ECE 448
Lecture 3

Combinational-Circuit Building Blocks

Data Flow Modeling of Combinational Logic

Reading

Required

- P. Chu, *FPGA Prototyping by VHDL Examples*

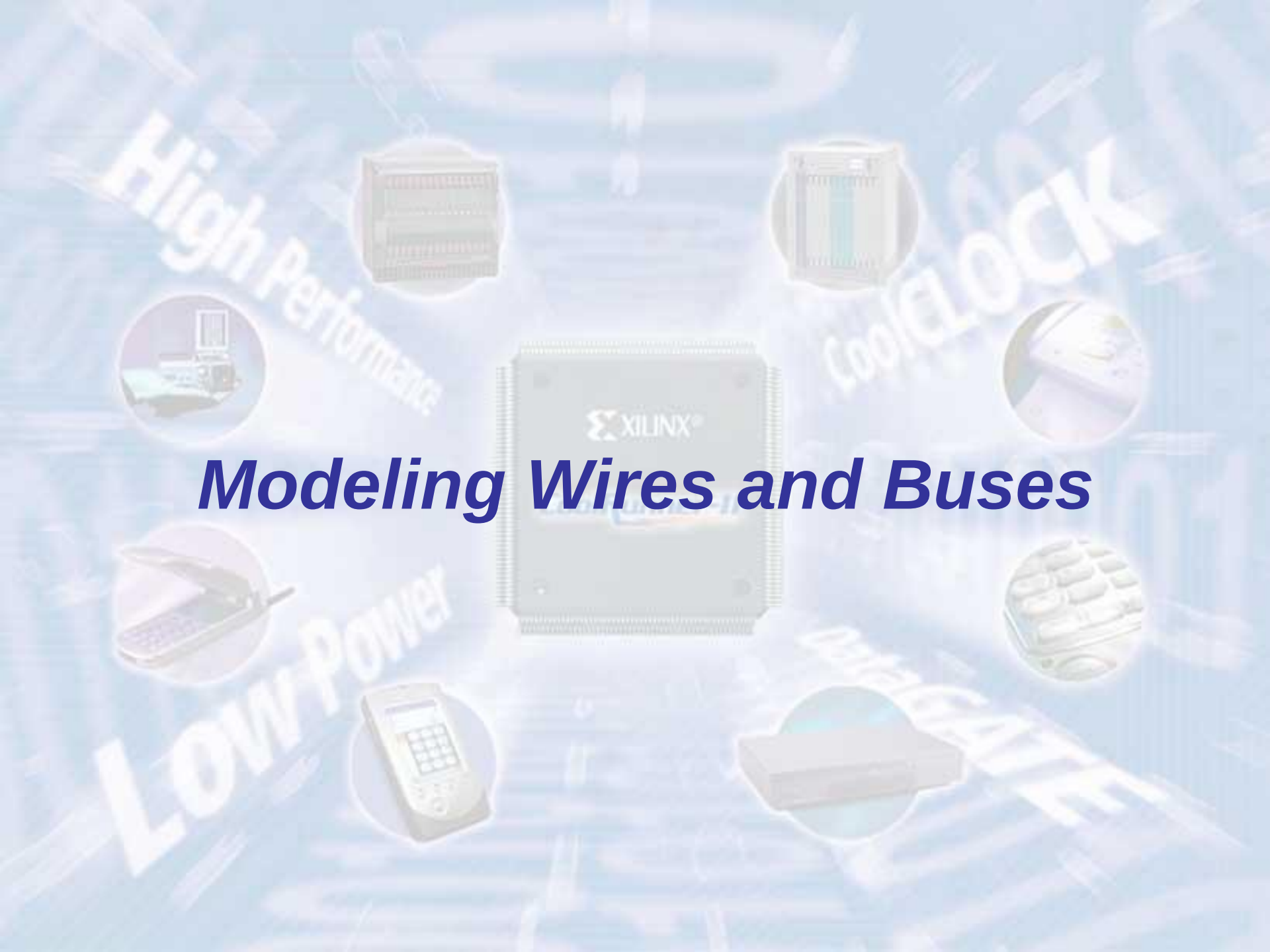
  Chapter 3, RT-level combinational circuit

Recommended

- S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design*

  Chapter 6, Combinational-Circuit Building Blocks

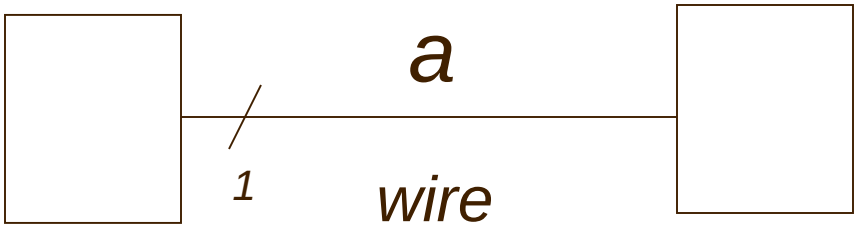
  Chapter 5.5, Design of Arithmetic Circuits Using CAD Tools



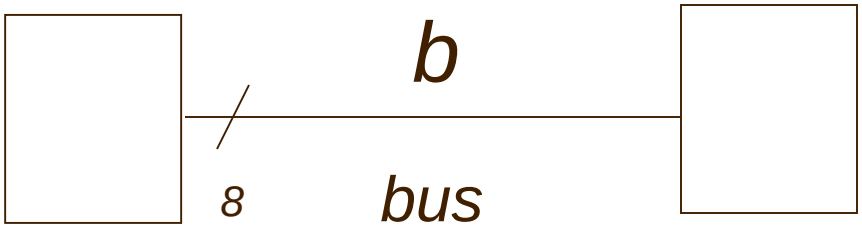
Modeling Wires and Buses

Signals

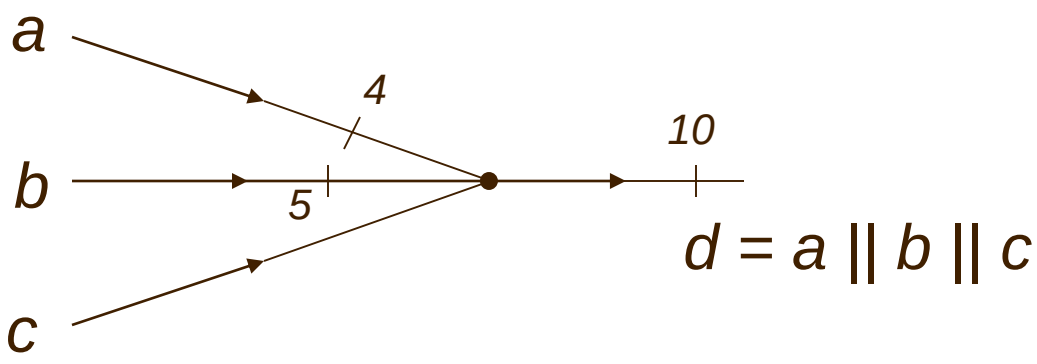
SIGNAL a : STD_LOGIC;



SIGNAL b : STD_LOGIC_VECTOR(7 DOWNTO 0);

