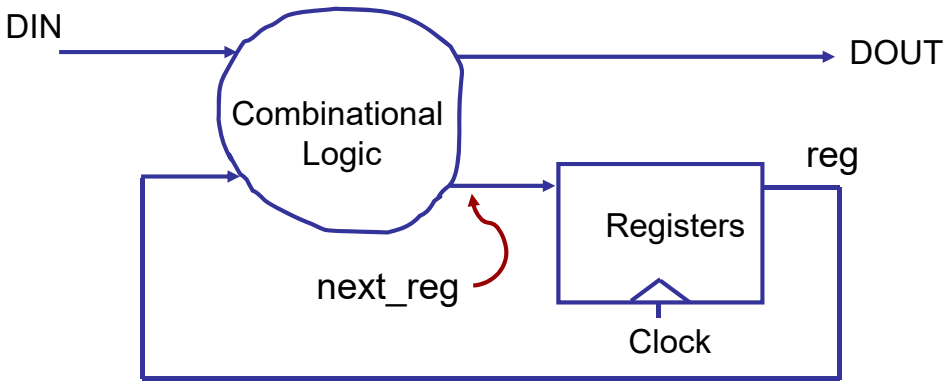
Chapter 6: Modeling at the RT Level

- A register transfer level (RTL) design consists of a set of registers connected by combinational logic.



6.1 Combinational circuit

- A combinational circuit, by definition, is a circuit whose output, after the initial transient period, is a function of current input.
- It has no internal state and therefore is "memoryless" about the past event (or past inputs).

```
signal A, B, Cin, Cout : bit;
…
process (A, B, Cin) is
begin
   Cout <= (A and B) or ((A xor B) and Cin);
end;
```

To describe a combinational circuit

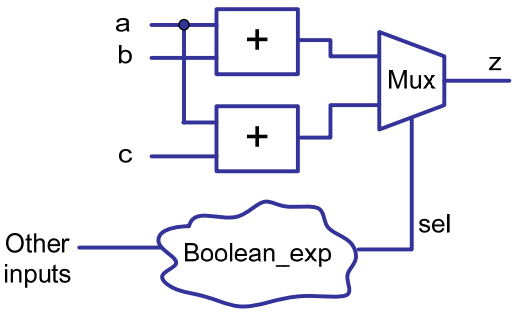
- The variables or signals in the process must not have initial values.
- A signal or a variable must be assigned a value before being referenced.
- The arithmetic operators (such as +, -, *, etc), relational operators (such as <, >, =, etc), and logic operators (such as and, or, not, etc) can be used in an expression.

Operator sharing

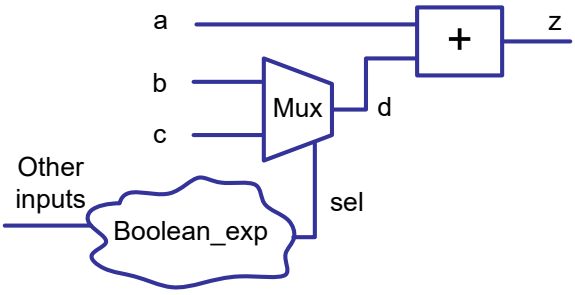
when a control signal is 1, z <= a + b;
Otherwise, z <= a + c;

- One way to reduce the overall size of synthesized hardware is to identify the resources that can be used by different operations. This is know as *resource sharing*.

```
sel <= c1 xor c2;
z <= a + b when sel='1' else
      a + c;
```




```
sel <= c1 xor c2;
d <= b when sel='1' else
      c;
z <= a + d;
```

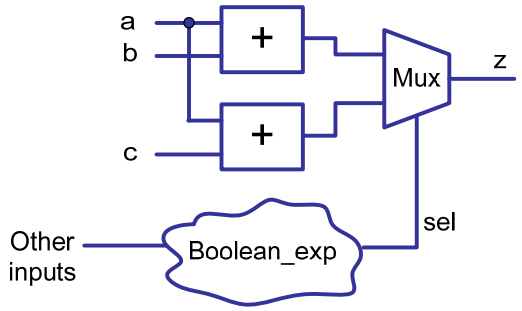


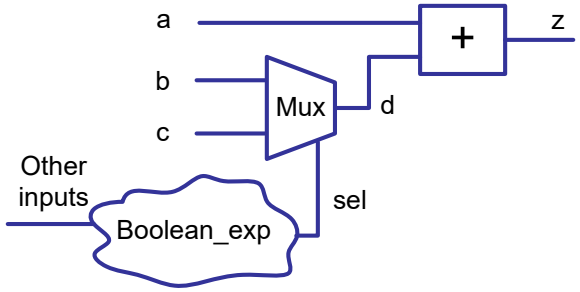
- Performing resource sharing normally introduces some overhead and may penalize performance.
  - In the above examples, assume T_{adder}, T_{mux}, $T_{boolean}$,
    - For the circuit not sharing the adders:

$$T = \max(T_{adder}, T_{boolean}) + T_{mux}$$

    - For the circuit sharing the adders:

$$T = T_{adder} + T_{boolean} + T_{mux}$$





Shaping the circuit

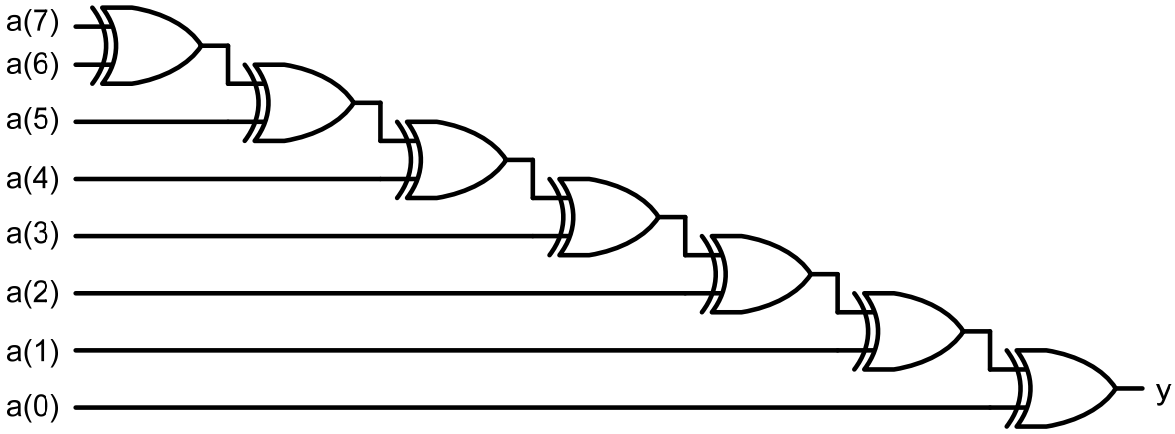
- Using VHDL code, it is possible to outline the general shape of the circuit.

Reduced-xor circuit

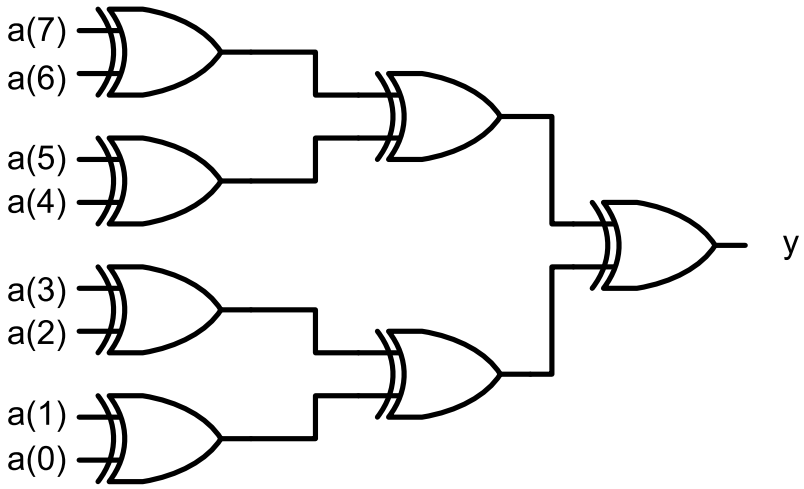
$$y = a_7 \oplus a_6 \oplus a_5 \oplus a_4 \oplus a_3 \oplus a_2 \oplus a_1 \oplus a_0$$

```
signal a: std_logic_vector (7 downto 0);
signal y: std_logic;
…
y <= a(7) xor a(6) xor a(5) xor a(4) xor a(3)
        xor a(2) xor a(1) xor a(0);
```

$$y <= (((((((a_7 \oplus a_6) \oplus a_5) \oplus a_4) \oplus a_3) \oplus a_2) \oplus a_1) \oplus a_0);$$



$$y <= ((a_7 \oplus a_6) \oplus (a_5 \oplus a_4)) \oplus ((a_3 \oplus a_2) \oplus (a_1 \oplus a_0));$$



Example:
Combinational adder-based multiplier

					a_4	a_3	a_2	a_1	a_0	**multiplicand**
x					b_4	b_3	b_2	b_1	b_0	**multiplier**
					a_4b_0	a_3b_0	a_2b_0	a_1b_0	a_0b_0	
				a_4b_1	a_3b_1	a_2b_1	a_1b_1	a_0b_1		
			a_4b_2	a_3b_2	a_2b_2	a_1b_2	a_0b_2			
+		a_4b_3	a_3b_3	a_2b_3	a_1b_3	a_0b_3				
	a_4b_4	a_3b_4	a_2b_4	a_1b_4	a_0b_4					
y_9	y_8	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0	**product**

The algorithm includes three tasks:

- Multiply the digits of the multiplier (b4, b3, b2, b1 and b0) by the multiplicand A = (a4, a3, a2, a1, a0) one at a time to obtain b4*A, b3*A, b2*A, b1*A and b0*A.

  bi * A = (a4 · bi, a3 · bi, a2 · bi, a1 · bi, a0 · bi)
- Shift bi * A to left by i position.
- Add the shifted bi * A terms to obtain the final product.

					a_4	a_3	a_2	a_1	a_0
x					b_4	b_3	b_2	b_1	b_0
					a_4b_0	a_3b_0	a_2b_0	a_1b_0	a_0b_0
				a_4b_1	a_3b_1	a_2b_1	a_1b_1	a_0b_1	
			a_4b_2	a_3b_2	a_2b_2	a_1b_2	a_0b_2		
+		a_4b_3	a_3b_3	a_2b_3	a_1b_3	a_0b_3			
	a_4b_4	a_3b_4	a_2b_4	a_1b_4	a_0b_4				
y_9	y_8	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0

Initial description of an adder-based multiplier

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

entity mult5 is
port (a, b : in std_logic_vector(4 downto 0);
        y: out std_logic_vector(9 downto 0));
end entity mult5;

architecture comb1_arch of mult5 is
constant WIDTH : integer := 5;
signal au, bv0, bv1, bv2, bv3, bv4: std_logic_vector(WIDTH–1 downto 0);
signal p0, p1, p2, p3, p4, prod: std_logic_vector(2*WIDTH–1 downto 0);
```

						a_4	a_3	a_2	a_1	a_0
x						b_4	b_3	b_2	b_1	b_0
						a_4b_0	a_3b_0	a_2b_0	a_1b_0	a_0b_0
					a_4b_1	a_3b_1	a_2b_1	a_1b_1	a_0b_1	
				a_4b_2	a_3b_2	a_2b_2	a_1b_2	a_0b_2		
+			a_4b_3	a_3b_3	a_2b_3	a_1b_3	a_0b_3			
		a_4b_4	a_3b_4	a_2b_4	a_1b_4	a_0b_4				
y_9	y_8	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0	

Array aggregate:
a VHDL construct to assign a value to an object of array data type.

```
v <=  "1011";
v <=  ('1', '0', '1', '1');
v <= (3=>'1', 2=>'0', 1=>'1', 0=>'1');
v <= (3|1|0=>'1', 2=>'0');
v <= (2=>'0', others=>'1');
v <= (others=>'0');
```

same

```
begin
    au <= a;
    bv0 <= (others => b(0));
    bv1 <= (others => b(1));
    bv2 <= (others => b(2));
    bv3 <= (others => b(3));
    bv4 <= (others => b(4));
    p0 <= "00000" & (bv0 and au);
    p1 <= "0000" & (bv1 and au) & '0';
    p2 <= "000" & (bv2 and au) & "00";
    p3 <= "00" & (bv3 and au) & "000";
    p4 <= '0' & (bv4 and au) & "0000";
    prod <= ((p0+p1)+(p2+p3))+p4;
    y <= prod;
end architecture comb1_arch;
```

						a_4	a_3	a_2	a_1	a_0
x						b_4	b_3	b_2	b_1	b_0
						a_4b_0	a_3b_0	a_2b_0	a_1b_0	a_0b_0
					a_4b_1	a_3b_1	a_2b_1	a_1b_1	a_0b_1	
				a_4b_2	a_3b_2	a_2b_2	a_1b_2	a_0b_2		
+			a_4b_3	a_3b_3	a_2b_3	a_1b_3	a_0b_3			
		a_4b_4	a_3b_4	a_2b_4	a_1b_4	a_0b_4				
y_9	y_8	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0	

					a_4	a_3	a_2	a_1	a_0	multiplicand
X					b_4	b_3	b_2	b_1	b_0	multiplier
					a_4b_0	a_3b_0	a_2b_0	a_1b_0	a_0b_0	
				$pp0_5$	$pp0_4$	$pp0_3$	$pp0_2$	$pp0_1$	$pp0_0$	partial product pp0
			+	a_4b_1	a_3b_1	a_2b_1	a_1b_1	a_0b_1		
			$pp1_5$	$pp1_4$	$pp1_3$	$pp1_2$	$pp1_1$	$pp1_0$		partial product pp1
		+	a_4b_2	a_3b_2	a_2b_2	a_1b_2	a_0b_2			
		$pp2_5$	$pp2_4$	$pp2_3$	$pp2_2$	$pp2_1$	$pp2_0$			partial product pp2
	+	a_4b_3	a_3b_3	a_2b_3	a_1b_3	a_0b_3				
	$pp3_5$	$pp3_4$	$pp3_3$	$pp3_2$	$pp3_1$	$pp3_0$				partial product pp3
+	a_4b_4	a_3b_4	a_2b_4	a_1b_4	a_0b_4					
$pp4_5$	$pp4_4$	$pp4_3$	$pp4_2$	$pp4_1$	$pp4_0$					partial product pp4
y_9	y_8	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0	

More efficient description of an adder-based multiplier

```
architecture comb2_arch of mult5 is
constant WIDTH : integer := 5;
signal au, bv0, bv1, bv2, bv3, bv4: std_logic_vector (WIDTH–1
    downto 0);
signal pp0, pp1, pp2, pp3, pp4: std_logic_vector (WIDTH downto 0);
signal prod: std_logic_vector (2*WIDTH–1 downto 0);
```

						a_4	a_3	a_2	a_1	a_0	multiplicand
X						b_4	b_3	b_2	b_1	b_0	multiplier
						a_4b_0	a_3b_0	a_2b_0	a_1b_0	a_0b_0	
					$pp0_5$	$pp0_4$	$pp0_3$	$pp0_2$	$pp0_1$	$pp0_0$	partial product pp0
				+	a_4b_1	a_3b_1	a_2b_1	a_1b_1	a_0b_1		
				$pp1_5$	$pp1_4$	$pp1_3$	$pp1_2$	$pp1_1$	$pp1_0$		partial product pp1
			+	a_4b_2	a_3b_2	a_2b_2	a_1b_2	a_0b_2			
		$pp2_5$	$pp2_4$	$pp2_3$	$pp2_2$	$pp2_1$	$pp2_0$				partial product pp2
	+	a_4b_3	a_3b_3	a_2b_3	a_1b_3	a_0b_3					
	$pp3_5$	$pp3_4$	$pp3_3$	$pp3_2$	$pp3_1$	$pp3_0$					partial product pp3
+	a_4b_4	a_3b_4	a_2b_4	a_1b_4	a_0b_4						
$pp4_5$	$pp4_4$	$pp4_3$	$pp4_2$	$pp4_1$	$pp4_0$						partial product pp4
y_9	y_8	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0		

```
begin
    au <= a;
    bv0 <= (others => b(0));
    bv1 <= (others => b(1));
    bv2 <= (others => b(2));
    bv3 <= (others => b(3));
    bv4 <= (others => b(4));
    pp0 <= '0' & (bv0 and au);
    pp1 <= ('0' & pp0(WIDTH downto 1))+ ('0' & (bv1 and au));
    pp2 <= ('0' & pp1(WIDTH downto 1))+ ('0' & (bv2 and au));
    pp3 <= ('0' & pp2(WIDTH downto 1))+ ('0' & (bv3 and au));
    pp4 <= ('0' & pp3(WIDTH downto 1))+ ('0' & (bv4 and au));
    prod <= pp4 & pp3(0) & pp2(0) & pp1(0) & pp0(0);
    y <= prod;
end architecture comb2_arch;
```

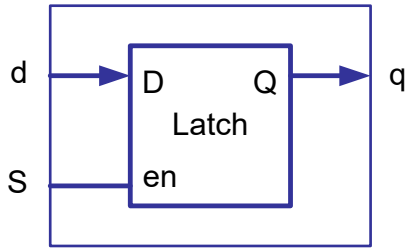

6.2 Sequential circuit

- A sequential circuit is a circuit that has an internal state, or memory.
- Its output is a function of current input as well as the internal state. Thus the output is affected by current input values as well as past input values.
- A synchronous sequential circuit, in which all memory elements are controlled by a global synchronizing signal, greatly simplifies the design process and is the most important design methodology.
- Flip-flops and latches are two commonly used one-bit memory devices.

6.2.1 Latch

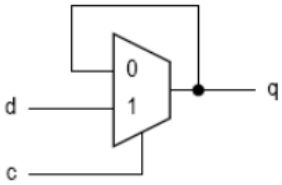
- A latch is a level-sensitive memory device.

```
signal S, d, q: bit
......
process (S, d) is
begin
    if (S='1') then
            q <= d;
    end if;
end process;
```

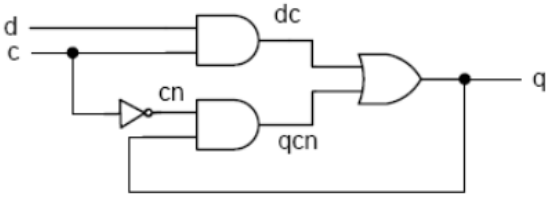


Truth table

S	q*
0	q
1	d



Conceptual diagram



Gate-level diagram

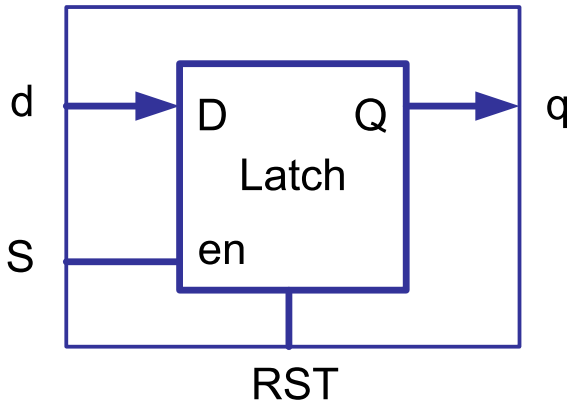
- In general, latches are synthesized from incompletely specified conditional expressions in a combinational description.
- Latch inferences occur normally with **if** statements or **case** statements.
- To avoid having a latch inferred, assign a value to the signal under all conditions.

```
signal S, d, q: bit
......
process (S, d) is
begin
    if (S='1') then q <= d;
    else q <= '0';
    end if;
end process;
```

asynchronous reset or preset

- An asynchronous reset (or preset) will change the output of a latch to 0 (or 1) immediately.

```
signal S, RST, d, q: bit
......
process (S, RST, d) is
begin
    if (RST = '1') then
        q <= '0';
    elsif (S='1') then
        q <= d;
    end if;
end process;
```



6.2.2 Flip-Flops (f/f)

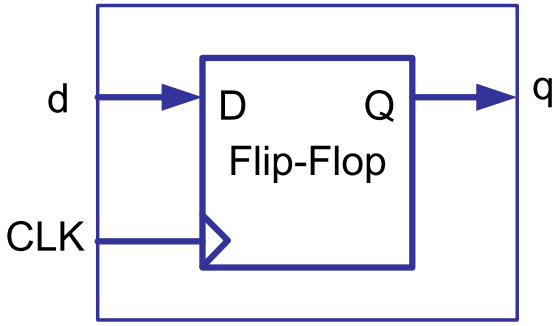
- A flip-flop is an edge-triggered memory device.
- To detect the rising edge (or falling edge), or the event occurred for a signal, we can make use of the attribute of a signal.

```
signal CLK : bit;
......
CLK'event                    true if CLK changes its value.
CLK'event and CLK = '1'      true for the CLK rising edge
CLK'event and CLK = '0'      true for the CLK falling edge
```

- The **event** attribute on a signal is the most commonly used edge-detecting mechanism. It operates on a signal and returns a Boolean value. The result is true if the signal shows a change in value.

An example of a simple flip-flop

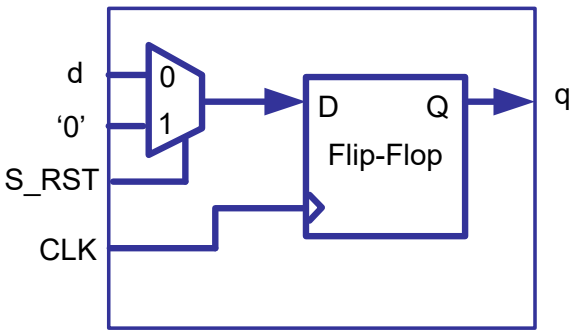
- An edge triggered flip-flop will be generated from a VHDL description if a signal assignment is executed on the rising (or falling) edge of another signal.




