EDA322
Digital Design

Lecture 7:
Sequential Logic - VHDL

Ioannis Sourdis

Lecture 7:

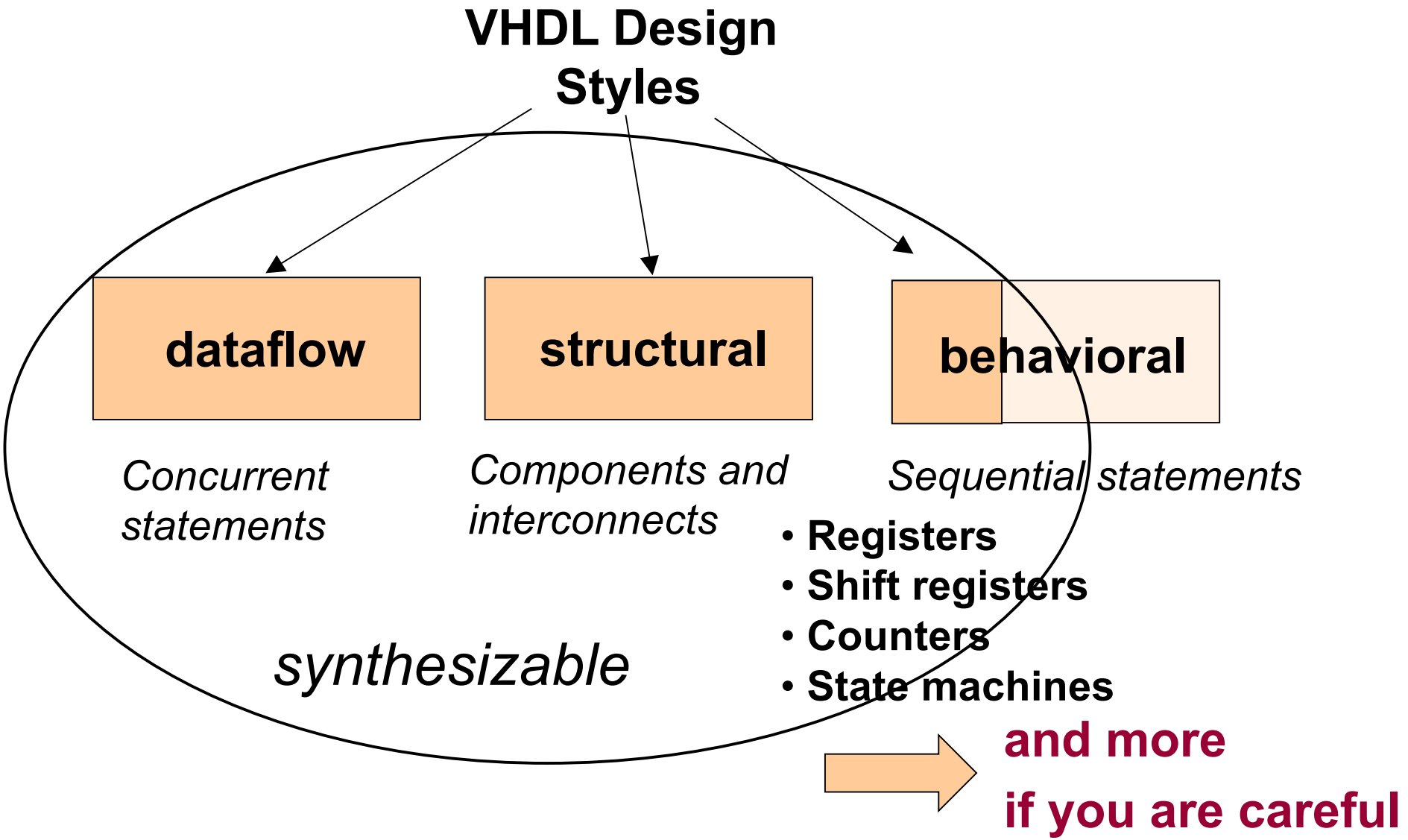
Sequential Circuits VHDL – Behavioral VHDL – Structural VHDL

Outline of Lecture 7

- Processes in VHDL
- Registers (latches, flip-flops)
- Counters, Shift registers
- Generic component instantiation
- Aliases and Constants
- Packages
- Structural VHDL
- Mixing design styles

Behavioral Design Style: Registers & Counters

VHDL Design Styles

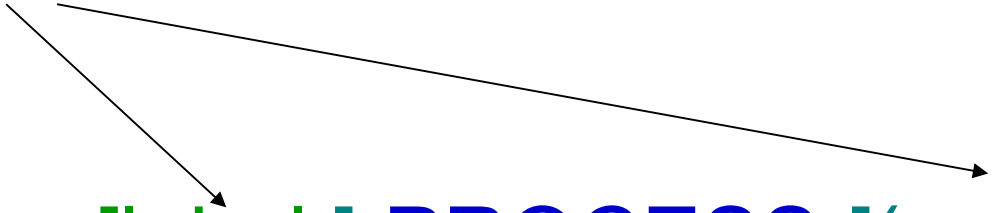


Processes in VHDL

- Processes Describe Sequential Behavior
- Processes in VHDL Are Very Powerful Statements
  - Allow to define an arbitrary behavior that may be difficult to represent by a real circuit
  - Not every process can be synthesized
- Use Processes with Caution in the Code to Be Synthesized
- Use Processes Freely in Testbenches

Anatomy of a Process

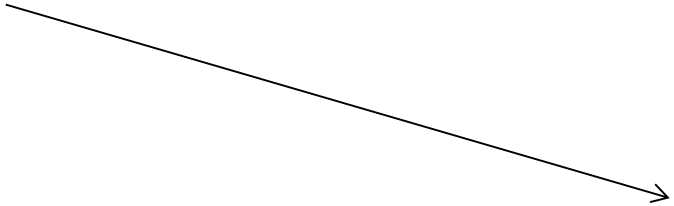
OPTIONAL



[label:] PROCESS [(*sensitivity list*)]
    [d*eclaration part*]
BEGIN
    statement part
END PROCESS [label];

PROCESS with a SENSITIVITY LIST

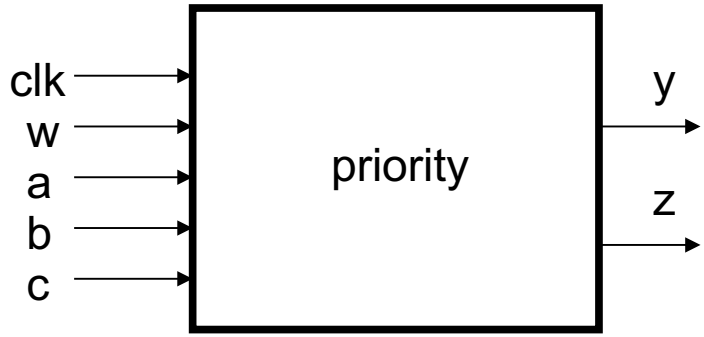
- List of signals to which the process is sensitive.
- Whenever there is an event on any of the signals in the sensitivity list, the process fires.
- Every time the process fires, it will run in its entirety.
- **WAIT statements are NOT ALLOWED in a processes with SENSITIVITY LIST.**



```
label: process (sensitivity list)
    declaration part
begin
    statement part
end process;
```

Component Equivalent of a Process

```
priority: PROCESS (clk)
BEGIN
        IF w(3) = '1' THEN
                y <= "11" ;
                z <= '1' ;
        ELSIF w(2) = '1' THEN
                y <= "10" ;
                z <= '1' ;
        ELSIF w(1) = c THEN
                y <= a and b;
                z <= '1' ;
        ELSE
                z <= '0' ;
        END IF ;
END PROCESS ;
```

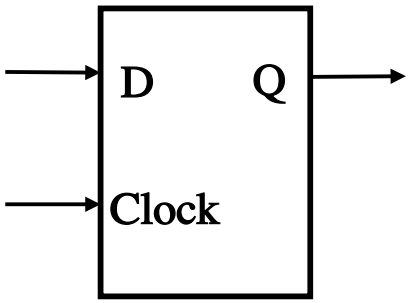


- All signals which appear on the left of signal assignment statement (<=) are outputs e.g. *y, z*
- All signals which appear on the right of signal assignment statement (<=) or in logic expressions are inputs e.g. *w, a, b, c*
- All signals which appear in the sensitivity list are inputs e.g. *clk*
- Note that not all inputs need to be included in the sensitivity list