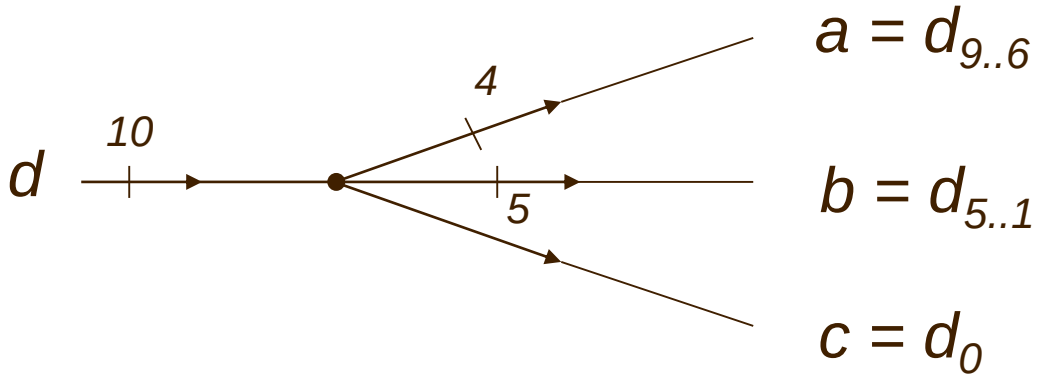


Merging wires and buses

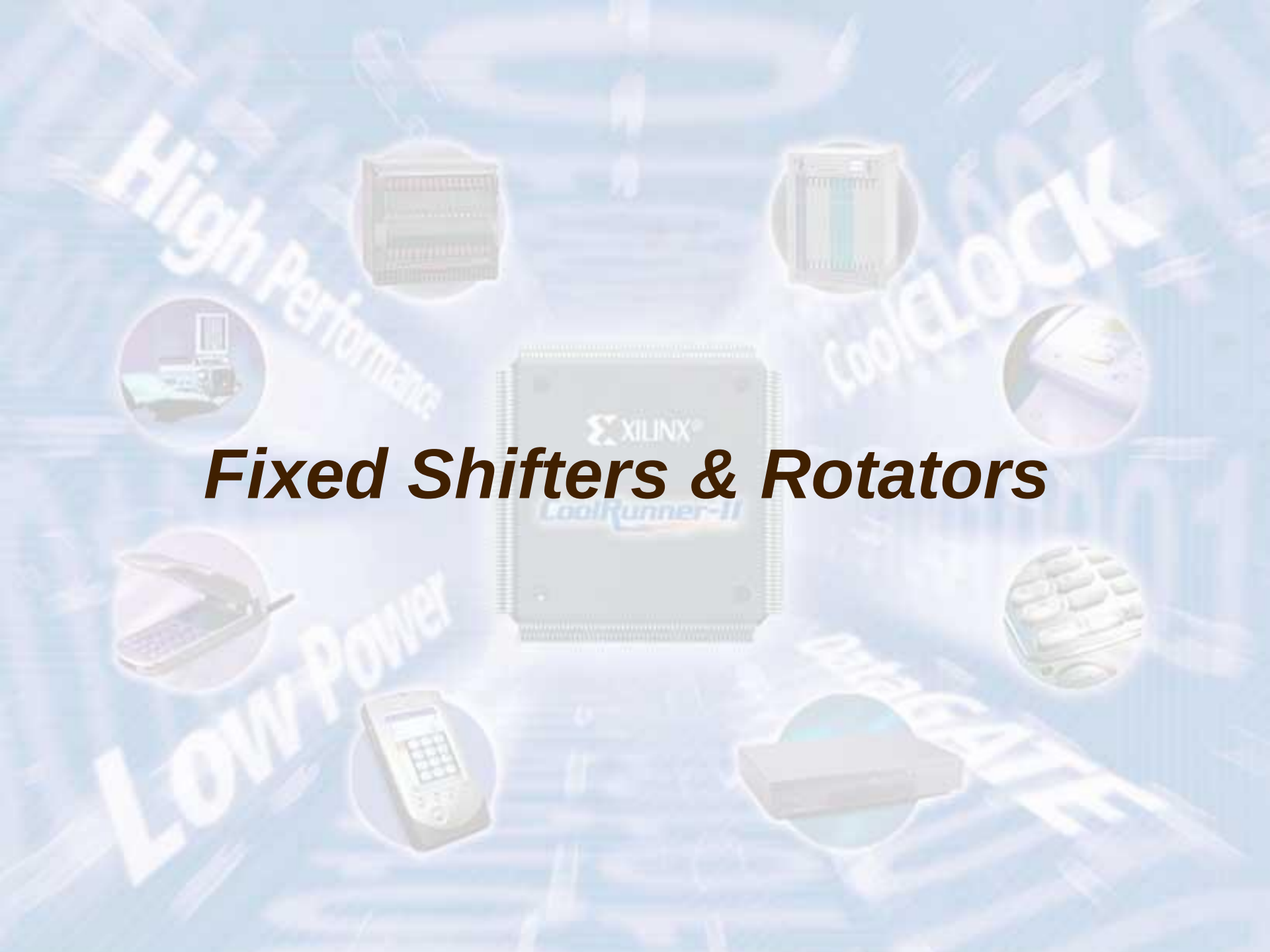



```
SIGNAL a: STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL b: STD_LOGIC_VECTOR(4 DOWNTO 0);
SIGNAL c: STD_LOGIC;
SIGNAL d: STD_LOGIC_VECTOR(9 DOWNTO 0);

d <= a & b & c;
```

Splitting buses



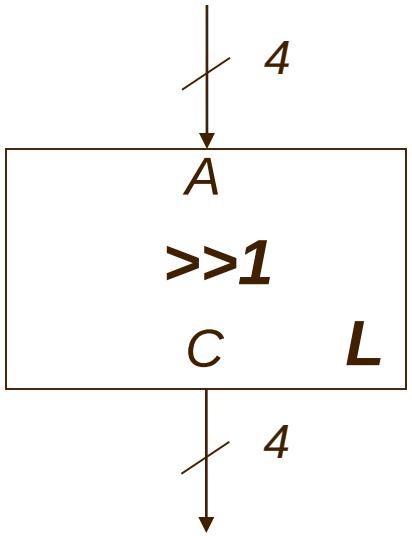

```
SIGNAL a: STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL b: STD_LOGIC_VECTOR(4 DOWNTO 0);
SIGNAL c: STD_LOGIC;
SIGNAL d: STD_LOGIC_VECTOR(9 DOWNTO 0);

a <= d(9 downto 6);
b <= d(5 downto 1);
c <= d(0);
```

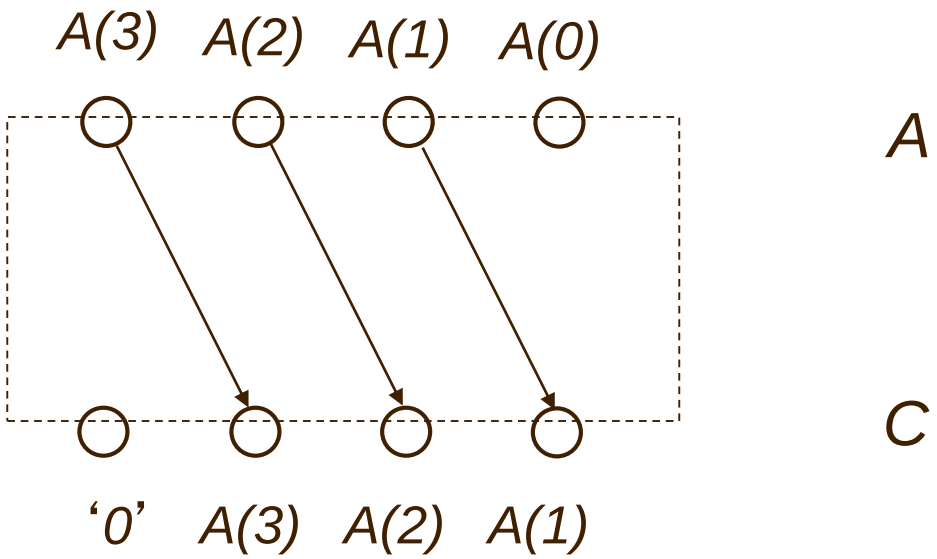


Fixed Shifters & Rotators

Fixed Logical Shift Right in VHDL

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

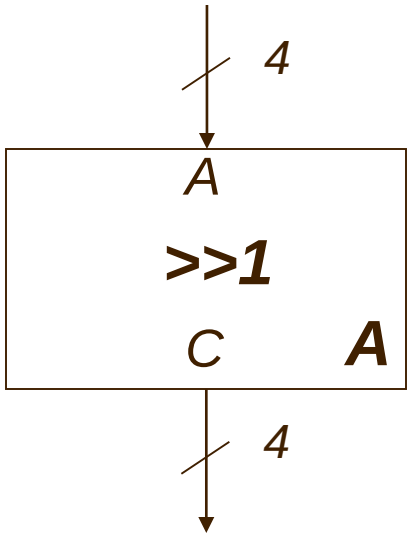


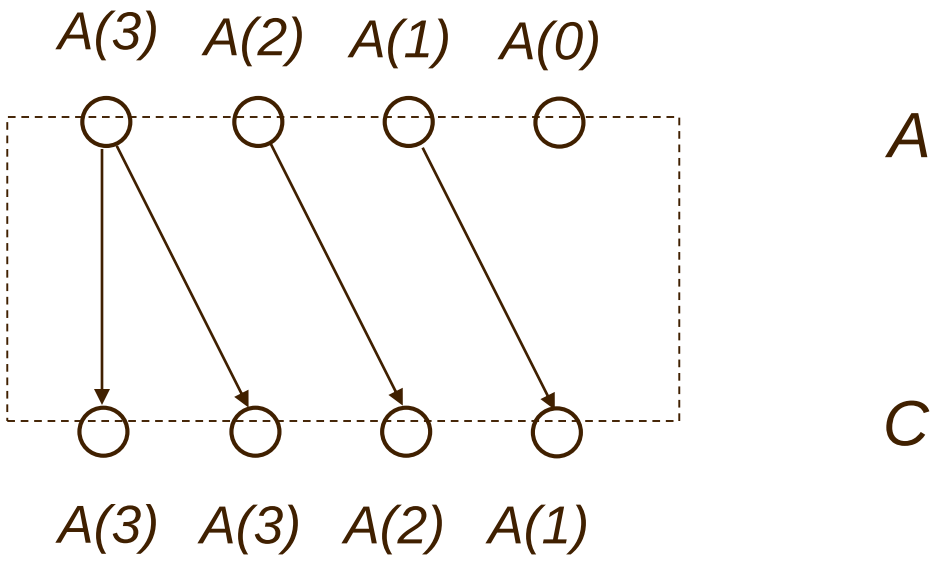


C <= '0' & A(3 downto 1);

