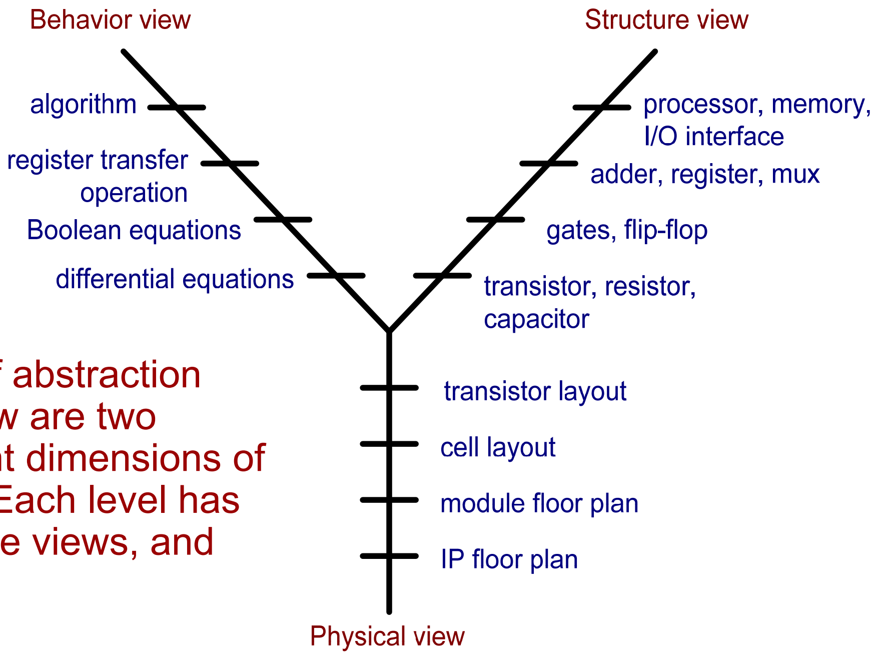
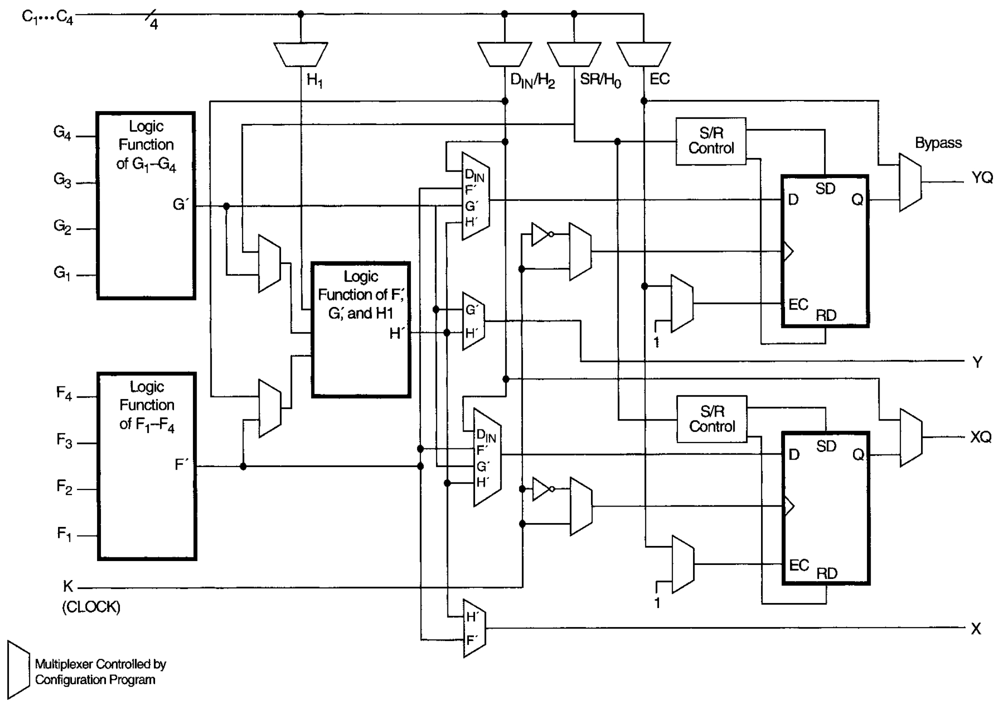
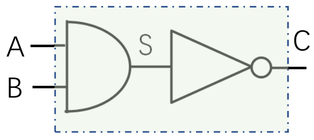


- The level of abstraction and the view are two independent dimensions of a system. Each level has its own three views, and vice verser.