```
entity dff is
port (d, CLK: in bit; q: out bit);
end entity dff;
architecture behavior of dff is
begin
process (CLK) is
begin
    if (CLK'event and CLK='1') then
        q <= d;
    end if;
end process;
end architecture behavior;
```


Synchronous sets and resets

- Synchronous inputs set (preset) or reset (clear) the output of flip-flops when they are asserted. The assignment will only take effect while the clock edge is active.

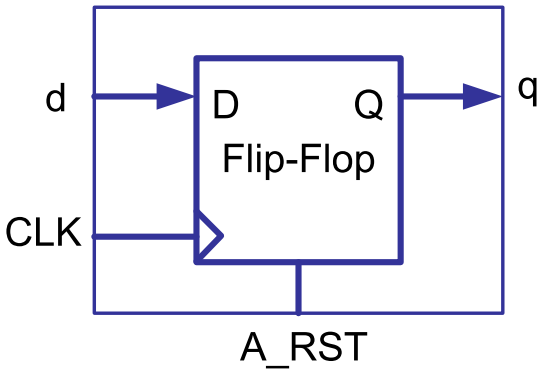



```
signal CLK, d, q, S_RST: bit;
......
process (CLK) is
begin
    if (CLK'event and CLK='1') then
        if (S_RST = '1') then
            q <= '0';
        else
            q <= d;
        end if;
    end if;
end process;
```

Asynchronous sets and resets

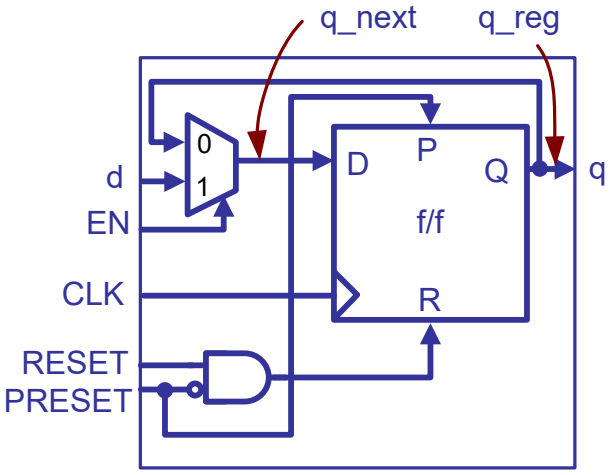
- Asynchronous inputs set (preset) or reset (clear) the output of flip-flops whenever they are asserted independent of the clock.

```
signal CLK , A_RST, d, q: bit;
......
process (CLK, A_RST) is
begin
    if (A_RST = '1') then
        q <= '0';
    elsif (CLK'event and CLK='1') then
        q <= d;
    end if;
end process;
```



A f/f with more than one asynchronous input

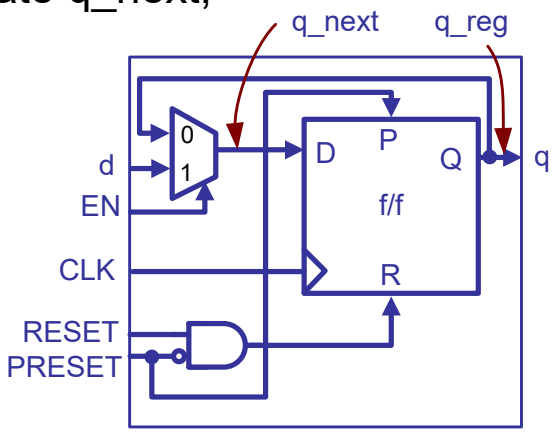
```
signal CLK , RST, PRST, EN: bit;
signal d, q: bit;
......
architecture two_seg of dff_en is
signal q_reg, q_next : bit;
begin
process (CLK, PRST, RST) is
begin
    if (PRST = '1') then
        q_reg <= '1';
    elsif (RST = '1') then
        q_reg <= '0';
    elsif (CLK'event and CLK='1') then
        q_reg <= q_next;
    end if;
end process;
```

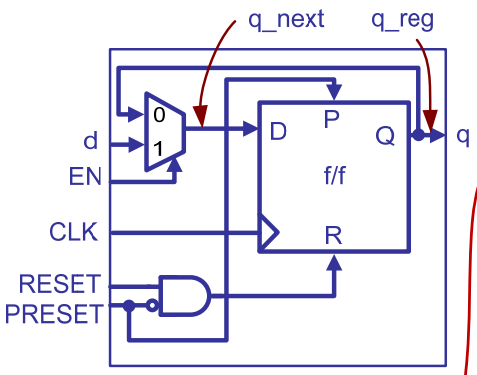



```
q_next <= d when EN = '1' else
            q_reg;
q <= q_reg;
end architecture two_seg;
```

6.2.3 VHDL templates for sequential circuits

- An RTL circuit can be described in two segments:
  - A synchronous section updates the register information at the rising edge of the clock.
    - q_reg <= q_next;
  - A combinational section describes combinational logics, for example, update q_next;





A synchronous section

```
if (CLK'event AND CLK='1') then
    q_reg <= q_next;
    -- CLK is the clock input to reg q
end if;
```

or

A synchronous section with asynchronous inputs

```
if (async_sig = '1') then
    q_reg <= '0';
    -- active high asynchronous reset
elsif (CLK's event AND CLK='1') then
    q_reg <= q_next;
    -- CLK is the clock input to reg Q
end if;
```

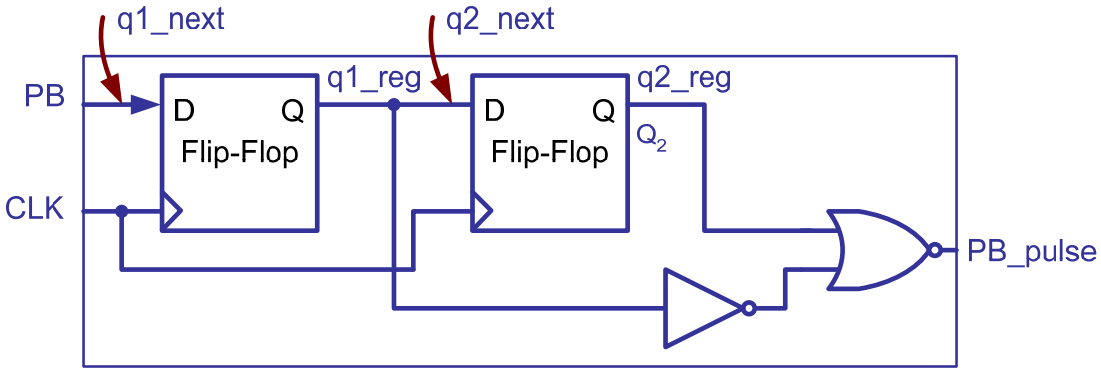
+

A combinational section

```
q_next <= expression;
-- other combinational logics.
```

An example

```
entity PULSER is
    port (CLK, PB : in bit ;
          PB_pulse : out bit);
end PULSER;

architecture BHV of PULSER is
    signal q1_reg, q2_reg,
           q1_next, q2_next : bit;
begin
    process (CLK) is
    begin
        if (CLK'event and CLK='1') then
            q1_reg <= q1_next;
            q2_reg <= q2_next;
        end if;
    end process;
        q1_next <= PB;
        q2_next <= q1_reg;
        PB_pulse <= (not q1_reg) nor q2_reg;
end architecture BHV;
```

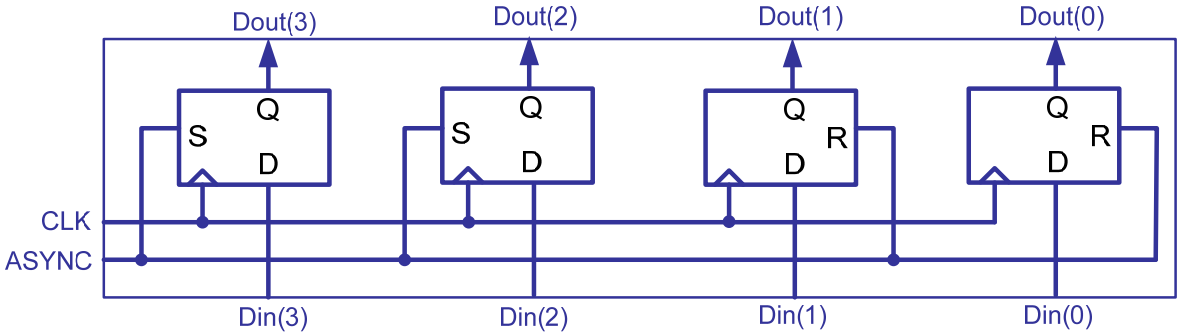


6.2.4 Registers

-- 4-bit simple register

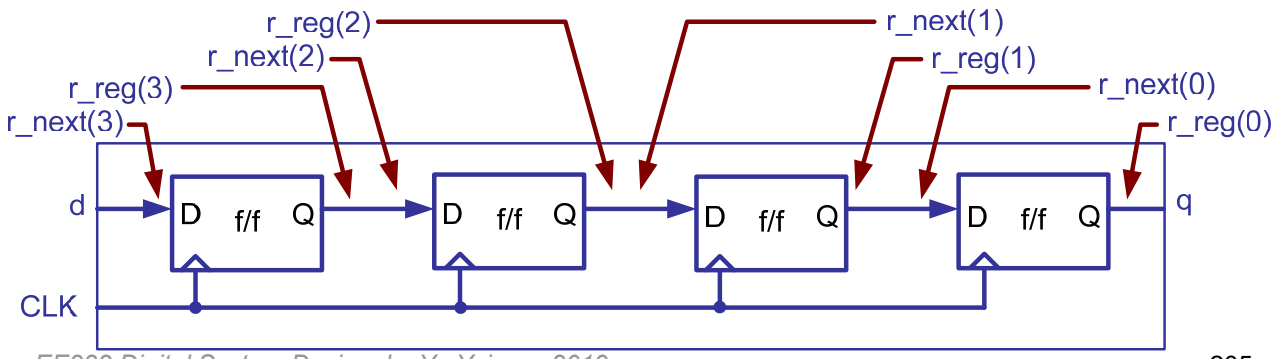
```
signal CLK , ASYNC : bit;
singal Din, Dout :
    bit_vector (3 down to 0);
......
```

```
process (CLK, ASYNC) is
begin
    if (ASYNC = '1') then
        Dout <= "1100";
    elsif (CLK'event and CLK='1') then
        Dout <= Din;
    end if;
end process;
```



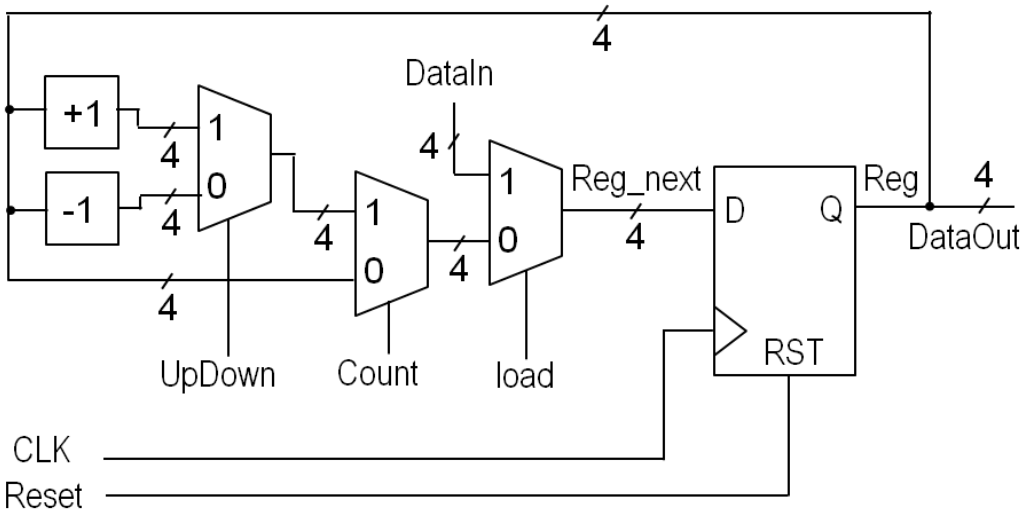
-- 4-bit serial-in and serial-out shift register

```
signal CLK ,d, q : bit;
......
architecture two_seg of shift_register is
signal r_reg, r_next: bit_vector (3 downto 0);
begin
process (CLK) is
begin
    if (CLK'event and CLK='1') then
        r_reg <= r_next;
    end if;
end process;
r_next <= d & r_reg(3 downto 1);
q <= r_reg(0);
end architecture two_seg;
```

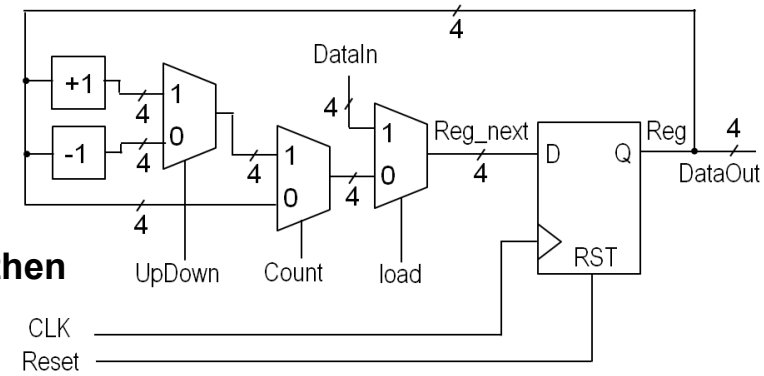


6.2.5 Synchronous counter

```
-- 4-bit synchronous counter
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity counter is
port (CLK, RESET: in std_logic;
      load, Count, UpDown: in std_logic;
      DataIn: in std_logic_vector(3 downto 0);
      DataOut: out std_logic_vector(3 downto 0));
end entity counter;

architecture two-seg of counter is
signal Reg, Reg_Next : std_logic_vector (3 downto 0);
```



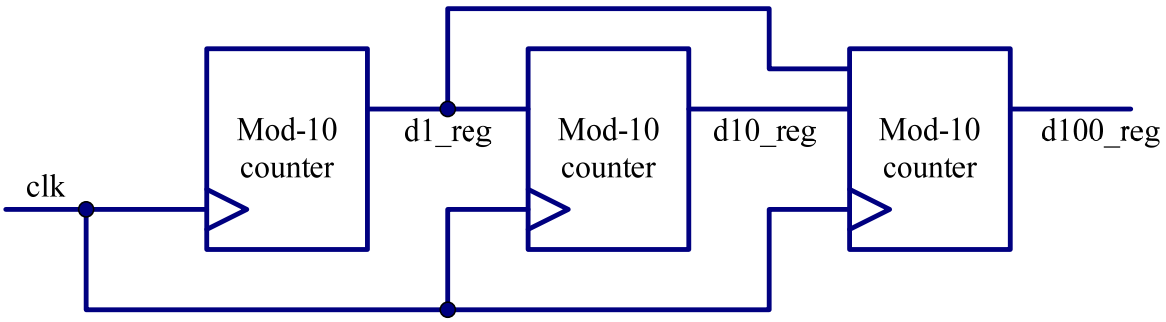
Conceptual Diagram




```
begin
process (CLK, RESET) is
begin
if RESET = '1' then
    Reg <="0000";
elsif CLK'event and CLK='1' then
    Reg <= Reg_Next;
end if;
end process;
-- next-state logic
Reg_next <= DataIn when load = '1' else
     (Reg+1) when (Count='1' and UpDown = '1') else
     (Reg–1) when (Count='1' and UpDown ='0') else
     Reg;
-- Output logic
DataOut <= Reg;
end architecture two-seg;
```

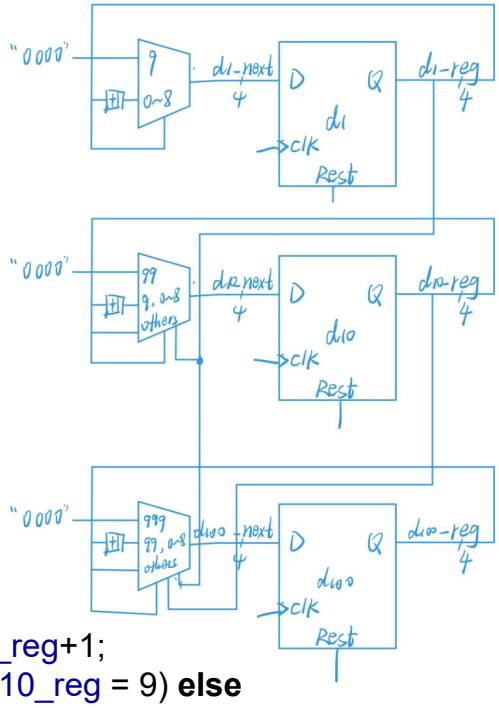
Decimal counter

- A decimal counter circulates the patterns in binary-coded decimal (BCD) format.
- The BCD code use 4 bits to represent a decimal number.



Three-digit decimal counter using conditional concurrent statements

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

entity decimal_counter is
port (CLK, RESET: in std_logic;
      d1, d10, d100: out std_logic_vector(3 downto 0));
end entity decimal_ounter;

architecture concurrent_arch of decimal_ounter is
signal d1_reg, d10_reg, d100_reg: std_logic_vector (3 downto 0);
signal d1_next, d10_next, d100_next: std_logic_vector (3 downto 0);

begin
```

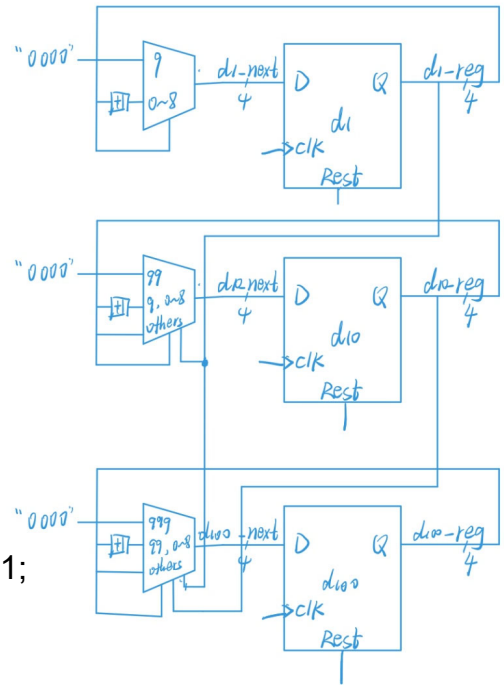



```
-- register
process (CLK, RESET) is
begin
if RESET = '1' then
    d1_reg <="0000";
    d10_reg <="0000";
    d100_reg <="0000";
elsif CLK'event and CLK='1' then
    d1_reg <= d1_next;
    d10_reg <= d10_next;
    d100_reg <= d100_next;
end if;
end process;
-- next-state logic
d1_next <= "0000" when d1_reg = 9 else d1_reg+1;
d10_next <= "0000" when (d1_reg = 9 and d10_reg = 9) else
                d10_reg+1 when d1_reg = 9 else d10_reg;
d100_next <= "0000" when (d1_reg=9 and d10_reg=9 and d100_reg=9) else
                d100_reg+1 when (d1_reg=9 and d10_reg=9) else d100_reg;
-- Output logic
d1 <= d1_reg; d10 <= d10_reg; d100 <= d100_reg;
end architecture concurrent_arch;
```


Three-digit decimal counter using a nested if statement

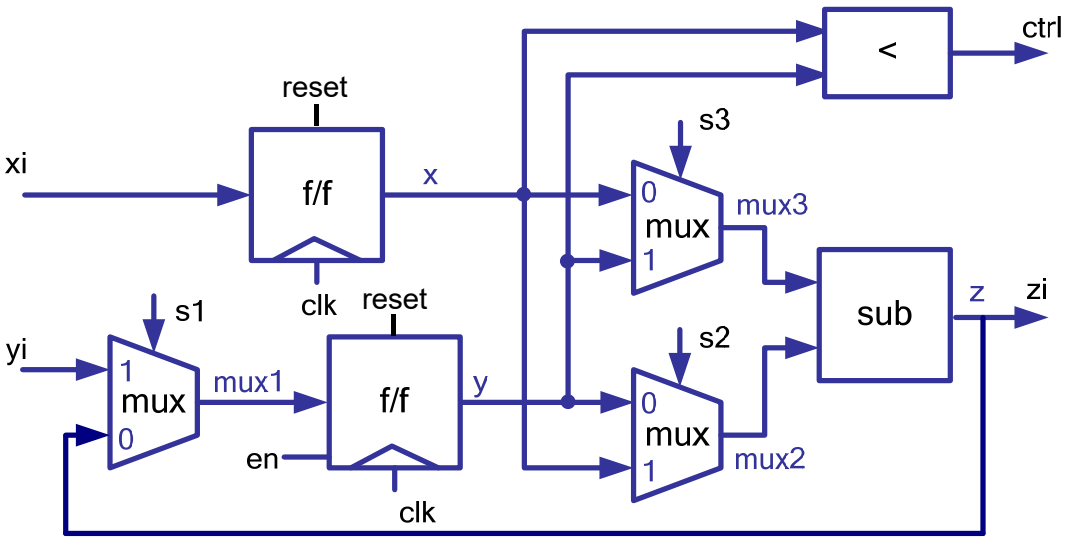
```
architecture if_arch of decimal_ounter is
    signal d1_reg, d10_reg, d100_reg: std_logic_vector (3 downto 0);
    signal d1_next, d10_next, d100_next: std_logic_vector (3 downto 0);
begin
process (CLK, RESET) is
begin
if RESET = '1' then
    d1_reg <="0000";
    d10_reg <="0000";
    d100_reg <="0000";
elsif CLK'event and CLK='1' then
    d1_reg <= d1_next;
    d10_reg <= d10_next;
    d100_reg <= d100_next;
end if;
end process;
```

```
-- next-state logic
process (d1_reg, d10_reg, d100_reg)
begin
    d10_next <= d10_reg;
    d100_next <= d100_reg;
    if d1_reg /= 9 then
        d1_next = d1_reg+1;
    else -- reach 9
        d1_next = "0000";
        if d10_reg /= 9 then
            d10_next <= d10_reg + 1;
        else -- reach 99
            d10_next <= "0000";
            if d100_reg /= 9 then
                d100_next <= d100_reg + 1;
            else -- reach 999
                d100_next <= "0000";
            end if;
        end if;
    end if;
end process;
-- Output logic
d1 <= d1_reg; d10 <= d10_reg; d100 <= d100_reg;
end architecture if_arch;
```

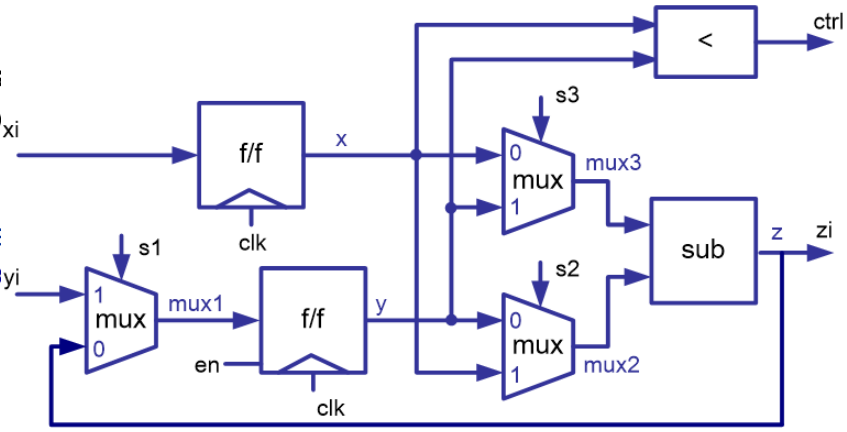


6.3 Netlist of RTL components

- A data path usually consists of a netlist of RTL components such as function units, multiplexers, comparators, registers, etc.



```
signal clk, en, s1, s2, s3 : s
signal xi, yi, zi : std_logic_v
signal ctrl : boolean;
……
architecture two_seg of da
signal x, y, z, x_next, y_ne
begin
process (clk) is
begin
if reset ='1' then
   x <= "00000000";
   y <= "00000000";
elsif (clk'event and clk='1') then -- registers
   x <= x_next;
   y <= y_next;
end if;
end process;
ctrl <= (x<y); -- comparator
x_next <= xi;
y_next <= mux1 when en = '1' else
          y;
```




```
mux1 <= z when s1 = '0' else
        yi;        -- multiplexer
mux2 <= y when s2 = '0' else
        x; -- multiplexer
mux3 <= x when s3 = '0' else
        y;         -- multiplexer
z <= mux3 – mux2;  -- subtracter
zi <= z;
end architecture two_seg;
```


6.4 Test benches for sequential system

- All synchronous system require a system clock signal.
- A reset signal is required. The reset signal is asserted at power on to place the sequential system in its initial state.