Fixed Arithmetic Shift Right in VHDL

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

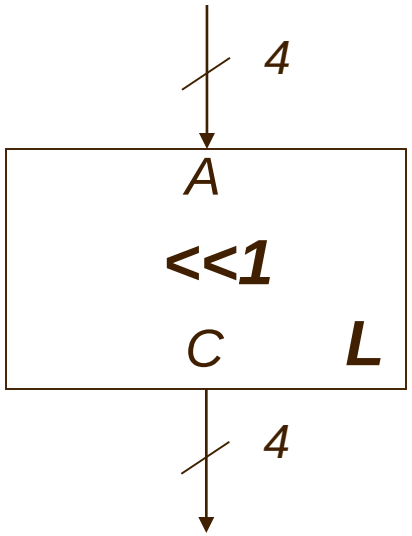


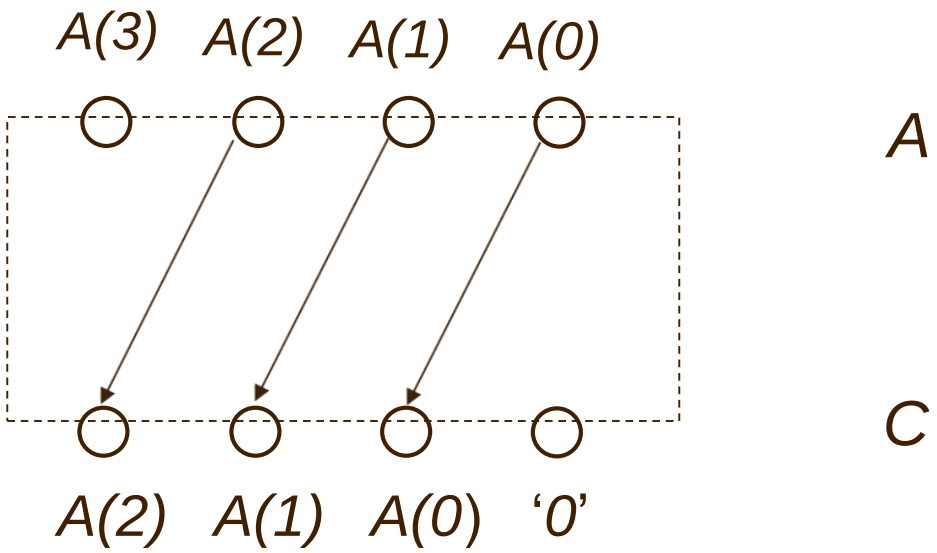


C = A(3) & A(3 downto 1);

Fixed Logical Shift Left in VHDL

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

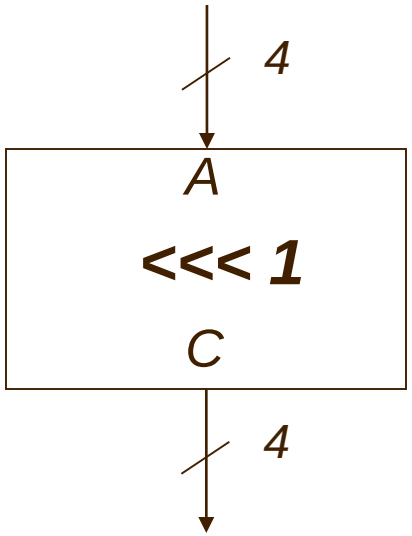


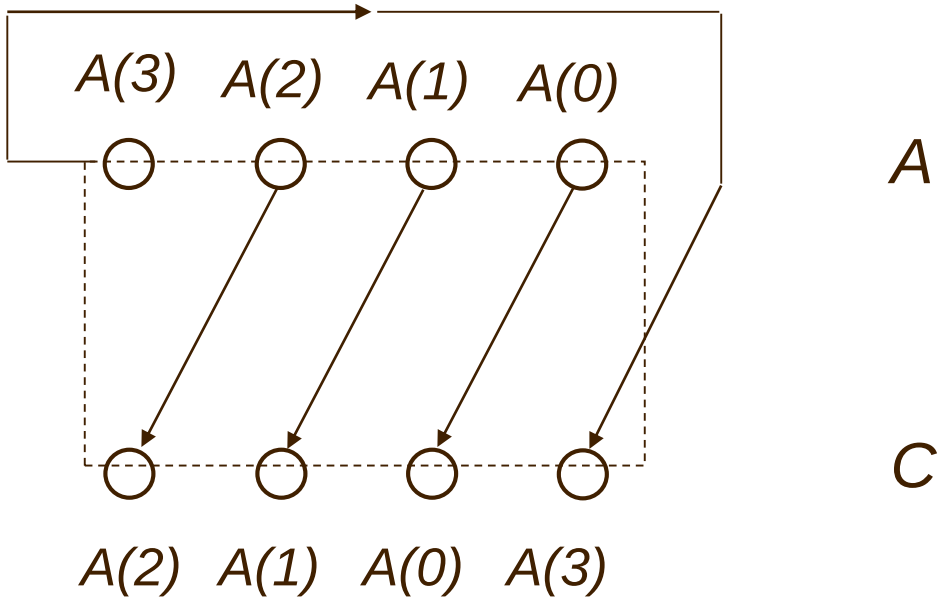


C = A(2 downto 0) & '0';

Fixed Rotation Left in VHDL

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





C = A(2 downto 0) & A(3);

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 ;*

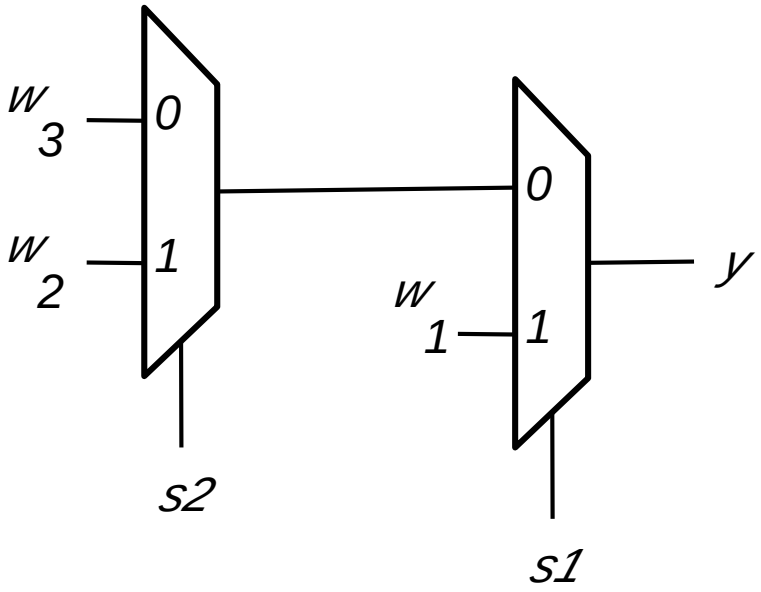
equivalent to

$y = (ab + c)d$

Correct

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

Cascade of two multiplexers



VHDL:

```
y <= w1  WHEN s1 =  '1' ELSE
     w2  WHEN s2 =  '1' ELSE
     w3 ;
```

Priority of logic and relational operators

compare a = bc

Incorrect

… when a = b ***and*** c else …

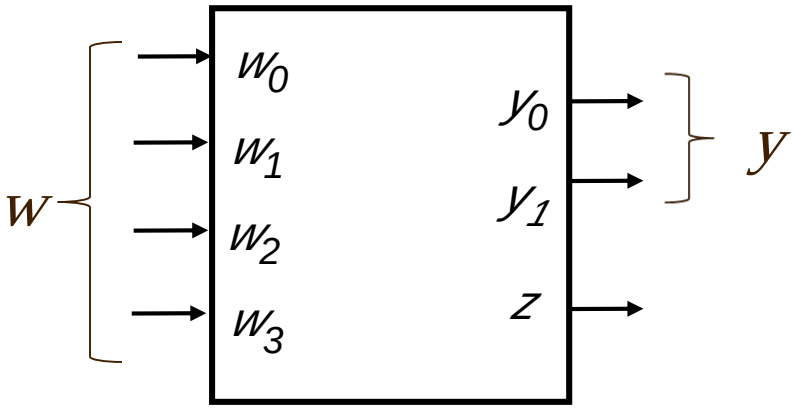
equivalent to

… when (a = b) ***and*** c else …

Correct

… when a = (b ***and*** c) else …

Priority Encoder



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

```
y <= "11" WHEN w(3) = '1' ELSE
     "10" WHEN w(2) = '1' ELSE
     "01" WHEN w(1) = '1' ELSE
     "00" ;
```

