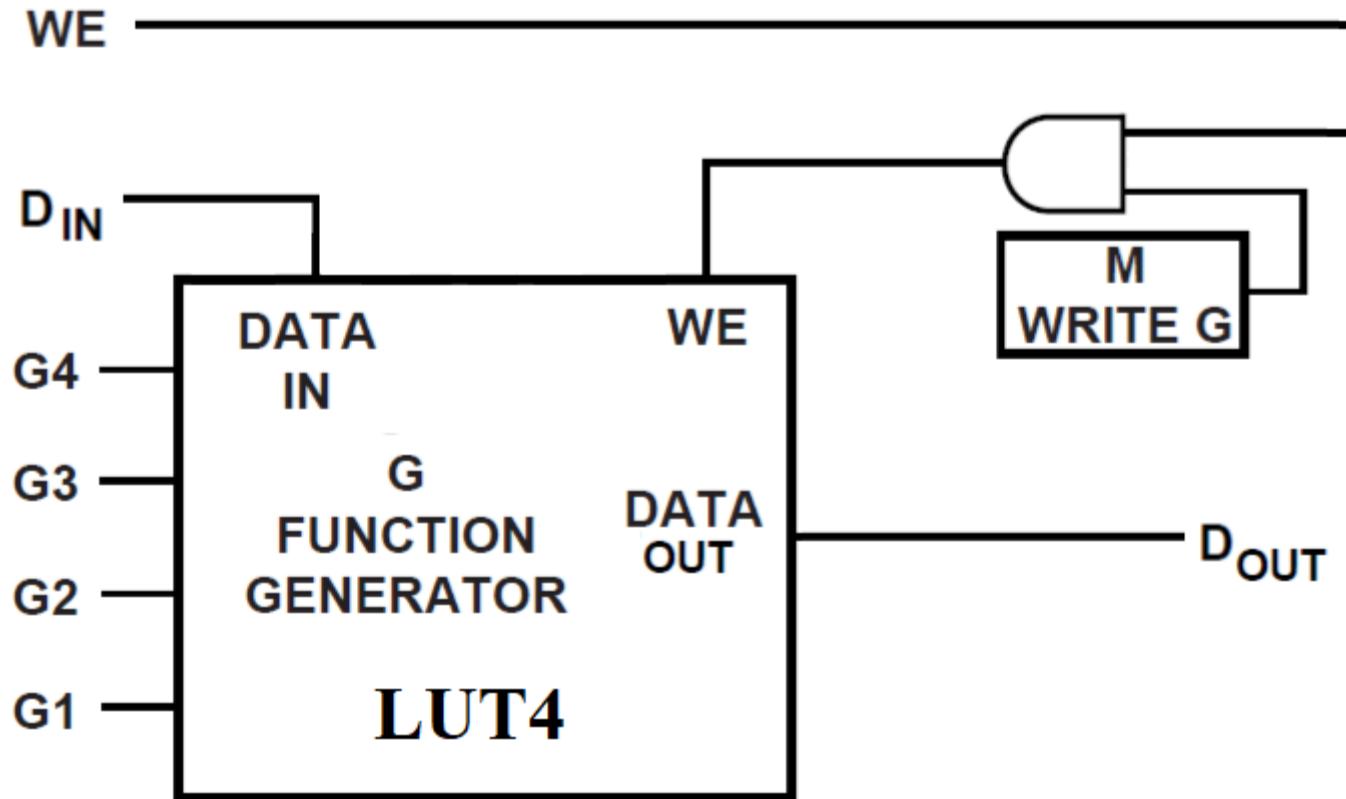


Creating Memory from LUTs

(W x 1 Rom Memory)



LUTs as a Read/Write Memory Cell



Creating Memory from LUTs

Verilog HDL Model that typically infers LUT-based memory

```
// Synchronous write Asynchronous Read Read/Write Memory
// Made from LUTs of the LUT Register Resources
module Memory  (input CLK, Memwrite, input [12:0] Address,
                input [7:0] Data_In, output [7:0] DATA_OUT);

    reg [7:0] DataMEM[0:8191]; // 8 K bytes of memory

    initial
        begin
            $readmemh("initial_ram.txt",DataMEM);
        end

    always @ (posedge CLK)
        begin
            if (Memwrite) DataMEM[Address] = Data_In; // Synchronous Write
        end

        assign DATA_OUT = DataMEM[Address];           // Asynchronous Read

endmodule
```