CLB - Configurable logic block





Specifies input and output signals

Entity name

```
entity NAND2 is
port (A, B:   in  bit;
      C    :   out  bit );
end entity NAND2;
```

Port mode

Port name

Signal type of port

Architecture name

Entity name

declaration

concurrent statements

```
architecture DATAFLOW  of NAND2 is
    signal S: bit;
begin
    S <= A and B;
    C <= not S;
end architecture DATAFLOW;
```

Data Object – Signal, Variable, and Constant

Where to declare?

- Entity declaration – Signal
- Declarative section of an architecture – Signal, Constant
- Process – Variable, Constant

How to declare?

signal/variable/constant name : data_type [:= initial_value];

Examples:

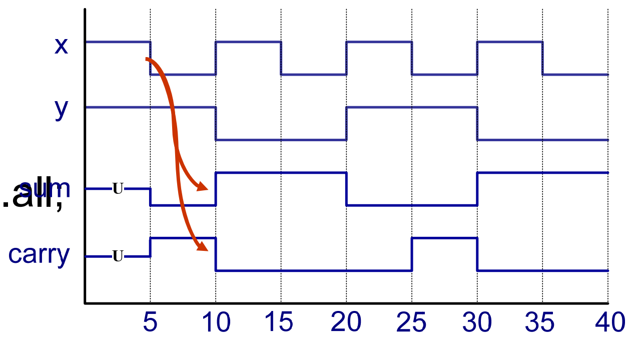
signal status : std_logic := ‘0’;
variable data : std_logic_vector (31 downto 0);
constant yes : **Boolean** := **True**;

```
entity nand2 is
port (A, B:   in  bit;
      C    :   out  bit );
end entity nand2;
```

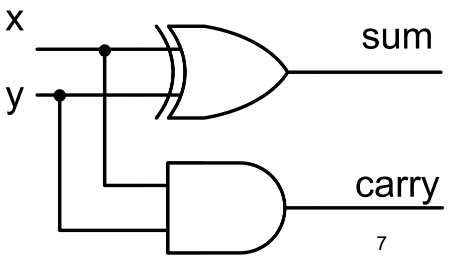
```
architecture dataflow  of
nand2 is
   signal S: bit;
begin
         S <= A and B;
         C <= not S;
end architecture dataflow;
```

Operator	Description	Data type of a	Data type of b	Data type of result
a ** b	exponentiation	integer		
abs a	absolute value	integer		
not a	negation	boolean, bit, bit_vector		
a * b, a / b, a **mod** b, a **rem** b	multiplication, division, modulo, remainder	integer		
+a, -a	identity, negation	integer		integer
a + b, a - b a & b	addition, subtraction, concatenation	integer		
		1D array, element		
a **sll** b, a **srl** b, a **sla** b, a **sra** b, a **rol** b, a **ror** b	shift-left (right) logical, shift-left (right) arithmetic, rotate left (right)	bit_vector	integer	bit_vector
a = b, a /= b, a < b, a <= b, a > b, a >= b		any	same as a	boolean
		scalar or 1D array	same as a	boolean
a **and** b, a **or** b, a **xor** b, a **nand** b, a **nor** b, a **xnor** b		boolean, bit, bit_vector	same as a	same as a

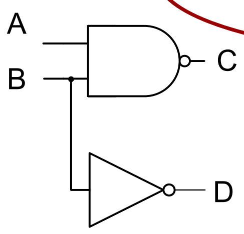
CSA example: Half_adder



```
library IEEE;
use IEEE.std_logic_1164.all;
entity half_adder is
port (x, y : in std_logic;
      sum, carry : out std_logic);
end entity half_adder;
architecture concurrent_behavior of half_adder
  is
begin
  sum <= ( x xor y) after 5 ns;
  carry <= (x and y) after 5 ns;
end architecture concurrent_behavior;
```



process



sensitivity list.

process declarative section

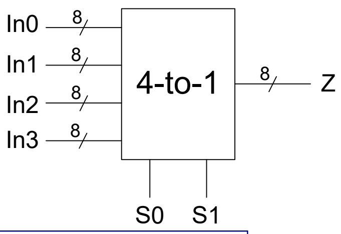
sequential statement

```
entity ex_proc is
port (A, B:   in  std_logic;  C, D:   out  std_logic );
end entity ex_proc;

architecture BEHAVIOR  of ex_proc is
begin
    process (A,B) is
      variable DELAYT : time : = 0 ns;
    begin
      DELAYT := DELAYT + 1 ns;
      if (A='1' and 'B'=1) then
        C <= '0' after DELAYT;
      else
        C <= '1' after DELAYT;
      end if;
    end process;
    D <= not B;
end architecture BEHAVIOR;
```