Registers

D latch

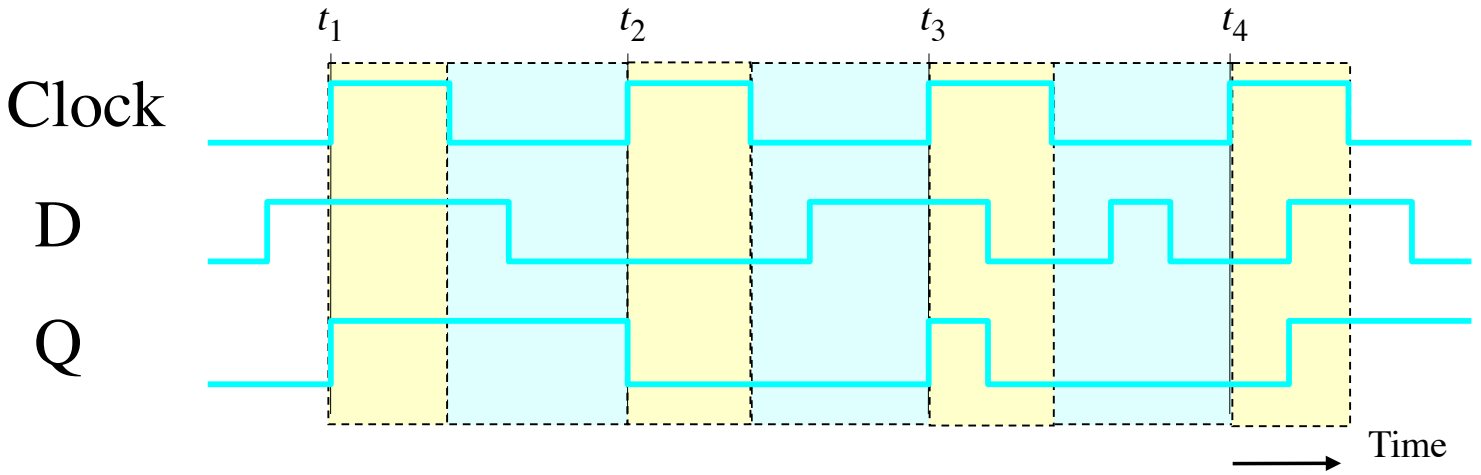
Graphical symbol



Truth table

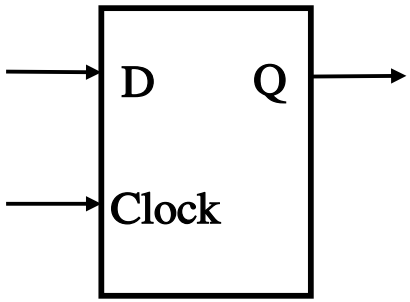
Clock	D	Q(*t*+1)
0	–	Q(*t*)
1	0	0
1	1	1

Timing diagram



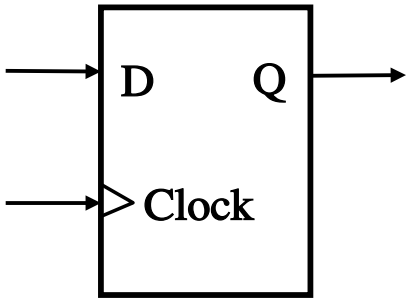
D latch

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

ENTITY latch IS
        PORT (   D, Clock  : IN       STD_LOGIC ;
                 Q         : OUT      STD_LOGIC) ;
END latch ;

ARCHITECTURE behavioral OF latch IS
BEGIN
        PROCESS ( D, Clock )
        BEGIN
                IF Clock = '1' THEN
                        Q <= D ;
                END IF ;
        END PROCESS ;
END behavioral;
```



D flip-flop

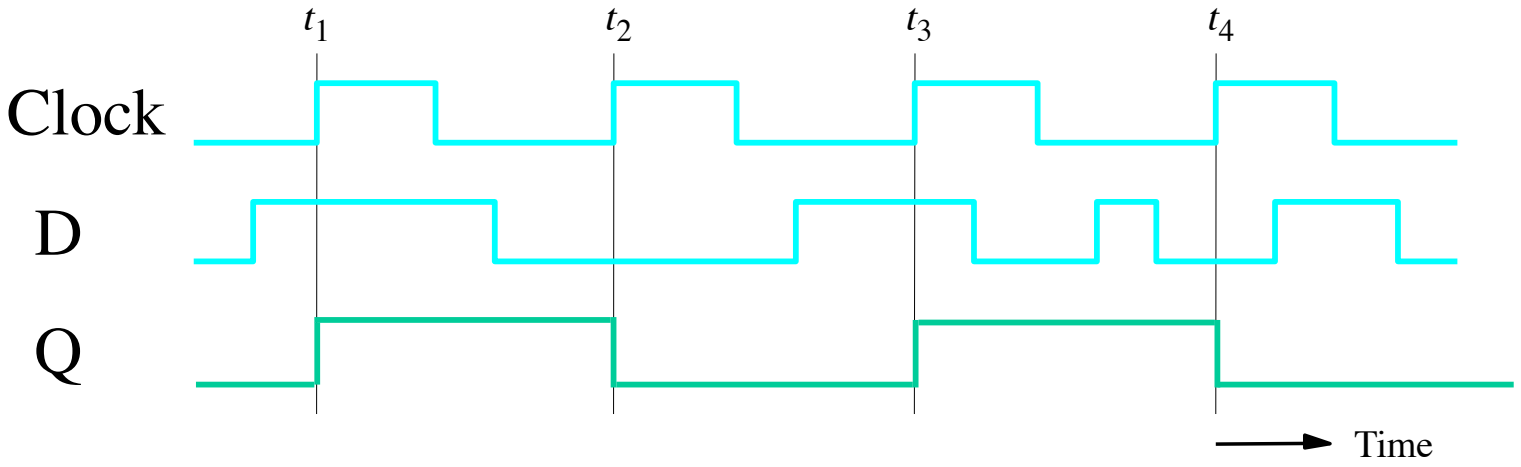
Graphical symbol



Truth table

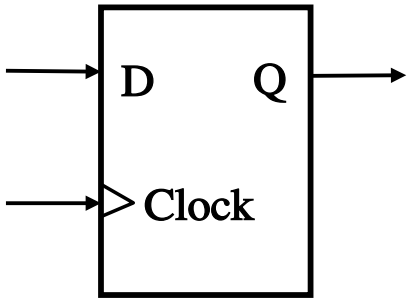
Clk	D	Q(t+1)
↑	0	0
↑	1	1
0	–	Q(t)
1	–	Q(t)

Timing diagram



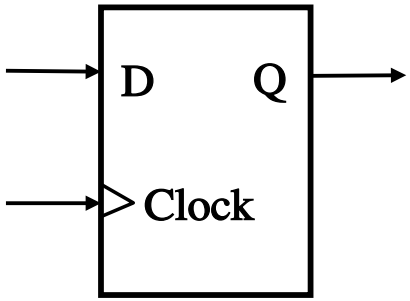
D flip-flop

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

ENTITY flipflop IS
      PORT (   D, Clock : IN     STD_LOGIC ;
               Q        : OUT    STD_LOGIC) ;
END flipflop ;

ARCHITECTURE behavioral OF flipflop IS
BEGIN
      PROCESS ( Clock )
      BEGIN
           IF Clock'EVENT AND Clock = '1' THEN
                Q <= D ;
           END IF ;
      END PROCESS ;
END behavioral ;
```



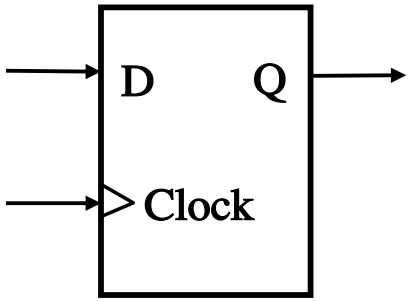
D flip-flop

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

ENTITY flipflop IS
      PORT (   D, Clock : IN     STD_LOGIC ;
               Q        : OUT    STD_LOGIC) ;
END flipflop ;