Creating Memory from LUTs

Verilog HDL Model that typically infers LUT-based memory

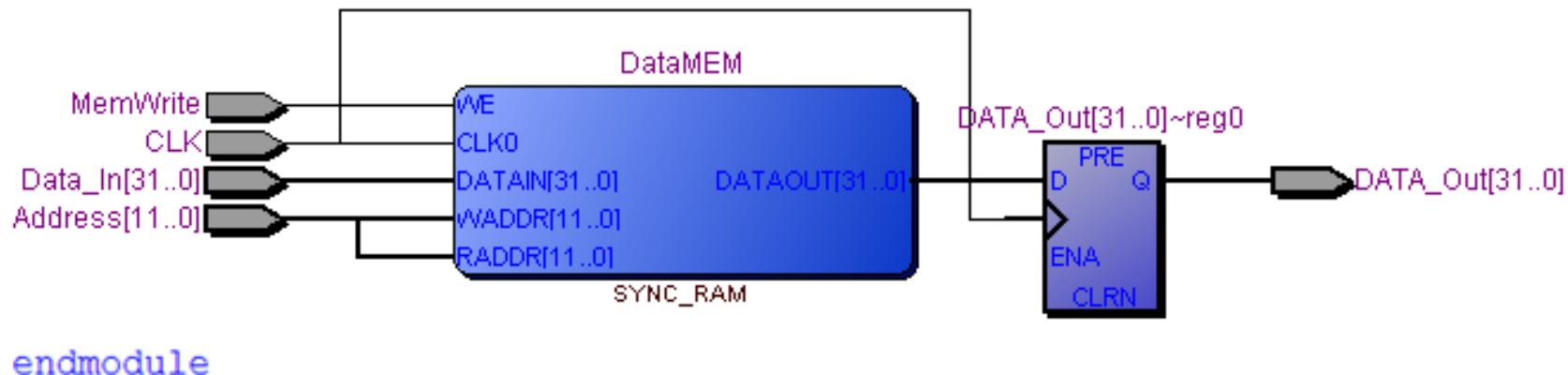
Flow Summary	
<<Filter>>	
Flow Status	Successful - Sun Apr 19 13:15:09 2020
Quartus Prime Version	16.1.0 Build 196 10/24/2016 SJ Lite Edition
Revision Name	Memory
Top-level Entity Name	Memory
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	84,363 / 114,480 (74 %)
Total registers	65536
Total pins	31 / 529 (6 %)
Total virtual pins	0
Total memory bits	0 / 3,981,312 (0 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

Took 74% of the 114,480
LEs on the IntelFPGA
EP4CE115 FPGA in the
DE2-115

Took 30 minutes to
Synthesize/Optimize/Place
and Route using Quartus
Prime CAD tool

Creating Memory Regions using Dedicated Embedded FPGA Synchronous RAM Memory

```
module Memory (input CLK, MemWrite, input [11:0] Address,  
    input [31:0] Data_In, output reg [31:0] DATA_Out);
```



Both inputs and outputs are activated by the clock

Dedicated Memory – Variable Data Widths (Cyclone IV E)

Width	Depth	Addr Bus	Data Bus
1	8192	13	1
2	4096	12	2
4	2048	11	4
8	1024	10	8
9	1024	10	9
16	512	9	16
18	512	9	18
32	256	8	32
36	256	8	36

Dedicated Memory – Variable Data Widths

(Cyclone IV E – Modes of Operation)

- Single Port
 - supports non-simultaneous read and write operations from a single address
- Simple Dual-Port
 - supports simultaneous read and write operations to different memory locations
- True Dual-Port
 - supports any combination of two-port operations: two reads, two writes, or one read and one write, at two different clock frequencies.
- Shift-register