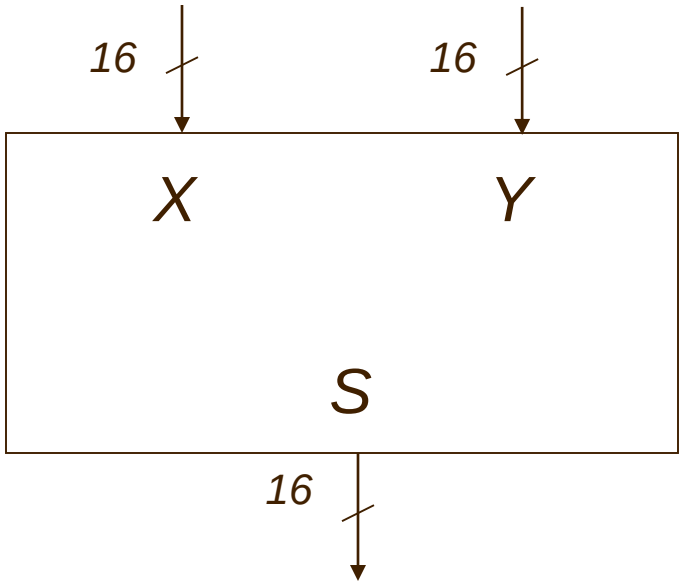
```
z <= '0' WHEN w = "0000" ELSE '1' ;
```

VHDL code for a Priority Encoder entity

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

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" ;
    z <= '0' WHEN w = "0000" ELSE '1' ;
END dataflow ;
```

Adders

Adder mod 2^{16}



VHDL code for an Adder mod 2^{16}

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

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

ARCHITECTURE dataflow OF adder16 IS
BEGIN
      S <= std_logic_vector(unsigned(X) + unsigned(Y));
END dataflow ;
```

Signed and Unsigned Types

Behave** exactly **like

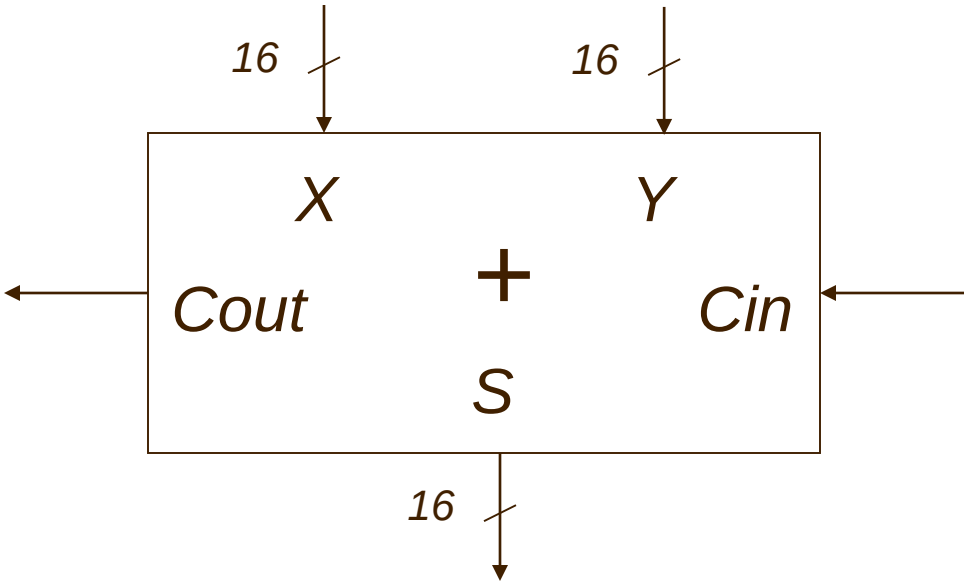
STD_LOGIC_VECTOR

plus, they determine whether a given vector should be treated as a signed or unsigned number.

Require

USE ieee.numeric_std.all;

16-bit Unsigned Adder



Addition of Unsigned Numbers (1)

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


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 (3)

```
ARCHITECTURE dataflow OF adder16 IS
        signal USum: unsigned(16 DOWNTO 0) ;
        signal UCin: unsigned(0 downto 0);
BEGIN
        UCin(0) <= Cin;
        USum <= unsigned('0' & X) + unsigned(Y) + UCin;
        S <= std_logic_vector(USum(15 downto 0));
        Cout <= USum(16) ;
END dataflow ;
```

IEEE numeric_std package

- *How to infer arithmetic operators?*
- *In standard VHDL:*

  ***signal** a, b, sum: integer;*

  . . .

  sum <= a + b;
- What's wrong with integer data type?

- *IEEE numeric_std package: define integer as a an array of elements of std_logic*
- *Two new data types: unsigned, signed*
- *The array interpreted as an unsigned or signed binary number*
- *E.g.,*

  ***signal** x, y: signed(15 **downto** 0);*
- *Need invoke package to use the data type*

  ***library** ieee;*

  ***use** ieee.std_logic_1164.**all**;*

  ***use** ieee.numeric_std.**all**;*

Overloaded operators in IEEE 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

- *E.g.,*

```
signal a, b, c, d: unsigned(7 downto 0);
. . .
a <= b + c;
d <= b + 1;
e <= (5 + a + b) - c;
```

Type conversion

- *Std_logic_vector, unsigned, signed are defined as an array of element of std_logic*
- *They considered as three different data types in VHDL*
- *Type conversion between data types:*
  - *type conversion function*
  - Type casting (for “closely related” data types)

Type conversion between number-related data types

data type of a	to data type	conversion function / type casting
unsigned, signed	std_logic_vector	std_logic_vector(a)
signed, std_logic_vector	unsigned	unsigned(a)
unsigned, signed	std_logic_vector	std_logic_vector(a)
unsigned, signed	integer	to_integer(a)
natural	unsigned	to_unsigned(a, size)
integer	signed	to_signed(a, size)

- *E.g.*

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
. . .
signal s1, s2, s3, s4, s5, s6: std_logic_vector(3 downto 0);
signal u1, u2, u3, u4, u6, u7: unsigned(3 downto 0);
signal sg: signed(3 downto 0);
```

– *Ok*

```
u3 <= u2 + u1;  --- ok, both operands unsigned
u4 <= u2 + 1;   --- ok, operands unsigned and natural
```

– *Wrong*

```
u5 <= sg;  -- type mismatch
u6 <= 5;   -- type mismatch
```

– *Fix*

```
u5 <= unsigned(sg);     -- type casting
u6 <= to_unsigned(5,4); -- conversion function
```

– *Wrong*