6.4.1 Generating a system clock

- 50% duty cycle clock

```
clock_gen: process
constant period : time := 100 ns;
begin
  clk <= '0';
  wait for period/2;
  clk <= '1';
  wait for period/2;
end process;
```


6.4.2 Generating the system reset

- The reset signal typically
  - starts in its asserted state at power on,
  - remains in that state for a specified period of time, then
  - changes to its unasserted state, and
  - remains there for as long as power continues to be applied to the system.

- The duration of the assertion of the reset signal is specified as
  - either a fixed time

```
reset <= '1', '0' after 160 ns;
```

  - or some multiple of the clock's period and is synchronized to the system clock

```
reset_process : process
begin
   reset <= '1';
   for i in 1 to 2 loop
      wait until clk = '1';
   end loop;
   reset <= '0';
   wait;
end process;
```

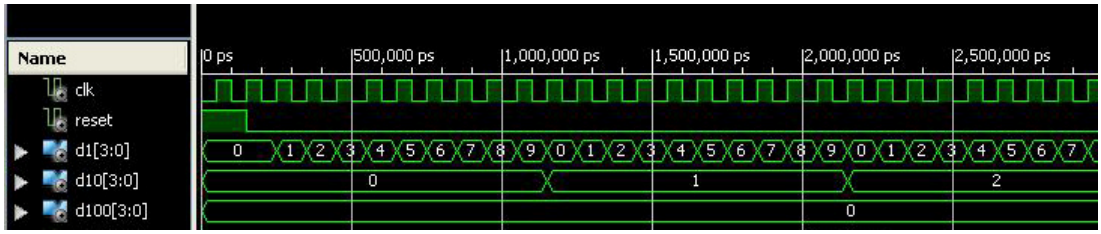

6.4.3 Synchronizing stimulus generation and monitoring

```
monitor : process
    constant n : integer := 1000;
    variable number : integer range 0 to 999 :=0;
begin
    wait until reset <= '0';
    wait for 1 ns;
    for i in 0 to n loop
       number := to_integer(unsigned(d100))*100+to_integer(unsigned(d10))*10
                 + to_integer(unsigned(d1));
       assert number = i mod n
       report "count of " & integer'image(i mod n) & " failed"
       severity error;
       wait until clk = '1';
       wait for 1 ns;
    end loop;
    wait;
end process;
```

```
-- register
process (CLK, RESET) is
begin
if RESET = '1' then
    d1_reg <="0000";
    d10_reg <="0000";
    d100_reg <="0000";
elsif CLK'event and CLK='1' then
    d1_reg <= d1_next;
    d10_reg <= d10_next;
    d100_reg <= d100_next;
end if;
end process;
```

```
-- next-state logic
d1_next <= "0000" when d1_reg = 9 else d1_reg+1;
d10_next <= "0000" when (d1_reg = 9 and d10_reg = 9) else
                d10_reg+1 when d1_reg = 9 else d10_reg;
d100_next <= "0000" when (d1_reg=9 and d10_reg=9 and d100_reg=9) else
                d100_reg+1 when (d1_reg=9 and d10_reg=9) else d100_reg;
-- Output logic
d1 <= d1_reg; d10 <= d10_reg; d100 <= d100_reg;
end architecture concurrent_arch;
```

Waveforms of clk and reset




```
use ieee.numeric_std.all;
signal x : std_logic_vector(3 downto 0); -- vector with element std_logic
signal y : unsigned(3 downto 0); -- vector with element std_logic
signal z : integer range 0 to 15;
```

conversion between std_logic_vector, signed, unsigned

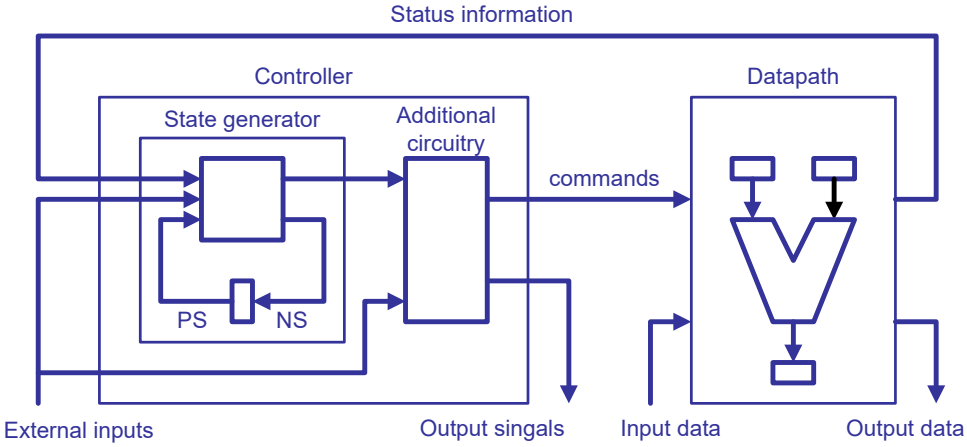
```
x <= y;  -- illegal assignment, type conflict
y <= x;  -- illegal assignment, type conflict

x <= std_logic_vector(y); -- legal assignment
y <= unsigned(x);  -- legal assignment
```

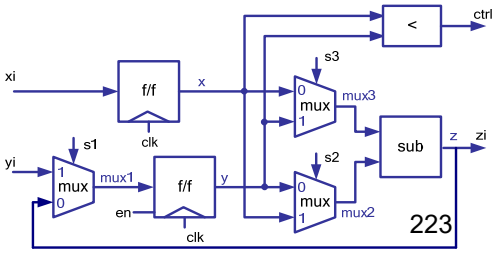
conversion between signed, unsigned, integer

```
z <= to_integer(y); -- legal assignment
z <= to_integer(unsigned(x)); -- legal assignment
y <= to_unsigned(z, 4); -- legal assignment
x <= std_logic_vector(to_unsigned(z, 4)); -- legal assignment
```

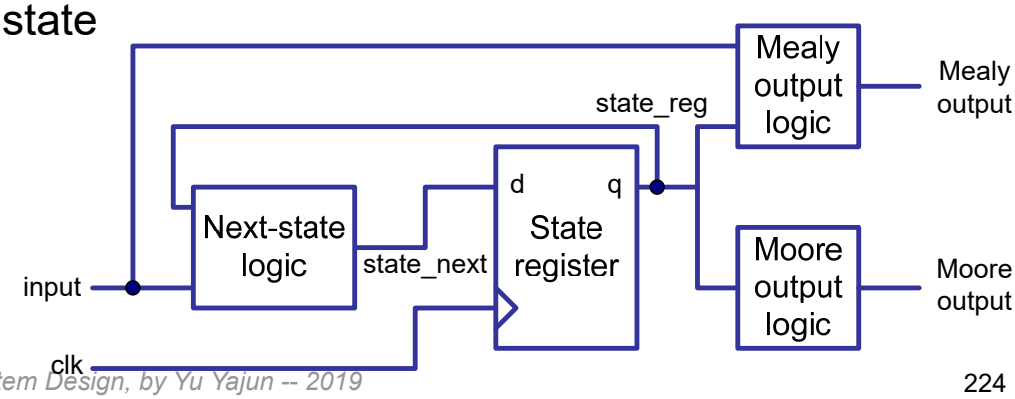
Chapter 7: Modeling at the FSMD Level

- A digital design is conceptually divided into two parts – a controller and a datapath.



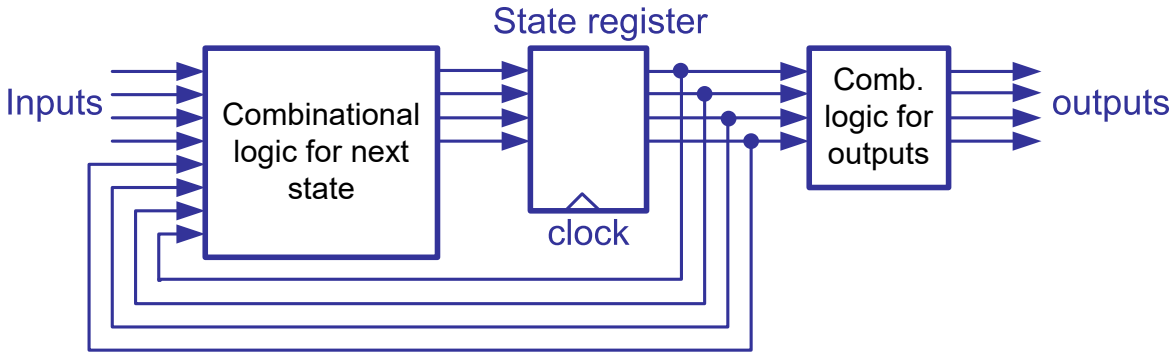


- A sequential circuit which is implemented in a fixed number of possible states is called a finite state machine (FSM).
- It contains five elements:
  - symbolic state
  - input signal
  - output signal
  - present state
  - next state
- Two types of FSM:
  - Moore machines
  - Mealy machines

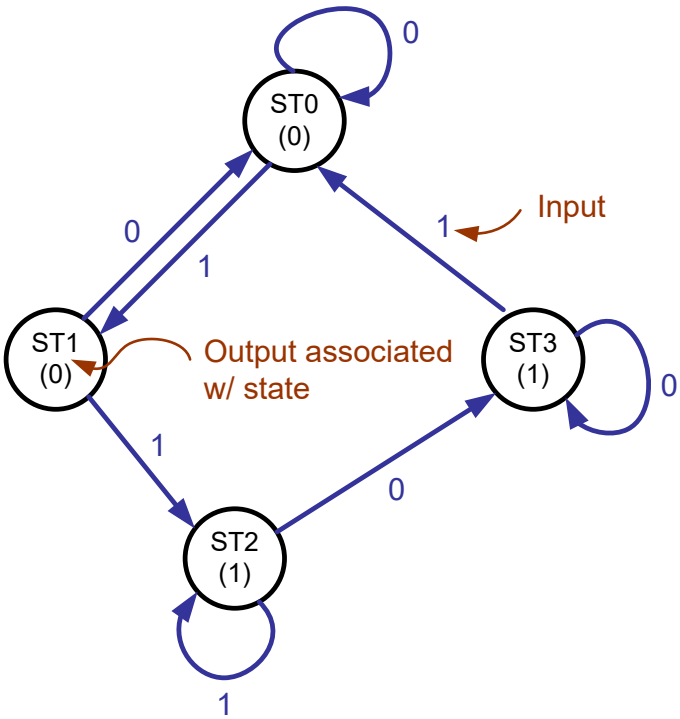


7.1 Moore machine

- In the Moore modal of sequential circuits, the outputs are the functions of the present state only.



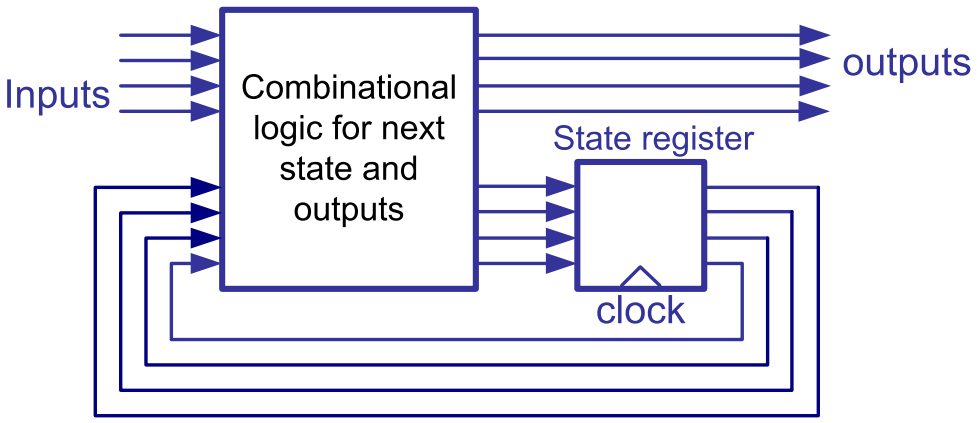
A state transition diagram of a Moore machine



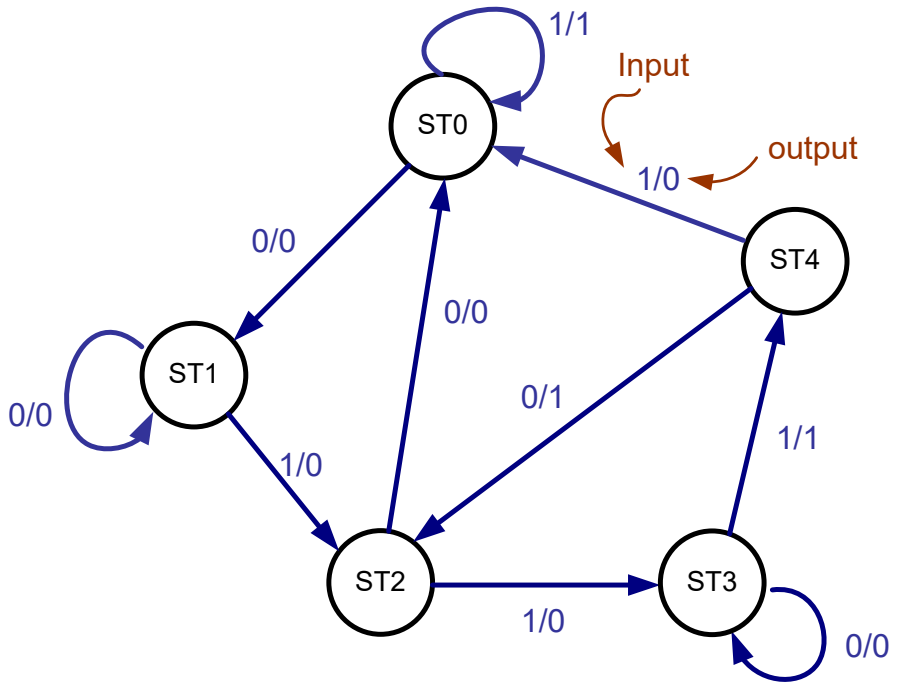
```
-- Moore machine
entity MOORE is
port (Clk, RST, I : in std_logic;
        O : out std_logic);
end entity MOORE;

architecture two_seg_arch of MOORE is
type state_type is (ST0, ST1, ST2, ST3);
signal State, Next_State :  state_type;
begin
  clk_proc: process (CLK, RST) is
  begin
        if (RST = '1') then
            State <= ST0;
        elsif (Clk'event and Clk = '1') then
            State <= Next_State;
        end if;
  end process clk_proc;
  comb_proc: process (State, I) is
  begin
case State is
  when ST0 =>
    O <= '0' ;
    if ( I ='0') then Next_State <= ST0;
    else              Next_State <= ST1;
    end if;
  when ST1 =>
    O <= '0' ;
    if ( I = '0' ) then Next_state <= ST0;
    else                Next_State <= ST2;
    end if;
  when ST2 =>
    O <= '1' ;
    if ( I ='0') then Next_State <= ST3;
    else              Next_State <= ST2;
    end if;
  when ST3 =>
    O <= '1' ;
    if ( I = '0' ) then Next_state <= ST3;
    else                Next_State <= ST0;
    end if;
end case; end process comb_proc;
end architecture two_seg_arch;
```


7.2 Mealy machine

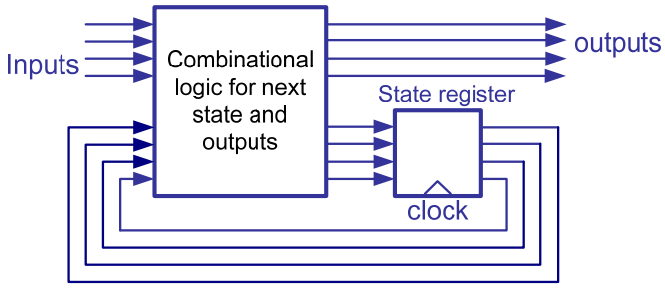
- In the Mealy modal, the outputs are the functions of both the present state and current inputs.