process statement

Two concurrent statements

4-to-1, 8-bit multiplexor



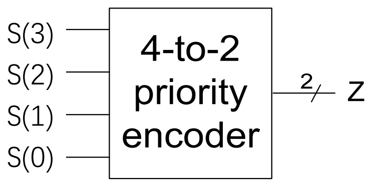
```
library IEEE;
use IEEE.std_logic_1164.all;

entity mux4 is
port (In0, In1, In2, In3 : in std_logic_vector (7 downto 0);
        S: in std_logic_vector(1 downto 0);
        Z : out std_logic_vector (7 downto 0) );
end entity mux4;
```

```
architecture con_arch of mux4 is
begin
   Z <=  In0 when S = “00” else
          In1 when S = “01” else
          In2 when S = “10” else
          In3;
end architecture con_arch1;
```

```
architecture sel_arch of mux4 Is
begin
   with S select
      Z <= In0 when “00”,
           In1 when “01”,
           In2 when “10”,
           In3 when others;
end architecture sel_behavioral;
```

```
architecture if_arch of mux4
is begin
Process (In0, In1, In2, In3, S)
begin
   if (S = “00”) then Z <= In0;
   elsif (S = “01”) then
      Z <= In1;
   elsif (S = “10”) then
      Z <= In2;  else Z <= In3;
   end if;
   end process;
end architecture if_arch;
```

```
architecture case_arch of
mux4 is begin
   process(In0, In1, In2, In3, S)
   begin case S is
         when “00” => Z <= In0;
         when “01” => Z <= In1;
         when “10” => Z <= In2;
         when others =>
            Z <= In3;
      end case;
   end process;
end architecture case_arch;
```



```
library IEEE;
use IEEE.std_logic_1164.all;

entity pr_encoder is
port (S: in std_logic_vector(3 downto 0);
        Z : out std_logic_vector (1 downto 0) );
end entity pr_encoder;
```

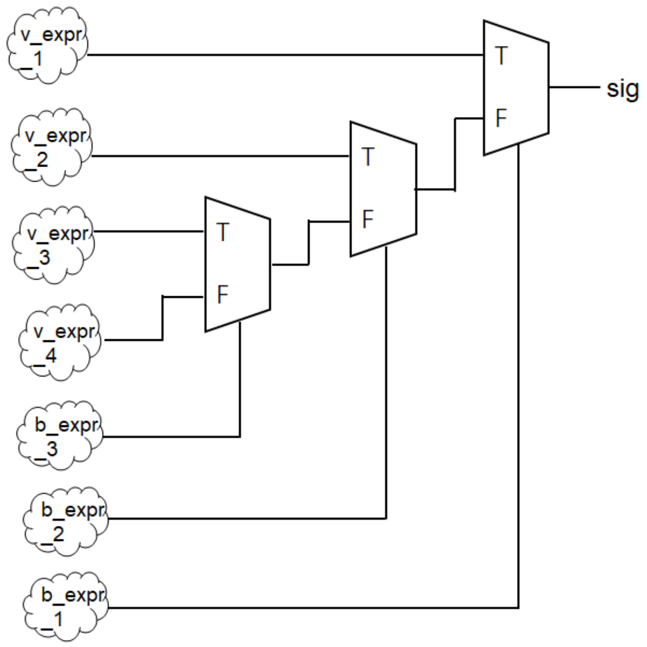
```
architecture con_behavioral of
   pr_encoder is
begin
   Z <= “11” when S(3) = ‘1’ else
          “10” when S(2) = ‘1’ else
          “01” when S(1) = ‘1’ else
          “00” when S(0) = ‘1’ else
          “00”;
end architecture con_behavioral;
```