```
u7 <= sg + u1;   -- + undefined over the types
```

– *Fix*

```
u7 <= unsigned(sg) + u1; -- ok, but be careful
```

– *Wrong*

```
s3 <= u3;  -- type mismatch
s4 <= 5;   -- type mismatch
```

– *Fix*

```
s3 <= std_logic_vector(u3); -- type casting
s4 <= std_logic_vector(to_unsigned(5,4));
```

– *Wrong*

s5 <= s2 + s1; + undefined over std_logic_vector

s6 <= s2 + 1; + undefined

– *Fix*

s5 <= std_logic_vector(unsigned(s2) + unsigned(s1));

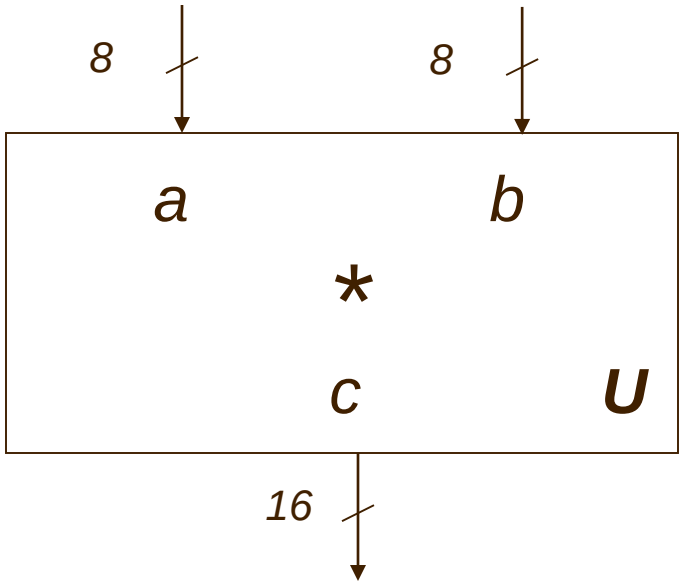
s6 <= std_logic_vector(unsigned(s2) + 1);

Multipliers

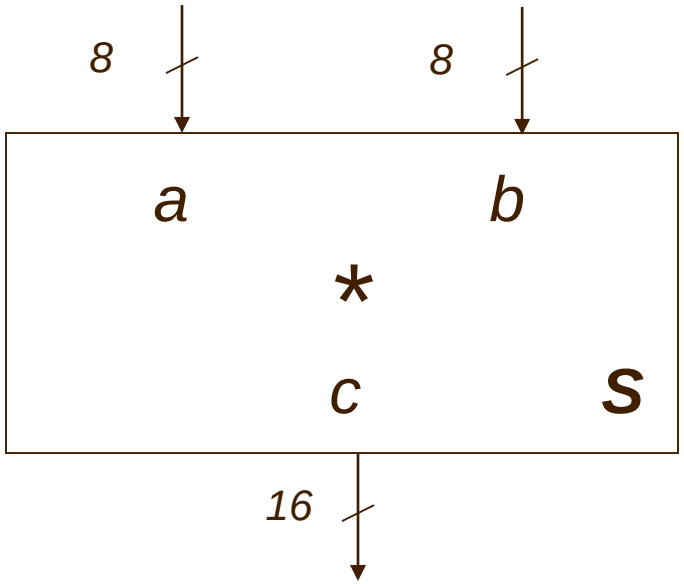
Unsigned vs. Signed Multiplication

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

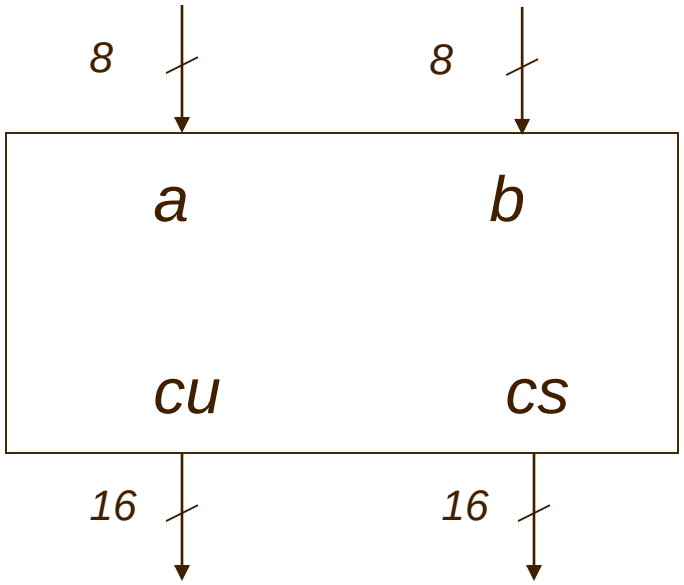
8x8-bit Unsigned Multiplier



8x8-bit Signed Multiplier

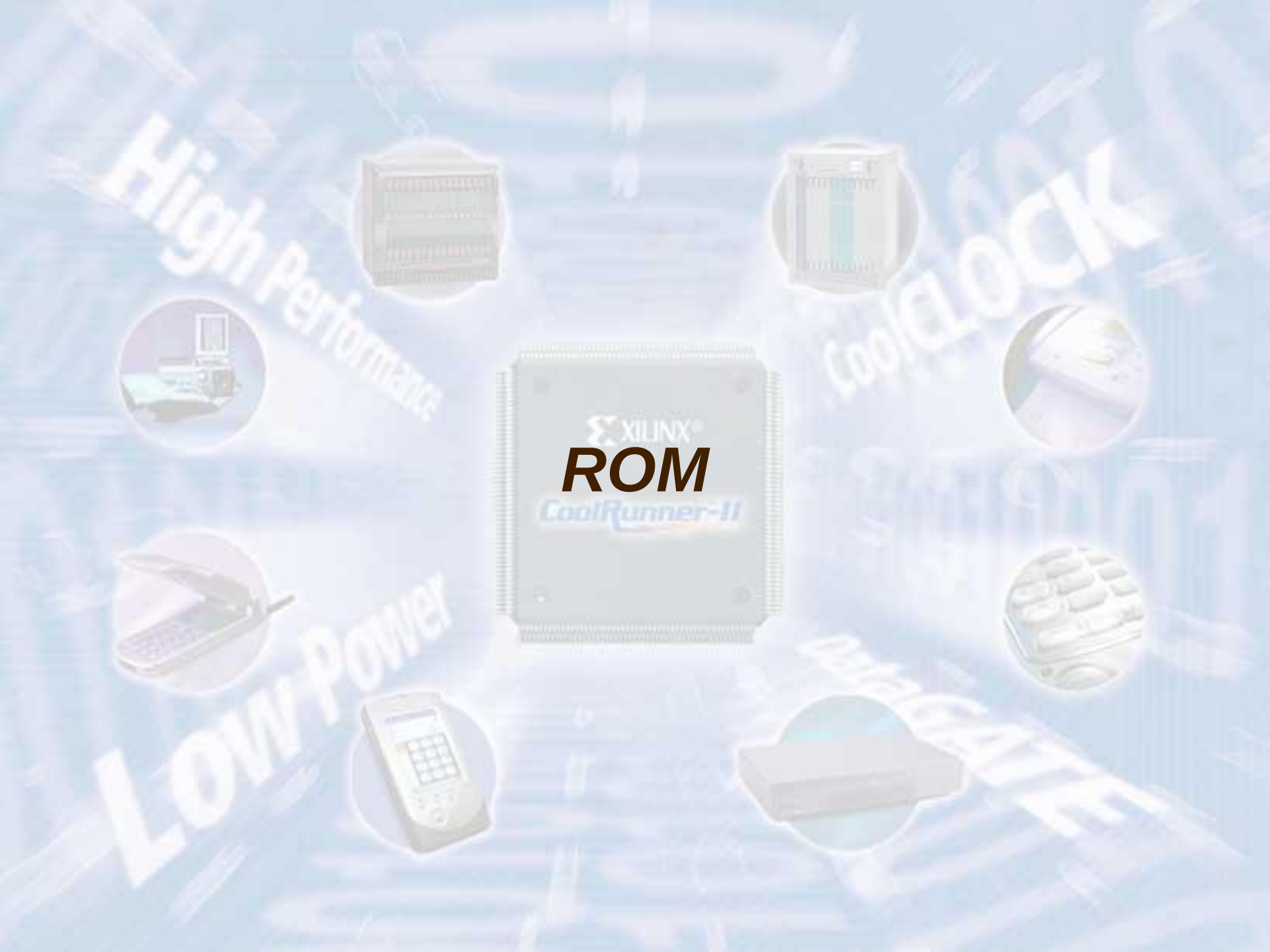


8x8-bit Unsigned and Signed Multiplier

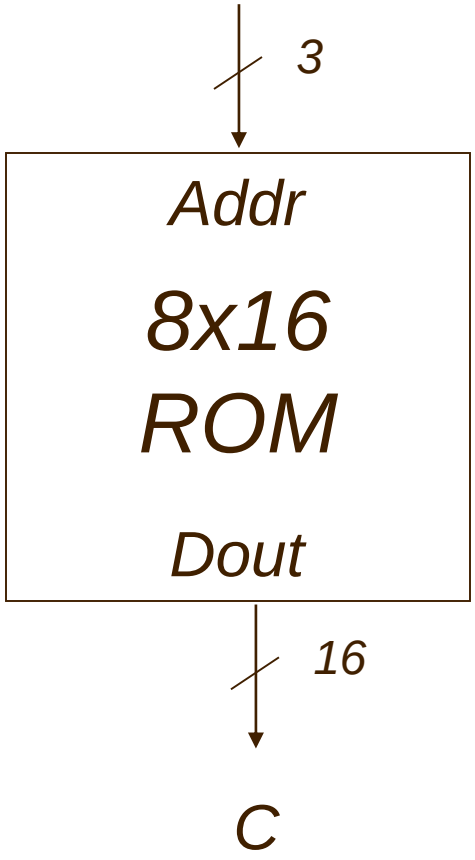


Multiplication of signed and unsigned numbers

```
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;
```


ROM
XILINX
CoolRunner-II
High Performance
Low Power
Cool Clock

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;
```

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;
```

ROM 8x16 example (4)

```
ARCHITECTURE dataflow OF rom IS
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

   Dout <= memory(to_integer(unsigned(Addr)));