A state transition diagram of a Mealy machine



-- two segments coding style




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

entity MEALY is
port (Clk, RST, I : in std_logic;
        O : out std_logic);
end entity MEALY;

architecture two_seg_arch of MEALY is
type state_type is (ST0, ST1, ST2, ST3, ST4);
signal State, Next_State :  state_type;
begin

  clk_proc: process (CLK, RST) is
  begin
        if (RST = '1') then
            State <= ST0;
        elsif (Clk'event and Clk = '1') then
            State <= Next_State;
        end if;
  end process clk_proc;
```

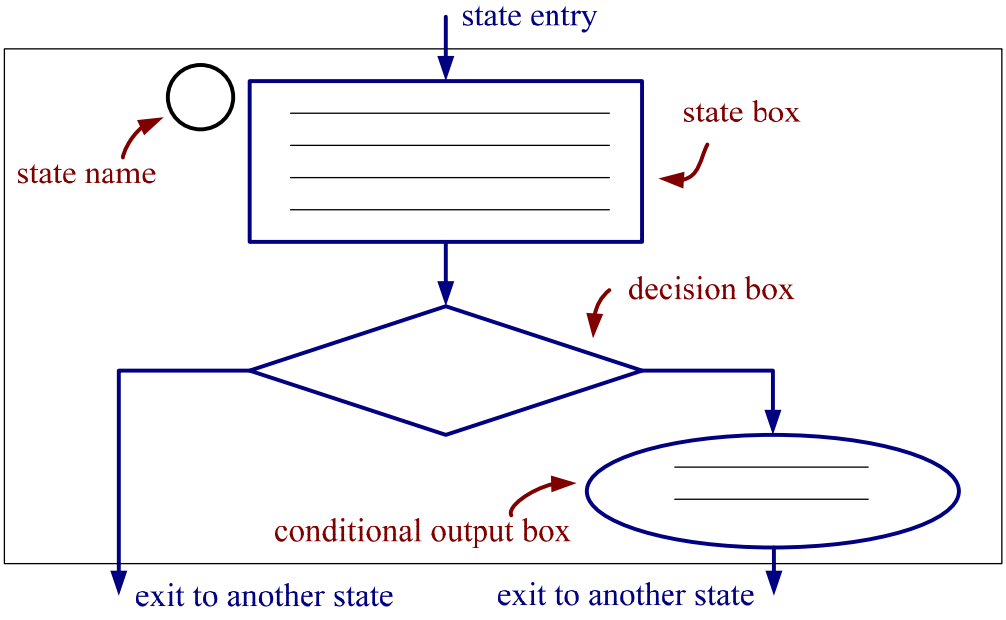
```
comb_proc: process (State, I) is
begin
case State is
  when ST0 =>
     if ( I ='0') then
          O <= '0' ;
          Next_State <= ST1;
     else
          O <= '1';
          Next_State <= ST0;
     end if;
  when ST1 =>
     if ( I ='0') then
          O <= '0' ;
          Next_State <= ST1;
     else
          O <= '0';
          Next_State <= ST2;
     end if;
  when ST2 =>
     if ( I ='0') then O <= '0'; Next_State <= ST0;
     else O <= '0';  Next_State <= ST3;
     end if;
  when ST3 =>
     if ( I ='0') then O <= '0'; Next_State <= ST3;
     else O <= '1';  Next_State <= ST4;
     end if;
  when ST4 =>
     if ( I ='0') then O <= '1' ;Next_State <= ST2;
     else O <= '0';  Next_State <= ST0;
     end if;
end case;
end process comb_proc;
end architecture two_seg_arch;
```


7.3 An FSM with a datapath (FSMD)

- A traditional FSM
  - cannot represent storage elements (register) except the state registers.
  - works well for a design with a few to several hundred states.
- An FSM with a datapath (FSMD) is an extension of a traditional FSM.
  - storage and signals can be declared.
  - Within a state expression, comparison, arithmetic or logic operations on these signals can be performed.

Algorithm state machine (ASM) chart

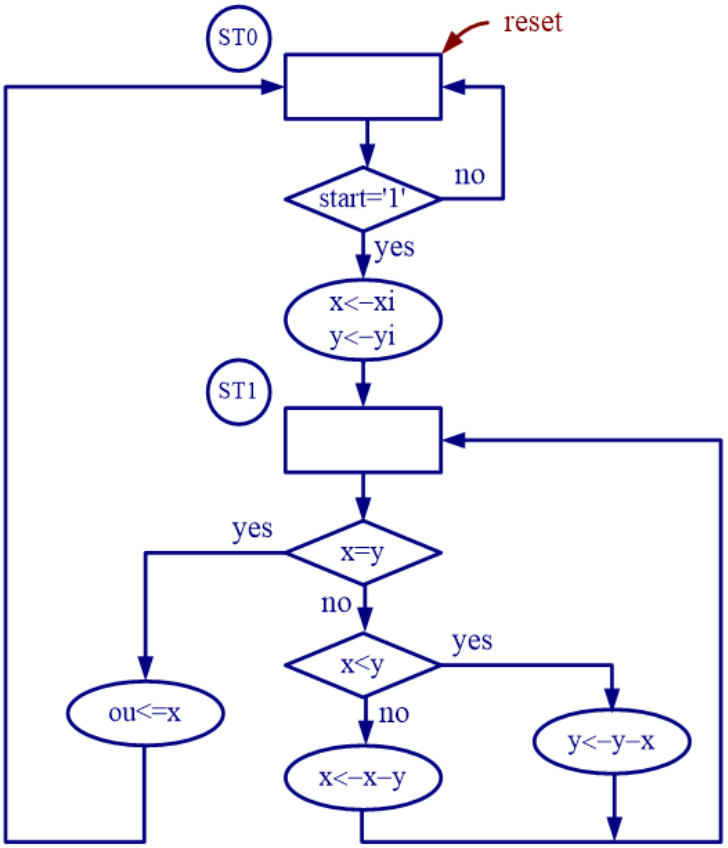
- The behavior of a FSMD can be represented as a flow-chart-like description – algorithm state machine (ASM) chart.
- ASM chart is constructed from ASM blocks;
- An ASM block consists of three basic elements:
  - the state box
  - the decision box
  - the conditional output box.

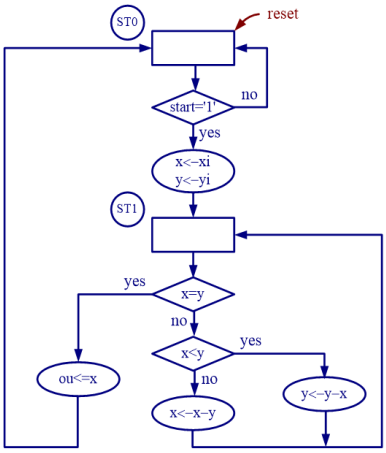


Example:

find the greatest common divisor of two eight-bit numbers xi and yi

```
        x = xi;
        y= yi;
St1:    If x=y then
            ou=x;
        Else {
            if x> y then
                x = x-y;
            Else y= y-x;
            Go to st1;
        }
```

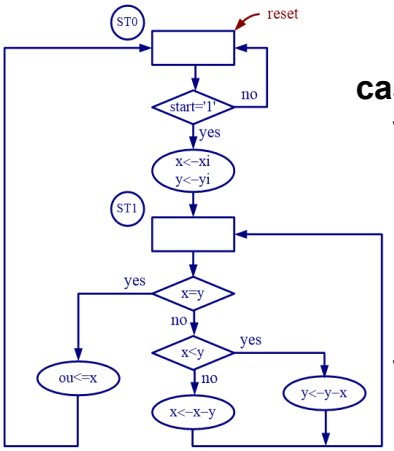




-- GCD calculator

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity GCD is
   Port ( xi, yi : in  std_logic_vector(7 downto 0);
          clk, reset, start : in  std_logic;
          ou : out  std_logic_vector (7 downto 0));
end entity GCD;

architecture FSMD of GCD is
    signal x, y, x_next, y_next: std_logic_vector(7 downto 0);
    type S_Type is (ST0, ST1);
    signal state, next_state : S_Type;
```

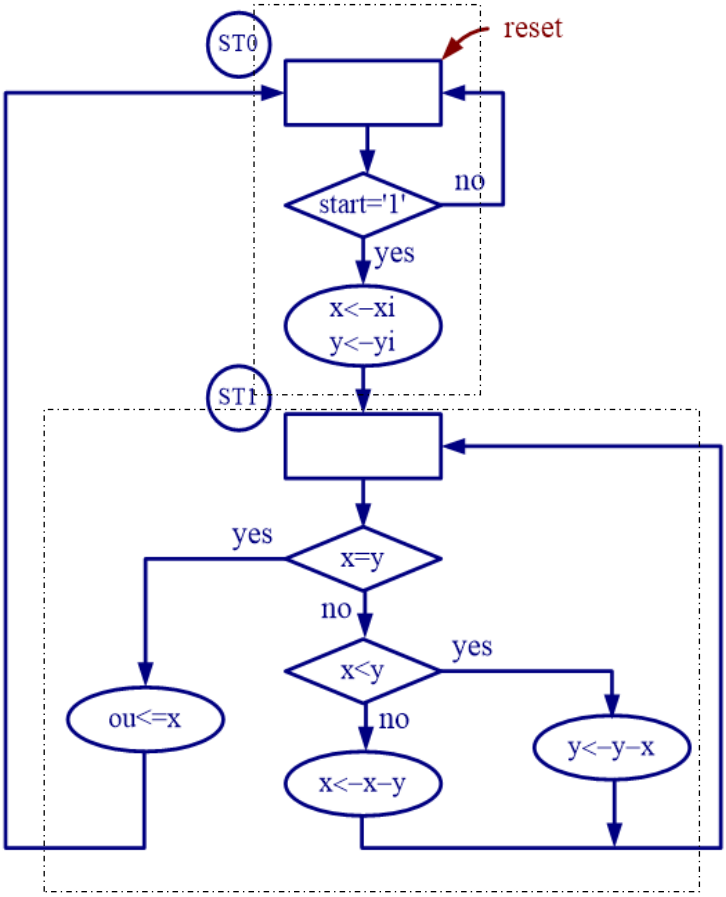
```
begin
 clkproc: process (clk, reset) is
   begin
       if (reset = '1') then
          state <= ST0;
          x <= 0;
          y <= 0;
       elsif (clk'event and clk = '1') then
          state <= next_state;
          x <= x_next;
          y <= y_next;
       end if;
   end process clkproc;

 operproc: process (state, xi, yi, x, y, start) is
 begin
   ou <= (others => '0');
   x_next <= x;
   y_next <= y;
   next_state <= ST0;
```




```
case state is
   when ST0 =>
       if ( start ='1') then
           x_next <= xi;
           y_next <= yi;
           next_state <= ST1;
       end if; -- (start = '1')
when ST1 =>
       if ( x = y ) then
           ou <= x;
       else
           if ( x < y) then
               y_next <= y – x;
           else -- ( x > y)
               x_next <= x – y;
           end if;
           next_state <= ST1;
       end if;
   end case;  -- State
end process operproc;
end architecture FSMD;
```


Example:

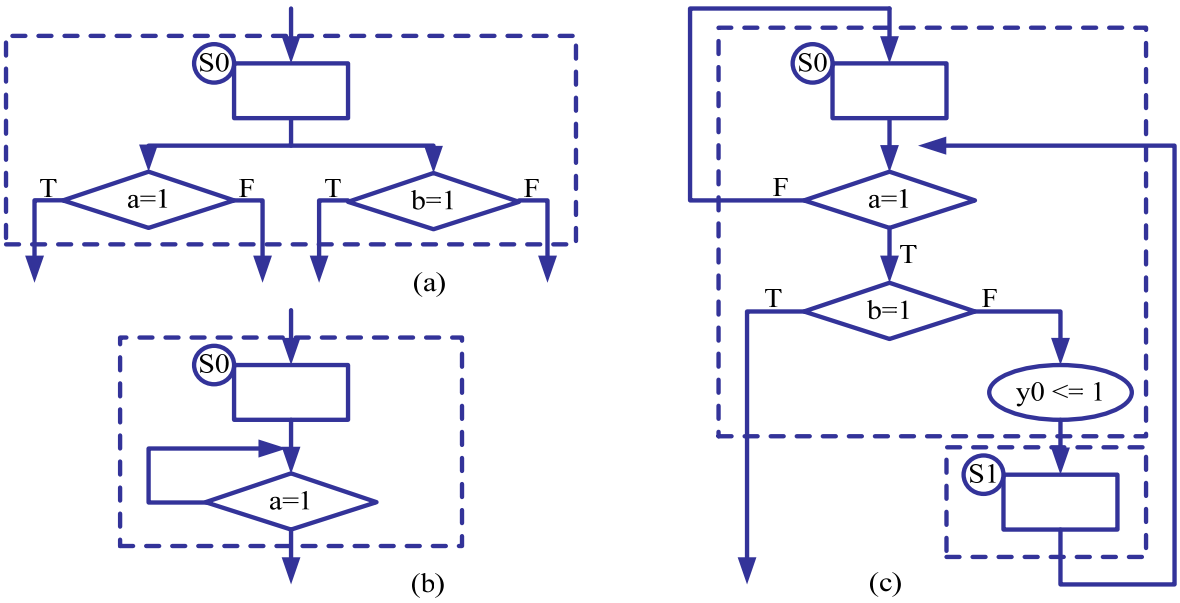
find the greatest common divisor of two eight-bit numbers xi and yi



Rules to Construct ASM Chart:

- For a given input combination, there is one unique exit path from the current AMS block.
- The exit path of an ASM block must always lead to a state box. The state box can be the state box of the current ASM block or a state box of another ASM block.

Common errors in ASM Chart Construction



Example:
FSMD design of a repetitive-addition multiplier

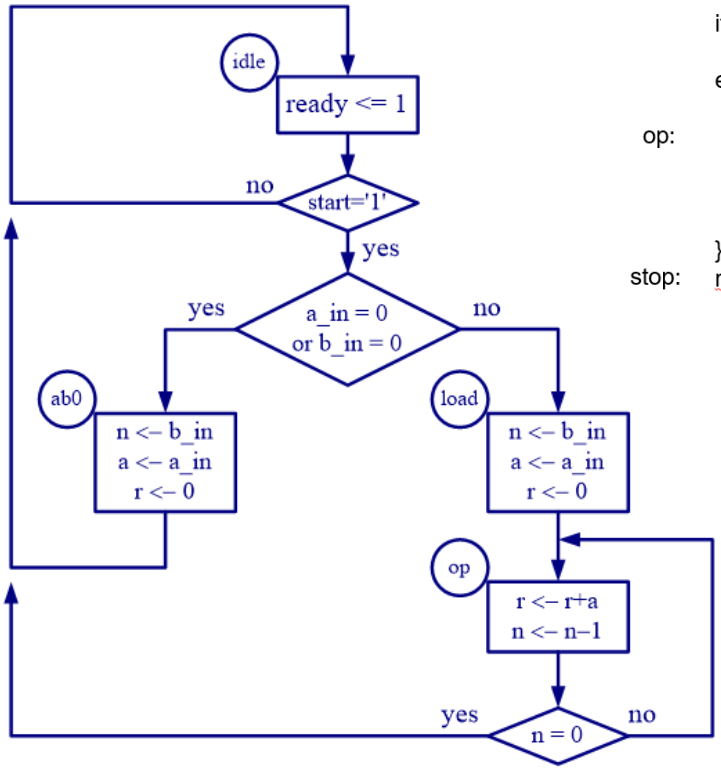
- Consider a multiplier with a_in and b_in, and with output r_out. The repetitive-addition algorithm can be formalized in the following pseudo-code:

```
        if (a_in =0 or b_in =0) then{
            r = 0;}
        else{
            a = a_in;  n = b_in;  r = 0;
op:         r = r + a;
            n = n - 1;
            if (n = 0) then {goto stop;}
            else {goto op;}
        }
stop:   r_out = r;
```


Step 1: Defining the input and output signals

- Input signals:
  - a_in and b_in: input operands. 8-bit signals with std_logic_vector data type and interpreted as unsigned integers
  - start: command. The multiplier starts operation when the start signal is activated.
  - clk: system clock;
  - reset: asynchronous reset signal for system initialization.
- Output signals
  - r_out: the product. 16-bit signals.
  - ready: external status signal. It is asserted when the multiplication circuit is idle and ready to accept new inputs.

Step 2: Converting the algorithm to an ASM chart



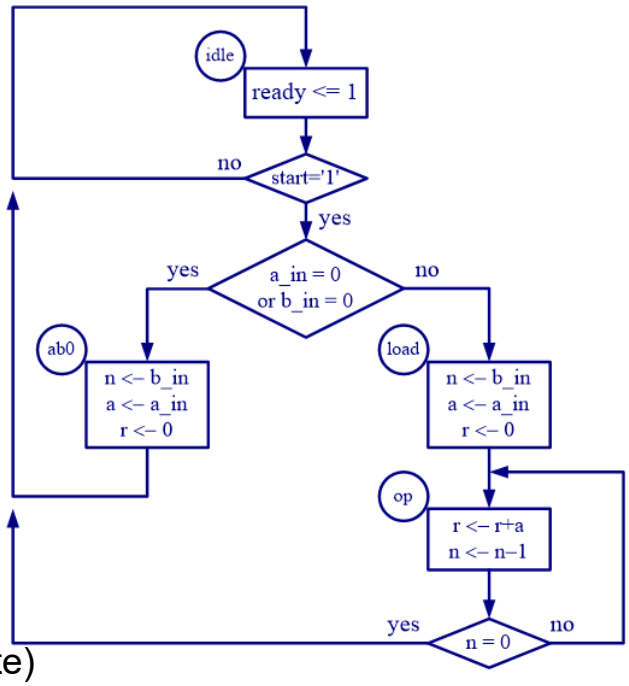

```
        if (a_in =0 or b_in =0) then{
           r = 0;}
        else{
           a = a_in;  n = b_in;  r = 0;
op:        r = r + a;
           n = n - 1;
           if (n = 0) then {goto stop;}
           else {goto op;}
        }
stop:   r_out = r;
```


Step 3: Constructing the FSMD

- Basic data path can be constructed as follows:
  1. List all possible RT operations in the ASM chart.
  2. Group RT operations according to their destination registers.
  3. Derive the circuit for each group RT operation.
  4. Add the necessary circuits to generate the status signals.

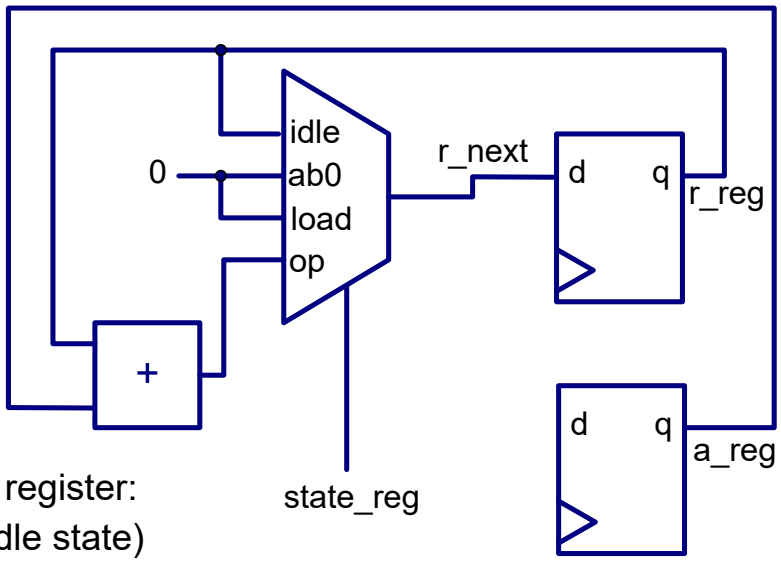
3.1 The circuit require 3 registers, to store signals r, n, and a respectively.

3.2. The RT operations:



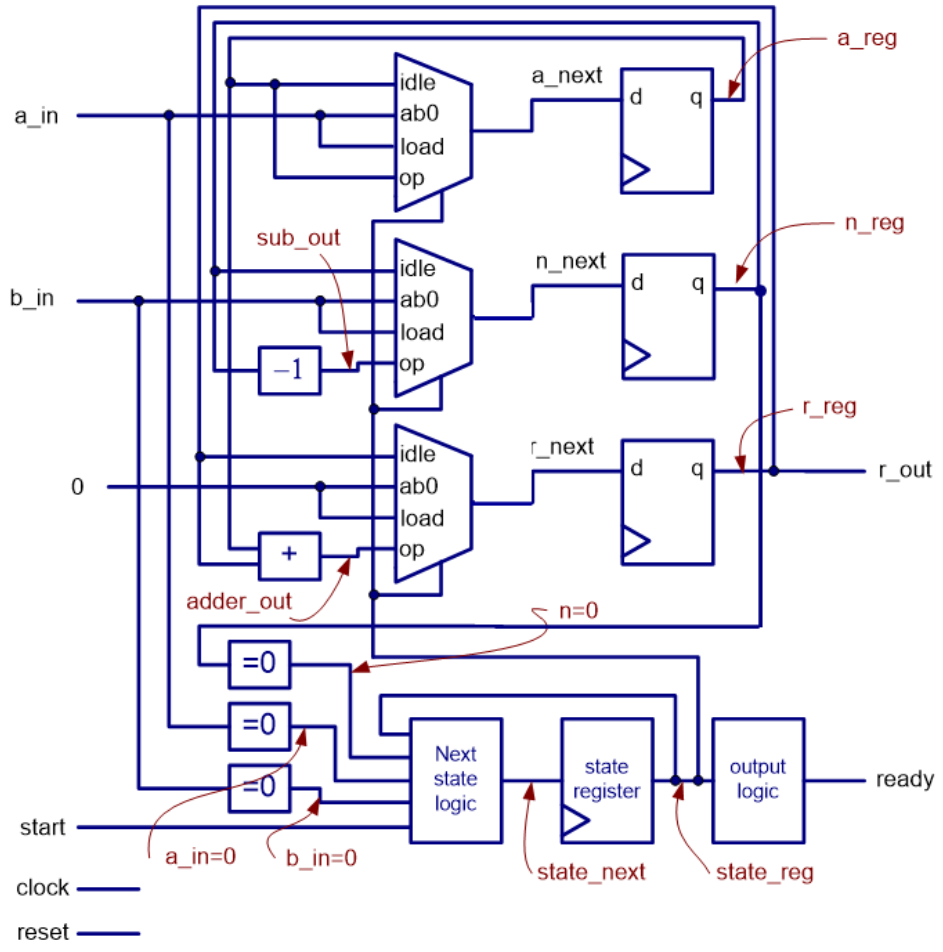
- RT operation with the r register:
  - r <– r (in the idle state)
  - r <– 0 (in the load and ab0 state)
  - r <– r + a (in the op state)
- RT operation with the n register:
  - n <– n (in the idle state)
  - n <– b_in (in the load and ab0 state)
  - n <– n – 1 (in the op state)
- RT operation with the a register:
  - a <– a (in the idle and op state)
  - a <– a_in (in the load and ab0 state)

3.3 the conceptual diagram of the circuit associated with the r register



- RT operation with the r register:
  - r <– r (in the idle state)
  - r <– 0 (in the load and ab0 state)
  - r <– r + a (in the op state)

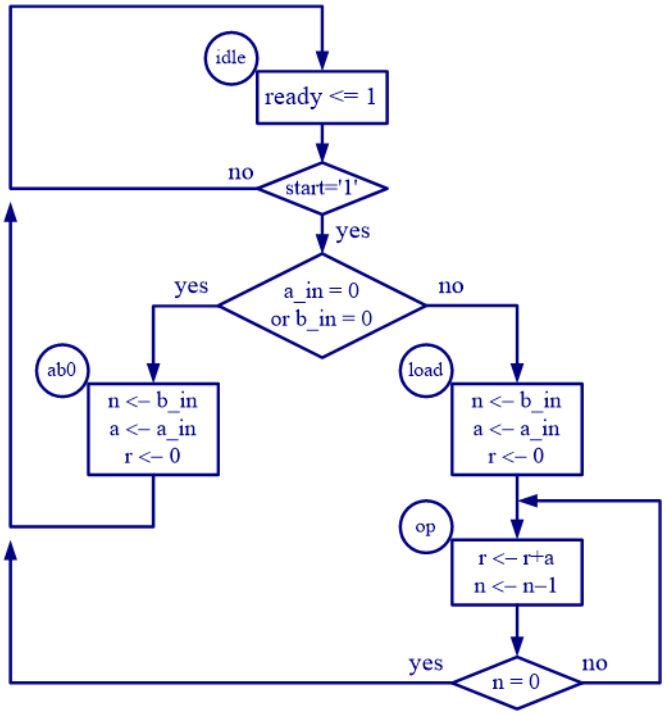
3.4 Complete block diagram of a repetitive-addition multiplier.



