CDA 4253/CIS 6930 FPGA System Design
Modeling of Combinational Circuits

Hao Zheng
Dept of Comp Sci & Eng
USF

Reading

- P. Chu, *FPGA Prototyping by VHDL Examples*
  - ***Chapter 3, RT-level combinational circuit***
  - ***Sections 3.1 - 3.2, 3.5 - 3.7.***
- *XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices*
  - **Chapter 3 and 7**

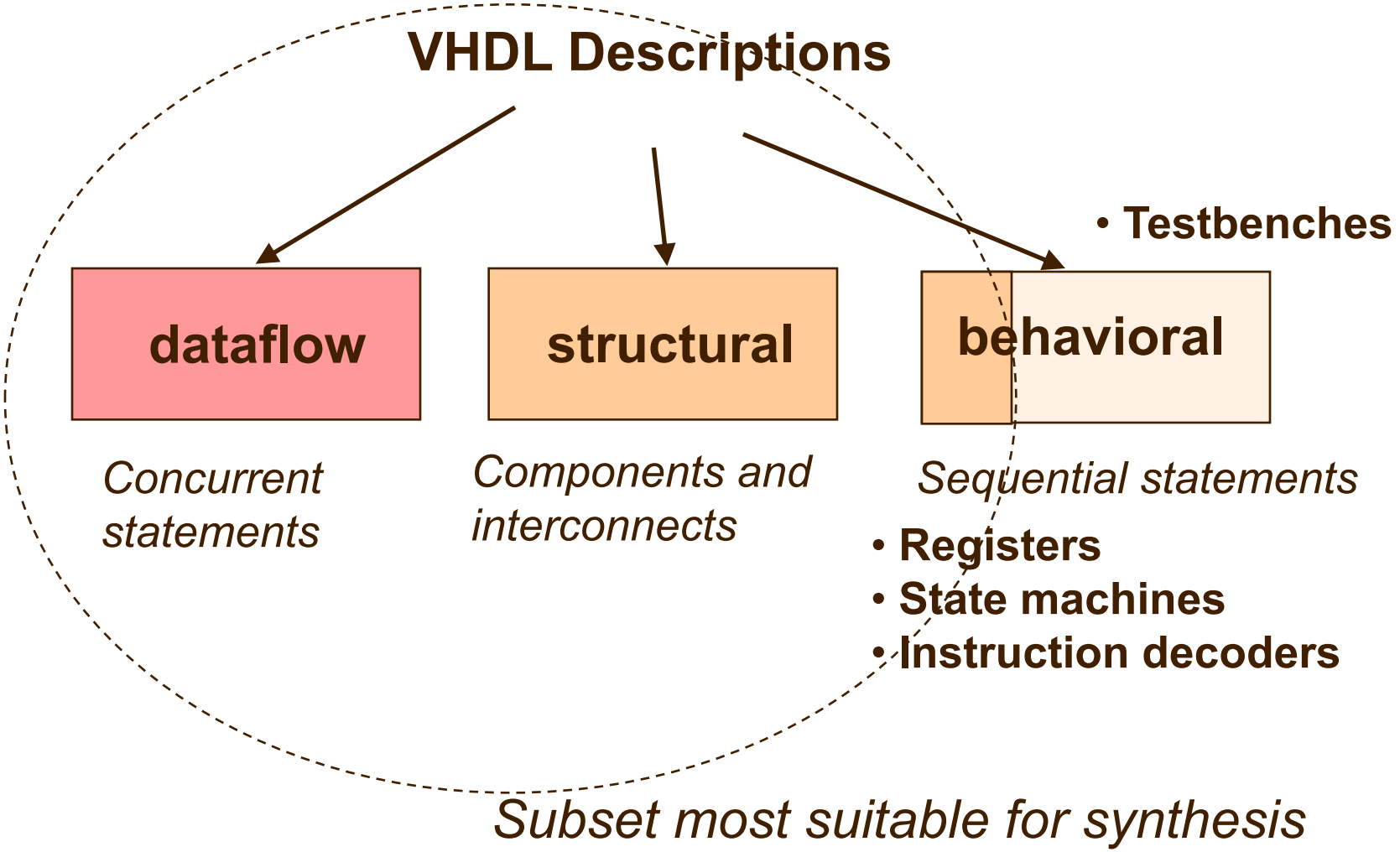
VHDL Model Template: Recap

```
library ieee;
use ieee.std_logic_1164.all;

entity entity_name is
     port declarations
end [entity] entity_name;

ARCHITECTURE architecture_name OF entity_name IS
     Signal & component declarations
BEGIN
     Concurrent statements
END [ARCHITECTURE] architecture_name;
```

Concurrent Statements

- **Simple concurrent signal assignment**
  - z <= a **xor** b
- **Conditional signal assignment (when-else)**
- **selected concurrent signal assignment (with-select-when)**
- **Process** statements
  - To be covered later

VHDL Modeling Styles

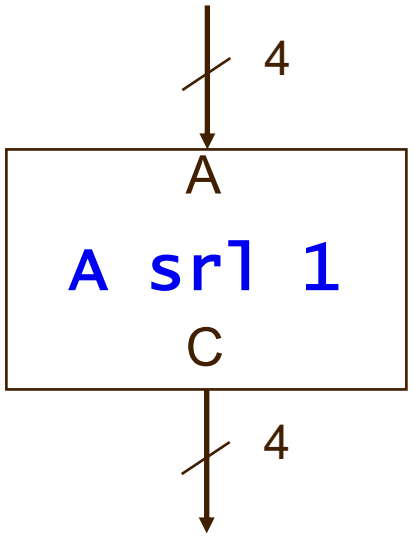


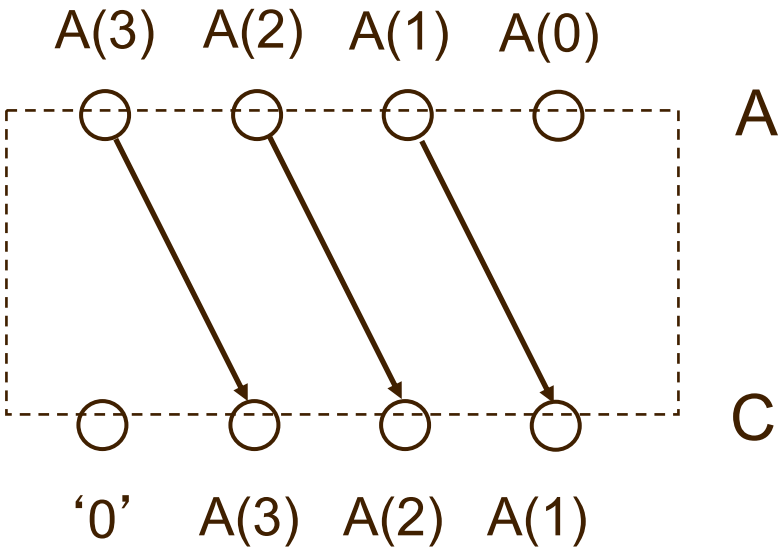
Combinational Circuit Building Blocks

Fixed Shifters & Rotators

Fixed Logical Shift Right in VHDL

```
SIGNAL A : STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL C : STD_LOGIC_VECTOR(3 DOWNTO 0);
```



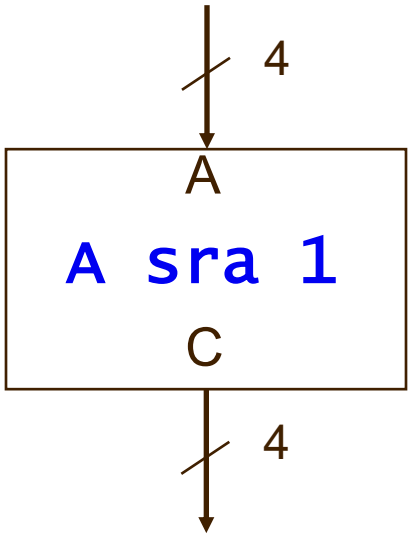


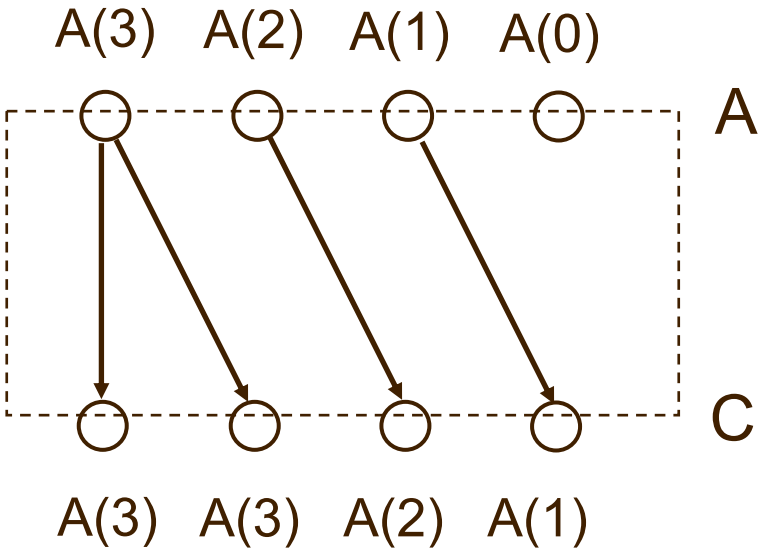
`srl`: logic shift right

```
C <= A srl 1;
C <= '0' & A(3 downto 1);
```

Fixed Arithmetic Shift Right in VHDL

```
SIGNAL A : STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL C : STD_LOGIC_VECTOR(3 DOWNTO 0);
```