ARCHITECTURE behavioral2 OF flipflop IS
BEGIN
      PROCESS ( Clock )
      BEGIN
           IF rising_edge(Clock) THEN
                Q <= D ;
           END IF ;
      END PROCESS ;
END behavioral2;
```



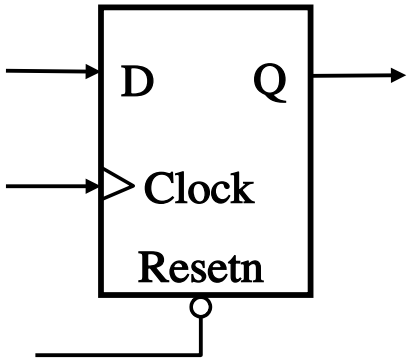
D flip-flop

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

ENTITY flipflop IS
      PORT (   D, Clock : IN     STD_LOGIC ;
               Q        : OUT    STD_LOGIC) ;
END flipflop ;

ARCHITECTURE behavioral3 OF flipflop IS
BEGIN
      PROCESS
      BEGIN
           WAIT UNTIL rising_edge(Clock) ;
           Q <= D ;
      END PROCESS ;
END behavioral3 ;
```



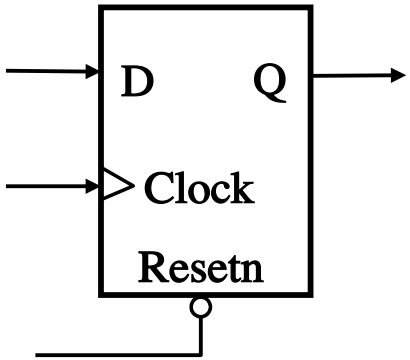
D flip-flop with asynchronous reset

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

ENTITY flipflop_ar IS
        PORT (    D, Resetn, Clock      : IN       STD_LOGIC ;
                  Q                     : OUT      STD_LOGIC) ;
END flipflop_ar ;

ARCHITECTURE behavioral OF flipflop_ar IS
BEGIN
        PROCESS ( Resetn, Clock )
        BEGIN
                IF Resetn = '0' THEN
                        Q <= '0' ;
                ELSIF rising_edge(Clock) THEN
                        Q <= D ;
                END IF ;
        END PROCESS ;
END behavioral ;
```



D flip-flop with synchronous reset

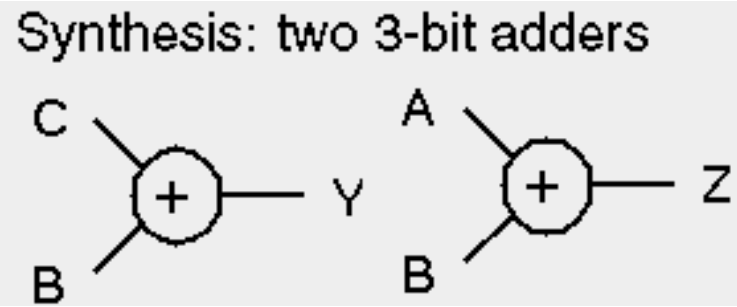
```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
ENTITY flipflop_sr IS
        PORT (    D, Resetn, Clock      : IN       STD_LOGIC ;
                  Q                     : OUT      STD_LOGIC) ;
END flipflop_sr ;

ARCHITECTURE behavioral OF flipflop_sr IS
BEGIN
        PROCESS(Clock)
        BEGIN
                IF rising_edge(Clock) THEN
                        IF Resetn = '0' THEN
                                Q <= '0' ;
                        ELSE
                                Q <= D ;
                        END IF ;
                END IF;
        END PROCESS ;
END behavioral ;
```



Variables vs. Signals

Variables

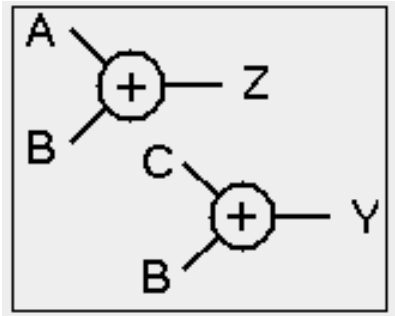
```
architecture RTL of XYZ is
        signal A, B, C : integer range 0 to 7;
        signal Y, Z :    integer range 0 to 15;
    begin
        process (A, B, C)
          variable M, N : integer range 0 to 7;
        begin
          M := A;
          N := B;
          Z <= M + N;
          M := C;
          Y <= M + N;
        end process;
    end RTL;
```



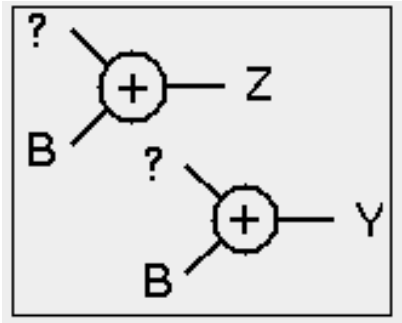
- Variables are available within processes, only named within process declarations known only in this process
- VHDL 93: shared variables
- immediate assignment keep the last value
- Possible assignments
  - signal to variable
  - variable to signal
  - types have to match

Variables vs. Signals

```
signal A, B, C, Y, Z : integer;
    begin
        process (A, B, C)
            variable M, N : integer;
        begin
            M  :=  A;
            N  :=  B;
            Z <= M + N;
            M  :=  C;
            Y <= M + N;
        end process;
```




```
signal A, B, C, Y, Z : integer;
    signal M, N  : integer;
    begin
        process (A, B, C, M, N)

        begin
            M  <=  A;
            N  <=  B;
            Z <= M + N;
            M  <=  C;
            Y <= M + N;
        end process;
```



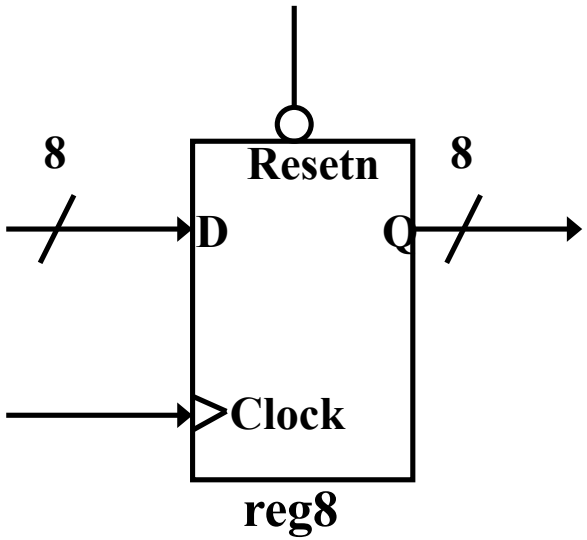
- Signal values are assigned after the process execution
- Only the last signal assignment is carried out
- M <= A;
  is overwritten by
  M <= C;
- The 2nd adder input is connected to C

Asychronous vs. Synchronous

- In the IF loop, asynchronous items are
  - **Before** the rising_edge(Clock) statement
- In the IF loop, synchronous items are
  - **After** the rising_edge(Clock) statement

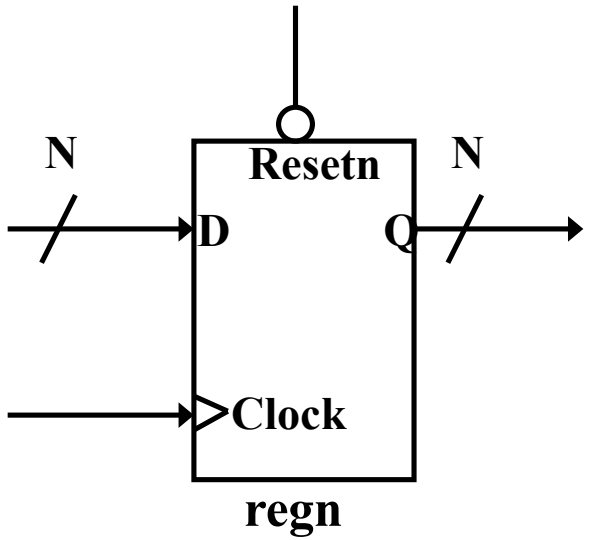
8-bit register with asynchronous reset

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

ENTITY reg8 IS
        PORT (    D                 : IN     STD_LOGIC_VECTOR(7 DOWNTO 0) ;
                  Resetn, Clock     : IN     STD_LOGIC ;
                  Q                 : OUT    STD_LOGIC_VECTOR(7 DOWNTO 0) ) ;
END reg8 ;

ARCHITECTURE behavioral OF reg8 IS
BEGIN
        PROCESS ( Resetn, Clock )
        BEGIN
                IF Resetn = '0' THEN
                        Q <= "00000000" ;
                ELSIF rising_edge(Clock) THEN
                        Q <= D ;
                END IF ;
        END PROCESS ;
END behavioral ;`
```