Using the Embedded Dedicated Memory

Verilog Model that typically infers Dedicated memory

```
// Synchronous Read/Write Memory
// Quartus will use Dedicated Memory instead of LUTs because
// the Dedicated Memory in our targeted FPGA has the same attributes
module Memory (input CLK, Memwrite, input [12:0] Address,
               input [7:0] Data_In, output reg [7:0] DATA_OUT);

    reg [7:0] DataMEM[0:8191]; // 8 k bytes of memory

    initial
        begin
            $readmemh("initial_ram.txt",DataMEM);
        end

    always @ (posedge CLK)
        begin
            if (Memwrite) DataMEM[Address] = Data_In; // synchronous write
            DATA_OUT = DataMEM[Address];           // synchronous Read
        end

endmodule
```

Using the Embedded Dedicated Memory

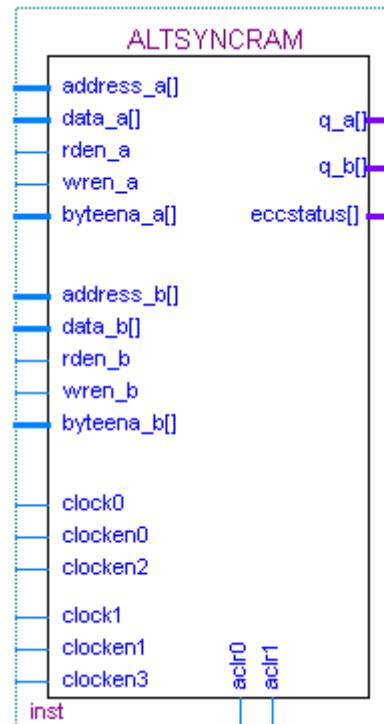
Verilog Model that typically infers Dedicated memory

Flow Summary	
 <<Filter>>	
Flow Status	Successful - Mon Apr 20 00:24:55 2020
Quartus Prime Version	16.1.0 Build 196 10/24/2016 SJ Lite Edition
Revision Name	Memory
Top-level Entity Name	Memory
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	9 / 114,480 (< 1 %)
Total registers	9
Total pins	31 / 529 (6 %)
Total virtual pins	0
Total memory bits	65,536 / 3,981,312 (2 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

Structural Technique for Implementing Memory Elements in the FPGA's Dedicated Block RAM Memory

```
// Memory module in Verilog HDL in a manner that explicitly utilizes
// the Altera Megafunction library in Quartus. This will utilize
// 64 M4K dedicated memory elements for a total storage of
// 262144 bits (It also uses 48 LE's).
module rom (input [14:0] address, input clock, output [7:0] q);
    altsyncram C1 (
        .clock0 (clock),
        .address_a (address),
        .q_a (q),
        .aclr0 (1'b0),
        .aclr1 (1'b0),
        .address_b (1'b1),
        .addressesstall_a (1'b0),
        .addressesstall_b (1'b0),
        .byteena_a (1'b1),
        .byteena_b (1'b1),
        .clock1 (1'b1),
        .clocken0 (1'b1),
        .clocken1 (1'b1),
        .clocken2 (1'b1),
        .clocken3 (1'b1),
        .data_a ({8{1'b1}}),
        .data_b (1'b1),
        .eccstatus (),
        .q_b (),
        .rden_a (1'b1),
        .rden_b (1'b1),
        .wren_a (1'b0),
        .wren_b (1'b0));

```



```
defparam
    C1.clock_enable_input_a = "BYPASS",
    C1.clock_enable_output_a = "BYPASS",
    C1.init_file = "E:/example/rom.hex",
    C1.intended_device_family = "Cyclone IV",
    C1.lpm_hint = "ENABLE_RUNTIME_MOD=NO",
    C1.lpm_type = "altsyncram",
    C1.numwords_a = 32768,
    C1.operation_mode = "ROM",
    C1.outdata_aclr_a = "NONE",
    C1.outdata_reg_a = "UNREGISTERED",
    C1.power_up_uninitialized = "FALSE",
    C1.ram_block_type = "M9K",
    C1.widthad_a = 15,
    C1.width_a = 8,
    C1.width_byteena_a = 1;
endmodule
```