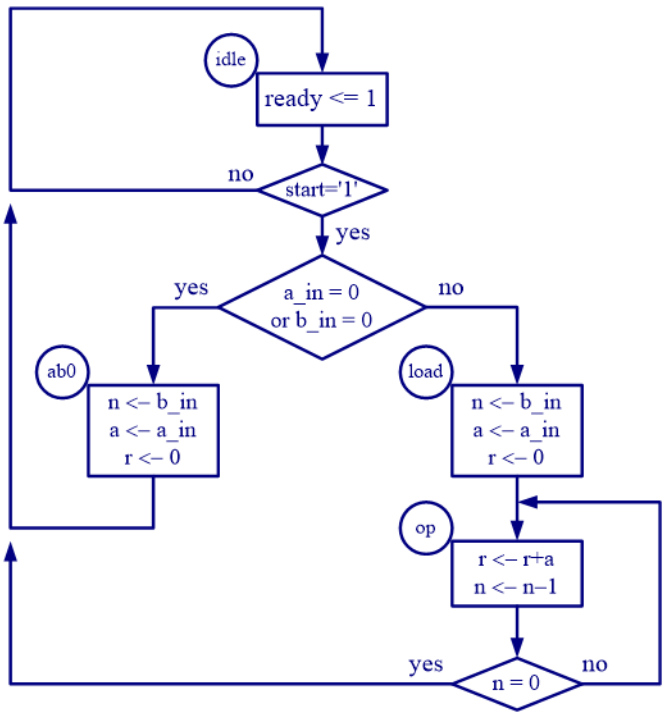
Step 4: VHDL descriptions of FSMD

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

entity seq_mult is
port (CLK, RESET, start: in std_logic;
      a_in, b_in: in std_logic_vector(7 downto 0);
      ready: out std_logic;
      r: out std_logic_vector(15 downto 0));
end entity seq_mult;

architecture seg_arch of seq_mult is
constant WIDTH : integer :=8;
type state_type is (idle, ab0, load, op);
signal state_reg, state_next : state_type;
signal a_reg, a_next, n_reg, n_next : std_logic_vector (WIDTH–1 downto 0);
signal r_reg, r_next : std_logic_vector (2*WIDTH–1 downto 0);
```

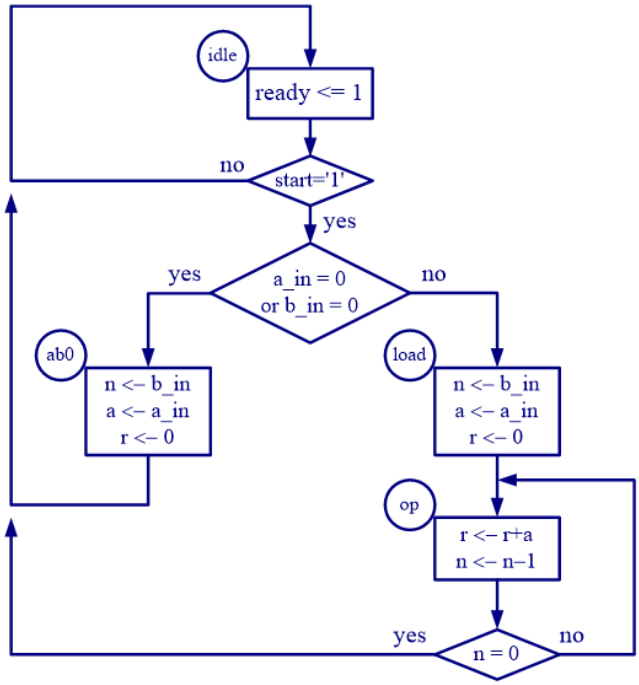
```
begin
-- state and data registers
process (CLK, RESET) is
begin
if RESET = '1' then
    state_reg <= idle;
    a_reg <= "00000000";
    n_reg <="00000000";
    r_reg <= x" 0000";
elsif CLK'event and CLK='1' then
    state_reg <= state_next;
    a_reg <= a_next;
    n_reg <= n_next;
    r_reg <= r_next;
end if;
end process;
```




```
-- combinational circuit
process (start, state_reg, a_reg, n_reg,
   r_reg, a_in, b_in) is
begin
    -- default value
    a_next <= a_reg;
    n_next <= n_reg;
    r_next <= r_reg;
    ready <= '0';
    case state_reg is
        when idle =>
            if start = '1' then
                if (a_in = "00000000" or
                    b_in = "00000000") then
                    state_next <= ab0;
                else
                    state_next <= load;
                end if;
            else
                state_next <= idle;
            end if;
            ready <= '1';
```



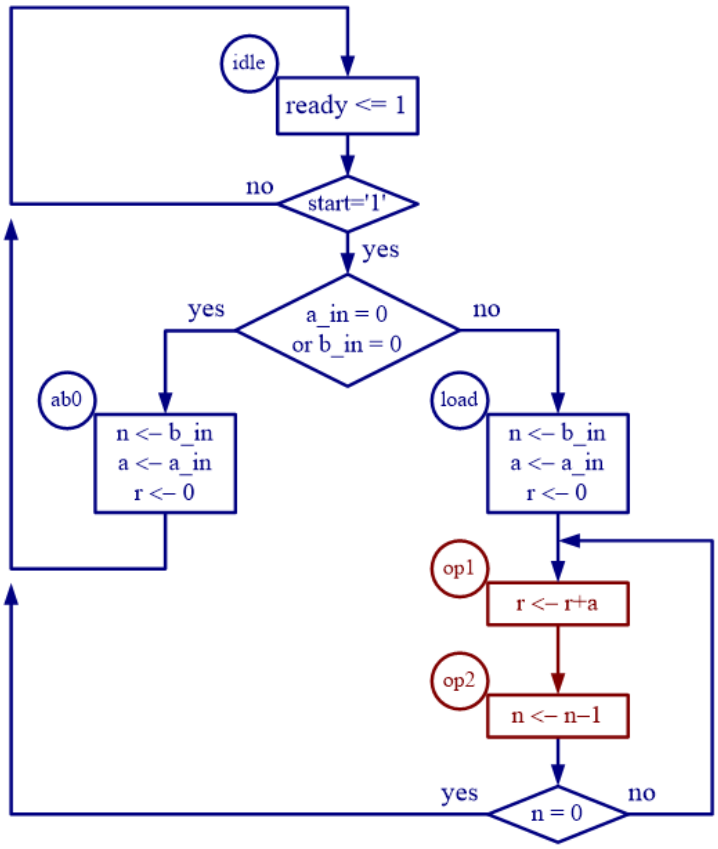
```
        when ab0 =>
            a_next <= a_in;
            n_next <= b_in;
            r_next <= x"0000";
            state_next <= idle;
        when load =>
            a_next <= a_in;
            n_next <= b_in;
            r_next <= x"0000";
            state_next <= op;
        when op =>
            n_next <= n_reg – 1;
            r_next <= ("00000000" & a_reg)
                        + r_reg;
            if (n_reg = "00000001" ) then
                state_next <= idle;
            else
                state_next <= op;
            end if ;
        end case;
    end process;
    r <= r_reg;
end architecture seg_arch;
```

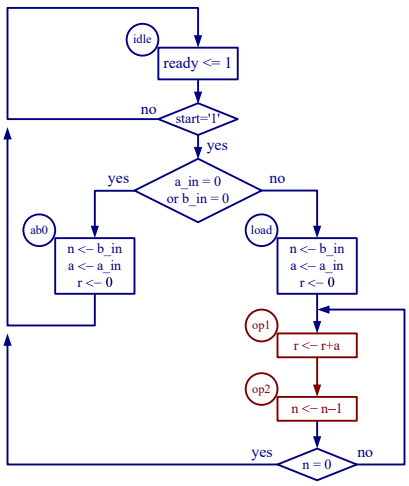


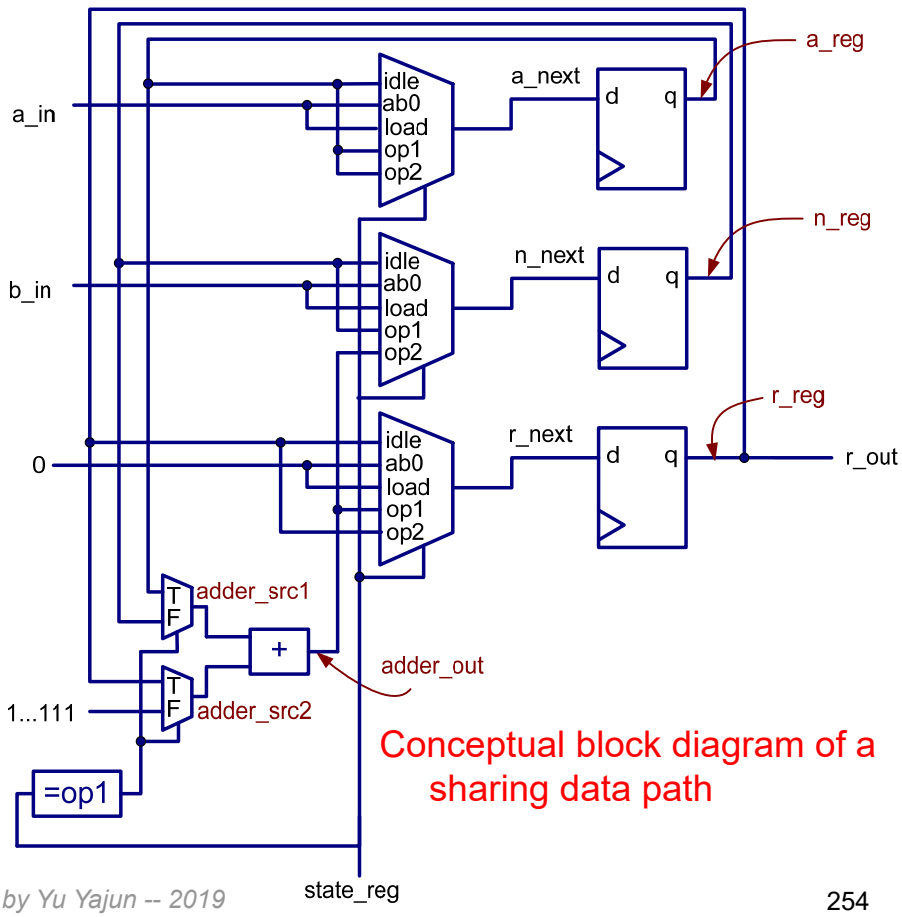
Resource sharing via FSMD example of repetitive-addition multiplier

- Many RT operations perform the same or similar function.
- Some function unit can be shared as long as these operations are scheduled in different states.
- the 16-bit adder and 8-bit decrementor are shared in the following example.

Modified ASM chart







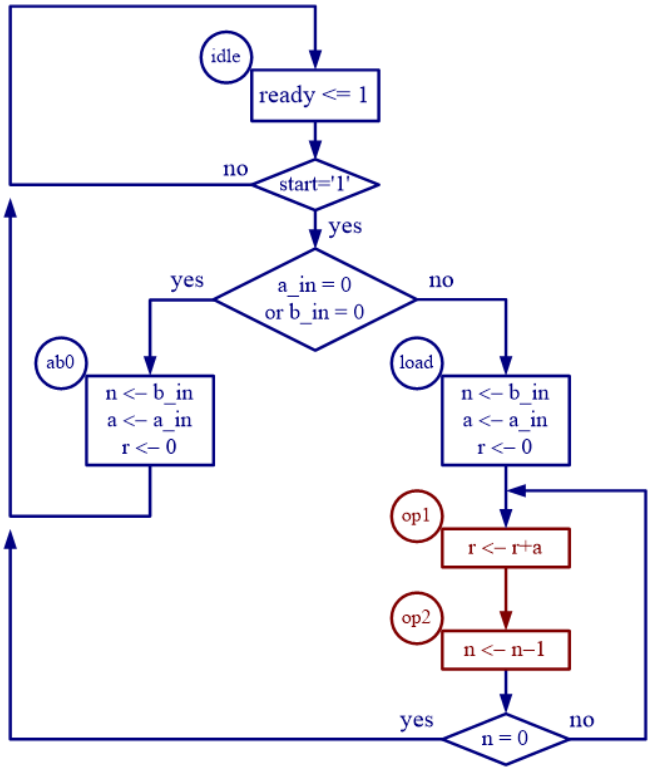
Conceptual block diagram of a sharing data path

sharing on a repetitive-addition multiplier

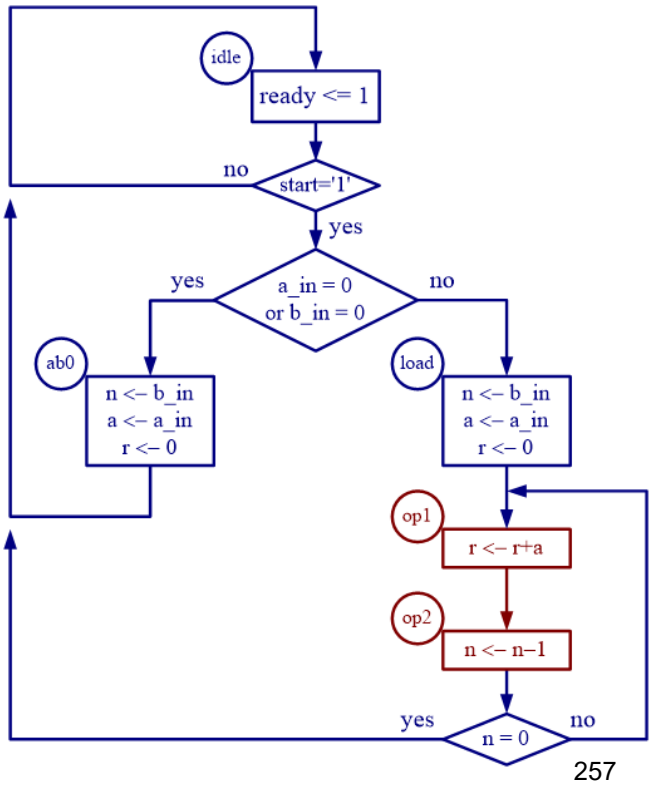
```
architecture sharing_arch of seq_mult is
constant WIDTH : integer :=8;
type state_type is (idle, ab0, load, op1, op2);
signal state_reg, state_next : state_type;
signal a_reg, a_next, n_reg, n_next : std_logic_vector (WIDTH–1 downto 0);
signal r_reg, r_next : std_logic_vector (2*WIDTH–1 downto 0);
signal adder_scr1, adder_scr2: std_logic_vector (2*WIDTH–1 downto 0);
signal adder_out: std_logic_vector (2*WIDTH–1 downto 0);
```



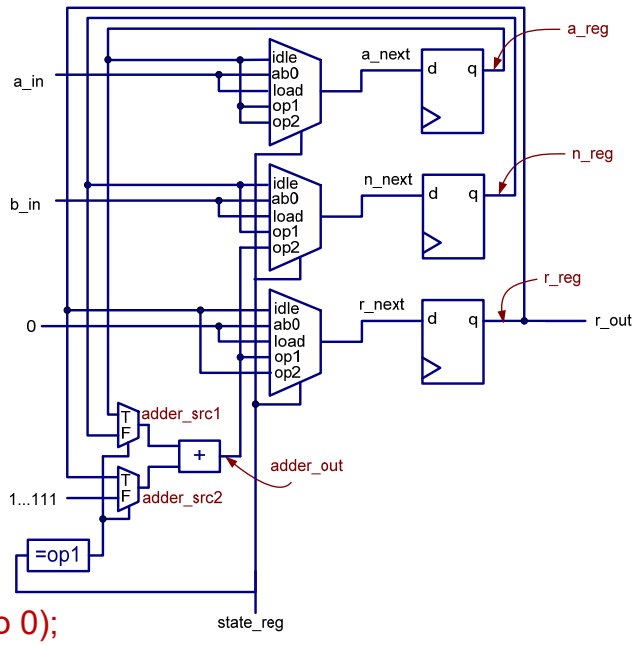
```
begin
-- state and data registers
process (CLK, RESET) is
begin
if RESET = '1' then
    state_reg <= idle;
    a_reg <= "00000000";
    n_reg <="00000000";
    r_reg <= x" 0000";
elsif CLK'event and CLK='1' then
    state_reg <= state_next;
    a_reg <= a_next;
    n_reg <= a_next;
    r_reg <= a_next;
end if;
end process;
```



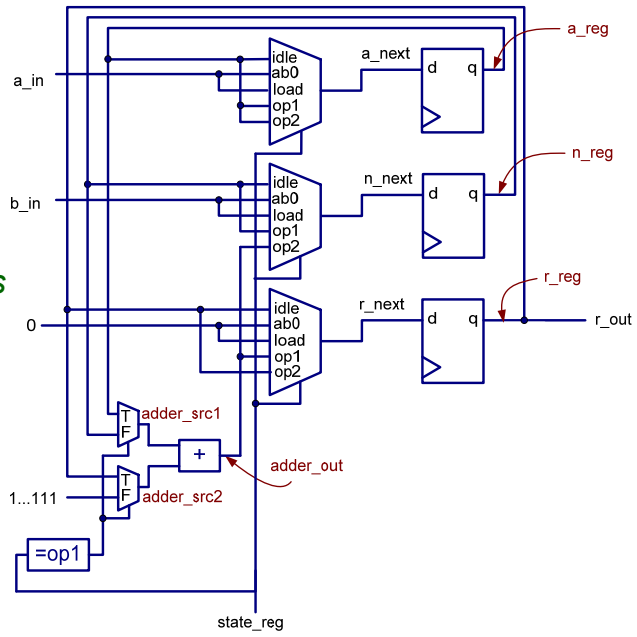
```
-- next-state, logic/output logic and data path routing
process (start, state_reg, a_reg, n_reg, r_reg, a_in, b_in, adder_out) is
begin
    -- default value
    a_next <= a_reg;
    n_next <= n_reg;
    r_next <= r_reg;
    ready <= '0';
    case state_reg is
        when idle =>
            if start = '1' then
                if (a_in = "00000000" or
                    b_in = "00000000") then
                    state_next <= ab0;
                else
                    state_next <= load;
                end if;
            else
                state_next <= idle;
            end if;
            ready <= '1';
```




```
    when ab0 =>
        a_next <= a_in;
        n_next <= b_in;
        r_next <= x"0000";
        state_next <= idle;
    when load =>
        a_next <= a_in;
        n_next <= b_in;
        r_next <= x"0000";
        state_next <= op1;
    when op1 =>
        r_next <= adder_out;
        state_next <= op2;
    when op2 =>
        n_next = adder_out (WIDTH -1 downto 0);
        if (n_reg = "00000001" ) then
            state_next <= idle;
        else
            state_next <= op1;
        end if ;
    end case;
end process;
```



```
-- datapath input routing and functional units
process (state_reg, r_reg, a_reg, n_reg) is
begin
    if (state_reg = op1) then
        adder_src1 <= r_reg;
        adder_src2 <= "00000000" & a_reg;
    else -- for op2 state
        adder_src1 <= "00000000" & n_reg;
        adder_src2 <= x"FFFF";
    end if;
end process;
adder_out <= adder_src1 + adder_src2;
-- output
r <= r_reg;
end architecture sharing_arch;
```



Chapter 8: Parameterized Design

- Goal: Design reuse
  - Ideally, we want to design some common modules that can be shared by many applications.
  - Since every application is different, it is desirable that a module can be customized to some degree to meet the specific need of an application.
  - Customization is normally specified by explicit or implicit parameters

Types of Parameters

- Width Parameters
  - The widths of data signals normally can be modified to meet different requirement.
  - The width parameters of a parameterized design specify the sizes (i.e., number of bits) of the relevant data signal.
- Feature Parameters
  - Specify the structure or organization of a design.
  - Defined on an ad hoc basis.
  - To include or exclude certain functionalities (i.e., features) from implementation or to select one particular version of the implementation

8.1 Generics

- The generic construct of VHDL is a mechanism to pass information into an entity and a component.
  - They are first declared in entity and component declaration and later assigned a value during component instantiation

```
entity para_binary_counter is
    generic (WIDTH: natural);
    port (
     clk, reset: in  std_logic;
      q: out  std_logic_vector(WIDTH–1 downto 0)
    );
end entity para_binary_counter;
```

- After the declaration, the generic can be used in the associated architecture bodies.
- A generic cannot be modified inside the architecture body and thus functions like a constant
  - It is sometimes referred to as a generic constant.