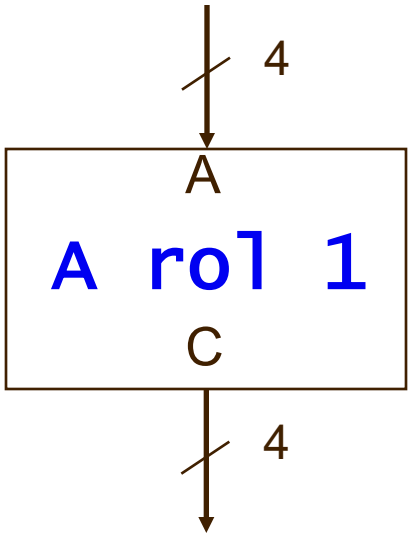


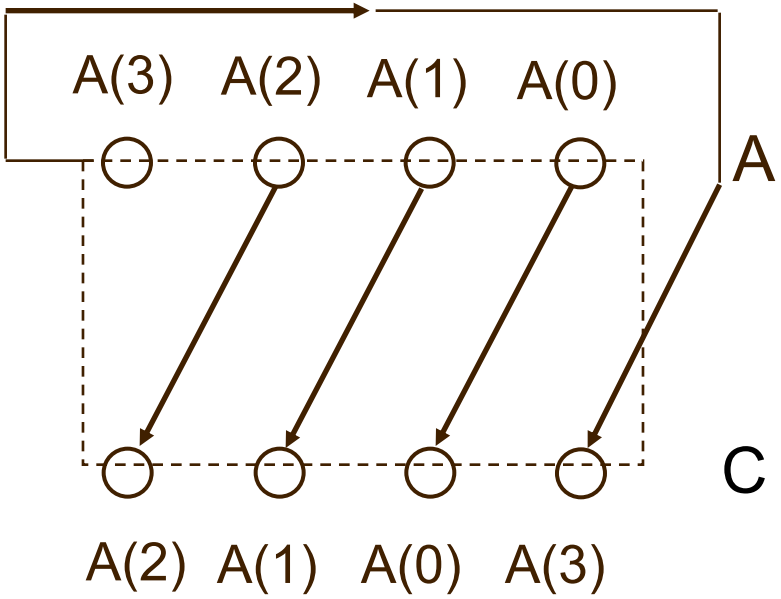
sra: arithmetic shift left

```
C <= A sra 1;
c <= A(3) & A(3 downto 1);
```

Fixed Rotation in VHDL

```
SIGNAL A : STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL C : STD_LOGIC_VECTOR(3 DOWNTO 0);
```