Multiplication in FPGAs

Using LUTS to Implement a 4x4 Multiplier (using LUTs)

```
module LUTmult(input [3:0] Mplier, Mcand, output reg [7:0] Product);

    reg [7:0] prod_rom [0:255];

    initial
        begin:ROM_LOAD
            reg [8:0] i;
            for (i=0;i<256;i=i+1)
                prod_rom[i] = {4'b0000,i[7:4]}*{4'b0000,i[3:0]};
        end
        ————— columns —————

        (x"00", x"00", x"00",
         x"00", x"01", x"02", x"03", x"04", x"05", x"06", x"07", x"08", x"09", x"0A", x"0B", x"0C", x"0D", x"0E", x"0F",
         x"00", x"02", x"04", x"06", x"08", x"0A", x"0C", x"0E", x"10", x"12", x"14", x"16", x"18", x"1A", x"1C", x"1E",
         x"00", x"03", x"06", x"09", x"0C", x"0F", x"12", x"15", x"18", x"1B", x"1E", x"21", x"24", x"27", x"2A", x"2D",
         x"00", x"04", x"08", x"0C", x"10", x"14", x"18", x"1C", x"20", x"24", x"28", x"2C", x"30", x"34", x"38", x"3C",
         x"00", x"05", x"0A", x"0F", x"14", x"19", x"1E", x"23", x"28", x"2D", x"32", x"37", x"3C", x"41", x"46", x"4B",
         x"00", x"06", x"0C", x"12", x"18", x"1E", x"24", x"2A", x"30", x"36", x"3C", x"42", x"48", x"4E", x"54", x"5A",
ROWS x"00", x"07", x"0E", x"15", x"1C", x"23", x"2A", x"31", x"38", x"3F", x"46", x"4D", x"54", x"5B", x"62", x"69",
        x"00", x"08", x"10", x"18", x"20", x"28", x"30", x"38", x"40", x"48", x"50", x"58", x"60", x"68", x"70", x"78",
        x"00", x"09", x"12", x"18", x"24", x"2D", x"36", x"3F", x"48", x"51", x"5A", x"63", x"6C", x"75", x"7E", x"87",
        x"00", x"0A", x"14", x"1E", x"28", x"32", x"3C", x"46", x"50", x"5A", x"64", x"6E", x"78", x"82", x"8C", x"96",
        x"00", x"0B", x"16", x"21", x"2C", x"37", x"42", x"4D", x"58", x"63", x"6E", x"79", x"84", x"8F", x"9A", x"A5",
        x"00", x"0C", x"18", x"24", x"30", x"3C", x"48", x"54", x"60", x"6C", x"78", x"84", x"90", x"9C", x"A8", x"B4",
        x"00", x"0D", x"1A", x"27", x"34", x"41", x"4E", x"5B", x"68", x"75", x"82", x"8F", x"9C", x"A9", x"86", x"C3",
        x"00", x"0E", x"1C", x"2A", x"38", x"46", x"54", x"62", x"70", x"7E", x"8C", x"9A", x"A8", x"86", x"C4", x"D2",
        x"00", x"0F", x"1E", x"2D", x"3C", x"48", x"5A", x"69", x"78", x"87", x"96", x"A5", x"84", x"C3", x"D2", x"E1");

    always @ (Mplier or Mcand)
        Product = prod_rom[{Mplier,Mcand}]; // read Product LUT

endmodule
```

Using LUTS to Implement a 4x4 Multiplier (using LUTs)

Flow Summary	
 <<Filter>>	
Flow Status	Successful - Mon Apr 20 00:52:44 2020
Quartus Prime Version	16.1.0 Build 196 10/24/2016 SJ Lite Edition
Revision Name	LUTmult
Top-level Entity Name	LUTmult
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	67 / 114,480 (< 1 %)
Total registers	0
Total pins	16 / 529 (3 %)
Total virtual pins	0
Total memory bits	0 / 3,981,312 (0 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