```
architecture arch of para_binary_counter is
    signal reg, reg_next : std_logic_vector (WIDTH–1
        downto 0);
begin
    process (clk, reset) is
    begin
        if reset = '1' then
            reg <= (others =>'0');
        elsif clk'event and clk='1' then
            reg <= reg_next;
        end if;
    end process;
-- next-state logic
    reg_next <= reg + 1;
    q <= std_logic_vector(reg);
end architecture arch;
```

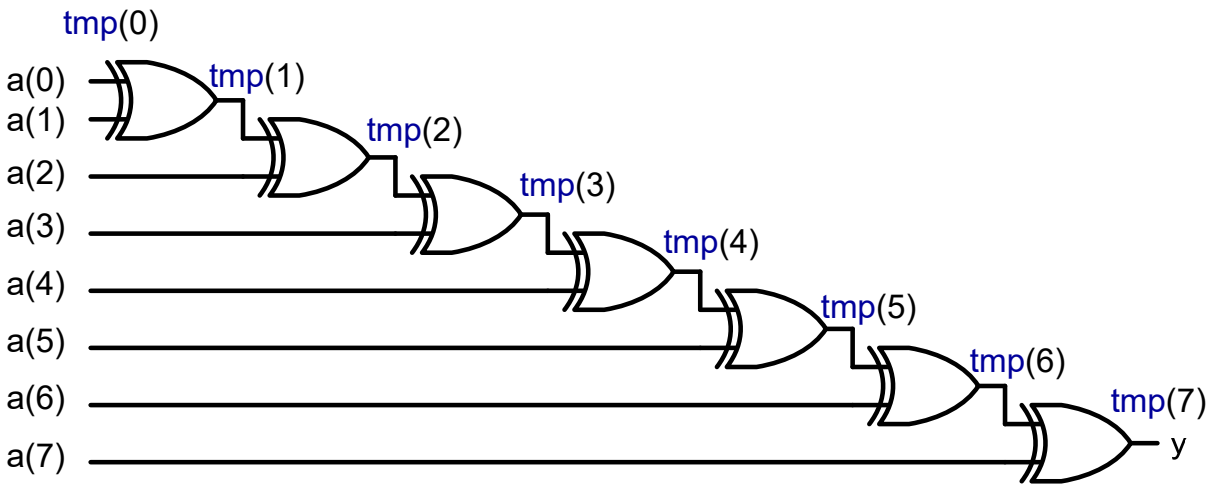
- To use the parameterized free-running binary counter in a hierarchical design, a similar component declaration should be included in the architecture declaration.
- The generic can then be assigned a value in the generic mapping section when a component instance is instantiated.
- Example of the use of generics

```
library ieee;
use ieee.std_logic_1164.all;
entity generic_demo is
    port(
        clk, reset: in std_logic;
        q_4: out std_logic_vector(3 downto 0);
        q_12: out std_logic_vector(11 downto 0)
    );
end entity generic_demo;
```



```
architecture arch of generic_demo is
    component para_binary_counter is
        generic (WIDTH: natural);
        port (
            clk, reset: in  std_logic;
            q: out  std_logic_vector(WIDTH–1 downto 0)
        );
    end component para_binary_counter;

begin
    four_bit: para_binary_counter
        generic map (WIDTH => 4)
        port map (clk => clk, reset => reset, q => q_4);
    twelve_bit: para_binary_count
        generic map (WIDTH => 12)
        port map (clk => clk, reset => reset, q => q_12);
end architecture arch;
```

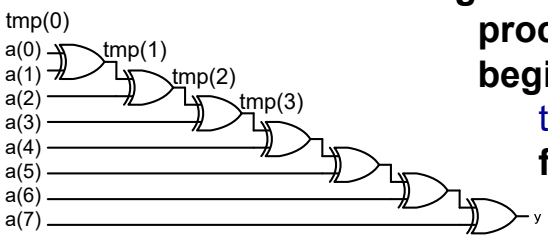
Example: Reduced-xor circuit



tmp(i) <= tmp(i-1) **xor** a(i)

Parameterized reduced-xor circuit using a generic

```
library ieee;  use ieee.std_logic_1164.all;
entity reduced_xor is
    generic (WIDTH: natural); -- generic declaration
    port(
        a: in std_logic_vector(WIDTH–1 downto 0);
        y: out std_logic
    );
end entity reduced_xor;

architecture loop_linear_arch of reduced_xor is
    signal tmp: std_logic_vector(WIDTH–1 downto 0);
begin
    process (a, tmp) is
    begin
        tmp(0) <= a(0); -- boundary bit
        for i in 1 to (WIDTH–1) loop
            tmp(i) <= a(i) xor tmp(i–1);
        end loop;
    end process;
    y <= tmp(WIDTH-1);
end architecture loop_linear_arch;
```



Slide 280

Slide 293

8.2 Array attribute

- A VHDL attribute provides information about a named item, such as a data type or a signal.
- We have used the 'event attribute, as in clk'event, express the changing edge of the clk signal.
- A set of attributes is associated with an object of an array data type. Let s be a signal with an array data type.
  - s'left, s'right: the left and right bounds of the index range of s.
  - s'low, s'high: the lower and upper bounds of the index range of s.
  - s'length: the length of the index range of s.
  - s'range: the index range of s.
  - s'reverse_range: the reversed index range of s.

- The attributes can be applied to the signal defined with std_logic_vector, unsigned and signed:

```
signal s1: std_logic_vector (31 downto 0);
signal s2: std_logic_vector (8 to 15);
```

The attributes of s1 are

s1'left = 31; s1'right = 0;
s1'low = 0; s1'high =31;
s1'length = 32;
s1'range = 31 **downto** 0
s1'reverse_range = 0 **to** 31

The attributes of s2 are

s2'left = 8; s2'right = 15;
s2'low = 8; s2'high =15;
s2'length = 8;
s2'range = 8 **to** 15
s2'reverse_range = 15 **downto** 8

Parameterized reduced-xor circuit using an attribute

```
library ieee;
use ieee.std_logic_1164.all;
entity reduced_xor is
   generic(W: natual);
   port(a: in std_logic_vector(W-1 downto 0);
        y: out std_logic
   );
end entity reduced_xor;
```

```
architecture loop_linear_arch of reduced_xor is
    signal tmp: std_logic_vector(a'length–1 downto 0);
begin
    process (a, tmp) is
    begin
        tmp(0) <= a(0);
        for i in 1 to (a'length–1) loop
            tmp(i) <= a(i) xor tmp(i–1);
        end loop;
    end process;
    y <= tmp(a'length–1);
end architecture loop_linear_arch;
```

- The range of the for loop can also be expressed as:
  - **for** i **in** a'low+1 **to** a'high **loop**
  - **for** i **in** a'right+1 **to** a'left **loop**
- The last signal assignment
  - y <= tmp (tmp'left);

8.3 Unconstrained Array

- Unconstrained array is defined as an array type with specified data type of the index value, but without specified exact bounds of the index value.
- Example:

  type std_logic_vector **is array** (natural **range** <>) **of** std_logic
- Similarly, we have unsigned and signed data types.
- If an object is declared with an unconstrained array data type, we must specify its index range when the data type is used, as 15 downto 0 in

  signal x: std_logic_vector(15 **downto** 0);

- A special case: the unconstrained array can be declared without specifying the range in port declaration.
- Example:

```
library ieee;  use ieee.std_logic_1164.all;
entity unconstrain_dff is
    port( clk: std_logic;
          d: in std_logic_vector;      -- the actual range is inferred
          q: out std_logic_vector      -- when an instance of
    );                                 -- unconstrain_dff is instantiated.
end entity unconstrain_dff;

architecture arch of unconstrain_dff is
begin
    process (clk) is
    begin
        if (clk'event and clk='1') then q <= d; end if;
    end process;
end architecture arch;
```


- The ranges of the actual signals become the ranges of d and q signals. Example: the dff16 instance is instantiated as a 16-bit register

```
...
signal din, qout: std_logic_vector(15 downto 0);
signal clk: std_logic;
...
dff16: unconstrain_dff
    port map( clk => clk, d => din, q => qout);
...
```

- Since no range is specified for d and q, the boundaries of the two signal will not be check in the analysis stage.

```
...
signal din: std_logic_vector(15 downto 0);
signal qout: std_logic(7 downto 0); -- syntactically correct.
...                  -- error may be detected during the synthesis
dff16: unconstrain_dff
    port map( clk => clk, d => din, q => qout);
...
```

Parameterized reduced-xor circuit using an unconstrained array

The code appears to be correct at first glance

```
library ieee;  use ieee.std_logic_1164.all;
entity unconstrain_reduced_xor is
    port(
        a: in std_logic_vector;
        y: out std_logic
    );
end entity unconstrain_reduced_xor;

architecture arch of unconstrain_reduced_xor is
    constant WIDTH: natural := a'length;
    signal tmp: std_logic_vector(WIDTH–1 downto 0);
begin
    process (a, tmp) is
    begin
        tmp(0) <= a(0);
        for i in 1 to (WIDTH–1) loop
            tmp(i) <= a(i) xor tmp(i–1);
        end loop;
    end process;
    y <= tmp(WIDTH-1);
end architecture arch;
```


- If we map the a signal to an actual signal with the type of std_logic_vector of 8 bits during component instantiation, we may have a to be:
  - std_logic_vector(7 **downto** 0);
  - std_logic_vector(0 **to** 7);
  - std_logic_vector(15 **downto** 8);
  - std_logic_vector(8 **to** 15);
- The code does not work properly for the last two formats.

Improved parameterized reduced-xor circuit using an unconstrained array

```
architecture better_arch of unconstrain_reduced_xor is
    constant WIDTH: natural := a'length;
    signal tmp: std_logic_vector(WIDTH–1 downto 0);
    signal aa: std_logic_vector(WIDTH-1 downto 0);
begin
    aa <= a;
    process (aa, tmp) is
    begin
        tmp(0) <= aa(0);
        for i in 1 to (WIDTH–1) loop
            tmp(i) <= aa(i) xor tmp(i–1);
        end loop;
    end process;
    y <= tmp(WIDTH-1);
end architecture better_arch;
```


8.3 Comparison

- The unconstrained array mechanism uses attributes to infer the relevant information from the actual signal.
  - More general and flexible than the generic mechanism, but also
  - More opportunities for errors.
  - Requires comprehensive error-checking code
- Generic mechanism is preferred, unless a module is extremely general and widely used.
  - More rigid
  - It clearly specifies the range, direction and width of each signal and avoids many subtle erroneous conditions.

8.4 Generate Statement

- The generate statements are concurrent statements with embedded internal concurrent statement, which can be interpreted as a circuit part.
- Two types of generated statements:
  - for generate statement: used to create a circuit by replicating the hardware part
  - conditional or if generate statement: used to specify whether or not to create an optional hardware part.

8.4.1 For Generate Statement

- Many digital circuits can be implemented as a repetitive composition of basic building blocks, exhibiting a regular structure, such as a one-dimensional cascading chain, a tree-shaped connection or a two-dimensional mesh.
- For generate statement syntax

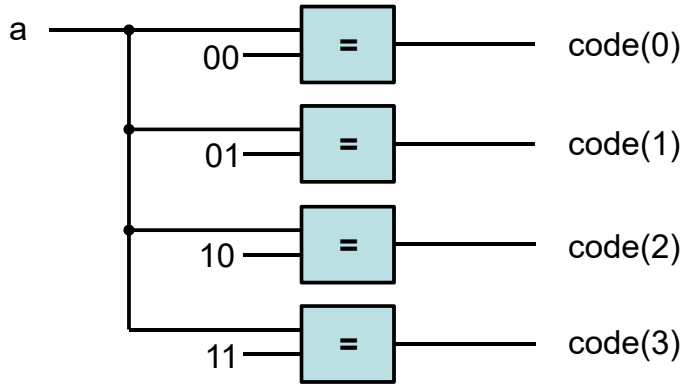
```
gen_label:      -- mandatory to identify to this
                -- particular generate statement
for loop_index in loop_range generate
    concurrent statements;
    -- describe a stage of the  iterative circuit
end generate;
```

Slide 268

- The loop_range has to be static. It is normally specified by the width parameters.

Example: Binary decoder

- A binary n-to-2^n decoder is circuit that asserts one of the 2^n possible output signal according to an n bit input signal.
- One way to view the binary decoder is to treat each bit of the decoded output as the result of a constant comparator.



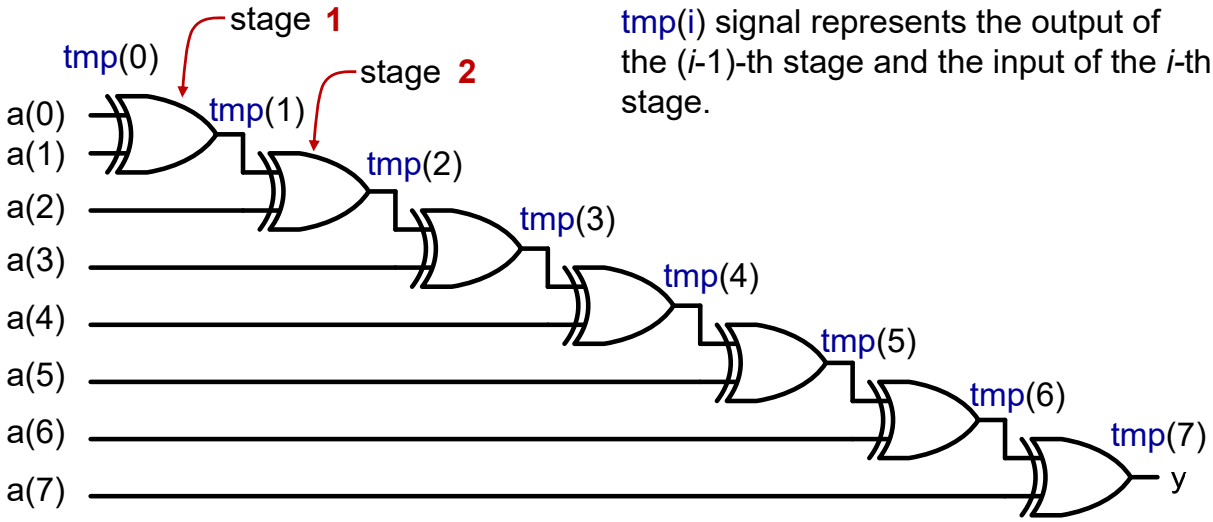

```
code(i) <= '1' when a = std_logic_vector(unsigned(i)) else
            '0';
```


Parameterized binary decoder using a for generate statement

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

entity bin_decoder is
    generic(WIDTH: natural);
    port(
        a: in std_logic_vector(WIDTH–1 downto 0);
        code: out std_logic_vector(2**WIDTH–1 downto 0)
    );
end bin_decoder;

architecture gen_arch of bin_decoder is
begin
    comp_gen:
    for i in 0 to (2**WIDTH–1) generate
        code(i) <= '1' when i = to_integer(unsigned(a)) else
                   '0';
    end generate;
end architecture gen_arch;
```

Slide 293

Example: Reduced-xor circuit



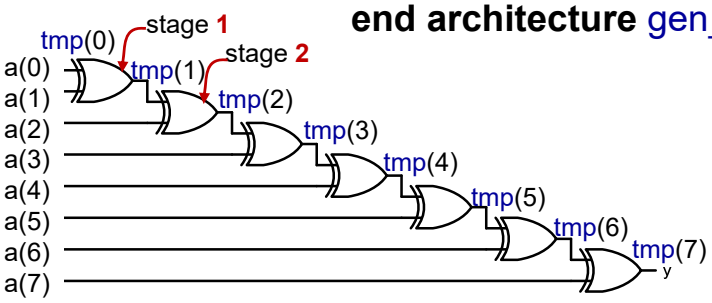
tmp(i) signal represents the output of the (*i*-1)-th stage and the input of the *i*-th stage.

tmp(i+1) <= tmp(i) **xor** a(i+1)

tmp(i) <= tmp(i-1) **xor** a(i)

Parameterized reduced-xor circuit using a for generate statement

```
architecture gen_linear_arch of reduced_xor is
    signal tmp: std_logic_vector(WIDTH–1 downto 0);
begin
    tmp(0) <= a(0);
    xor_gen:
        for i in 1 to (WIDTH–1) generate
            tmp(i) <= a(i) xor tmp(i–1);
        end generate;
     y <= tmp(WIDTH-1);
end architecture gen_linear_arch;
```



Slide 293

- In an iterative structure, the boundary stages interface to the external input and output signals, and sometimes their connections are different from the regular blocks.

8.4.2 Conditional Generate Statement

- The conditional generate statement is used to specify an optional circuit that can be included or excluded in the final implementation.
- Conditional generate statement syntax

```
gen_label:     -- mandatory
if boolean_exp generate -- boolean_exp must be static
    concurrent statements;
end generate;
```

- There is no else branch in conditional generate statement.
- If we want to include one of the two possible circuits in an implementation, we must use two separate if generate statements.

Reduced-xor circuit revisited

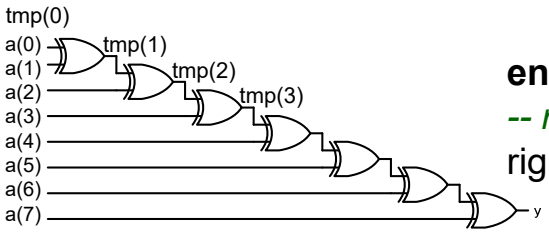
- One common use of the conditional generate statement is to describe the "irregular" stages in a for generate statement.
- For example, two statements

```
tmp(0) <= a(0);
y <= tmp(WIDTH-1);
```

  are used to rename the input and output signals in the for generate statement examples.
- To eliminate these statements, we can use conditional generate statements inside the for generate statement.

Parameterized reduced-xor circuit with a conditional generate statement

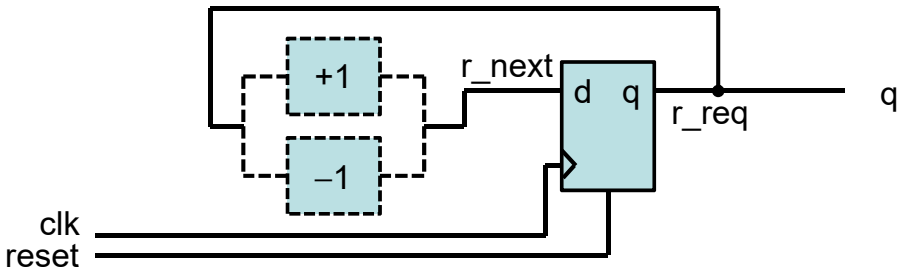
```
tmp(0) <= a(0);
xor_gen:
    for i in 1 to (WIDTH–1) generate
        tmp(i) <= a(i) xor tmp(i–1);
    end generate;
 y <= tmp(WIDTH-1);
```




```
architecture gen_if_arch of reduced_xor is
    signal tmp: std_logic_vector(WIDTH–2 downto 1);
begin
    xor_gen:
    for i in 1 to (WIDTH–1) generate
        -- leftmost stage
        left_gen: if i = 1 generate
            tmp(i) <= a(i) xor a(0);
        end generate;
        -- middle stage
        middle_gen: if (i>1) and (i<(WIDTH –1)) generate
            tmp(i) <= a(i) xor tmp(i–1);
        end generate;
        -- rightmost stage
        right_gen: if i = (WIDTH –1) generate
            y <= a(i) xor tmp(i–1);
        end generate;
    end generate;
end architecture gen_if_arch;
```


Example: Up-or-down free-running binary counter

- An up-or-down binary counter is a counter that can be instantiated in a specific mode.
- Note that the "**or**" here means that only one mode of operation, either counting up or counting down but not both, can be implemented in the final circuit.



- We use the UP generic as the feature parameter to specify the desired mode.

Up-or-down free-running binary counter

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

entity up_or_down_counter is
    generic(WIDTH: natural;  UP: natural);
    port(clk, reset: in std_logic;
        q: out std_logic_vector(WIDTH–1 downto 0)
    );
end up_or_down_counter;

architecture arch of up_or_down_counter is
    signal r_reg, r_next: unsigned(WIDTH–1 downto 0);
begin
    -- register
    process (clk, reset)
    begin
        if (reset = '1') then
            r_reg <= (others => '0')
        elsif (clk'event and clk='1') then
            r_reg <= r_next;
        end if;
    end process;
```



```
    -- next-state logic
    inc_gen: -- incrementor
    if UP = 1 generate
        r_next <= r_reg + 1;
    end generate;
    dec_gen: -- decrementor
    if UP /= 1 generate
        r_next <= r_reg – 1;
    end generate;
    q <= std_logic_vector(r_reg);  -- output logic
end architecture arch;
```

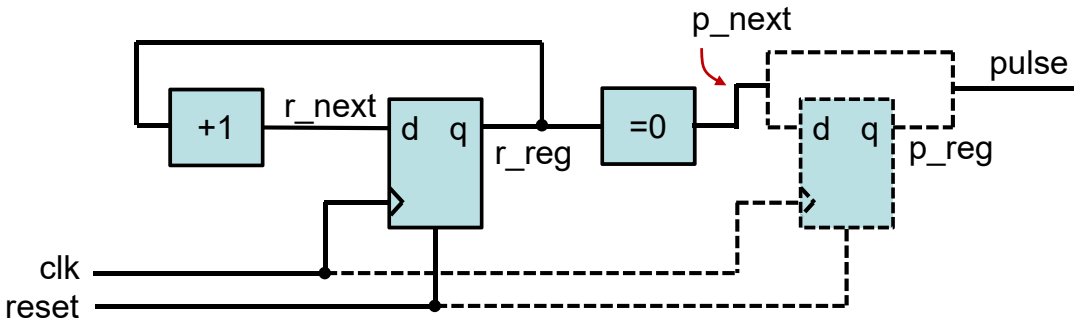
Up-and-down free-running binary counter

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

entity up_and_down_counter is
    generic map (WIDTH: natural)
    port map(clk, reset: in std_logic; mode: in std_logic;
        code: out std_logic_vector(2**WIDTH–1 downto 0)
    );
end up_and_down_counterr;
```

```
architecture arch of up_and_down_counter is
    signal r_reg, r_next: unsigned(WIDTH–1 downto 0);
begin
    -- register
    process (clk, reset)
    begin
        if (reset = '1') then
            r_reg <= (others => '0')
        elsif (clk'event and clk='1') then
            r_reg <= r_next;
        end if;
    end process;
    -- next-state logic
    r_next <= r_reg + 1 when mode ='0' else
                r_reg -1;
    -- output logic
    q <= std_logic_vector(r_reg);
end architecture arch;
```


Counter with an optional output buffer

- An output buffer can remove glitches from the signal.
- Since the buffer is only needed for certain application, it will be convenient to include the buffer as an optional part of the circuit.