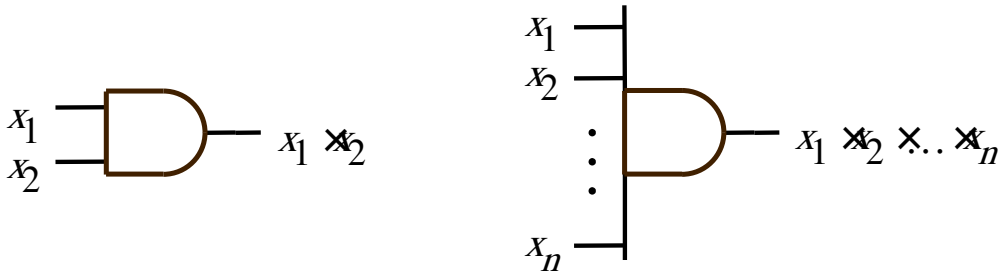


`rol:` rotation to left

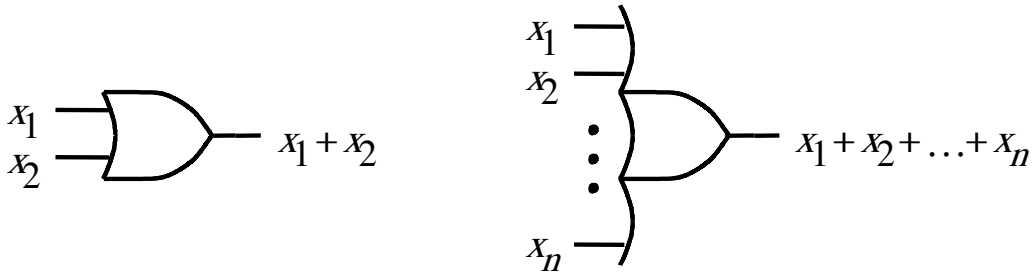
```
C <= A rol 1
```

Logic Gates

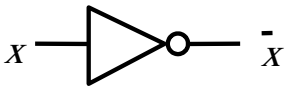
Basic Gates – AND, OR, NOT



(a) AND gates

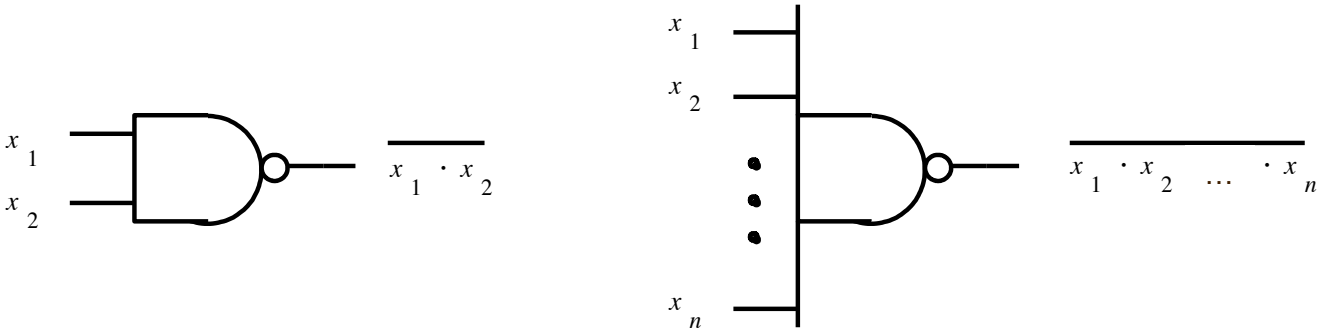


(b) OR gates

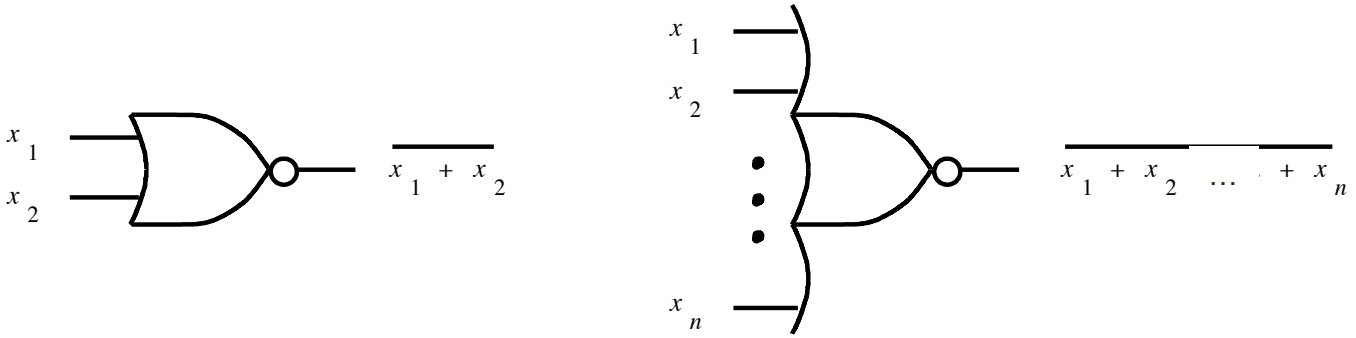


(c) NOT gate

Basic Gates – NAND, NOR



(a) NAND gates

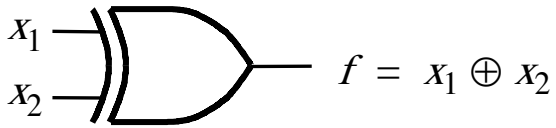


(b) NOR gates

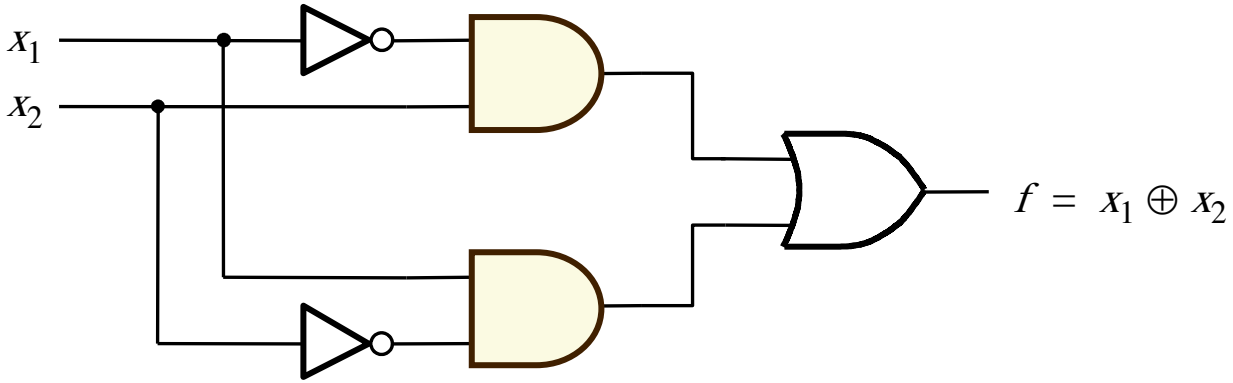
Basic Gates – XOR

x_1	x_2	$f = x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

(a) Truth table



(b) Graphical symbol

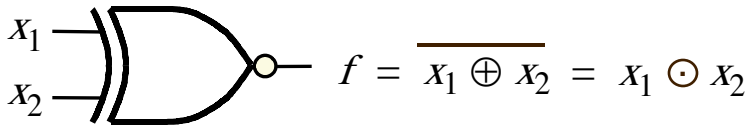


(c) Sum-of-products implementation

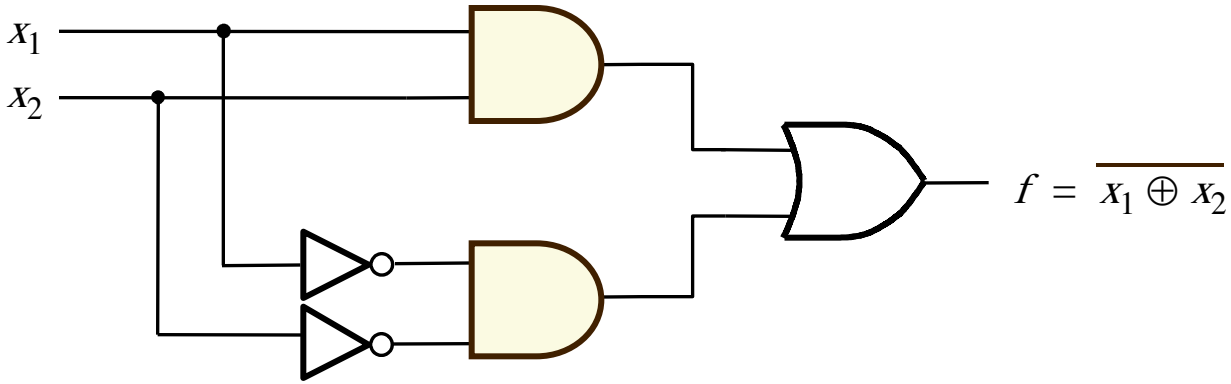
Basic Gates – XNOR

x_1	x_2	$f = \overline{x_1 \oplus x_2}$
0	0	1
0	1	0
1	0	0
1	1	1

(a) Truth table

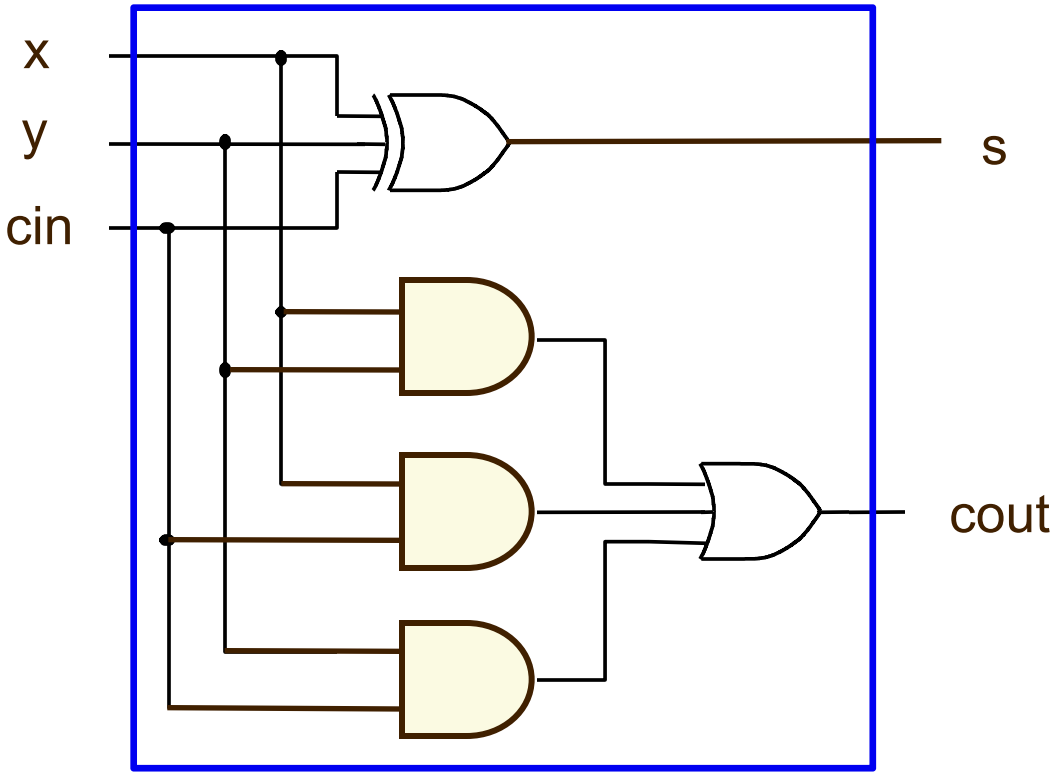


(b) Graphical symbol



(c) Sum-of-products implementation

1-Bit Full Adder

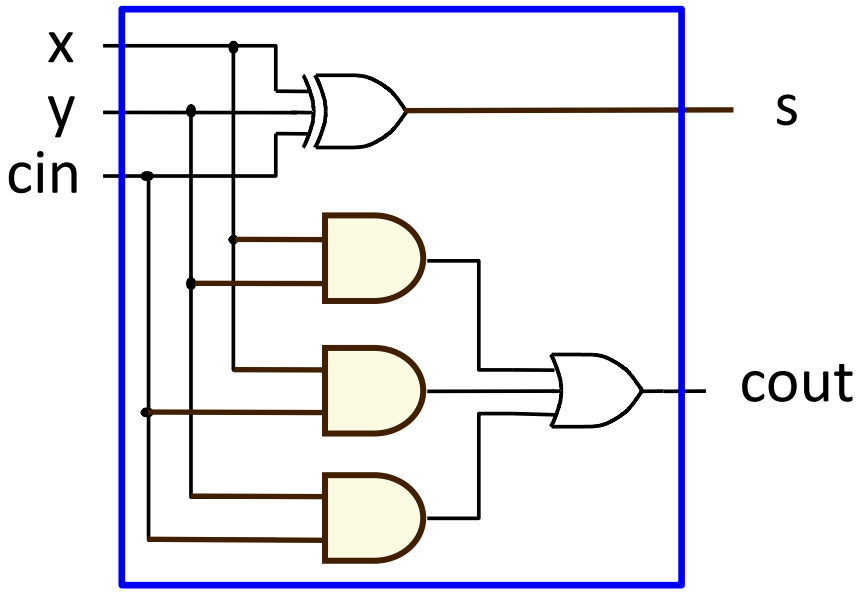


1-Bit Full Adder

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY fa1b IS
     PORT(    x : IN STD_LOGIC ;
              y : IN STD_LOGIC ;
            cin : IN STD_LOGIC ;
              s : OUT STD_LOGIC ;
           cout : OUT STD_LOGIC ) ;
END fa1b;
```

1-Bit Full Adder

```
ARCHITECTURE dataflow OF fa1b IS
BEGIN
      s      <=  x XOR y XOR cin ;
      cout   <=  (x AND y) OR (cin AND x)
                 OR (cin AND y) ;
END dataflow ;
```



Logic Operators

- Logic operators

```
and   or   nand   nor   xor   not   xnor
```

- Logic operators precedence

Highest

Lowest

```
                    not
and    or    nand    nor    xor   xnor
```

No Implied Precedence

Wanted: $y = ab + cd$

Incorrect

y <= a **and** b **or** c **and** d;

equivalent to

y <= ((a **and** b) **or** c) **and** d;

equivalent to

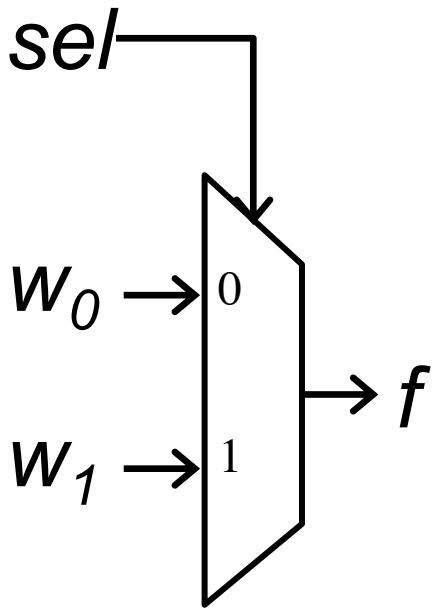
$y = (ab + c)d$

Correct

y <= (a **and** b) **or** (c **and** d);

Modeling Routing Structures
with
Conditional Concurrent Signal Assignment
(when-else)

2-to-1 Multiplexer



(a) Graphical symbol

sel	f
0	w_0
1	w_1

(b) Truth table

2-to-1 Multiplexer

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
   PORT( w0, w1, sel  : IN     STD_LOGIC ;
         f            : OUT    STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE dataflow OF mux2to1 IS
BEGIN
   f  <= w0 WHEN sel = '0' ELSE
         w1;
END dataflow ;
```

Conditional Concurrent Signal Assignment

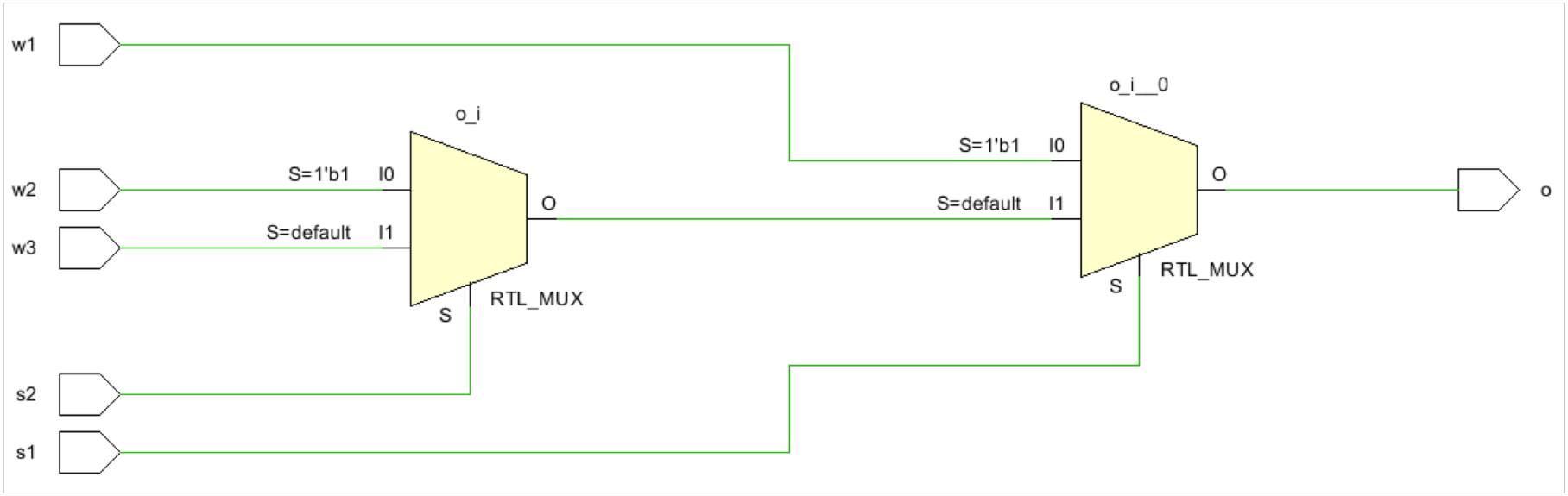
```
target_signal <= value1  when condition1 else
                 value2  when condition2 else
                   . . .
                 value_N+1 when condition_N+1 else
                 value_N;
```

- Branches are evaluated one by one from top to bottom.
- Induces priority among branches

Cascade of Multiplexers

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux_cascade IS
   PORT (w1, w2, w3     : IN  STD_LOGIC ;
         s1, s2         : IN  STD_LOGIC ;
         f              : OUT STD_LOGIC ) ;
END mux_cascade ;