Using Dedicated Block RAM to Implement a 4x4 Multiplier

```
module LUTmult(input clk, input [3:0] Mplier, Mcand, output reg [7:0] Product);

reg [7:0] prod_rom [0:255];

initial
begin:ROM_LOAD
reg [8:0] i;
for (i=0;i<256;i=i+1)
    prod_rom[i] = {4'b0000,i[7:4]}*{4'b0000,i[3:0]};
end

----- columns -----

(x"00", x"00", x"00",
x"00", x"01", x"02", x"03", x"04", x"05", x"06", x"07", x"08", x"0A", x"0B", x"0C", x"0D", x"0E", x"0F",
x"00", x"02", x"04", x"06", x"08", x"0A", x"0C", x"0E", x"10", x"12", x"14", x"16", x"18", x"1A", x"1C", x"1E",
x"00", x"03", x"06", x"09", x"0C", x"0F", x"12", x"15", x"18", x"1B", x"1E", x"21", x"24", x"27", x"2A", x"2D",
x"00", x"04", x"08", x"0C", x"10", x"14", x"18", x"1C", x"20", x"24", x"28", x"2C", x"30", x"34", x"38", x"3C",
x"00", x"05", x"0A", x"0F", x"14", x"19", x"1E", x"23", x"28", x"2D", x"32", x"37", x"3C", x"41", x"46", x"4B",
x"00", x"06", x"0C", x"12", x"18", x"1E", x"24", x"2A", x"30", x"36", x"3C", x"42", x"48", x"4E", x"54", x"5A",
ROWS x"00", x"07", x"0E", x"15", x"1C", x"23", x"2A", x"31", x"38", x"3F", x"46", x"4D", x"54", x"5B", x"62", x"69",
x"00", x"08", x"10", x"18", x"20", x"28", x"30", x"38", x"40", x"48", x"50", x"58", x"60", x"68", x"70", x"78",
x"00", x"09", x"12", x"18", x"24", x"20", x"36", x"3F", x"48", x"51", x"5A", x"63", x"6C", x"75", x"7E", x"87",
x"00", x"0A", x"14", x"1E", x"28", x"32", x"3C", x"46", x"50", x"5A", x"64", x"6E", x"78", x"82", x"8C", x"96",
x"00", x"0B", x"16", x"21", x"2C", x"37", x"42", x"4D", x"58", x"63", x"6E", x"79", x"84", x"8F", x"9A", x"A5",
x"00", x"0C", x"18", x"24", x"30", x"3C", x"48", x"54", x"60", x"6C", x"78", x"84", x"90", x"9C", x"A8", x"84",
x"00", x"0D", x"1A", x"27", x"34", x"41", x"4E", x"5B", x"68", x"75", x"82", x"8F", x"9C", x"A9", x"B6", x"C3",
x"00", x"0E", x"1C", x"2A", x"38", x"46", x"54", x"62", x"70", x"7E", x"8C", x"9A", x"A8", x"B6", x"C4", x"D2",
x"00", x"0F", x"1E", x"2D", x"3C", x"48", x"5A", x"69", x"78", x"87", x"96", x"A5", x"B4", x"C3", x"D2", x"E1");

always @ (posedge clk)
Product = prod_rom[{Mplier,Mcand}]; // read Product LUT
// synchronously

endmodule
```