END dataflow;
```

ECE 448
Lecture 4

Sequential-Circuit Building Blocks

Constants & Packages

Mixing Description Styles

Reading

Required

- P. Chu, *FPGA Prototyping by VHDL Examples*

  Chapter 4, Regular Sequential Circuit

Recommended

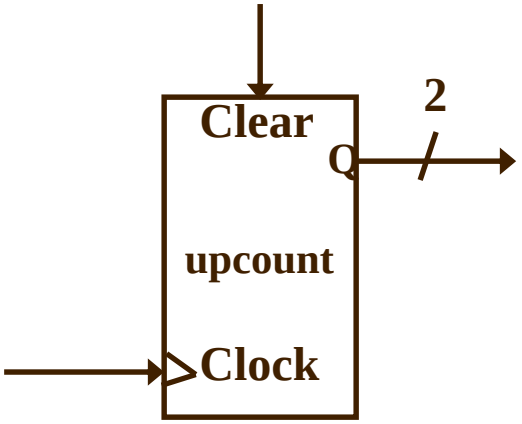
- S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design*

  Chapter 7, Flip-Flops, Registers, Counters, and a Simple Processor

Counters

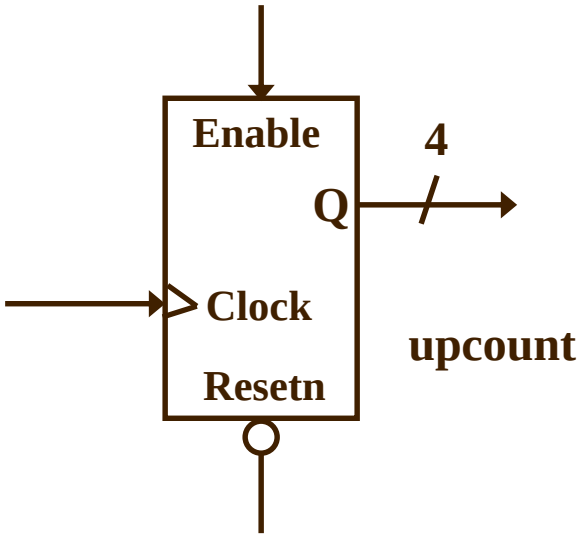
2-bit up-counter with synchronous reset

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.numeric_std.all ;
ENTITY upcount IS
        PORT (      Clear, Clock      : IN          STD_LOGIC ;
                    Q                 : OUT     STD_LOGIC_VECTOR(1 DOWNTO 0) ) ;
END upcount ;

ARCHITECTURE behavioral OF upcount IS
   SIGNAL Count : unsigned(1 DOWNTO 0);
BEGIN
        upcount: PROCESS ( Clock )
        BEGIN
                IF rising_edge(Clock) THEN
                        IF Clear = '1' THEN
                                Count <= "00" ;
                        ELSE
                                Count <= Count + 1 ;
                        END IF ;
                END IF;
        END PROCESS;
       Q <= std_logic_vector(Count);
END behavioral;
```



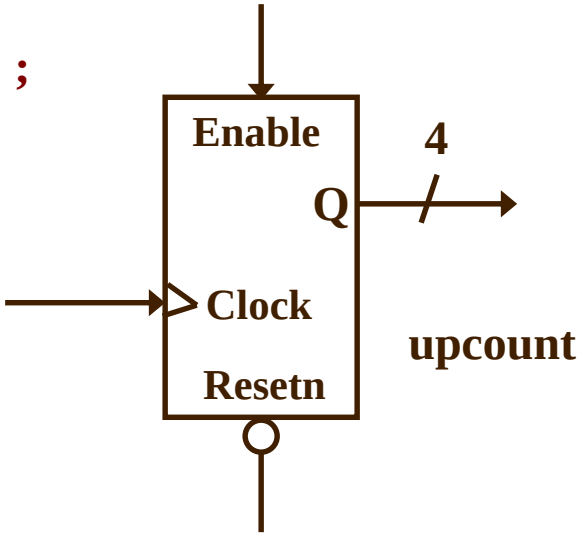
4-bit up-counter with asynchronous reset (1)

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

ENTITY upcount_ar IS
      PORT (   Clock, Resetn, Enable   : IN       STD_LOGIC ;
               Q                       : OUT  STD_LOGIC_VECTOR (3 DOWNTO 0)) ;
END upcount_ar ;
```



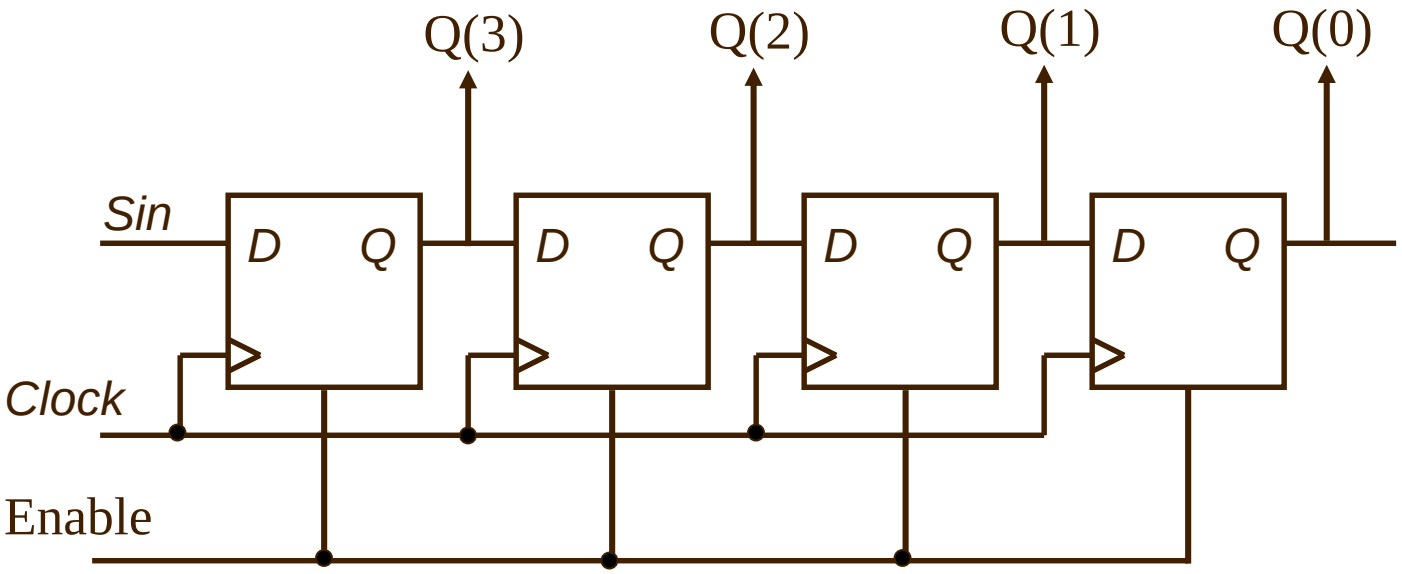
4-bit up-counter with asynchronous reset (2)