N-bit register with asynchronous reset

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

ENTITY regn IS
        GENERIC ( N : INTEGER := 8 ) ;
        PORT (    D                 : IN      STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
                  Resetn, Clock     : IN      STD_LOGIC ;
                  Q                 : OUT     STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
END regn ;

ARCHITECTURE behavioral OF regn IS
BEGIN
        PROCESS ( Resetn, Clock )
        BEGIN
                IF Resetn = '0' THEN
                        Q <= (OTHERS => '0') ;
                ELSIF rising_edge(Clock) THEN
                        Q <= D ;
                END IF ;
        END PROCESS ;
END behavioral ;
```



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 can 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

OTHERS

OTHERS stand for any index value that has not been previously mentioned.

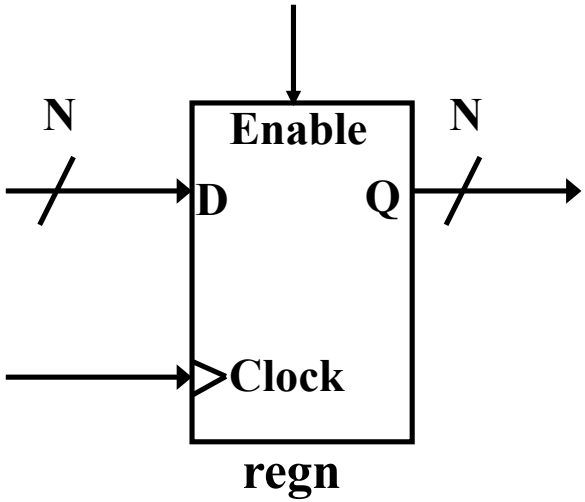
Q <= “00000001” can be written as Q <= (0 => ‘1’, OTHERS => ‘0’)

Q <= “10000001” can be written as Q <= (7 => ‘1’, 0 => ‘1’, OTHERS => ‘0’)
or Q <= (7 | 0 => ‘1’, OTHERS => ‘0’)

Q <= “00011110” can be written as Q <= (4 downto 1=> ‘1’, OTHERS => ‘0’)

N-bit register with enable

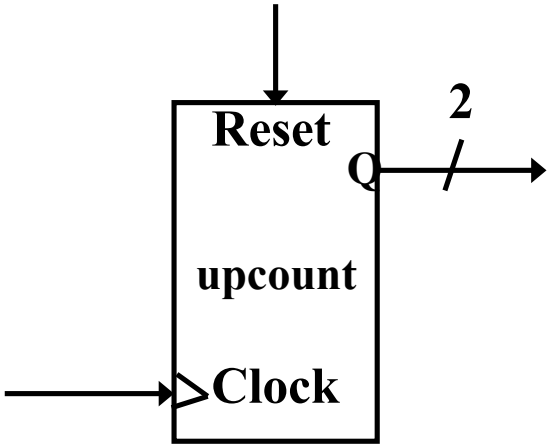
```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

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

ARCHITECTURE behavioral OF regne IS
BEGIN
        PROCESS (Clock)
        BEGIN
                IF rising_edge(Clock) THEN
                        IF Enable = '1' THEN
                                Q <= D ;
                        END IF ;
                END IF;
        END PROCESS ;
END behavioral ;
```



Counters

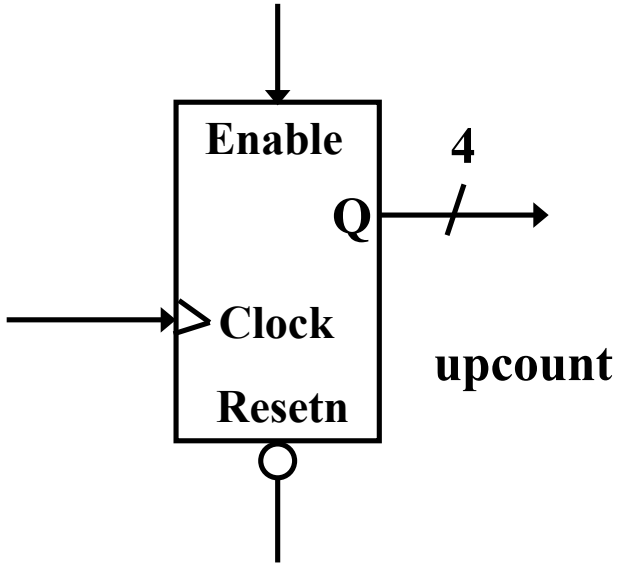
2-bit up-counter with synchronous reset

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

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



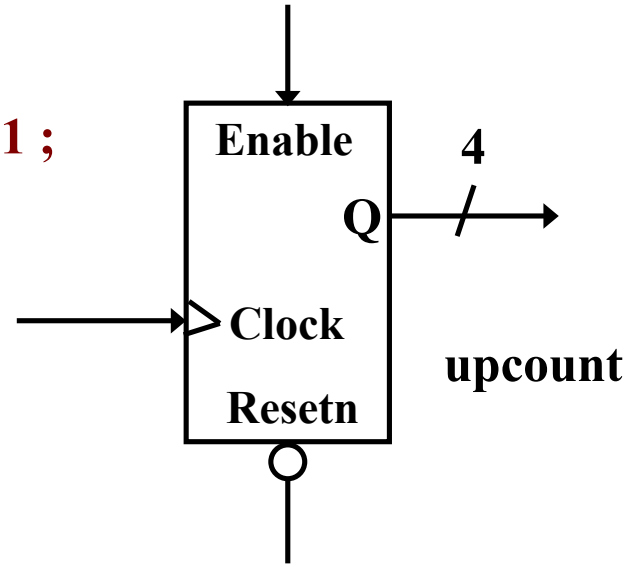
4-bit up-counter with asynchronous reset (1)

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

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



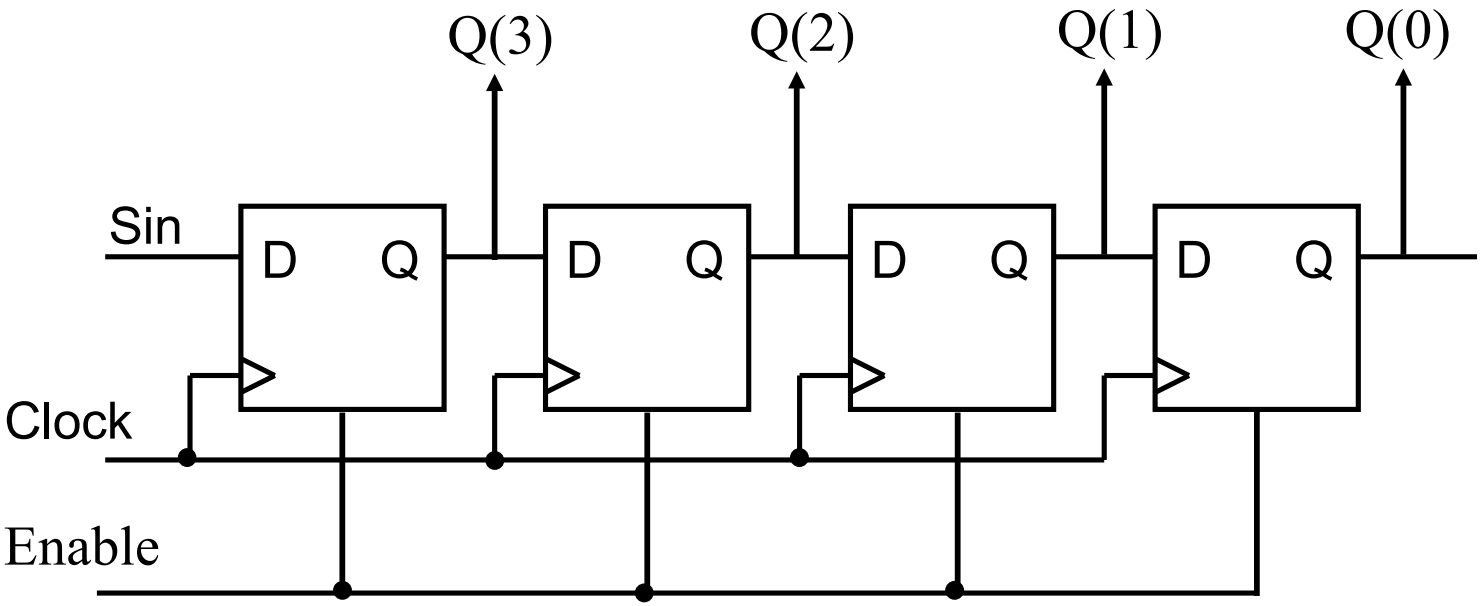
4-bit up-counter with asynchronous reset (2)