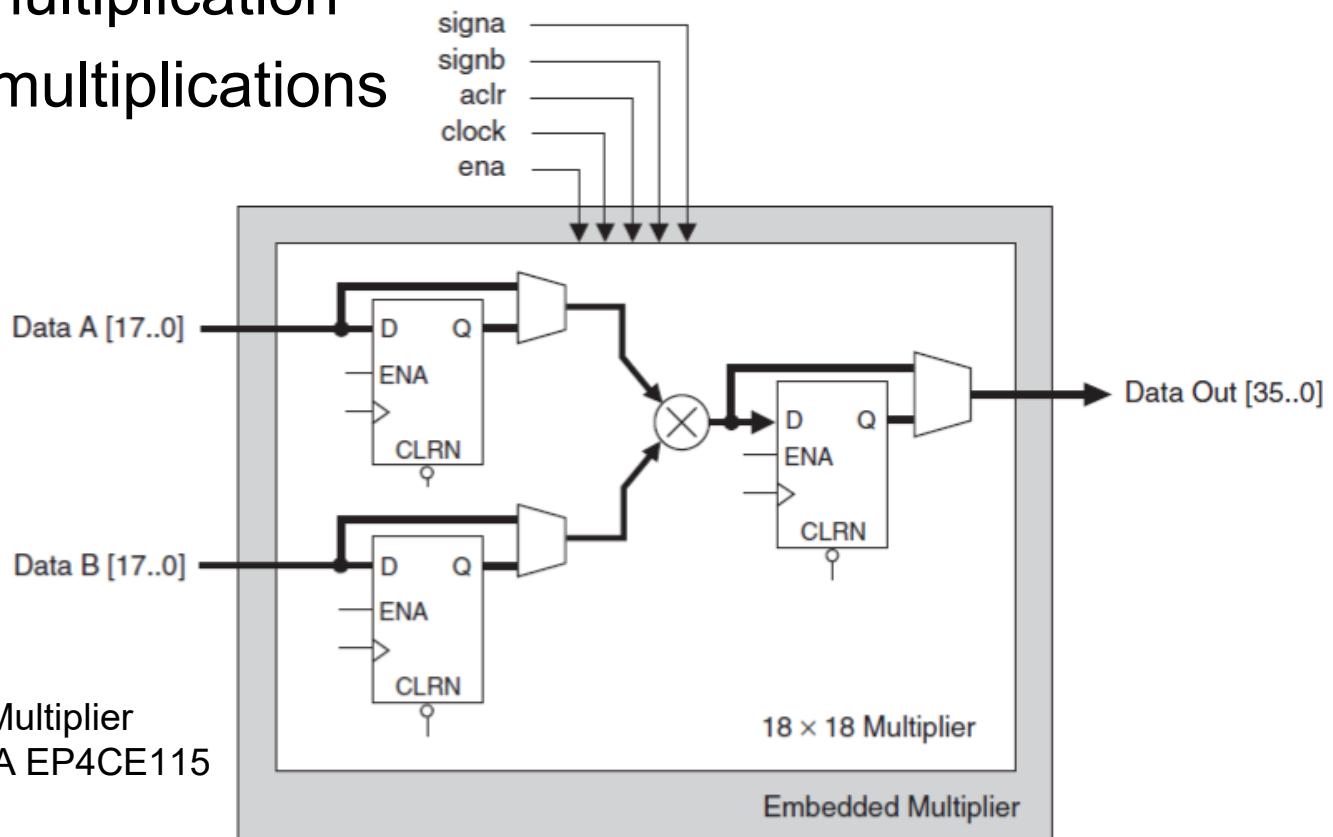
Using Dedicated Block RAM to Implement a 4x4 Multiplier

Flow Summary	
 <<Filter>>	
Flow Status	Successful - Mon Apr 20 01:09:21 2020
Quartus Prime Version	16.1.0 Build 196 10/24/2016 SJ Lite Edition
Revision Name	LUTmult
Top-level Entity Name	LUTmult
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	0 / 114,480 (0 %)
Total registers	0
Total pins	17 / 529 (3 %)
Total virtual pins	0
Total memory bits	2,048 / 3,981,312 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

Embedded Multiplier Blocks

(Cyclone IV E – 18x18 bit mode)

- Two Modes for each 18x18 bit multiplier
 - 18x18 bit multiplication
 - Two 9 x 9 multiplications