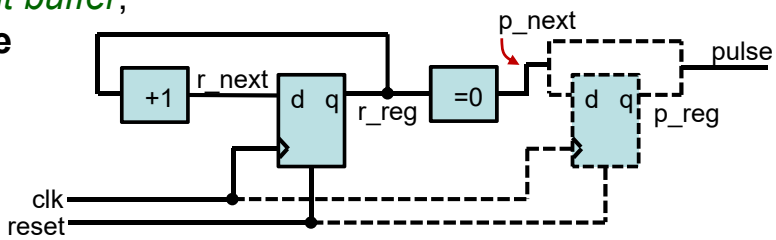


Counter with an optional output buffer

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

entity op_buf_counter is
    generic(WIDTH: natural; BUFF: natural);
    port(clk, reset: in std_logic;
        pulse: out std_logic);
end op_buf_counterr;

architecture arch of op_buf_counter is
    signal r_reg, r_next: unsigned(WIDTH–1 downto 0);
    signal p_reg, p_next: std_logic;
begin
    -- register
    process (clk, reset)
    begin
        if (reset = '1') then r_reg <= (others => '0')
        elsif (clk'event and clk='1') then r_reg <= r_next;
        end if;
    end process;
```



```
    -- next-state logic
    r_next <= r_reg + 1;
    -- output logic
    p_next <= '1' when r_reg = 0 else '0';
    buf_gen:
    if BUFF = 1 generate
        process (clk, reset)
        begin
            if (reset = '1') then p_reg <= '0'
            elsif (clk'event and clk = '1') then p_reg <= p_next;
            end if;
        end process;
        pulse <= p_reg;
    end generate;
    no_buf_gen: -- without buffer;
    if BUFF /= 1 generate
        pulse <= p_next;
    end generate;
end architecture arch;
```



8.5 Comparison

- To create a circuit with a selectable feature:
  - use conditional generate statement
  - a full-featured circuit with some input control signal connected to constant values to permanently enable the desired feature
  - use the configuration construct
- Assume we need a 16-bit up counter in a design.

```
count16up: up_or_down_counter
    generic map (WIDTH => 16, UP =>1)
    port map (clk => clk, reset => reset, q=>q);
```

or

```
count16up: up_and_down_counter
    generic map (WIDTH => 16)
    port map (clk => clk, reset => reset, mode => '1', q=>q);
```


Difference

- The up-or-down counter instance
  - creates a circuit with only the needed features.
  - The selected portion of code is passed to the synthesis stage, i.e., the synthesis software only needs to synthesize the selected portion.
- The up-and-down counter instance
  - creates a circuit that consists of all features and uses an external control signal to selectively enable a portion of the circuit.
  - The entire VHDL code is passed to synthesis stage. The synthesis software eliminates the unused portion through logic optimization.
- In general, use of the feature parameters and conditional generate statement is better than the full-featured approach.

- The selected hardware creation can also be achieved by configuration where multiple architecture bodies are constructed, each containing a specific feature, e.g., architectures up_arch and down_arch of the same entity updown_counter, for counting up and counting down, respectively.
- And the following instantiation can be used to select the counting up circuit.

```
count16up: work.updown_counter(up_arch)
    generic map(WIDTH =>16)
    port map (clk => clk, reset => reset, q => q);
```


Up-or-down counter with two architecture bodies

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

entity updown_counter is
    generic(WIDTH: natural);
    port(clk, reset: in std_logic;
        q: out std_logic_vector(WIDTH–1 downto 0)
    );
end updown_counter;

architecture up_arch of updown_counter is
    signal r_reg, r_next: unsigned(WIDTH–1 downto 0);
begin
    -- register
    process (clk, reset)
    begin
        if (reset = '1') then r_reg <= (others => '0')
        elsif (clk'event and clk='1') then r_reg <= r_next;
        end if;
    end process;
```

```
        -- next-state logic
        r_next <= r_reg + 1;
        -- output logic
        q <= std_logic_vector(r_reg);
    end architecture up_arch;

    architecture down_arch of updown_counter is
        signal r_reg, r_next: unsigned(WIDTH–1 downto 0);
    begin
        -- register
        process (clk, reset)
        begin
            if (reset = '1') then r_reg <= (others => '0')
            elsif (clk'event and clk='1') then r_reg <= r_next;
            end if;
        end process;
        -- next-state logic
        r_next <= r_reg – 1;
        -- output logic
        q <= std_logic_vector(r_reg);
    end architecture down_arch;
```


- Conversely, we can merge the logic from several architecture bodies into a single body and use a feature generic and conditional generate conditions to select the desired portion.
- There is no rule about when to use a feature parameter and when to use a configuration construct. In general,
  - code with a feature parameter is more difficult to develop and comprehend, but on the other hand, if we use a separate architecture body for each distinctive feature, the number of architecture bodies will grow exponentially and becomes difficult to manage.
  - when a feature parameter leads to significant modification or addition of the no-feature codes and starts to make the code incomprehensible, it is probably a good idea to use separate architecture bodies and the configuration construct.

8.6 For Loop Statement

- The for loop statement is a sequential statement and is the only sequential loop construct that can be synthesized.

must be static

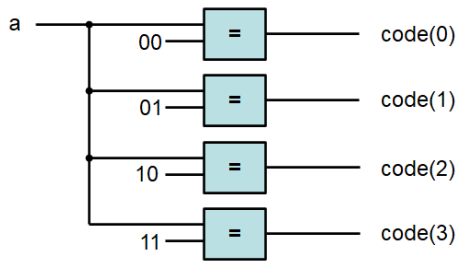
```
for index in loop_range loop
    sequential statements;
end loop;
```

- The basic way to synthesize a for loop statement is to unroll or flatten the loop. Unrolling a loop means to replace the loop structure by explicitly listing all iterations.

Example: Binary decoder

- The code is very similar to the for generate version Slide 273

```
architecture loop_arch of bin_decoder is
begin
    process (a)
    begin
        for i in 0 to (2**WIDTH–1) loop
            if i = to_integer(unsigned(a)) then code(i) <= '1';
            else code(i) <= '0';
            end if;
        end loop;
    end process;
end architecture gen_arch;
```



Example: Reduced-xor circuit

- For loop version: Slide 259
- For generate version: Slide 275

Example: Priority Encoder

- Recall that a signal can be assigned with multiple times inside process and only the last assignment takes effect.
- A priority encoder is a circuit that returns the binary code of the highest-priority request.
- Assume that the input is an array of r(WIDTH-1 **downto** 0), and r(WIDTH-1) has the highest priority.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numberic_std.all;
use work.util_pkg.all;

entity prio_encoder is
    generic(WIDTH: natural);
    port(r: in std_logic_vector(WIDTH-1 downto 0);
        bcode: out std_logic_vector(log2c(WIDTH)-1 downto 0);
        valid: out std_logic);
end prio_encoder;
```



```
architecture loop_linear_arch of prio_encoder is
    constant B: natural := log2c(WIDTH);
    signal tmp: std_logic_vector(WIDTH-1 downto 0);
begin
    process (r) --binary code
    begin
        bcode <= (others => '0');
        for i in 0 to (WIDTH-1) loop
            if r(i) = '1' then
                bcode <= std_logic_vector(to_unsigned(i, B));
            end if;
        end loop;
    end process;

    process(r, tmp) -- reduced – or circuit
    begin
        tmp(0) <= r(0);
        for i in 1 to (WIDTH – 1) loop
            tmp(i) <= r(i) or tmp(i-1);
        end loop;
    end process;
    valid <= tmp (WIDTH-1);
end architecture loop_linear_arch;
```

8.7 Comparison

- Both the for generate and for loop statements are used to describe replicated structures.
- For generate statement:
  - can only use concurrent statements.
  - start a design with a conceptual diagram of a few stages; the diagram is used to identify the basic building block and connection pattern, and then the code of the loop body is derived.
- For loop statement:
  - can only use sequential statements.
  - the body of the loop statement can be more general and versatile.
  - may lead to unnecessarily complex implementation or even an unsynthesizable description.

Chapter 9: Pipelined Design

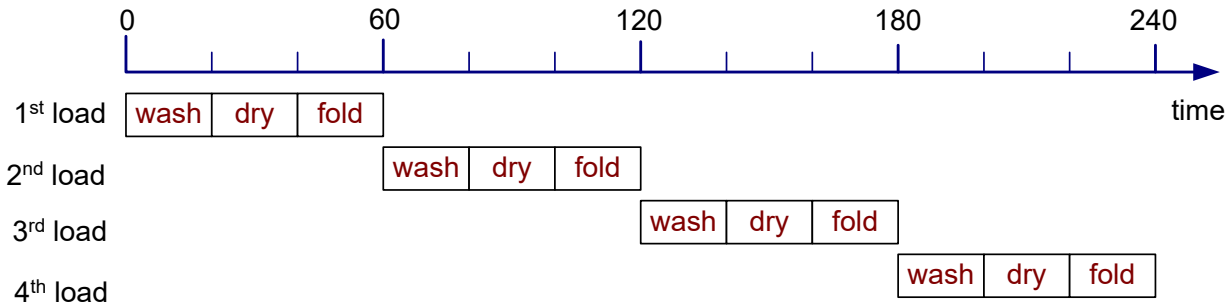
- Pipeline is an important technique to increase the performance of a system.
- The basic idea is to overlap the processing of several tasks so that more tasks can be completed in the same amount of time.

9.1 Delay and throughput

- Delay and throughput are the two criteria used to examine the performance of a system
  - Delay: the time required to complete one task.
  - Throughput: the number of tasks that can be completed per unit time.
- Adding pipeline to a combinational circuit can increase the throughput but not reduce the delay.

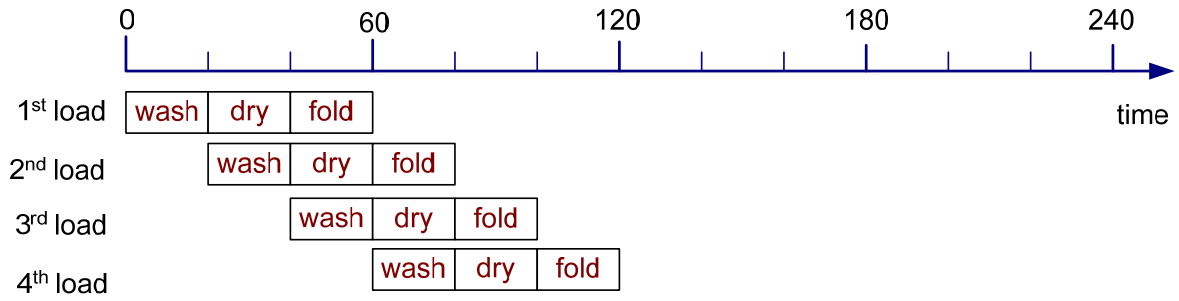
9.2 Overview on pipelined design

- The pipelining technique can be applied to a task that is processed in stages.
  - An example: Pipelined laundry.
  - A complete laundry includes the stages of washing, drying and folding, for example.



Non-pipelined sequence

- For non-pipelined process, a new load cannot start until the previous load is completed.
  - It takes 240 minutes to complete the four loads.
  - The delay of processing one load is 60 minutes.
  - The throughput is 1/60 load per minute.

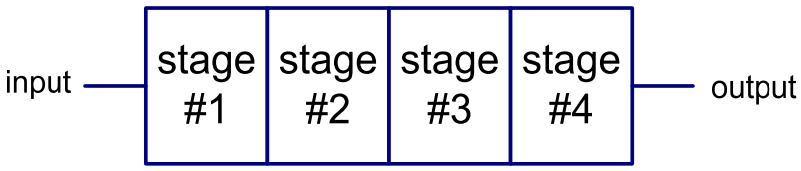


pipelined sequence

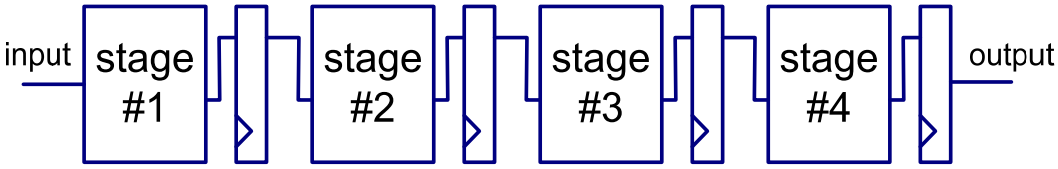
- For pipelined process,
  - It takes 120 minutes to complete the four loads.
  - The delay in processing one load **remains** 60 minutes.
  - The throughput increases to 2/60 load per minute.
  - To process k loads, it will take 40+20k minutes.
  - The throughput becomes k/(40+20k) load per minute –> 1/20 load per minute when k is large.

Pipelined combinational circuit

- A combinational circuit can be divided into stages so that the processing of different tasks can be overlapped.
- To ensure that the signals in each stage flow in the correct order and prevent any potential race, registers must be added between successive stages.
- The registers ensures that the signals can be passed to the next stage only at predetermined points.



Original combinational circuit



Pipelined circuit

Assume: propagation delay for each stage: T1, T2, T3 and T4.

T_{max} = max(T1, T2, T3, T4);

Thus, the minimum clock period has to accommodate the longest delay plus the overhead introduced by the buffer register in each stage:

$T_c = T_{max} + T_r$;

The effectiveness of the circuit

- Propagation delay:
  - non-pipelined circuit: T_{comb} = T1+T2+T3+T4
  - pipelined circuit: $T_{pipe} = 4T_c = 4T_{max} + 4T_r$
- Throughput:
  - non-pipelined circuit: $1/T_{comb}$
  - pipelined circuit: $k/(3T_c+kT_c) \rightarrow 1/T_c$.

Ideally, for an *N*-stage circuit

- The propagation delay of each stage is identical (i.e., $T_{max} = T_{comb}/N$)
- The register overhead (T_r) is comparably small
  - $T_{pipe} = NT_c = NT_{max} = T_{comb}$
  - Throughput: $1/T_c = 1/T_{max} = N/T_{comb}$
- Ideally, it is desirable to have more stages in the pipeline. However, when *N* becomes large,
  - the propagation delay of each stage becomes smaller, but T_r remains the same; its effect cannot be ignored.
  - In reality, it is difficult to keep dividing the original combinational circuit into smaller and smaller stages.

9.3 Adding pipeline to a combinational circuit

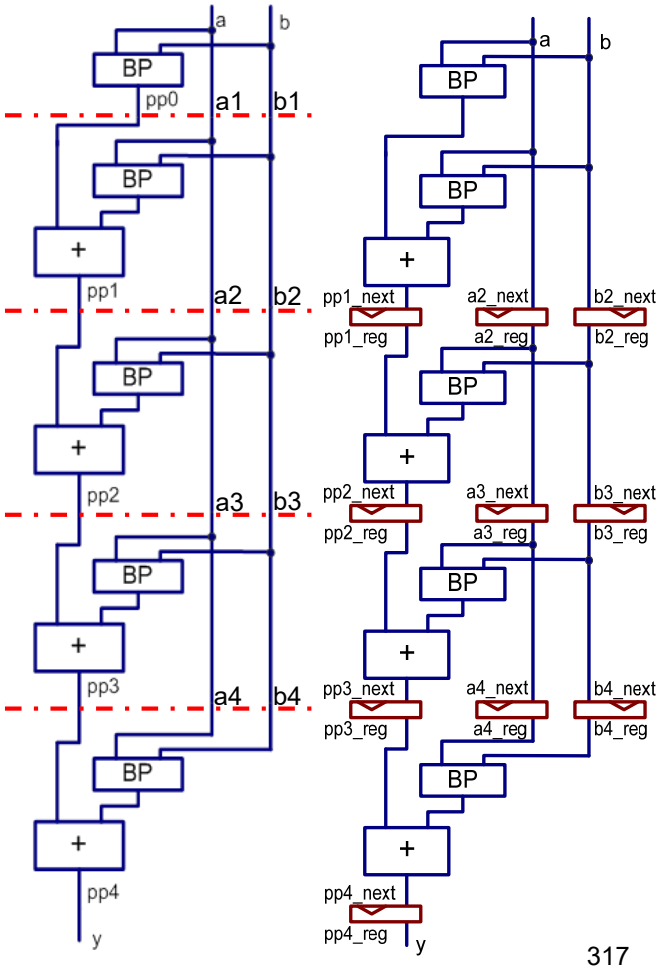
- The candidate circuits for effective pipeline design should include the following characteristics:
  - There is enough input data to feed the pipeline circuit.
  - The throughput is the main performance criterion.
  - The combinational circuit can be divided into stages with similar propagation delay.
  - The propagation delay of a stage is much longer than the delay incurred due to the register.

The procedure to convert a combinational circuit to a pipelined design

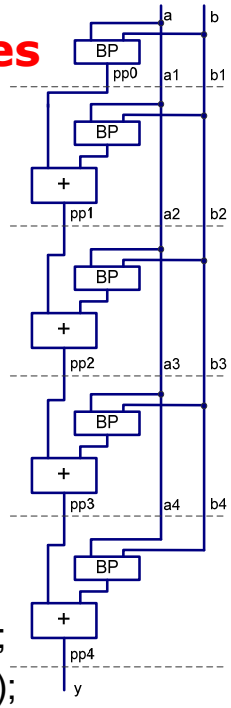
- Derive the block diagram of the original combinational circuit and arrange it as a cascading chain.
- Identify the major components and estimate the relative propagation delays of these components.
- Divide the chain into stages of similar propagation delays.
- identify the signals that cross the boundary of the chain.
- Insert registers for these signals in the boundary.

Example: Simple pipelined adder-based multiplier

x						a_4 b_4	a_3 b_3	a_2 b_2	a_1 b_1	a_0 b_0
						a_4b_0	a_3b_0	a_2b_0	a_1b_0	a_0b_0
					a_4b_1	a_3b_1	a_2b_1	a_1b_1	a_0b_1	
				a_4b_2	a_3b_2	a_2b_2	a_1b_2	a_0b_2		
+			a_4b_3	a_3b_3	a_2b_3	a_1b_3	a_0b_3			
		a_4b_4	a_3b_4	a_2b_4	a_1b_4	a_0b_4				
	y_9	y_8	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0

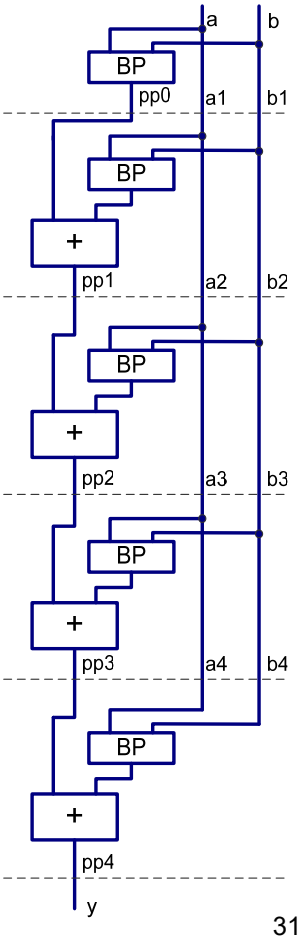


Non-pipelined multiplier in cascading stages

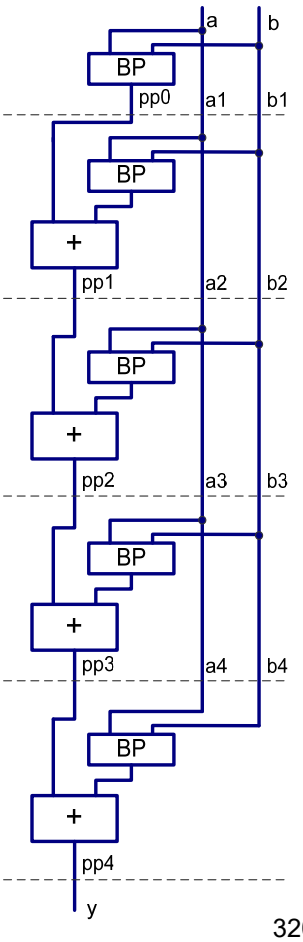


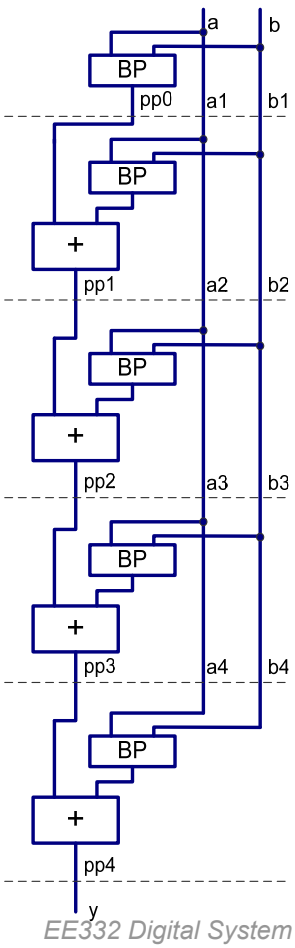

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity mult5 is
port (clk, reset : in std_logic;
        a, b : in std_logic_vector(4 downto 0);
        y: out std_logic_vector(9 downto 0));
end entity mult5;

architecture comb_arch of mult5 is
constant WIDTH : integer := 5;
signal a1, a2, a3, a4 : std_logic_vector (WIDTH-1 downto 0);
signal b1, b2, b3, b4 : std_logic_vector (WIDTH–1 downto 0);
signal bv0, bv1, bv2, bv3, bv4: std_logic_vector (WIDTH–1 downto 0);
signal bp0, bp1, bp2, bp3, bp4: std_logic_vector (2*WIDTH–1 downto 0);
signal pp0, pp1, pp2, pp3, pp4: std_logic_vector (2*WIDTH–1 downto 0);
```

```
begin
    -- stage 0
    bv0 <= (others => b(0));
    bp0 <= "00000" & (bv0 and a);
    pp0 <= bp0;
    a1 <= a;
    b1 <= b;
    -- stage 1
    bv1 <= (others => b1(1));
    bp1 <= "0000" & (bv1 and a1) & "0";
    pp1 <= pp0 + bp1;
    a2 <= a1;
    b2 <= b1;
    -- stage 2
    bv2 <= (others => b2(2));
    bp2 <= "000" & (bv2 and a2) & "00";
```




```
    pp2 <= pp1 + bp2;
    a3 <= a2;
    b3 <= b2;
    -- stage 3
    bv3 <= (others => b3(3));
    bp3 <= "00" & (bv3 and a3) & "000";
    pp3 <= pp2 + bp3;
    a4 <= a3;
    b4 <= b3;
    -- stage 4
    bv4 <= (others => b4(4));
    bp4 <= "0" & (bv4 and a4) & "0000";
    pp4 <= pp3 + bp4;
    -- output
    y <= pp4;
end architecture comb_arch;
```