ARCHITECTURE dataflow OF mux_cascade IS
BEGIN
   f  <= w1 WHEN s1 ='1'    ELSE
         w2 WHEN s2 ='1'    ELSE
         w3;
END dataflow ;
```

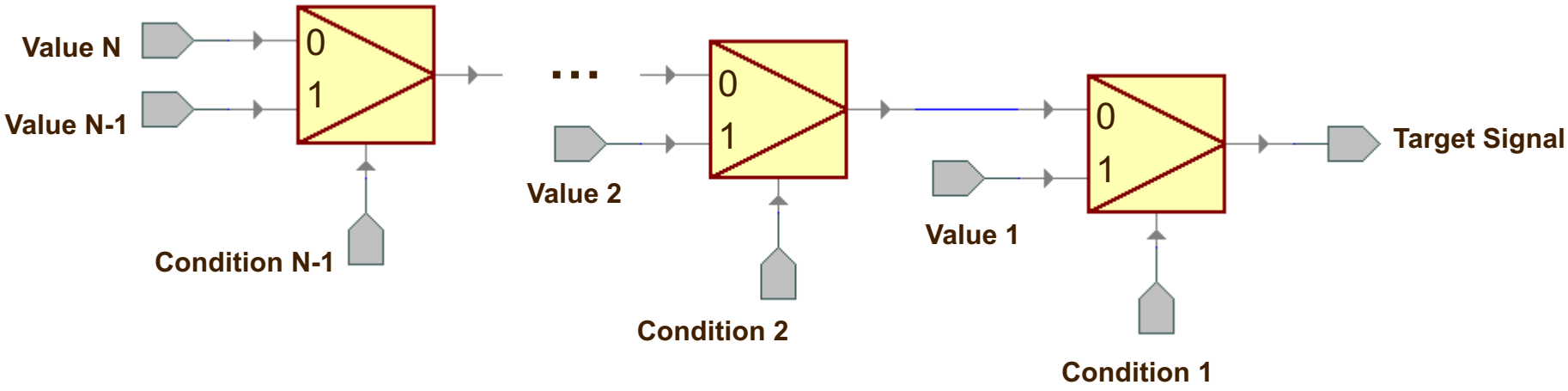
Cascade of Multiplexers



Notice the priority of selection.

Conditional Concurrent Signal Assignment

```
target_signal <= value1  when condition1 else
                 value2  when condition2 else
                    . . .
                 value_N+1 when condition_N+1 else
                 value_N;
```



More Operators

- Relational operators

=	/=	<	<=	>	>=

- Logic and relational operators precedence

Highest

Lowest

not					
=	/=	<	<=	>	>=
and	or	nand	nor	xor	xnor

Precedence of Logic and Relational Operators

Comparison $a = bc$

Incorrect

```
... when a = b and c else ...
```

equivalent to

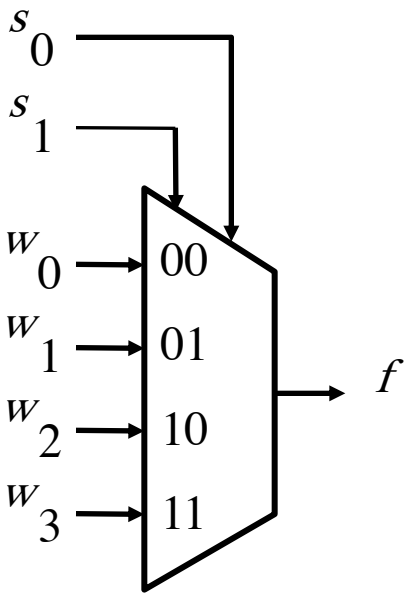
```
... when (a = b) and c else ...
```

Correct

```
... when a = (b and c) else ...
```

Modeling Routing Structures with Selected Concurrent Signal Assignment (with-select-when)

4-to-1 Multiplexer



(a) Graphic symbol

s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

(b) Truth table

No priority, and choices are disjoint.

A 4-to-1 Multiplexer

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux4to1 IS
    PORT( w0, w1, w2, w3     : IN     STD_LOGIC ;
          s                  : IN   STD_LOGIC_VECTOR(1 DOWNTO 0);
          f                  : OUT  STD_LOGIC ) ;
END mux4to1 ;

ARCHITECTURE dataflow OF mux4to1 IS
BEGIN
    WITH s SELECT
          f  <=      w0 WHEN "00",
                     w1 WHEN "01",
                     w2 WHEN "10",
                     w3 WHEN OTHERS;
END dataflow;
```

default condition

Selected Concurrent Signal Assignment

```
with choice_expression select
    target <= expression1 when choices_1,
              expression2 when choices_2,
                        . . .
              expressionN when choices_N;
```

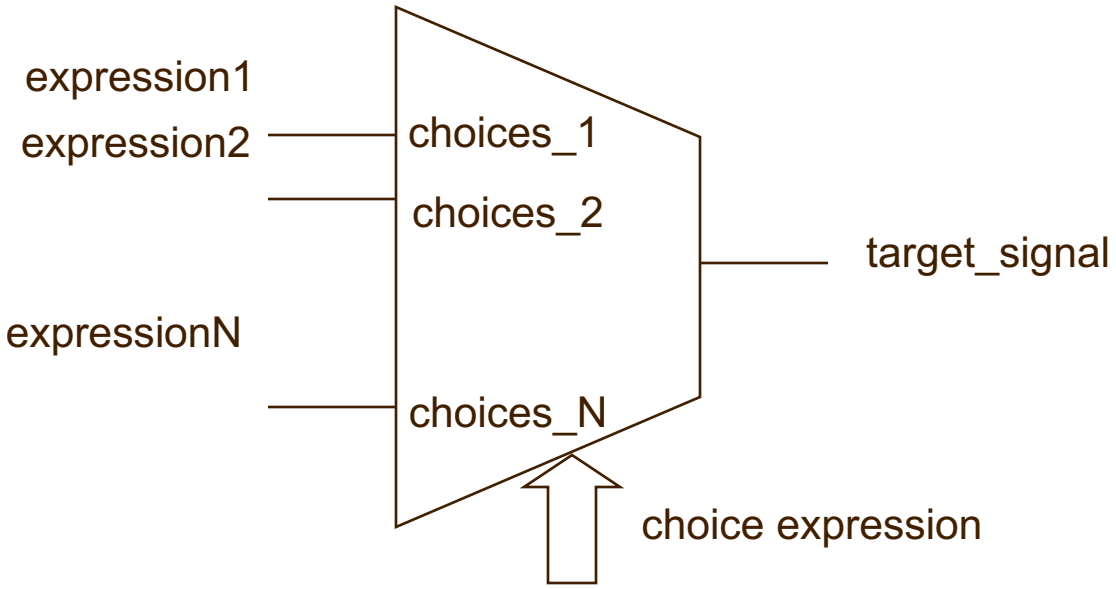
All choices are mutually exclusive

and

cover all values of choice_expression.

Selected Concurrent Signal Assignment

```
with choice_expression select
    target <= expression1 when choices_1,
              expression2 when choices_2,
                        . . .
              expressionN when choices_N;
```



Formats of Choices

- **when** *Expr*
- **when** *Expr*_1 | .... | *Expr*_N
  - this branch is taken if any of *Expr*_x matches *choice_expression*
- **when others**

Formats of Choices - Example

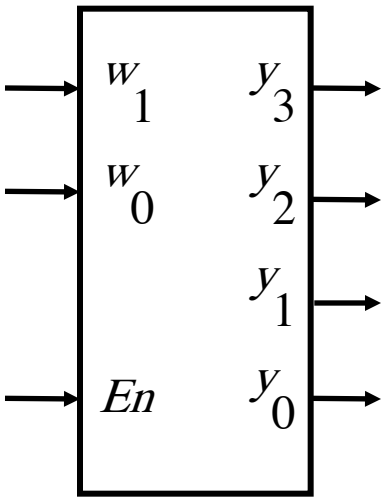
```
with sel select
     y <= a when "000",
          c when "001" | "111",
          d when others;
```

Decoders

2-to-4 Decoder

En	w_1	w_0	y_3	y_2	y_1	y_0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0
0	x	x	0	0	0	0

(a) Truth table



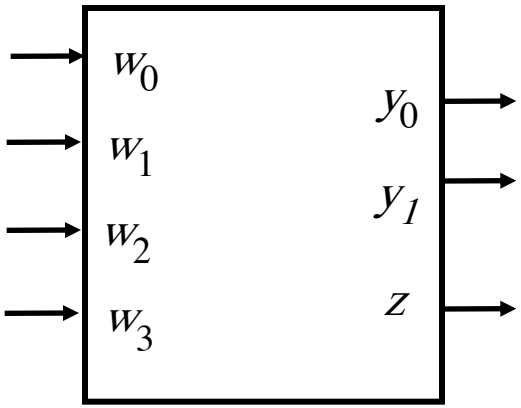
(b) Graphical symbol

VHDL Code for a 2-to-4 Decoder

```
-- LIBRARY not shown
ENTITY dec2to4 IS
    PORT ( w   : IN   STD_LOGIC_VECTOR(1 DOWNTO 0) ;
           En  : IN   STD_LOGIC ;
           y   : OUT  STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END dec2to4 ;

ARCHITECTURE dataflow OF dec2to4 IS
    SIGNAL Enw : STD_LOGIC_VECTOR(2 DOWNTO 0) ;
BEGIN
    Enw <= En & w ;
    WITH Enw SELECT
        y   <= "0001" WHEN "100",
               "0010" WHEN "101",
               "0100" WHEN "110",
               "1000" WHEN "111",
               "0000" WHEN OTHERS ;
END dataflow ;
```

Encoders

Priority Encoder



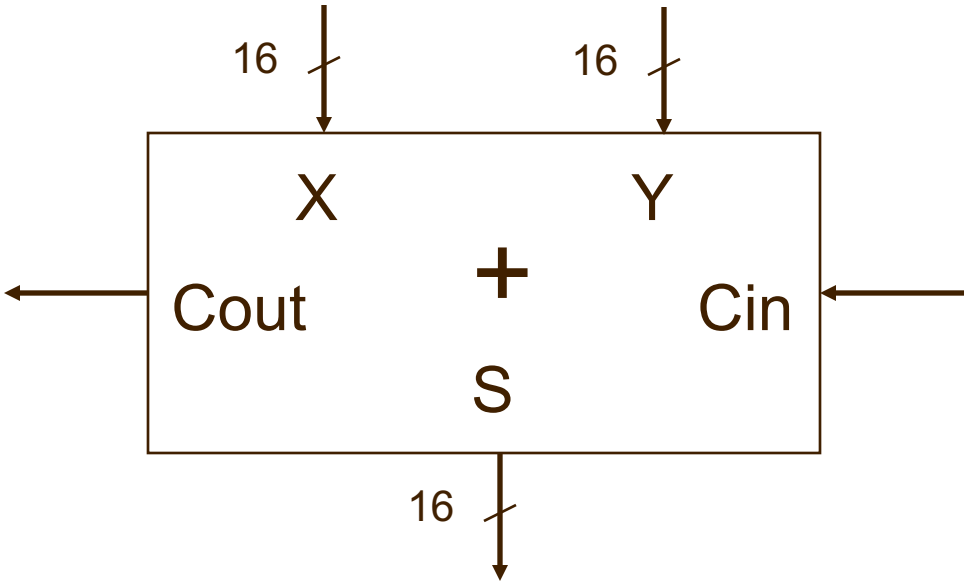
w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	x	x	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