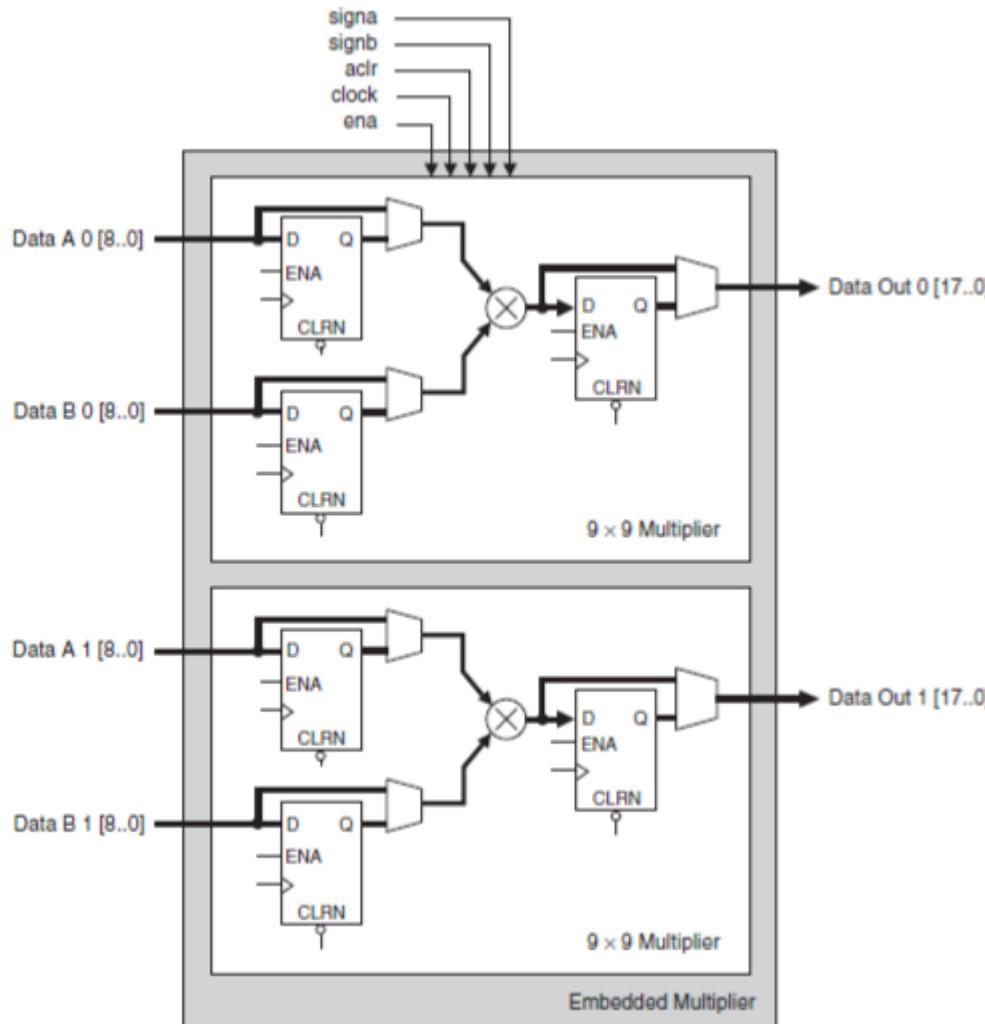


Embedded Multiplier Blocks (Cyclone IV E – Dual 9x9 bit mode)

266 Embedded 18x18
Multiplier Blocks on the
IntelFPGA EP4CE115
FPGA in the DE2-115

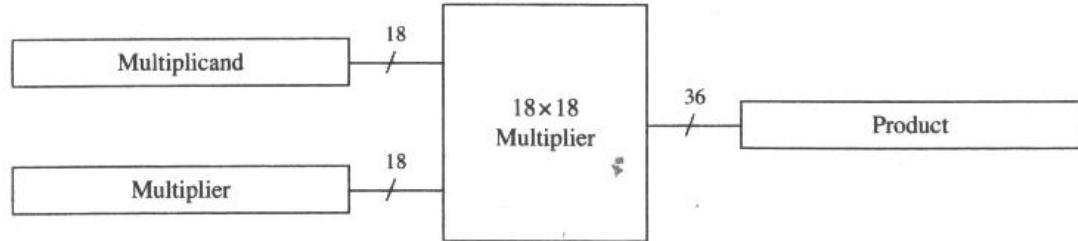
OR

532 Embedded 9x9
Multiplier Blocks on the
IntelFPGA EP4CE115
FPGA in the DE2-115
(when all are operated in
the 9x9 mode)



Using Embedded Multiplier Blocks to Implement a 32 x 32 bit Multiplication

```
module mult_embedded (input [31:0] A,B, output [63:0] C);  
    assign C = A * B;  
endmodule
```



Flow Summary	
<<Filter>>	
Flow Status	Successful - Mon Apr 20 02:47:39 2020
Quartus Prime Version	16.1.0 Build 196 10/24/2016 SJ Lite Edition
Revision Name	mult_embedded
Top-level Entity Name	mult_embedded
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	79 / 114,480 (< 1 %)
Total registers	0
Total pins	128 / 529 (24 %)
Total virtual pins	0
Total memory bits	0 / 3,981,312 (0 %)
Embedded Multiplier 9-bit elements	8 / 532 (2 %)
Total PLLs	0 / 4 (0 %)