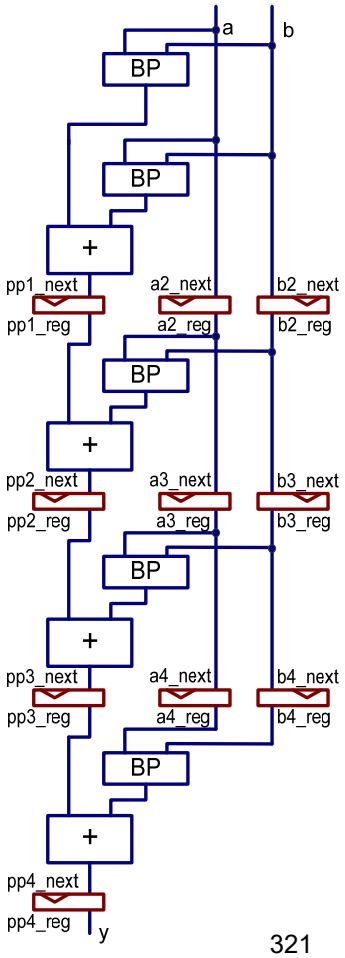


Non-pipelined circuit:

```
-- stage 2
pp2 <= pp1 + bp2;
-- stage 3
pp3 <= pp2 + bp3;
```

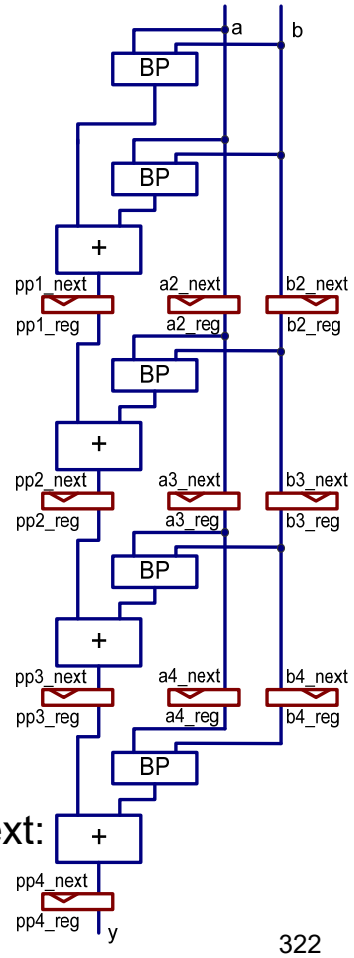
pipelined circuit:

```
-- register
if (reset = '1') then
      pp2_reg <= (others => '0');
      pp3_reg <= (others => '0');
elsif (clk'event and clk='1') then
      pp2_reg <= pp2_next;
      pp3_reg <= pp3_next;
end if;
…
-- stage 2
pp2_next <= pp1_reg + bp2;
-- stage 3
pp3_next <= pp2_reg + bp3;
```

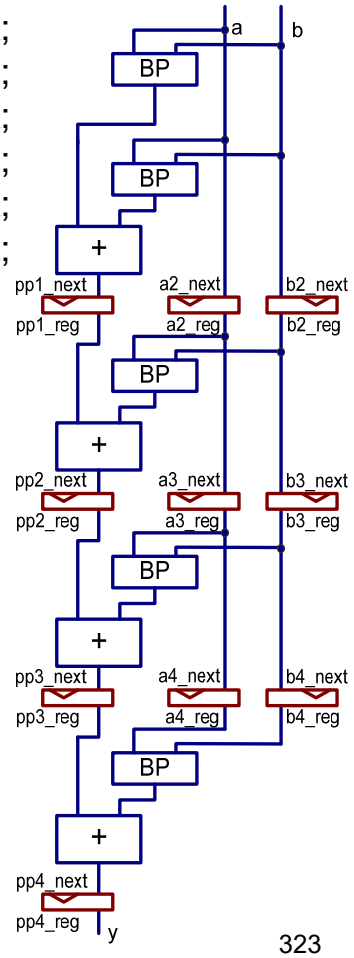


Pipelined multiplier

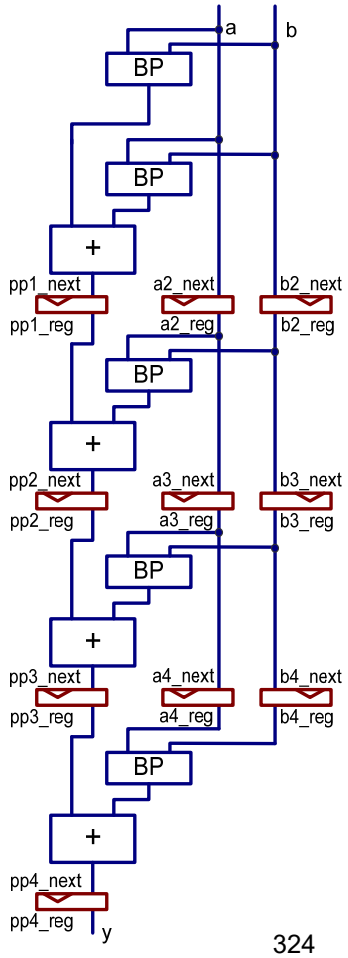
```
architecture pipe_arch of mult5 is
constant WIDTH : integer := 5;
signal a2_reg, a3_reg, a4_reg,
    b2_reg, b3_reg, b4_reg :
    std_logic_vector (WIDTH-1 downto 0);
signal a1, a2_next, a3_next, a4_next:
    std_logic_vector (WIDTH-1 downto 0);
signal b1, b2_next, b3_next, b4_next:
    std_logic_vector (WIDTH–1 downto 0);
signal bv0, bv1, bv2, bv3, bv4:
    std_logic_vector (WIDTH–1 downto 0);
signal bp0, bp1, bp2, bp3, bp4:
    std_logic_vector (2*WIDTH–1 downto 0);
signal pp1_reg, pp2_reg, pp3_reg, pp4_reg:
    std_logic_vector (2*WIDTH–1 downto 0);
signal pp0, pp1_next, pp2_next, pp3_next, pp4_next:
    std_logic_vector (2*WIDTH–1 downto 0);
```



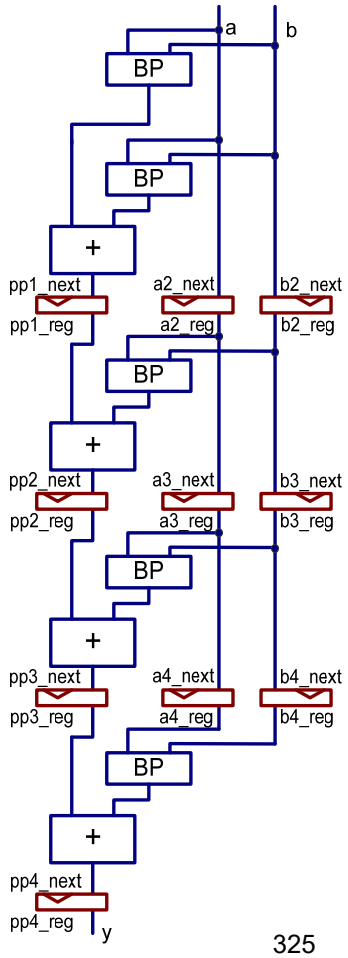
```
begin
  -- pipeline registers
  process (clk, reset)
  begin
    if (reset = '1') then
        pp1_reg <= (others => '0');
        pp2_reg <= (others => '0');
        pp3_reg <= (others => '0');
        pp4_reg <= (others => '0');
        a2_reg <= (others => '0');
        a3_reg <= (others => '0');
        a4_reg <= (others => '0');
        b2_reg <= (others => '0');
        b3_reg <= (others => '0');
        b4_reg <= (others => '0');
    elsif (clk'event and clk = '1') then
        pp1_reg <= pp1_next;
        pp2_reg <= pp2_next;
        pp3_reg <= pp3_next;
        pp4_reg <= pp4_next;
        a2_reg <= a2_next;
        a3_reg <= a3_next;
        a4_reg <= a4_next;
        b2_reg <= b2_next;
        b3_reg <= b3_next;
        b4_reg <= b4_next;
    end if;
  end process;
```




```
-- merged stage 0 & 1 for pipeline
bv0 <= (others => b(0));
bp0 <= "00000" & (bv0 and a);
pp0 <= bp0;
a1 <= a;
b1 <= b;
--
bv1 <= (others => b1(1));
bp1 <= "0000" & (bv1 and a1) & "0";
pp1_next <= pp0 + bp1;
a2_next <= a1;
b2_next <= b1;
-- stage 2
bv2 <= (others => b2_reg(2));
bp2 <= "000" & (bv2 and a2_reg) & "00";
```



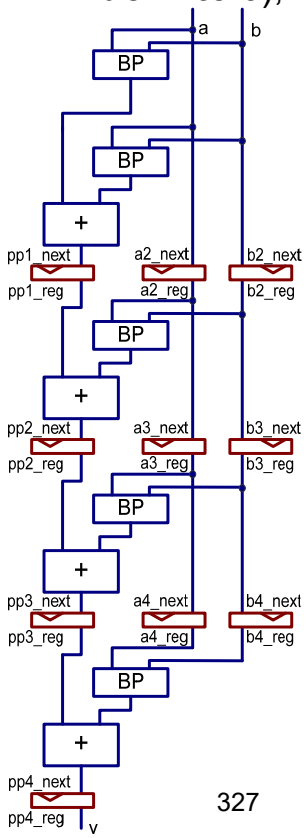
```
        pp2_next <= pp1_reg + bp2;
        a3_next <= a2_reg;
        b3_next <= b2_reg;
        -- stage 3
        bv3 <= (others => b3_reg(3));
        bp3 <= "00" & (bv3 and a3_reg) & "000";
        pp3_next <= pp2_reg + bp3;
        a4_next <= a3_reg;
        b4_next <= b3_reg;
        -- stage 4
        bv4 <= (others => b4_reg(4));
        bp4 <= "0" & (bv4 and a4_reg) & "0000";
        pp4_next <= pp3_reg + bp4;
        -- output
        y <= pp4_reg;
end architecture pipe_arch;
```



More efficient Pipelined multiplier

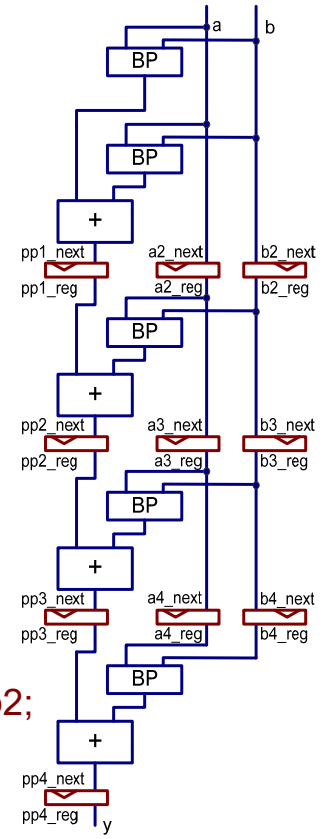
- Use a smaller (n+1)-bit adder to replace the 2n-bit adder in an n-bit multiplier.
- Reduce the size of the partial-product register
- Reduce the size of the registers that hold the b signal.

```
architecture efficient_pipe_arch of mult5 is
signal a2_reg, a3_reg, a4_reg: std_logic_vector(WIDTH-1 downto 0);
signal a1, a2_next, a3_next, a4_next:std_logic_vector(WIDTH-1 downto 0);
signal b1: std_logic_vector(4 downto 1);
signal b2_next, b2_reg: std_logic_vector (4 downto 2);
signal b3_next, b3_reg: std_logic_vector (4 downto 3);
signal b4_next, b4_reg: std_logic_vector (4 downto 4);
signal bv0, bv1, bv2, bv3, bv4:
     std_logic_vector (4 downto 0);
signal bp0, bp1, bp2, bp3, bp4:
     std_logic_vector (5 downto 0);
signal pp0: std_logic_vector  (5 downto 0);
signal pp1_next, pp1_reg:std_logic_vector (6 downto 0);
signal pp2_next, pp2_reg: std_logic_vector (7 downto 0);
signal pp3_next, pp3_reg: std_logic_vector (8 downto 0);
signal pp4_next, pp4_reg: std_logic_vector (9 downto 0);
```

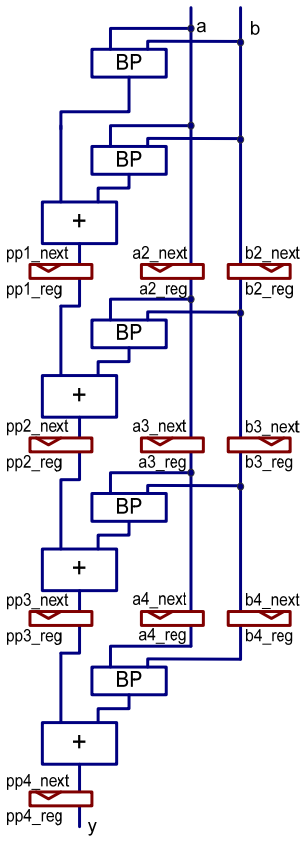



```
begin
    -- pipeline registers
    process (clk, reset)
    begin
        if (reset = '1') then
            pp1_reg <= (others => '0');
            pp2_reg <= (others => '0');
            pp3_reg <= (others => '0');
            pp4_reg <= (others => '0');
            a2_reg <= (others => '0');
            a3_reg <= (others => '0');
            a4_reg <= (others => '0');
            b2_reg <= (others => '0');
            b3_reg <= (others => '0');
            b4_reg <= (others => '0');
        elsif (clk'event and clk = '1') then
            pp1_reg <= pp1_next;
            pp2_reg <= pp2_next;
            pp3_reg <= pp3_next;
            pp4_reg <= pp4_next;
            a2_reg <= a2_next;
            a3_reg <= a3_next;
            a4_reg <= a4_next;
            b2_reg <= b2_next;
            b3_reg <= b3_next;
            b4_reg <= b4_next;
        end if;
    end process;
```

```
-- merged stage 0 & 1 for pipeline
bv0 <= (others => b(0));
bp0 <= "0" & (bv0 and a);
pp0 <= bp0;
a1 <= a;
b1 <= b (4 downto 1);
--
bv1 <= (others => b1(1));
bp1 <= "0" & (bv1 and a1);
pp1_next(6 downto 1) <= ("0" & pp0(5 downto 1)) + bp1;
pp1_next(0) <= pp0(0);
a2_next <= a1;
b2_next <= b1(4 downto 2);
-- stage 2
bv2 <= (others => b2_reg(2));
bp2 <= "0" & (bv2 and a2_reg);
pp2_next(7 downto 2) <= ("0" & pp1_reg(6 downto 2)) + bp2;
pp2_next(1 downto 0) <= pp1_reg(1 downto 0);
a3_next <= a2_reg;
b3_next <= b2_reg(4 downto 3);
```




```
-- stage 3
bv3 <= (others => b3_reg(3));
bp3 <= "0" & (bv3 and a3_reg);
pp3_next(8 downto 3) <=
        ("0" & pp2_reg(7 downto 3)) + bp3;
pp3_next(2 downto 0) <= pp2_reg(2 downto 0);
a4_next <= a3_reg;
b4_next <= b3_reg(4);
-- stage 4
bv4 <= (others => b4_reg(4));
bp4 <= "0" & (bv4 and a4_reg);
pp4_next(9 downto 4) <=
        ("0" & pp3_reg(8 downto 4)) + bp4;
pp4_next(3 downto 0) <= pp3_reg(3 downto 0);
-- output
y <= pp4_reg;
end architecture efficient_pipe_arch;
```



Chapter 10. Subprograms, packages and libraries

- For complex design, VHDL provides mechanics for structuring programs, reusing modules
  - Subprograms such as functions and procedures for encapsulating commonly used operations
  - Packages and libraries for sharing large bodies of code.

10.1 Functions

- Functions are used to compute a value based on the values of the input parameters.

- Function declaration:

function func_name (parameter_list) **return** data_type**;**

formal parameter

return value type

Parameter values in functions are used, but not changed within the function.

- Structure of a function

```
function rising_edge (clock: std_logic) return boolean is
--
-- declarative region: declare variables local to the function
--
begin
-- body
-- sequential statement;
-- sequential statement;
-- --
return (value);
end function rising_edge;
```


- Calling a function in a VHDL module:

```
rising_edge (enable);    -- positional association
```

actual parameter,
the type must match to the formal parameter, i.e., enable must be a **signal** of type std_logic.

```
rising_edge (clock => enable); -- name association
```

- Functions execute in zero simulation time.
  - Wait statement are not permitted in functions.

```
entity dff is
port (    D_in, CLK: in std_logic;
          D_out: out std_logic);
end entity dff;
architecture behavior of dff is
    function rising_edge(signal clock: in std_logic) return boolean is
          variable edge : boolean :=FALSE;
    begin
          edge := (clock='1' and clock'event);
          return (edge);
    end function rising_edge;
begin
    process (CLK) is
    begin
          if (rising_edge(CLK)) then
                    D_out <= D_in;
          end if;
    end process;
end architecture behavior;
```

function definition

A majority function

- It returns '1' if two or more of the 3 input parameters, *a*, *b* and *c* are '1'.
- It can be treated as a shorthand for the $a \cdot b + a \cdot c + b \cdot c$ expression

```
architecture arch of …
    -- declaration
    function maj(a, b, c: std_logic) return std_logic is
          variable result: std_logic;
    begin
          result := (a and b) or (a and c) or (b and c);
          return result;
    end function maj;
    signal i1, i2, i3, i4, x, y: std_logic;
begin
    …
    x <= maj(i1, i2, i3) or i4;
    y <= i1 when maj(i2, i3, i4)='1' else …
end arch;
```

Type conversion functions

- To make assignment from an object of one type to an object of another type.
  - for example: bit_vector and std_logic_vector.

```
function to_bitvector (svalue: std_logic_vector) return bit_vector is
   variable outvalue : bit_vector(svalue'length – 1 downto 0);
begin
   for i in svalue' range loop -- scan all elements of the array
         case svalue(i) is
               when '0' => outvalue (i) := '0';
               when '1' => outvalue (i) := '1';
               when others => outvalue (i) := '0';
         end case;
   end loop;
   return outvalue;
end function to_bitvector;
```


A Function performing $\lceil \log_2 n \rceil$

```
function log2c (n: integer) return integer is
   variable m, p: integer;
begin
   m := 0;
   p := 1;
   while p < n loop
      m := m+1;
      p := p*2;
   end loop;
   return m;
end function log2c;
```

Summary

- Unlike entity and architecture, functions (and procedures) are **not design units** and thus cannot be processed independently.
- In synthesis, functions should not be used to specify the design hierarchy, but should **be treated as a shorthand** for simple, repeatedly used operations.
- A function can be thought of as **an extension of the expression** and can be "called" whenever an expression is used.

9.2 Package

- The primary purpose of a package is to collect elements that can be shared (globally) among two or more design units. A package is represented by:
  - a package declaration and, optionally,
  - a package body.

contains a set of declarations that may possibly be shared by many design units. It defines items that can be made visible to other design units, for example, a function declaration.

- Package declaration and package body are design units of VHDL.

An example of a package declaration

```
package SYNTH_PACK is
    constant LOW2HIGH : TIME := 20ns;
    type ALU_OP is (ADD, SUB, MUL, DIV, EQL);
    type MVL is ('U', '0', '1', 'Z');
    component NAND2
            port (A, B : in MVL; C : out MVL);
    end component;
    -- subprogram, type, constant, signal, variable, component …,
    -- and use clause can be declared in package declaration
end package SYNTH_PACK;
```

- If the declarations include items such as functions or procedure declarations, the behavior of the function and procedure are specified in a separate design unit called the package body.


```
use WORK.SYNTH_PACK.all;

package PROGRAM_PACK is
    constant PROP_DELAY : TIME;
    function ISZERO(A: MVL) return boolean;
end package PROGRAM_PACK;
```

- In this case, a package body is required.
- A package body primarily contains the behavior of the subprograms declared in a package declaration. It may also contain other declarations.

An example of a package body

```
use WORK.SYNTH_PACK.all;

package PROGRAM_PACK is
    constant PROP_DELAY : TIME;
    function ISZERO(A: MVL) return boolean;
end package PROGRAM_PACK;

package body PROGRAM_PACK is
    constant PROP_DELAY : TIME := 15ns;

    function ISZERO(A: MVL) return boolean is
    begin
        if (A='0') return TRUE;
        else return FALSE;
        end if;
    end function ISZERO;
end package body PROGRAM_PACK;
```

Package declaration

package body name: must be the same as of its corresponding package declaration.

Package body

Note:

- An item declared inside a package body has its scope restricted to be within the package body, and this item cannot be made visible in other design unit. (This is in contrast to items declared in a package declaration).

10.4 Design libraries

- Each design unit - entity, architecture, configuration, package declaration, package body is analyzed (complied) and placed in a design library for subsequent use.
- To use a design library, the library must be declared by its logical name.

  library logical-library-name1, logical-library-name2,...;
- In VHDL, the libraries STD and WORK are implicitly declared.

standard packages provided with the VHDL distributions

working directory

- Once a library is declared, all of the functions, procedures, and type declaration of a package in that library can be made accessible to a VHDL model through the **use** clause.

  library IEEE

  use IEEE.std_logic_1164.**all**;
- These clauses apply only to the immediate entity-architecture pair! ***Visibility must be established for other design units separately***!