VHDL code for a Priority Encoder

```
-- library not shown
ENTITY priority IS
   PORT (   w     : IN  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
            y   : OUT       STD_LOGIC_VECTOR(1 DOWNTO 0) ;
            z   : OUT       STD_LOGIC ) ;
END priority ;

ARCHITECTURE dataflow OF priority IS
BEGIN
   y <= "11" when w(3) = '1' else
        "10" when w(2) = '1' else
        "01" when w(1) = '1' else
        "00" when others;

   z <= '0' when w = "0000" else
        '1' when others;
END dataflow ;
```

Adders

16-bit Unsigned Adder



$S = X + Y$

Operations on Unsigned Numbers

For operations on **unsigned** numbers

USE

ieee.numeric_std.all

and

signals of the type UNSIGNED

and

conversion functions std_logic_vector(), unsigned()

OR USE

ieee.std_logic_unsigned.all

and

signals of the type STD_LOGIC_VECTOR

16-bit Unsigned Adder

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all;--non-IEEE standard

ENTITY adder16 IS
    PORT(  Cin          : IN   STD_LOGIC ;
           X            : IN   STD_LOGIC_VECTOR(15 DOWNTO 0) ;
           Y            : IN   STD_LOGIC_VECTOR(15 DOWNTO 0) ;
           S            : OUT  STD_LOGIC_VECTOR(15 DOWNTO 0) ;
           Cout         : OUT  STD_LOGIC ) ;
END adder16 ;

ARCHITECTURE dataflow OF adder16 IS
    SIGNAL Sum : STD_LOGIC_VECTOR(16 DOWNTO 0) ;
BEGIN
    Sum  <= ('0' & X) + Y + Cin ;
    S    <= Sum(15 DOWNTO 0) ;
    Cout <= Sum(16) ;
END dataflow ;
```

Addition of Unsigned Numbers (1)

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.numeric_std.all; -- IEEE standard

ENTITY adder16 IS
    PORT( Cin           : IN   STD_LOGIC ;
          X             : IN   STD_LOGIC_VECTOR(15 DOWNTO 0) ;
          Y             : IN   STD_LOGIC_VECTOR(15 DOWNTO 0) ;
          S             : OUT  STD_LOGIC_VECTOR(15 DOWNTO 0) ;
          Cout          : OUT  STD_LOGIC ) ;
END adder16 ;
```

Addition of Unsigned Numbers (2)

```
ARCHITECTURE dataflow OF adder16 IS
   SIGNAL Xu, Yu  : UNSIGNED(15 DOWNTO 0);
   SIGNAL Su      : UNSIGNED(16 DOWNTO 0) ;
BEGIN
   Xu <= unsigned(X);
   Yu <= unsigned(Y);
   Su <= ('0' & Xu) + Yu + unsigned('0' & Cin)
;
   S  <= std_logic_vector(Su(15 DOWNTO 0)) ;
   Cout <= Su(16) ;
END dataflow ;
```

Signed *and* ***unsigned*** *are arrays of std_logic.*

Operations on Signed Numbers

For operations on signed numbers

- Either use
  ieee.numeric_std.all,
  signals of the type SIGNED, and
  conversion std_logic_vector(), signed()

- Or use
  ieee.std_logic_signed.all, and
  signal type STD_LOGIC_VECTOR

Signed/Unsigned Types in numeric_std

- Behave exactly like

  std_logic_vector

- They determine whether a given vector should be treated as a signed or unsigned number.
- Prefer to use

  ieee.numeric_std.all;

- Use either numeric_std or std_logic_unsigned (or signed).
  - Do NOT mix them together.

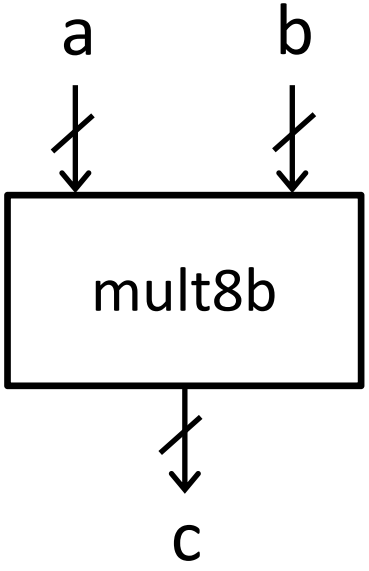
Multipliers

Unsigned vs. Signed Multiplication

Unsigned		Signed	
1111	15	1111	-1
x 1111	x 15	x 1111	x -1
11100001	225	00000001	1

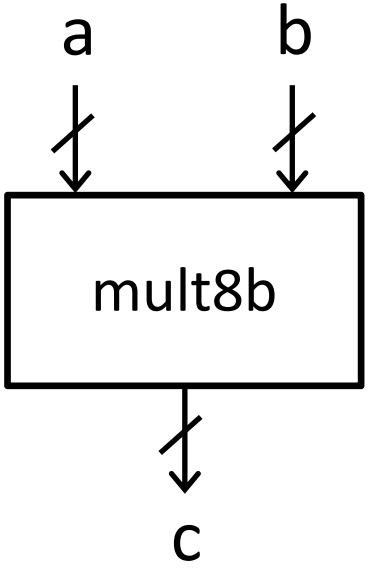
In Xilinx, a multiplier can be implemented either in a DSP or CLB

8x8-bit Unsigned Multiplier




```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

entity mult8b is
        port(…);
end mult8b;

architecture arch of mult8b is
begin
      c <= a * b;
end arch;
```

8x8-bit Signed Multiplier




```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;

entity mult8b is
        port(…);
end mult8b;

architecture arch of mult8b is
begin
      c <= a * b;
end arch;
```

Signed/Unsigned Multiplication

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all ;

entity multiply is
    port(       a : in STD_LOGIC_VECTOR(7 downto 0);
                b : in STD_LOGIC_VECTOR(7 downto 0);
                cu : out STD_LOGIC_VECTOR(15 downto 0);
                cs : out STD_LOGIC_VECTOR(15 downto 0));
end multiply;

architecture dataflow of multiply is
begin
-- signed multiplication
   cs <= std_logic_vector(signed(a)*signed(b));
-- unsigned multiplication
   cu <=
        std_logic_vector(unsigned(a)*unsigned(b));
end dataflow;
```

Multiplication with Constants

- If either *A* or *B* in *A* * *B* is a constant, more efficient implementation with shifts and additions.

$$A * 9$$

can be implemented as

$$A << 3 + A$$

Operators in numeric_std Package

overloaded operator	description	data type of operand a	data type of operand b	data type of result
abs a - a	absolute value negation	signed		signed
a * b a / b a **mod** b a **rem** b a + b a - b	arithmetic operation	unsigned unsigned, natural signed signed, integer	unsigned, natural unsigned signed, integer signed	unsigned unsigned signed signed
a = b a /= b a < b a <= b a > b a >= b	relational operation	unsigned unsigned, natural signed signed, integer	unsigned, natural unsigned signed, integer signed	boolean boolean boolean boolean

Parameterized Models

Design Reuse

- How to design for the 32-bit problem below?

$$O = A + B + C$$

- Create a new 32-bit adder
  - waste of effort
- Reuse previously designed adder
  - but it is 16-bit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity gen_add_w_carry is
    generic(N : integer := 4);
    port(
        a, b : in  std_logic_vector(N - 1 downto 0);
        cout : out std_logic;
        sum  : out std_logic_vector(N - 1 downto 0)
    );
end gen_add_w_carry;

architecture arch of gen_add_w_carry is
    signal a_ext, b_ext, sum_ext : unsigned(N downto 0);
begin
    a_ext   <= unsigned('0' & a);
    b_ext   <= unsigned('0' & b);
    sum_ext <= a_ext + b_ext;
    sum     <= std_logic_vector(sum_ext(N - 1 downto 0));
    cout    <= sum_ext(N);
end arch
```

Instances of Generic Models

```
-- instantiate 8-bit adder
adder_8_unit: work.gen_add_w_carry(arch)
   generic map(N=>8)
   port map(a=>a8, b=>b8, cout=>c8, sum=>sum8));
-- instantiate 16-bit adder
adder_16_unit: work.gen_add_w_carry(arch)
   generic map(N=>16)
   port map(a=>a16, b=>b16, cout=>c16, sum=>sum16));
-- instantiate 4-bit adder
-- (generic mapping omitted, default value 4 used)
adder_4_unit: work.gen_add_w_carry(arch)
   port map(a=>a4, b=>b4, cout=>c4, sum=>sum4));
```

A Word on Generics

- Generics are typically **integer** values
  - In this class, the entity inputs and outputs should be std_logic or std_logic_vector.
  - But the generics should be **integer.**
- Generics are given a default value
  - **GENERIC (N : INTEGER := 16)** ;
  - This value can be overwritten when entity is instantiated as a component
- Generics are very useful when instantiating an often-used component
  - Need a 32-bit register in one place, and 16-bit register in another
  - Can use the same generic code, just configure them differently

Constants – Make Code More Readable

Syntax:

```
constant name : type := value;
```

Examples:

```
constant init_val : STD_LOGIC_VECTOR(3 downto 0) := "0100";
constant ANDA_EXT : STD_LOGIC_VECTOR(7 downto 0) := X"B4";
constant counter_width : INTEGER := 16;
constant buffer_address : INTEGER := x”FFFE”;
constant clk_period : TIME := 20 ns;
constant strobe_period : TIME := 333.333 ms;
```

Constants vs Generics

- Constants:
  - Create symbolic names
  - Make code more readable
  - Declared in packages, entity, or architecture.
  - Cannot create generic designs: still need two design entities for Adder_8b and Adder_32b.
- Generics:
  - Can be passed through design hierarchy through component instantiation
  - Used for creating generic designs: a single design entity Adder for Adder_8b and Adder_32b.