```
ARCHITECTURE behavioral OF upcount _ar IS
        SIGNAL Count : UNSIGNED (3 DOWNTO 0) ;
BEGIN
        PROCESS ( Clock, Resetn )
        BEGIN
                IF Resetn = '0' THEN
                        Count <= "0000" ;
                ELSIF rising_edge(Clock) THEN
                        IF Enable = '1' THEN
                                Count <= Count + 1 ;
                        END IF ;
                END IF ;
        END PROCESS ;
        Q <= std_logic_vector(Count) ;
END behavioral ;
```

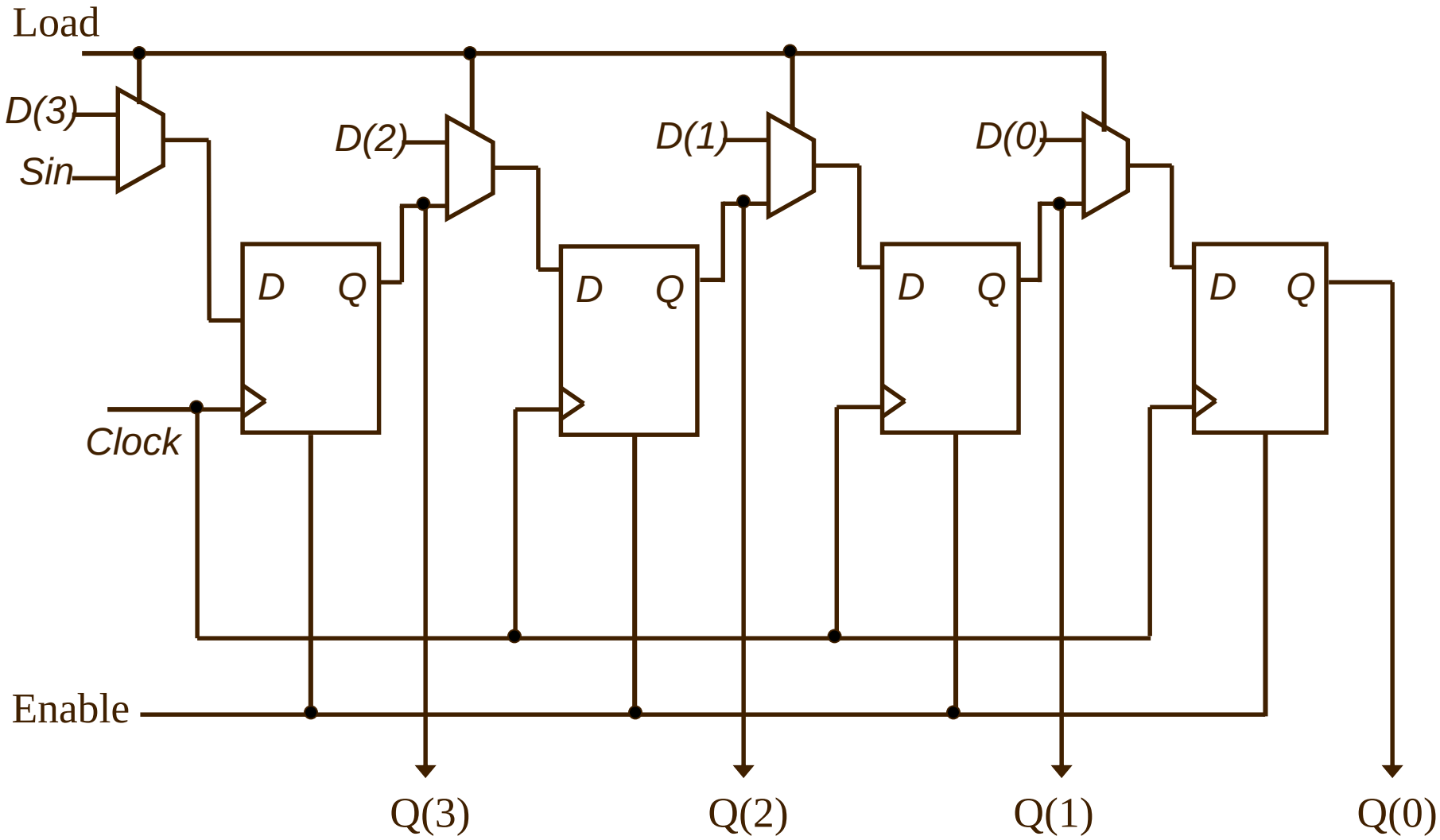


Shift Registers

Shift register – internal structure

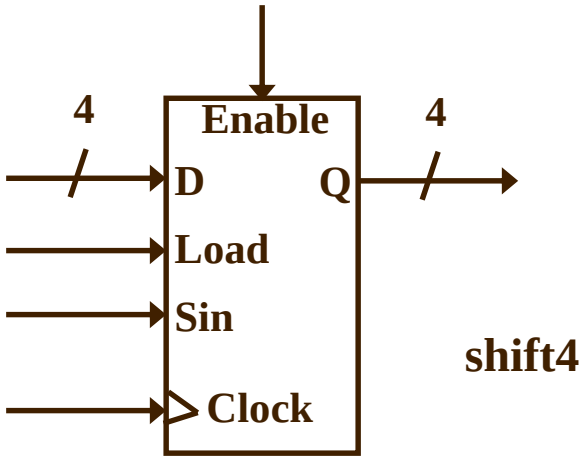


Shift Register With Parallel Load



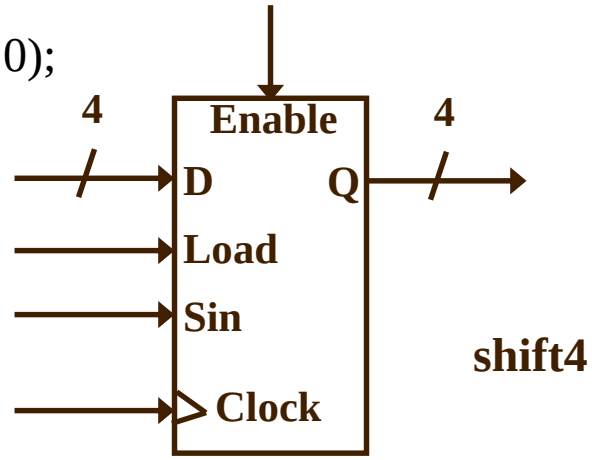
4-bit shift register with parallel load (1)

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

ENTITY shift4 IS
      PORT (   D        : IN      STD_LOGIC_VECTOR(3 DOWNTO 0) ;
               Enable   : IN      STD_LOGIC ;
               Load     : IN      STD_LOGIC ;
               Sin      : IN      STD_LOGIC ;
               Clock    : IN      STD_LOGIC ;
               Q        : OUT     STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END shift4 ;
```



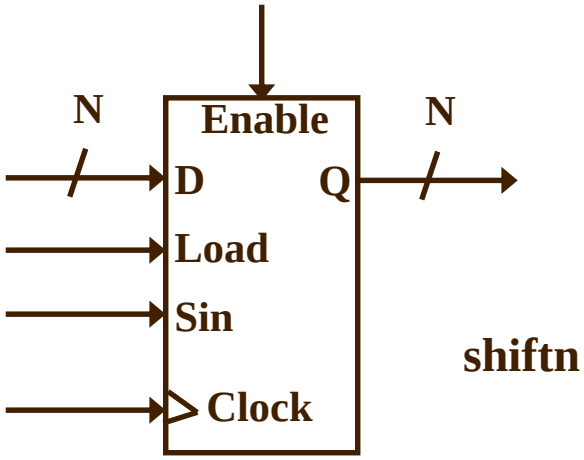
4-bit shift register with parallel load (2)

```
ARCHITECTURE behavioral OF shift4 IS
        SIGNAL Qt : STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN
        PROCESS (Clock)
        BEGIN
                IF rising_edge(Clock) THEN
                        IF Enable = '1' THEN
                                IF Load = '1' THEN
                                        Qt <= D ;
                                ELSE
                                        Qt <= Sin & Qt(3 downto 1);
                                END IF ;
                        END IF;
                END IF ;
        END PROCESS ;
        Q <= Qt;
END behavioral ;
```



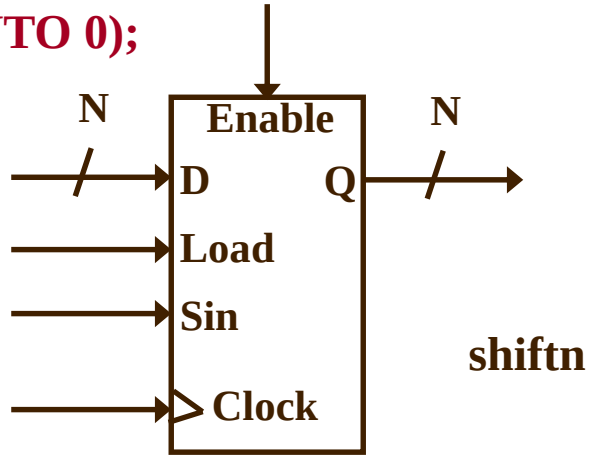
N-bit shift register with parallel load (1)

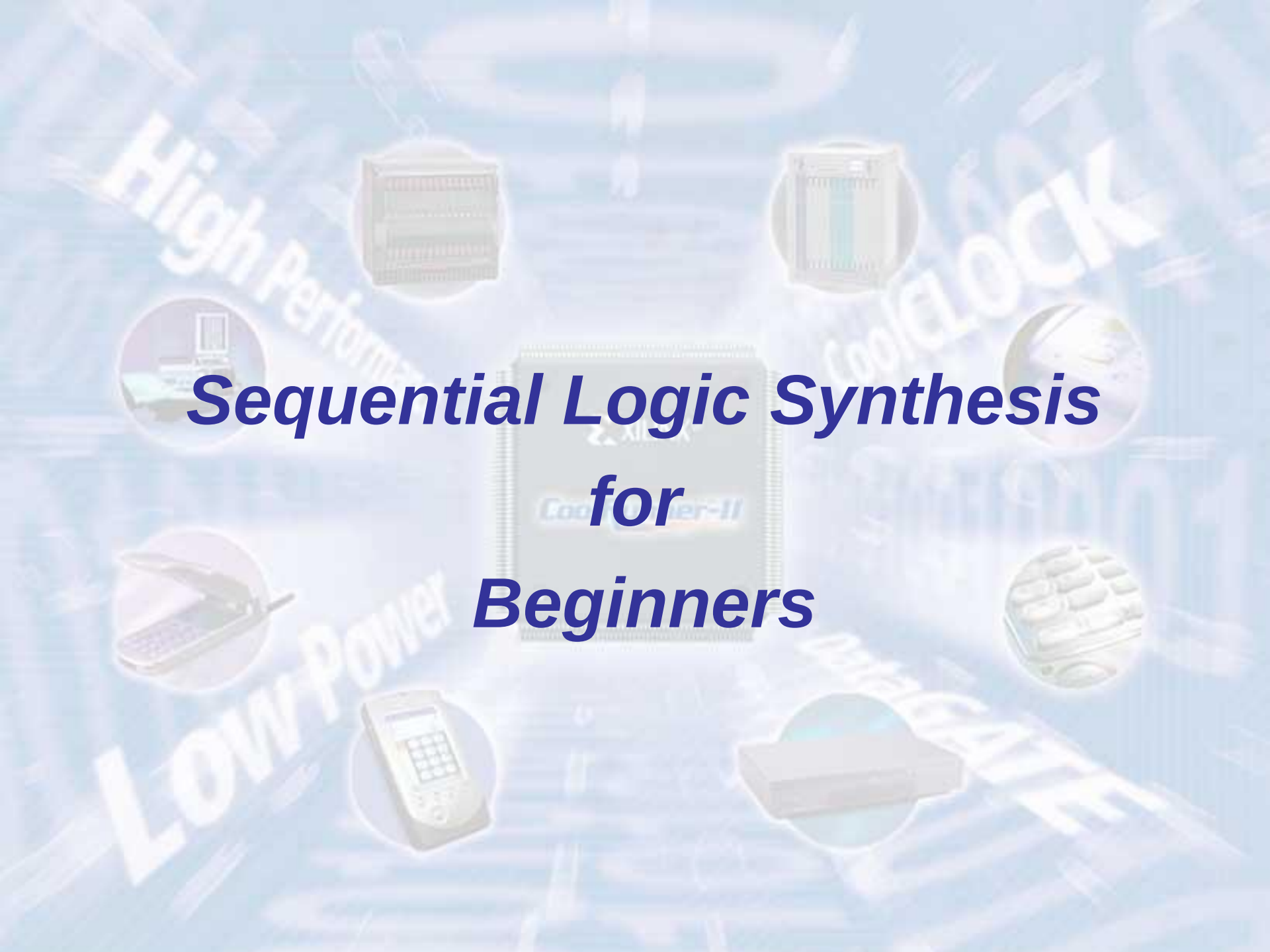
```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY shiftn IS
        GENERIC ( N : INTEGER := 8 ) ;
        PORT (  D : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
                Enable  : IN    STD_LOGIC ;
                Load    : IN    STD_LOGIC ;
                Sin     : IN    STD_LOGIC ;
                Clock   : IN    STD_LOGIC ;
                Q       : OUT   STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
END shiftn ;
```