```
ARCHITECTURE behavioral OF upcount _ar IS
        SIGNAL Count : STD_LOGIC_VECTOR (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 <= Count ;
END behavioral ;
```

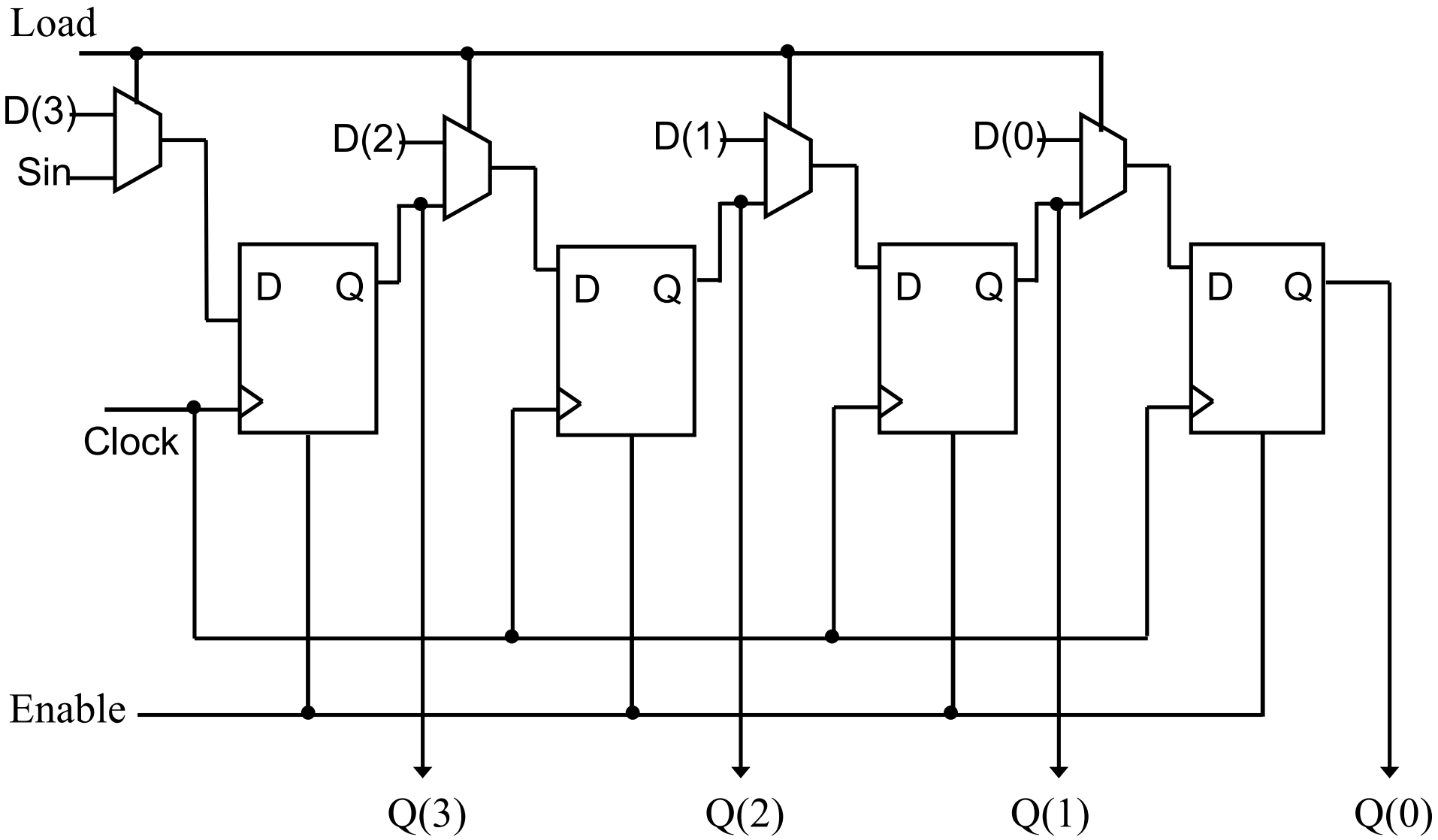


Shift Registers

Shift register – internal structure

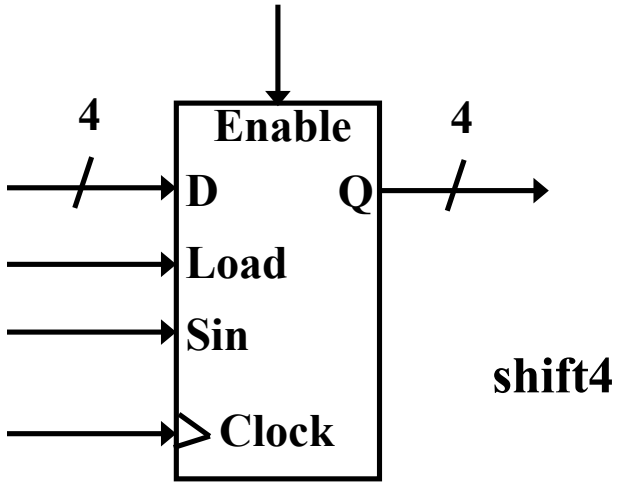


Shift Register With Parallel Load



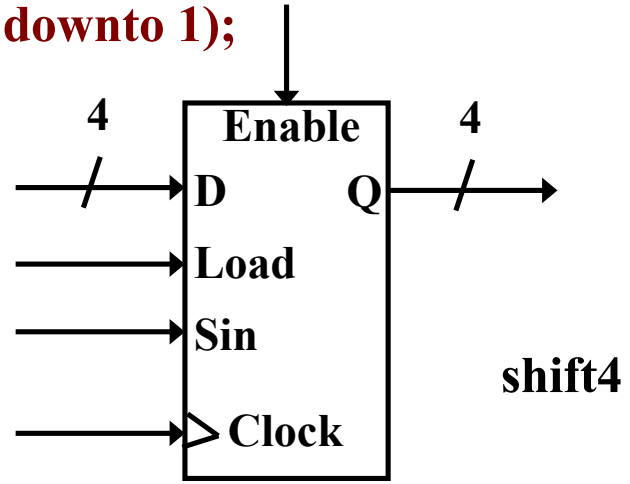
4-bit shift register with parallel load (1)

```
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 ;
```



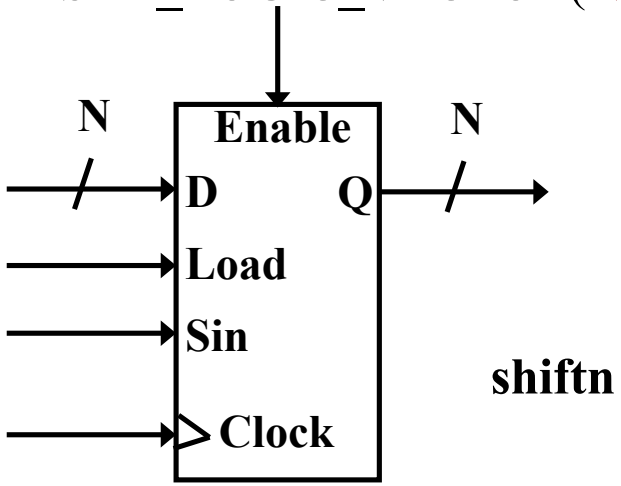
4-bit shift register with parallel load (2)

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



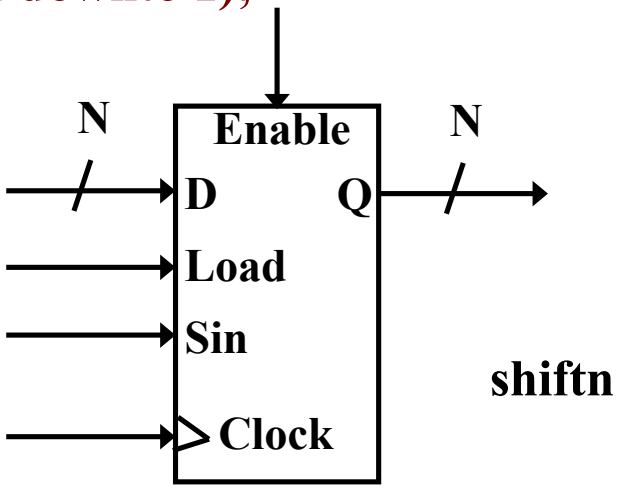
N-bit shift register with parallel load (1)