Binary to BCD Conversion

Shift and Add-3 (Double-Dabble)

```
for(i=0; i<8; i++) {
      // add 3 to a column if it is >= 5
      for each column
            if (column >= 5)
                  column += 3;
      // shift binary digits left 1
      Hundred << 1;
      Hundreds[0] = Tens[3];
      Tens << 1;
      Tens[0] = Ones[3];
      Ones << 1;
      Ones[0] = Binary[7];
      Binary << 1;
}
```

Shift and Add-3 (Double-Dabble)

1. If the binary value in any of the BCD columns is 5 or greater, add 3 to that value in that BCD column.
2. Shift the binary number left one bit.
3. If 8 shifts have taken place, the BCD number is in the *Hundreds*, *Tens*, and *Ones* column. Terminate
4. Otherwise, go to 1.

Example:

Hundreds	Tens	Ones	Binary
0000	0000	0000	11110011

Shift and Add-3 (Double-Dabble)

100's	10's	1's	Binary	Operation
0000	0000	0000	10100010	

Shift and Add-3 (Double-Dabble)

100's	10's	1's	Binary	Operation
0000	0000	0000	10100010	
0000	0000	0001	0100010	<< 1

Shift and Add-3 (Double-Dabble)

100's	10's	1's	Binary	Operation
0000	0000	0000	10100010	
0000	0000	0001	0100010	<< 1
0000	0000	0010	100010	<< 1

Shift and Add-3 (Double-Dabble)

100's	10's	1's	Binary	Operation
0000	0000	0000	10100010	
0000	0000	0001	0100010	<< 1
0000	0000	0010	100010	<< 1
0000	0000	0101	00010	<< 1
0000	0000	1000	00010	+3

Shift and Add-3 (Double-Dabble)

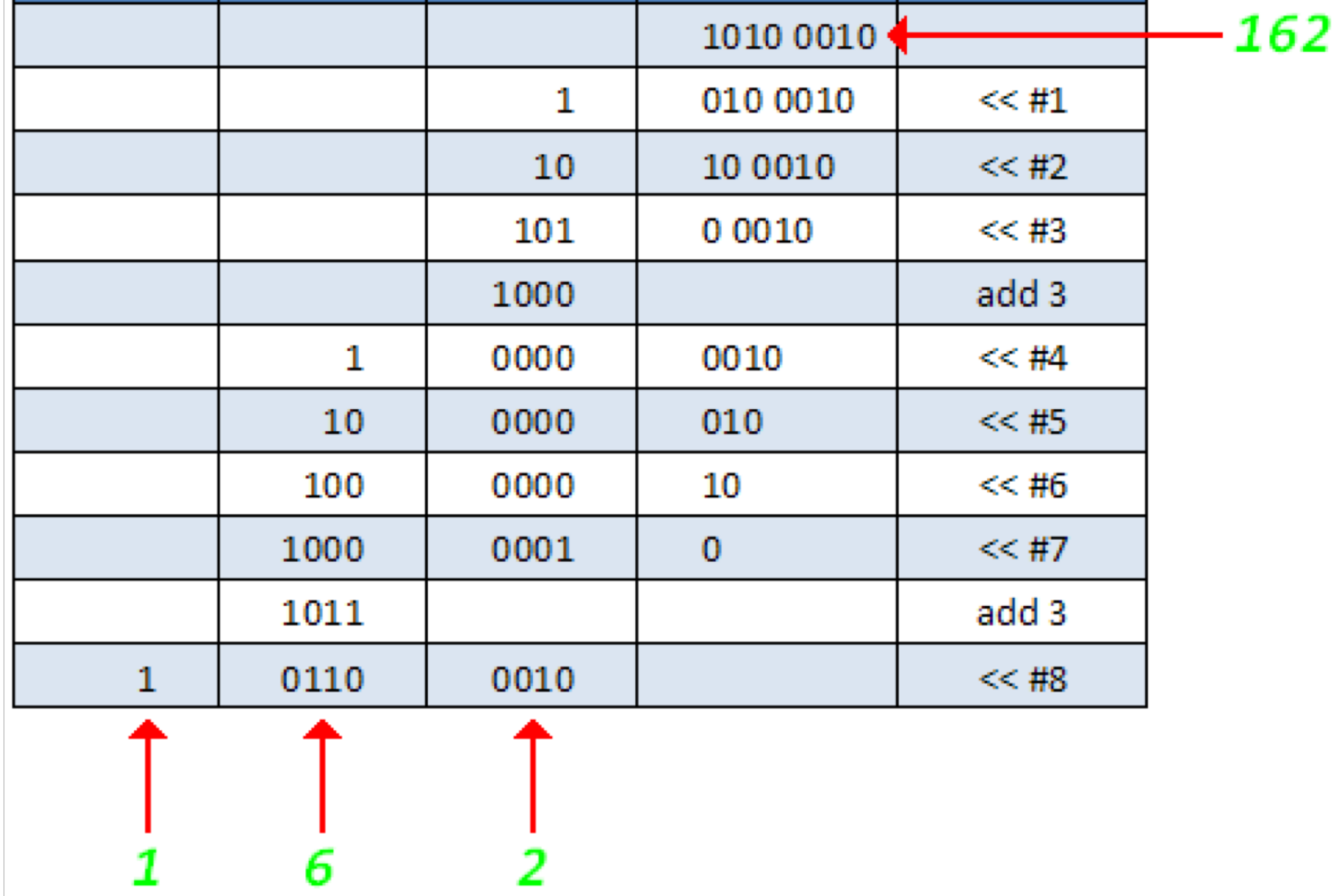
100's	10's	1's	Binary	Operation
0000	0000	0000	10100010	
0000	0000	0001	0100010	<< 1
0000	0000	0010	100010	<< 1
0000	0000	0101	00010	<< 1
0000	0000	1000	00010	+3
0000	0001	0000	0010	<< 1
0000	0010	0000	010	<< 1
0000	0100	0000	10	<< 1

Shift and Add-3 (Double-Dabble)

100's	10's	1's	Binary	Operation
0000	0000	0000	10100010	
0000	0000	0001	0100010	<< 1
0000	0000	0010	100010	<< 1
0000	0000	0101	00010	<< 1
0000	0000	1000	00010	+3
0000	0001	0000	0010	<< 1
0000	0010	0000	010	<< 1
0000	0100	0000	10	<< 1
0000	1000	0001	0	<< 1
0000	1011	0001	0	+3

Shift and Add-3 (Double-Dabble)

100's	10's	1's	Binary	Operation
			1010 0010	
		1	010 0010	<< #1
		10	10 0010	<< #2
		101	0 0010	<< #3
		1000		add 3
	1	0000	0010	<< #4
	10	0000	010	<< #5
	100	0000	10	<< #6
	1000	0001	0	<< #7
	1011			add 3
1	0110	0010		<< #8



Goto wiki for more information and VHDL implementation

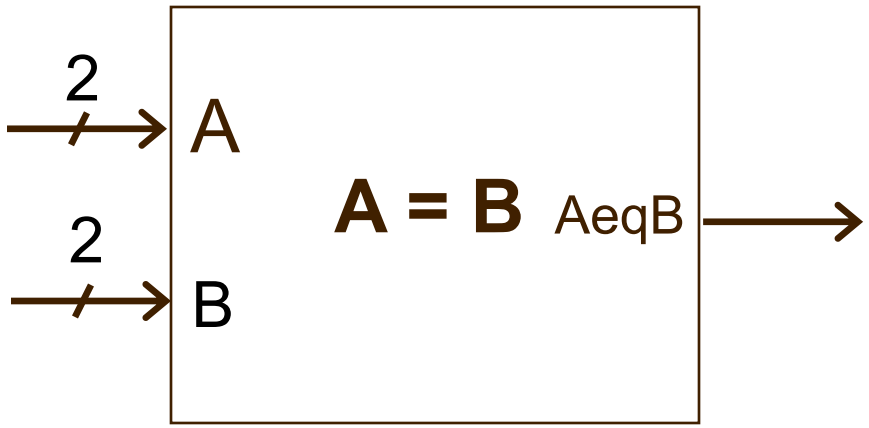
Summary

- More concurrent statements for DF modeling
  - describing routing structures
- Modeling of basic combinational circuit blocks
  - Adders, multipliers, muxes, encoder/decoder
- Generic design modeling
  - Using VHDL generics

Backup

Comparators

2-bit Number Comparator



4-bit Unsigned Number Comparator

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all ;

entity compare is
   port( A, B      : in     STD_LOGIC_VECTOR(1 downto 0);
         AeqB      : out    STD_LOGIC );
end compare ;

architecture dataflow of compare is
begin
   AeqB <=   '1' when A = B else
             '0';
end dataflow ;
```

4-bit Unsigned Number Comparator

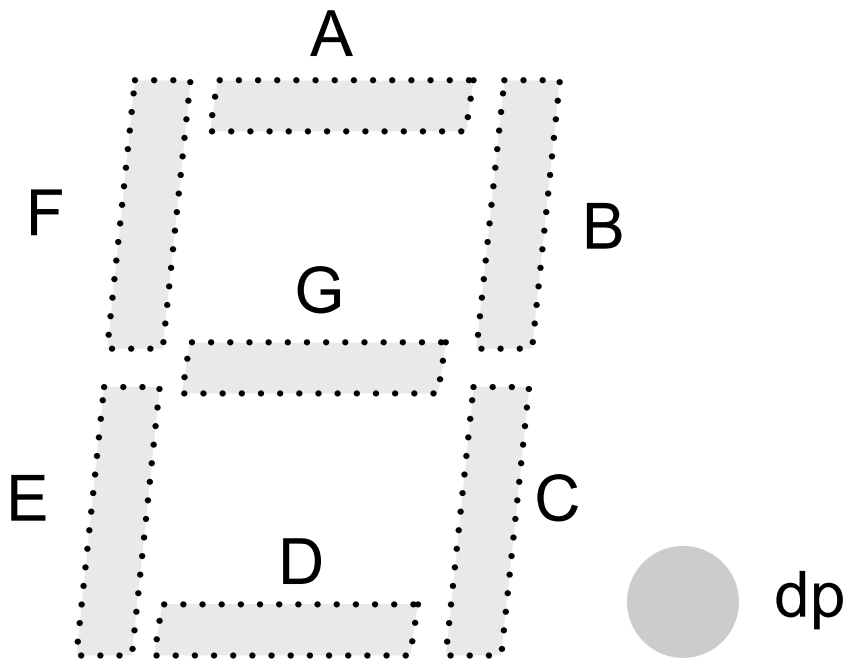
```
library ieee;
use ieee.std_logic_1164.all;

entity compare is
   port( A, B      : in     STD_LOGIC_VECTOR(1 downto 0);
         AeqB      : out    STD_LOGIC );
end compare ;
-- Create a different model?
architecture dataflow of compare is
begin




end dataflow ;
```

4-bit Signed Number Comparator

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity compare is
   port( A, B        : in  STD_LOGIC_VECTOR(1 downto 0);
         AeqB        : out STD_LOGIC);
end compare ;

architecture dataflow of compare is
begin
   AeqB  <= '1' when A = B else
            '0';
end dataflow ;
```