N-bit shift register with parallel load (2)

```
ARCHITECTURE behavioral OF shiftn IS
        SIGNAL Qt: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
BEGIN
        PROCESS (Clock)
        BEGIN
                IF rising_edge(Clock) THEN
                        IF Enable = '1' THEN
                                IF Load = '1' THEN
                                        Qt <= D ;
                                ELSE
                                        Qt <= Sin & Qt(N-1 downto 1);
                                END IF ;
                        END IF;
                END IF ;
        END PROCESS ;
        Q <= Qt;
END behavior al;
```





Sequential Logic Synthesis for Beginners

For Beginners

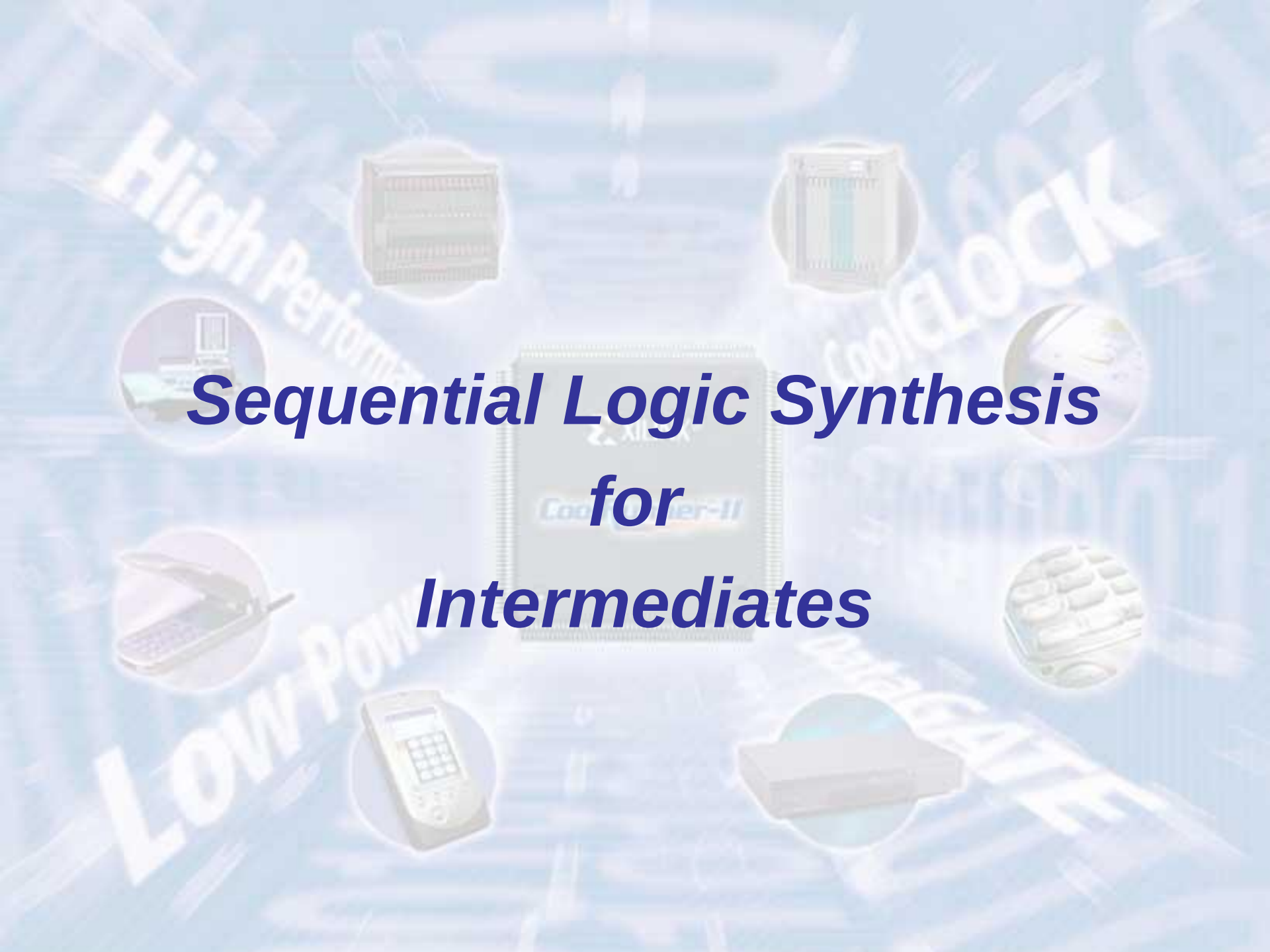
Use processes with very simple structure only to describe

- *registers*
- *shift registers*
- *counters*
- *state machines.*

Use examples discussed in class as a template.

*Create **generic** entities for registers, shift registers, and counters, and instantiate the corresponding components in a higher level circuit using GENERIC MAP PORT MAP.*

Supplement sequential components with combinational logic described using concurrent statements.



Sequential Logic Synthesis for Intermediates

For Intermmediates

1. *Use Processes with IF and CASE statements only. Do not use LOOPS or VARIABLES.*
2. *Sensitivity list of the PROCESS should include **only** signals that can by themsleves change the outputs of the sequential circuit (typically, clock and asynchronous set or reset)*
3. *Do not use PROCESSes without sensitivity list*

   (they can be synthesizable, but make simulation inefficient)

For Intermmediates (2)

Given a single signal, the assignments to this signal should only be made within a single process block in order to avoid possible conflicts in assigning values to this signal.

```
Process 1: PROCESS (a, b)
BEGIN
   y <= a AND b;
END PROCESS;

Process 2: PROCESS (a, b)
BEGIN
   y <= a OR b;
END PROCESS;
```

Non-synthesizable VHDL

Delays

Delays are not synthesizable

Statements, such as

***wait for** 5 ns*

*a <= b **after** 10 ns*

will not produce the required delay, and should not be used in the code intended for synthesis.

Initializations

Declarations of signals (and variables) with initialized values, such as

SIGNAL a : STD_LOGIC ***:= '0' ;***

cannot be synthesized, and thus should be avoided.

If present, they will be ignored by the synthesis tools.

Use set and reset signals instead.

Dual-edge triggered register/counter (1)

In FPGAs register/counter can change only at either rising (default) or falling edge of the clock.

Dual-edge triggered clock is not synthesizable correctly, using either of the descriptions provided below.

Dual-edge triggered register/counter (2)

```
PROCESS (clk)
BEGIN
   IF (clk'EVENT AND clk='1' ) THEN
         counter <= counter + 1;
   ELSIF (clk'EVENT AND clk='0' ) THEN
         counter <= counter + 1;
   END IF;
END PROCESS;
```

Dual-edge triggered register/counter (3)

```
PROCESS (clk)
BEGIN
    IF (clk'EVENT) THEN
        counter <= counter + 1;
    END IF;
END PROCESS;
```

```
PROCESS (clk)
BEGIN
        counter <= counter + 1;
END PROCESS;
```