```
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)

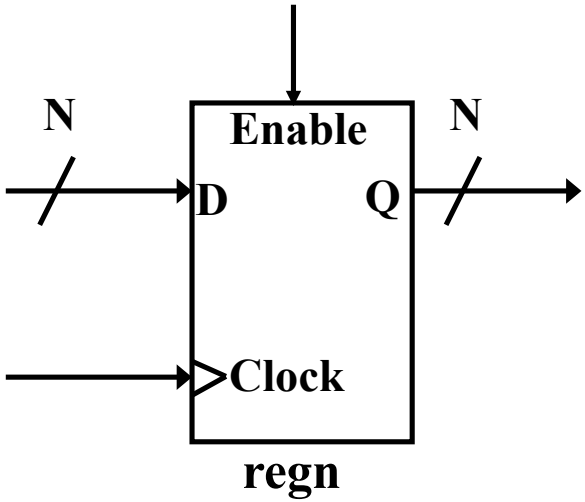
```
ARCHITECTURE behavioral OF shiftn IS
        SIGNAL Qt: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
BEGIN
        PROCESS (Clock)
        BEGIN
                IF rising_edge(Clock) THEN
                        IF Load = '1' THEN
                                Qt <= D ;
                        ELSIF Enable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



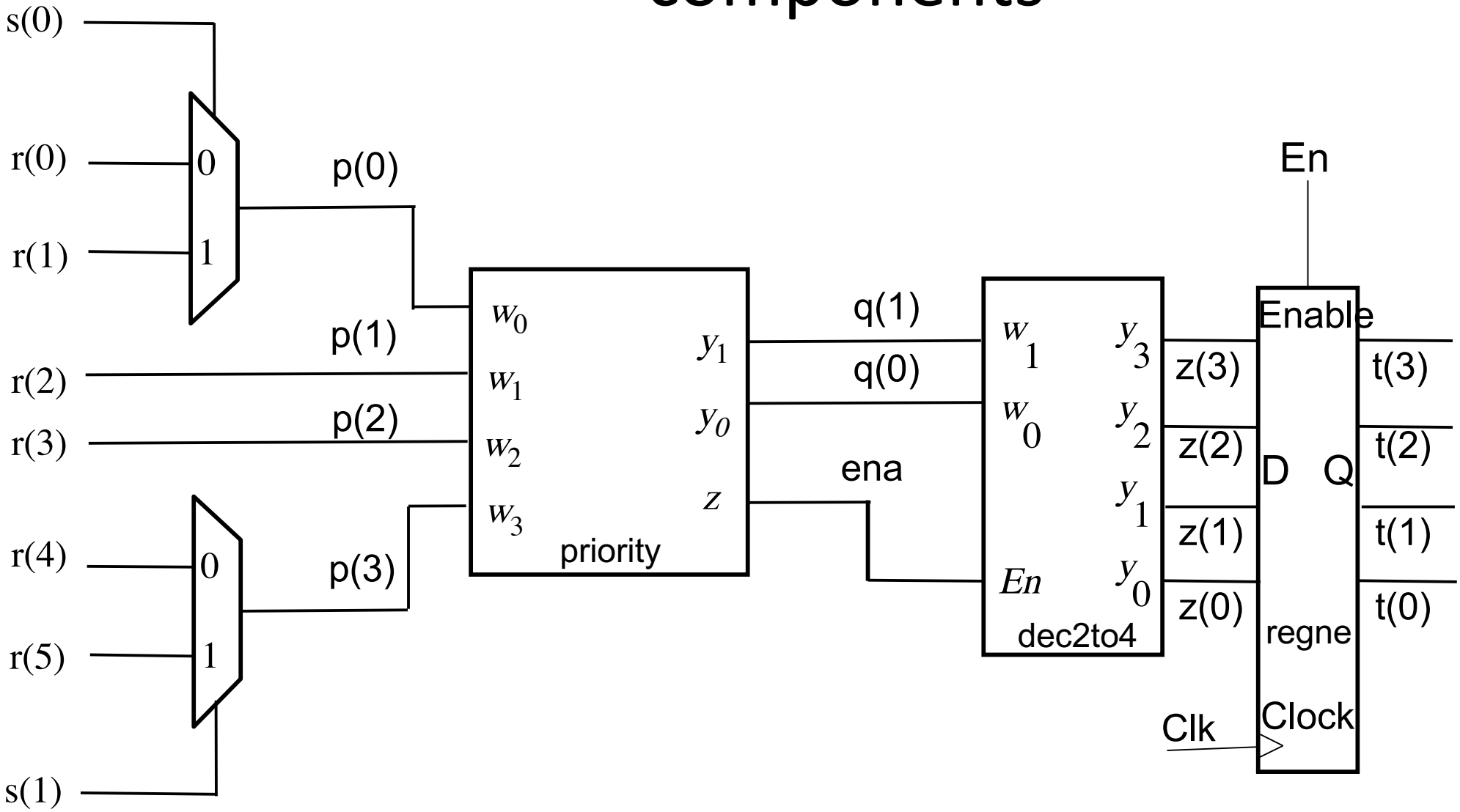
Generic Component Instantiation

N-bit register with enable

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

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

ARCHITECTURE Behavior OF regn IS
BEGIN
        PROCESS (Clock)
        BEGIN
                IF (Clock'EVENT AND Clock = '1' ) THEN
                        IF Enable = '1' THEN
                                Q <= D ;
                        END IF ;
                END IF;
        END PROCESS ;
END Behavior ;
```



Circuit built of medium scale components



Structural description – example (1)

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

ENTITY priority_resolver IS
    PORT (r          : IN      STD_LOGIC_VECTOR(5 DOWNTO 0) ;
          s          : IN      STD_LOGIC_VECTOR(1 DOWNTO 0) ;
          clk        : IN      STD_LOGIC;
          en         : IN      STD_LOGIC;
          t          : OUT     STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END priority_resolver;

ARCHITECTURE structural OF priority_resolver IS

SIGNAL  p : STD_LOGIC_VECTOR (3 DOWNTO 0) ;
SIGNAL  q : STD_LOGIC_VECTOR (1  DOWNTO 0) ;
SIGNAL  z : STD_LOGIC_VECTOR (3 DOWNTO 0) ;
SIGNAL  ena : STD_LOGIC ;
```

Structural description – example (2) VHDL-93

```
BEGIN

    u1:  work.mux2to1(dataflow)
                    PORT MAP (w0 => r(0) ,
                                    w1 => r(1),
                                   s => s(0),
                                   f => p(0));
    p(1) <= r(2);
    p(2) <= r(3);

    u2:  work.mux2to1(dataflow)
                    PORT MAP (w0 => r(4) ,
                                    w1 => r(5),
                                  s => s(1),
                                  f => p(3));

    u3: work.priority(dataflow)
                   PORT MAP (w => p,
                                  y => q,
                                    z => ena);
```

Structural description – example (3) VHDL-93