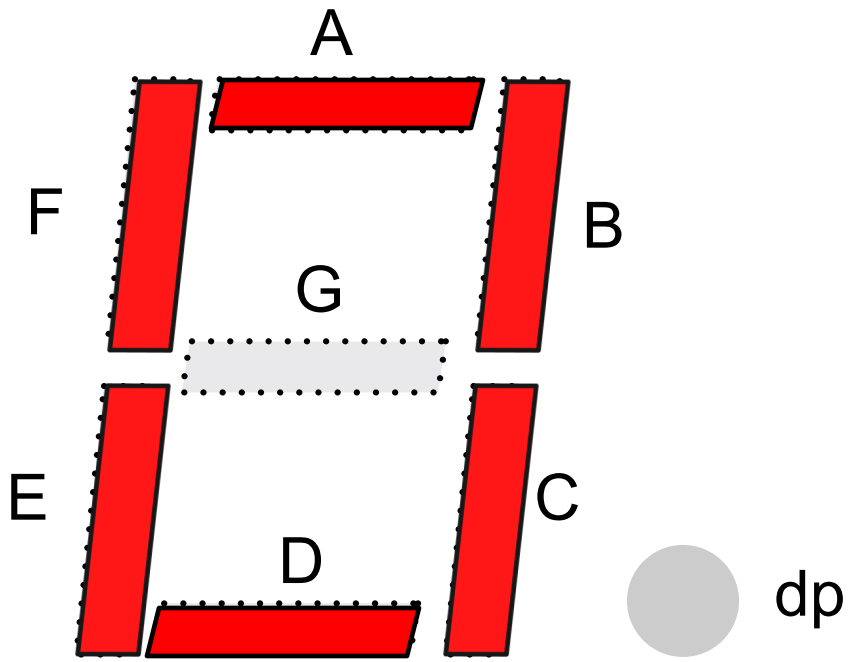
Hexadecimal to 7-Segment Display

7-Segment Display



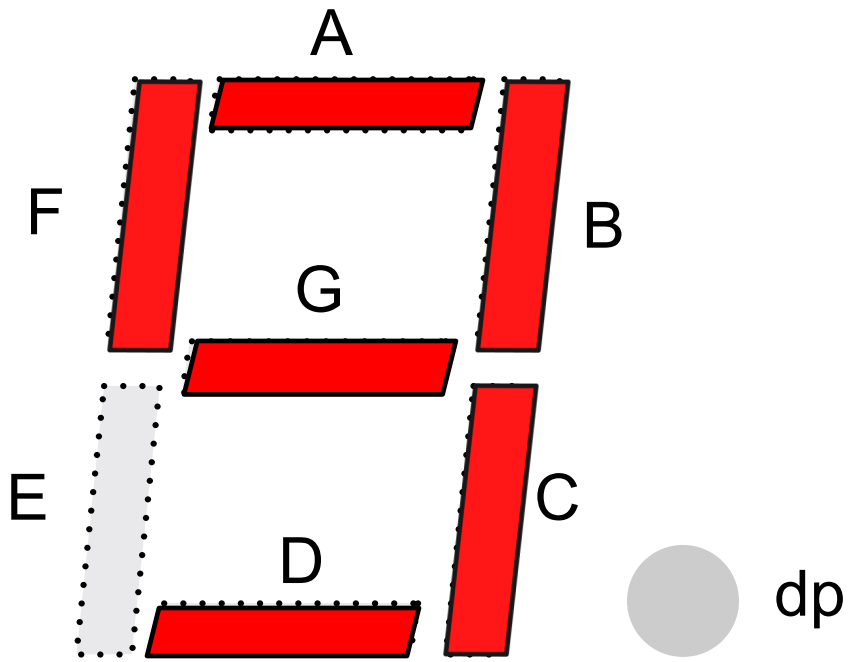
To illuminate a segment, the corresponding control signal should be driven low.

7-Segment Display



To illuminate a segment, the corresponding control signal should be driven low – A = ‘0,’ ..., F = ‘0’, G = ‘1’

7-Segment Display

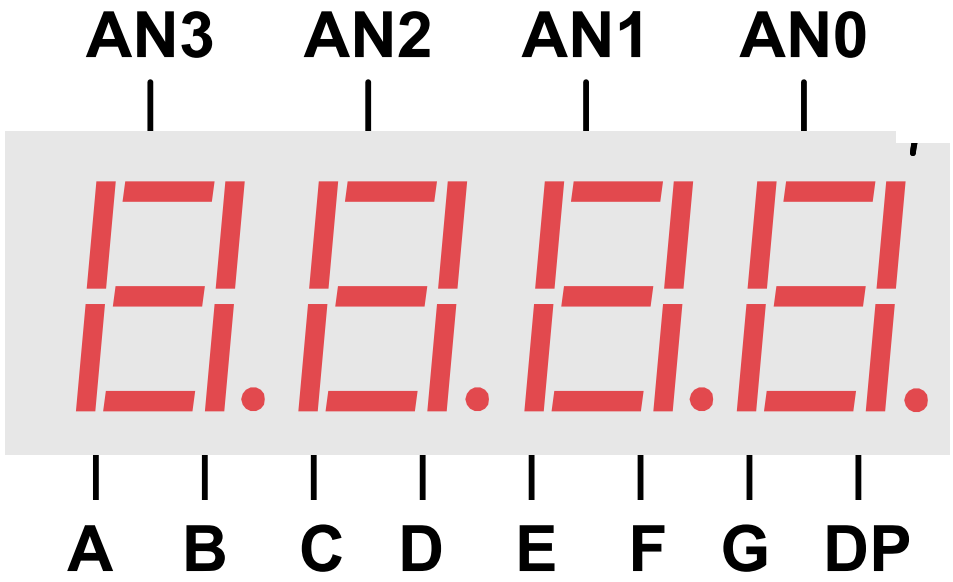


To illuminate a segment, the corresponding control signal should be driven low – A = ‘0,’ ..., E = ‘1’, F = ‘0’, G = ‘0’

Hex to 7-Segment

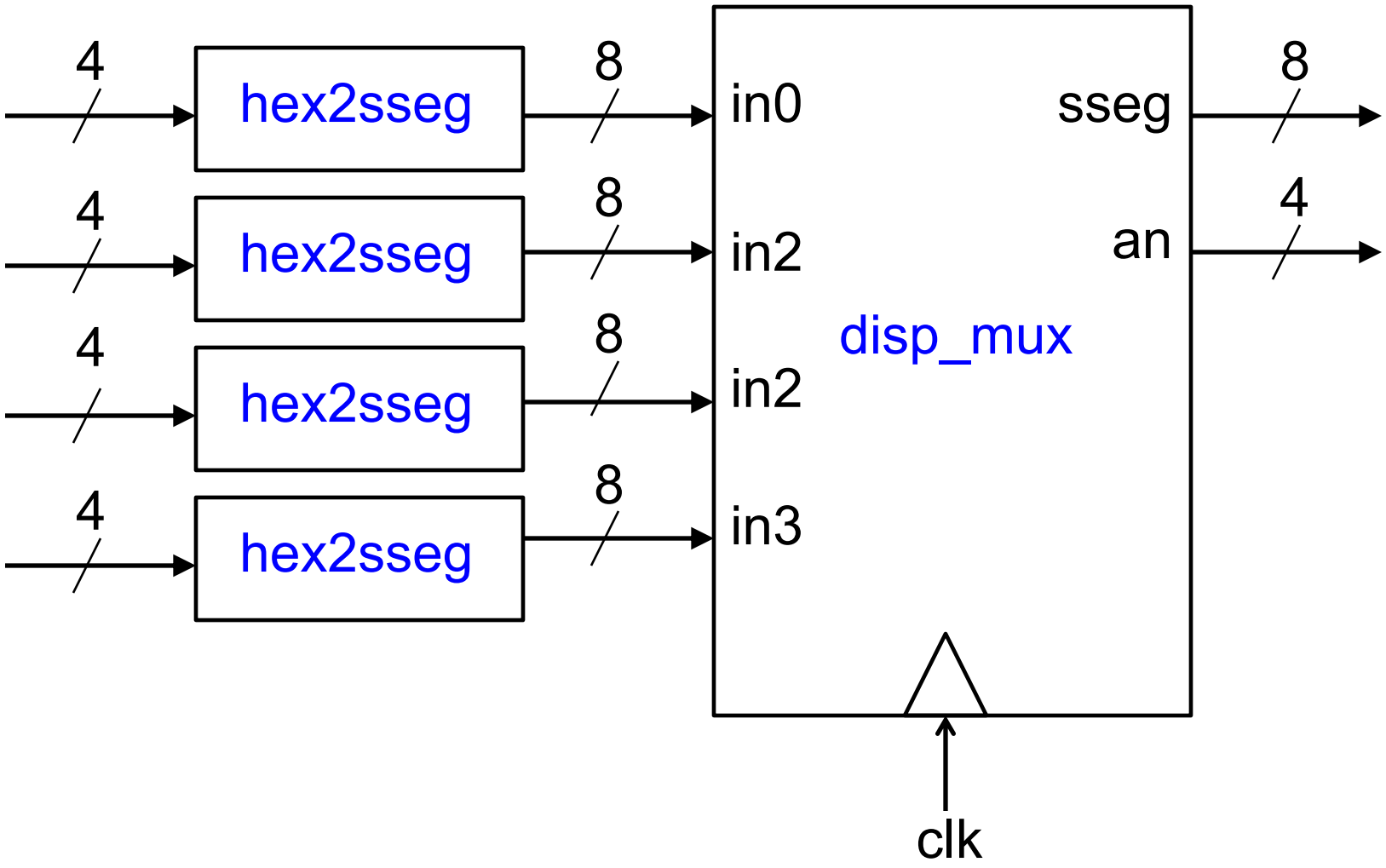
Hex Input	7-Segment Control GFE...BCA
0000 (0)	1000000
0001 (1)	1111001
0010 (2)	0100100
0011 (3)	0110000
0100 (4)	0011001
0101 (5)	0010010
0110 (6)	0000010
0111 (7)	1111000
1000 (8)	0000000
1001 (9)	0010000
1010 (A)	0001000

7-Segment Display



- All four displays share common segment control signals.
- Only one display can be illuminated at a time when signal *ANx* is driven high.

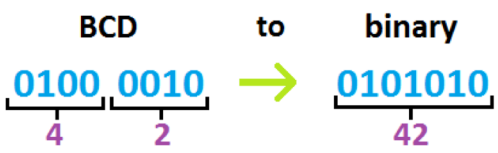
7-Segment Display Controller



BCD to Binary Conversion

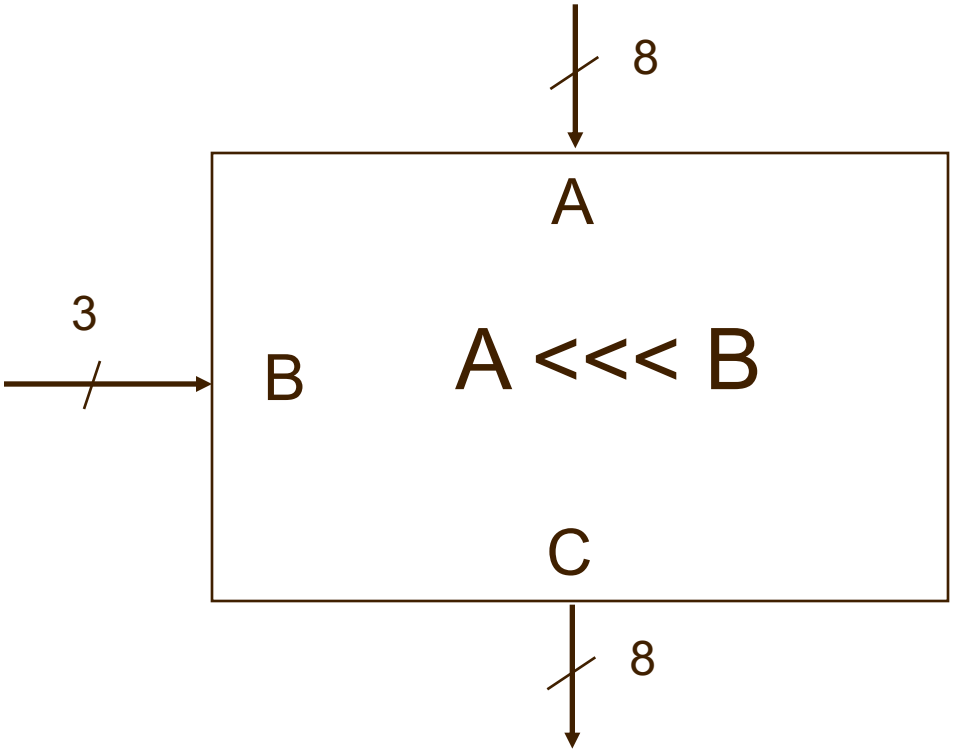
1. Right shift bcd1, with the LSB shifting to the MSB of bcd0.
2. Right shift bcd0, with the LSB shifting to the MSB of bin.
3. If bcd0 is now > 4, subtract 3
4. repeat steps 1-3, 7 times.

BCD to Binary Conversi



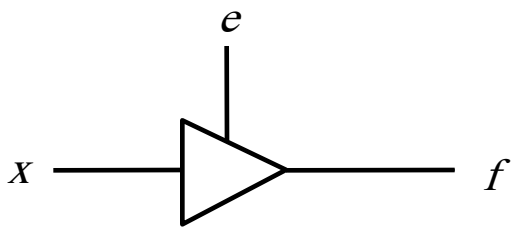
Iteration:	BCD	bin
1	01000010	0000000
>>	0100001	0000000
1 ≯ 4		
2		
>>	010000	1000000
0 ≯ 4		
3		
>>	01000	0100000
8 > 4, -3	00101	0100000
4		
>>	0010	1010000
2 ≯ 4		
5		
>>	001	0101000
1 ≯ 4		
6		
>>	00	1010100
0 ≯ 4		
7		
>>	0	0101010
0 ≯ 4		

8-bit Variable Rotator Left



Tri-State Buffers

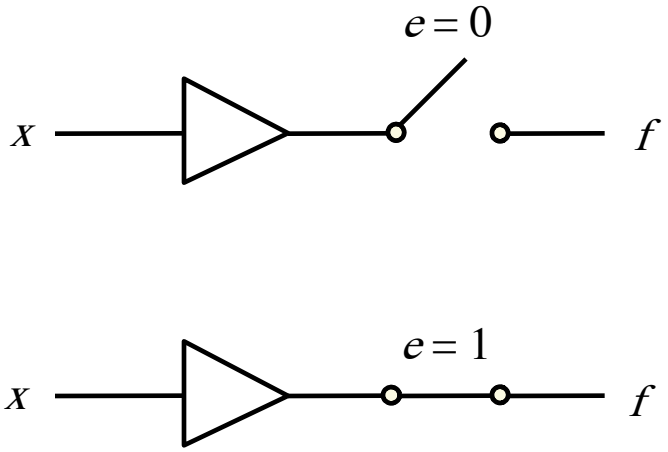
Tri-State Buffers



(a) A tri-state buffer

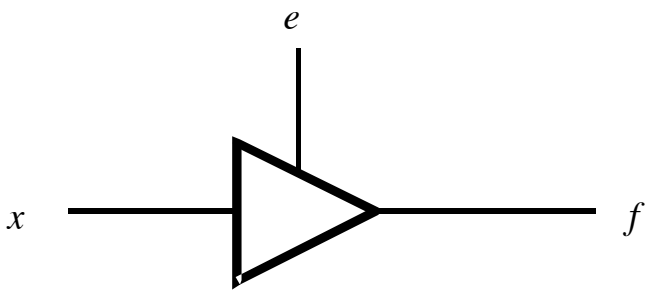
e	x	f
0	0	Z
0	1	Z
1	0	0
1	1	1

(c) Truth table

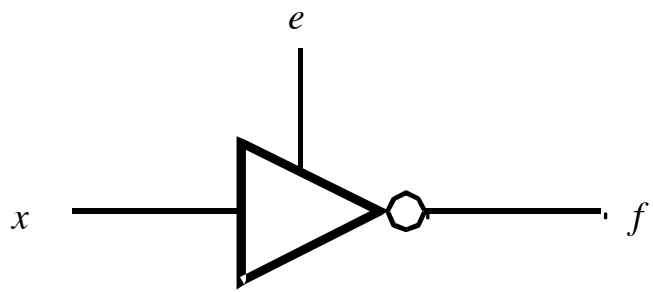


(b) Equivalent circuit

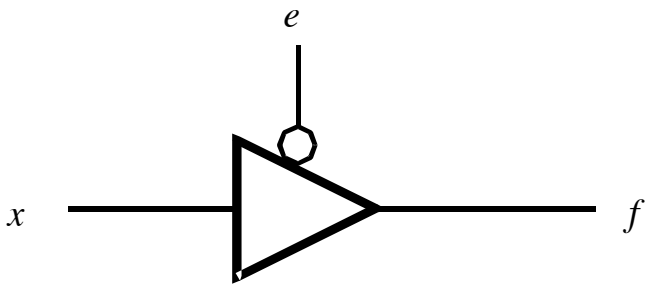
Four types of Tri-state Buffers



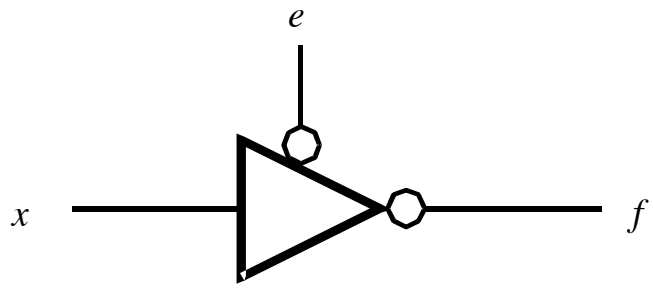
(a)



(b)



(c)



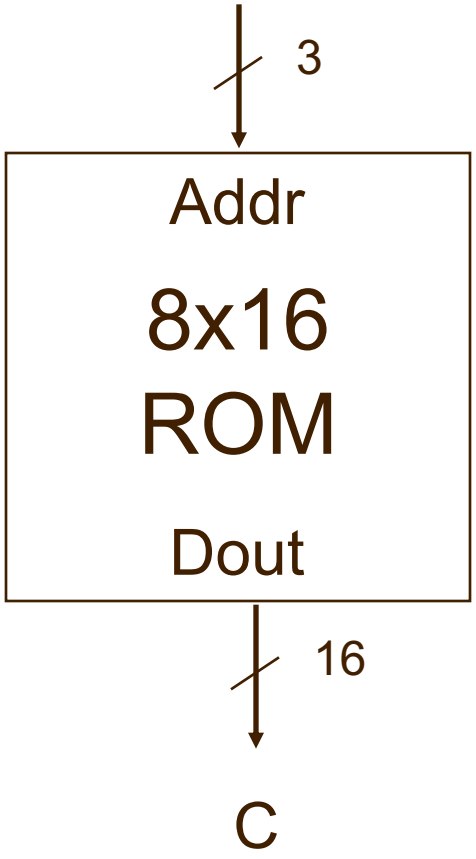
(d)

Tri-state Buffer – Example (1)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

entity tri_state is
   port(    ena, input :      IN STD_LOGIC;
            output     :      OUT STD_LOGIC);
end tri_state;

architecture dataflow of tri_state is
begin
    output <=   input when (ena = '1') else
                'Z';
end dataflow;
```

ROM

ROM 8x16 example (1)



ROM 8x16 example (2)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

entity rom is
   port (    Addr : in   STD_LOGIC_VECTOR(2 downto 0);
             Dout : out  STD_LOGIC_VECTOR(15 downto 0));
end rom;

-- architecture body is defined on the next slide
```

ROM 8x16 example (3)

```
architecture dataflow of rom is
   signal temp: integer range 0 to 7;
   type vector_array is array(0 to 7) of
                          std_logic_vector(15 downto 0);
   constant memory : vector_array :=  (    X”800A",
                                           X"D459",
                                           X"A870",
                                           X"7853",
                                           X"650D",
                                           X"642F",
                                           X"F742",
                                           X"F548");

begin
   temp     <= to_integer(unsigned(Addr));
   Dout     <= memory(temp);
end dataflow;
```