```
u4: work.dec2to4 (dataflow)
        PORT MAP (w => q,
                  En => ena,
                  y => z);

u5: work.regne(behavioral)
        GENERIC MAP (N => 4)
        PORT MAP (D => z ,
                  Enable => En ,
                  Clock => Clk,
                  Q => t );
END structural;
```

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.

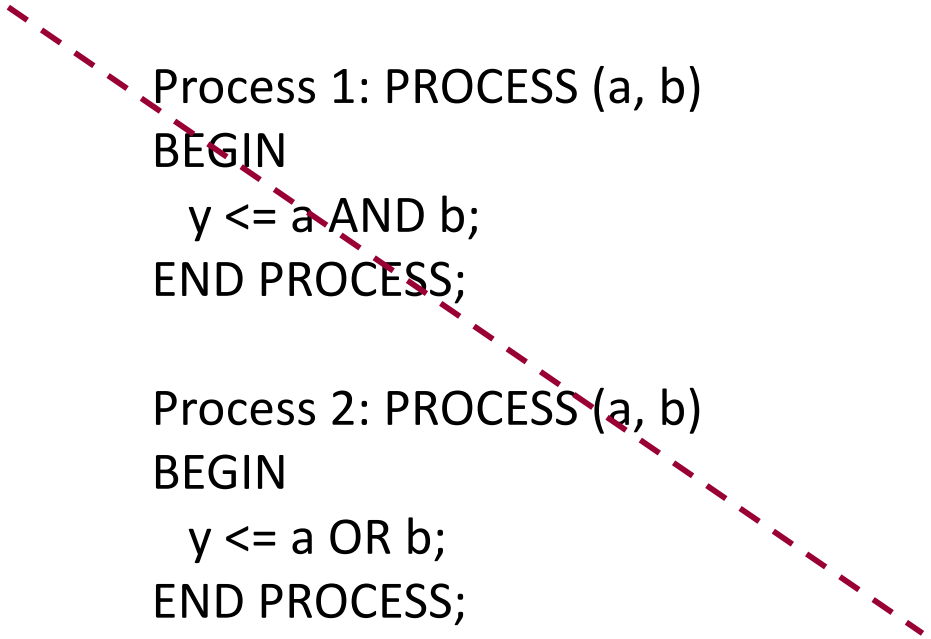
Sequential Logic Synthesis for Intermediates

For Intermmediates

- Use Processes with IF and CASE statements only. Do not use LOOPS or VARIABLES.
- Sensitivity list of the PROCESS should include **only** signals that can by themselves change the outputs of the sequential circuit (typically, clock and asynchronous set or reset)
- 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;
```

Aliases

Aliases

Syntax:

ALIAS name : type := expression;

Example:

```
signal IR : std_logic_vector(31 downto 0);

alias IR_opcode     : std_logic_vector(5 downto 0) is IR(31 downto 26);
alias IR_reg1_addr : std_logic_vector(4 downto 0) is IR(25 downto 21);
alias IR_reg2_addr : std_logic_vector(4 downto 0) is IR(20 downto 16);
```

Constants

Constants

Syntax:

CONSTANT name : type := value;

Examples:

```
CONSTANT init_value : 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 := 16#FFFE#;
CONSTANT clk_period : TIME := 20 ns;
CONSTANT strobe_period : TIME := 333.333 ms;
```

Constants - features

Constants can be declared in a
PACKAGE, ENTITY, ARCHITECTURE

When declared in a PACKAGE, the constant is truly global, for the package can be used in several entities.

When declared in an ARCHITECTURE, the constant is local, i.e., it is visible only within this architecture.

When declared in an ENTITY declaration, the constant can be used in all architectures associated with this entity.

Packages

Explicit Component Declaration versus Package

- Explicit component declaration is when you declare components in main code
  - When have only a few component declarations, this is fine
  - When have many component declarations, use packages for readability

- Packages also help with portability and sharing of libraries among many users in a company

- **Remember, the actual instantiations always take place in main code**
  - Only the declarations can be in main code or package

Explicit Component Declaration Tips

- For simple projects put entity .vhd files all in same directory
- Declare components in main code
- Compilation order
  - Modelsim figures it out by itself
  - For other tools make sure compiler knows the correct hierarchy
    - From lowest to highest

METHOD #2: Package component declaration

- Components declared in package
- Actual instantiations and port maps always in main code

Packages

- Instead of declaring all components in main code, you can declare all components in a PACKAGE, and INCLUDE the package once
  - This makes the top-level entity code cleaner
  - It also allows that complete package to be used by another designer
- A package can contain
  - **Components**
  - Functions, Procedures
  - Types, **Constants**

Package – example (1)

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

PACKAGE GatesPkg IS

    COMPONENT mux2to1
       PORT (w0, w1, s   : IN    STD_LOGIC ;
                   f             : OUT   STD_LOGIC ) ;
    END COMPONENT ;

    COMPONENT priority
       PORT (w   : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
         y       : OUT   STD_LOGIC_VECTOR(1 DOWNTO 0) ;
         z       : OUT   STD_LOGIC ) ;
    END COMPONENT ;
```

Package – example (2)

```
COMPONENT dec2to4
   PORT (w   : IN    STD_LOGIC_VECTOR(1 DOWNTO 0) ;
     En      : IN    STD_LOGIC ;
     y       : OUT   STD_LOGIC_VECTOR(0 TO 3) ) ;
END COMPONENT ;

COMPONENT regn
   GENERIC ( N : INTEGER := 8 ) ;
   PORT (    D     : IN              STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
             Enable, Clock   : IN            STD_LOGIC ;
             Q     : OUT     STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
END COMPONENT ;
```

Package – example (3)

```
    constant ADDAB : std_logic_vector(3 downto 0) := "0000";
    constant ADDAM : std_logic_vector(3 downto 0) := "0001";
    constant SUBAB : std_logic_vector(3 downto 0) := "0010";
    constant SUBAM : std_logic_vector(3 downto 0) := "0011";
    constant NOTA : std_logic_vector(3 downto 0) := "0100";
    constant NOTB : std_logic_vector(3 downto 0) := "0101";
    constant NOTM : std_logic_vector(3 downto 0) := "0110";
    constant ANDAB : std_logic_vector(3 downto 0) := "0111";

END GatesPkg;
```

Package usage (1)

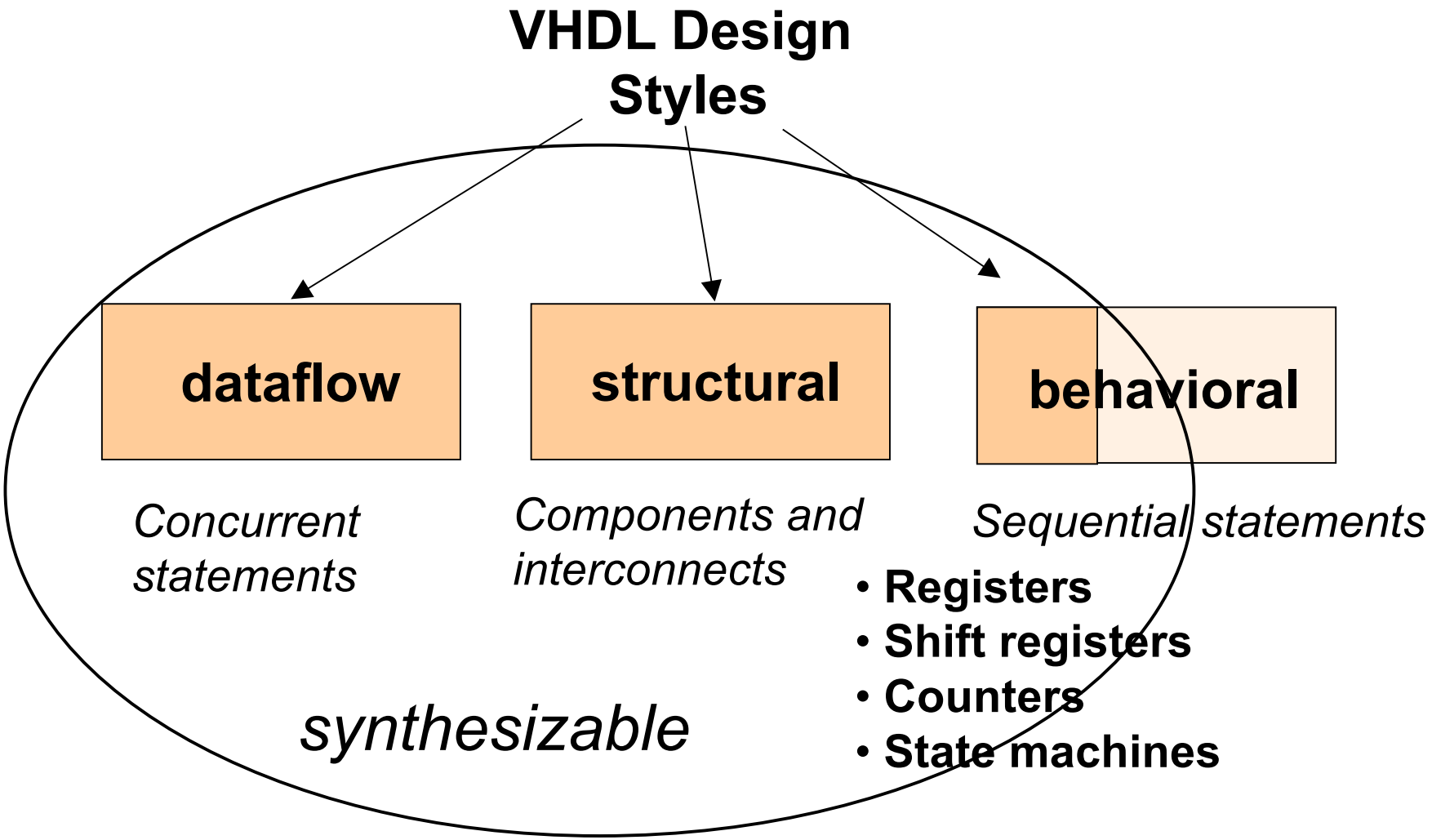
```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

USE work.GatesPkg.all;

ENTITY priority_resolver1 IS
   PORT (r    : IN   STD_LOGIC_VECTOR(5 DOWNTO 0) ;
         s    : IN   STD_LOGIC_VECTOR(1 DOWNTO 0) ;
                clk        : IN         STD_LOGIC;
                en         : IN        STD_LOGIC;
         t    : OUT  STD_LOGIC_VECTOR(3 DOWNTO 0) ) ;
END priority_resolver1;

ARCHITECTURE structural OF priority_resolver1 IS

SIGNAL  p : STD_LOGIC_VECTOR (3 DOWNTO 0) ;
SIGNAL  q : STD_LOGIC_VECTOR (1  DOWNTO 0) ;
SIGNAL  z : STD_LOGIC_VECTOR (3 DOWNTO 0) ;
SIGNAL  ena : STD_LOGIC ;
```

Package usage (2)

```
BEGIN
  u1: mux2to1 PORT MAP (w0 => r(0) ,
                        w1 => r(1),
                        s => s(0),
                        f => p(0));
  p(1) <= r(2);
  p(2) <= r(3);

  u2: mux2to1 PORT MAP (w0 => r(4) ,
                        w1 => r(5),
                        s => s(1),
                        f => p(3));

      u3: priority PORT MAP (w => p,
                             y => q,
                             z => ena);

  u4: dec2to4 PORT MAP (w => q,
                        En => ena,
                        y => z);

  u5: regn GENERIC MAP (N => 4)

           PORT MAP (D => z ,

                     Enable => En ,

                     Clock => Clk,

                     Q => t );

END structural;
```

Mixing Design Styles Inside of an Architecture

VHDL Design Styles



Mixed Style Modeling

architecture ARCHITECTURE_NAME **of** ENTITY_NAME **is**

– Here you can declare signals, constants, types, etc.
– Component declarations

begin

Concurrent statements: **dataflow**

- Concurrent simple signal assignment
- Conditional signal assignment
- Selected signal assignment
- Generate statement

- Component instantiation statement **structural**

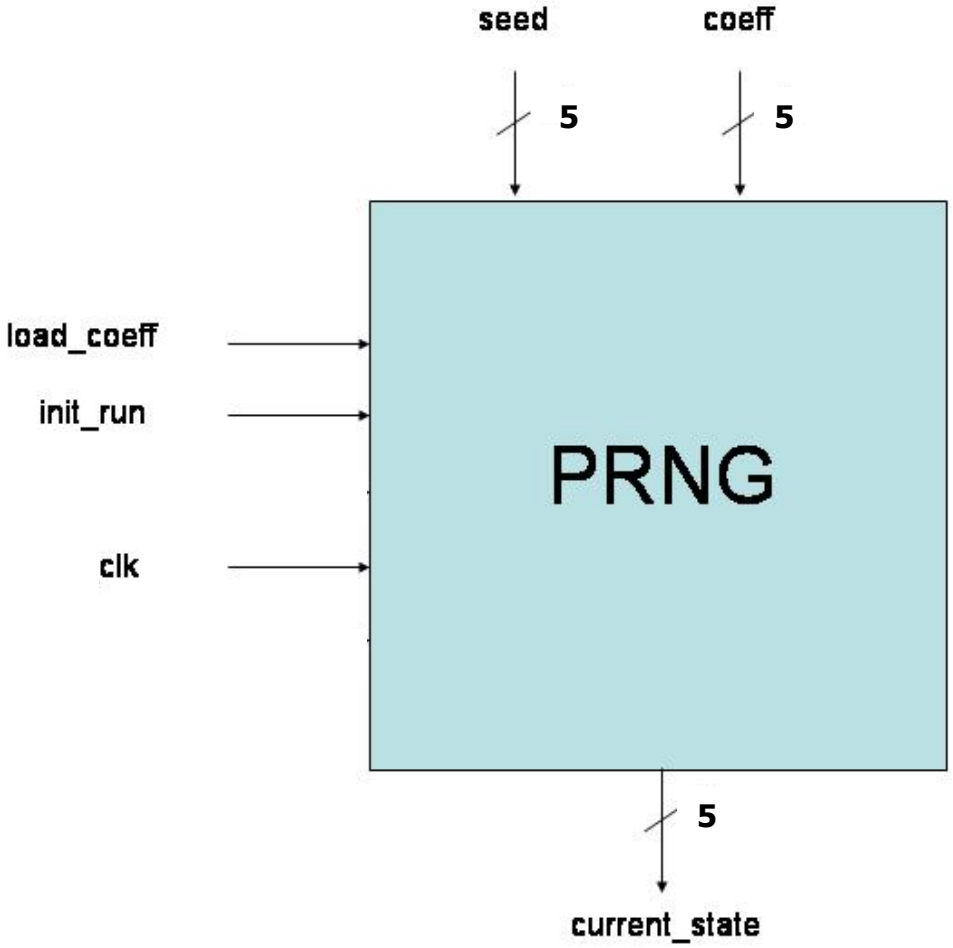
- Process statement **behavioral**
  – **inside process you can use only sequential statements**

Concurrent Statements

end ARCHITECTURE_NAME;

PseudoRandom Number Generator example

Name	Width	Description
seed	5	Initial state of a PRNG
coeff	5	Coefficients of the underlying LFSR (c0, c1, c2, c3, c4)
load_coeff	1	Control signal active when loading coefficients of the underlying LFSR
init_run	1	Choice between the initialization mode (loading seed), and execution mode (running PRNG)
clk	1	clock
current_state	5	Current state of the PRNG



PRNG Example (1)

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use work.prng_pkg.all;

ENTITY PRNG IS
    PORT( Coeff          : in  std_logic_vector(4 downto 0);
          Load_Coeff     : in  std_logic;
          Seed           : in  std_logic_vector(4 downto 0);
          Init_Run       : in  std_logic;
          Clk            : in  std_logic;
          Current_State  : out std_logic_vector(4 downto 0));
END PRNG;

ARCHITECTURE mixed OF PRNG is
    signal Ands     : std_logic_vector(4 downto 0);
    signal Sin      : std_logic;
    signal Coeff_Q  : std_logic_vector(4 downto 0);
    signal Shift5_Q : std_logic_vector(4 downto 0);
```

PRNG Example (2)

```
BEGIN
-- Data Flow
    G: FOR i IN 0 TO 4 GENERATE
        Ands(i) <= Coeff_Q(i) AND Shift5_Q(i);
    END GENERATE;
    Sin <= Ands(0) XOR Ands(1) XOR Ands(2) XOR Ands(3) XOR Ands(4);
    Current_State <= Shift5_Q;

    -- Behavioral
    Coeff_Reg: PROCESS(Clk)
    BEGIN
        IF Clk'EVENT and Clk = '1'  THEN
            IF Load_Coeff = '1'   THEN
                Coeff_Q <= Coeff;
            END IF;
        END IF;
    END PROCESS;

    -- Structural
    Shift5_Reg : Shift5 PORT MAP ( D     => Seed,
                                   Load  => Init_Run,
                                   Sin   => Sin,
                                   Clock => Clk,
                                   Q     => Shift5_Q);
END mixed;
```

Summary of Lecture 7

- Processes in VHDL
- Registers (latches, flip-flops)
- Counters, Shift registers
- Generic component instantiation
- Aliases and Constants
- Packages
- Structural VHDL
- Mixing design styles

- Next Lecture 8:
  - FSMs