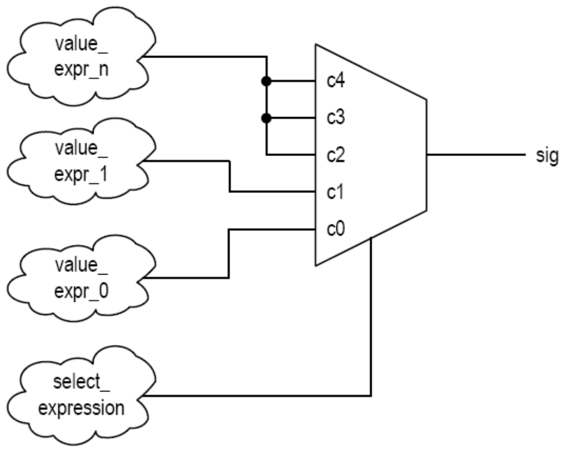
```
architecture sel_arch of
   pr_encoder is
begin
   with S select
   Z <= “11” when “1000” | “1001” |
             “1010” | “1011”| “1100” |
             “1101” | “1110” | “1111”,
         “10” when “0100” |“0101”|
             “0110” | “0111”,
         “01” when “0010” |“0011”,
         “00” when others;
end architecture behavioral;
```

```
architecture if_arch of
pr_encoder is begin
process(S)
   begin
      if (S(3) = ‘1’) then
         Z <= “11”;
      elsif (S(2) = ‘1’) then
         Z <= “10”;
      elsif (S(1) = ‘1’) then
         Z <= “01”;
      else Z <= “00”;
      end if;
   end process;
end if_arch;
```

```
sig <=  v_expr_1 when b_expr_1 else
        v_expr_2 when b_expr_2 else
        v_expr_3 when b_expr_3 else
        v_expr_4;
```

```
with slect_expression select
   sig <= value_expr_0 when c0,
          value_expr_1 when c1,
          value_expr_n when others;
```





Sequential statements

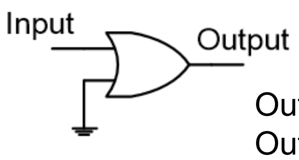
signal-object <= *expression* [**after** *delay-value*];

variable-object := *expression*;

null;

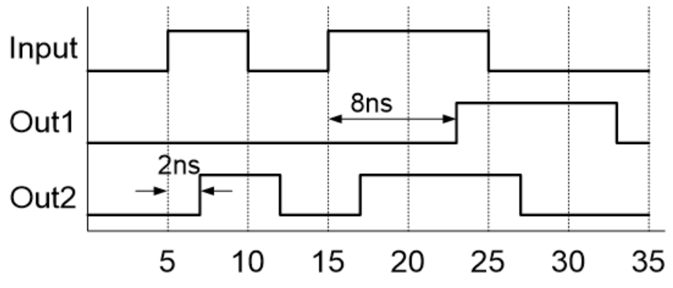
if *condition* **then**
   sequential-statements
{ **elsif** *condition* then
   sequential-statements }
[**else**
   sequential-statements]
end if;

case *expression* **is**
   when *choices* =>
     {*sequential-statements* }
   { **when** choices =>
     { *sequential-statements* } }
end case;

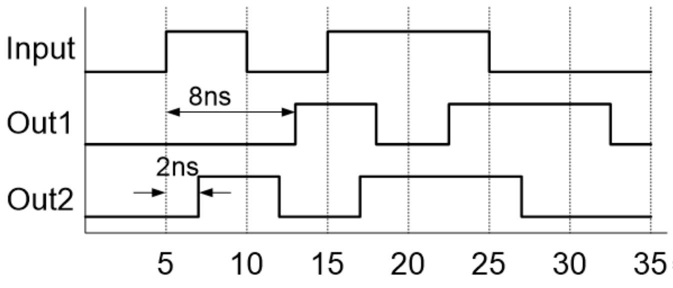


Understanding delays

```
Out1 <= Input or '0' after 8 ns;
Out2 <= Input or '0' after 2 ns;
```



```
Out1 <= transport (Input or '0') after 8 ns;
Out2 <= transport (Input or '0') after 2 ns;
```

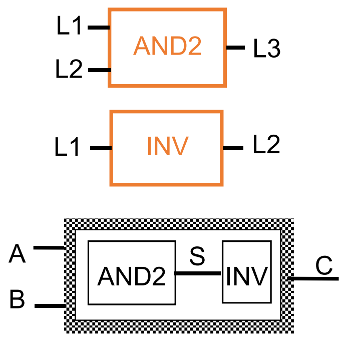


Example: Signal assignment with process

```
library IEEE;
use IEEE.std_logic_1164.all;
entity sig_var is
port (x, y, z : in std_logic; res1, res2 : out std_logic);
end entity sig_var;
architecture behavior of sig_var is
signal sig_s1, sig_s2 : std_logic;
begin
   proc1: process (x, y, z) is
   variable var_s1, var_s2 : std_logic;
   begin
         L1: var_s1 := x and y;
         L2: var_s2 := var_s1 xor z;
         L3: res1 <= var_s1 nand var_s2;
   end process proc1;
   proc2: process (x, y, z) is
   begin
         L1: sig_s1 <= x and y;
         L2: sig_s2 <= sig_s1 xor z;
         L3: res2 <= sig_s1 nand sig_s2;
   end process proc2;
end architecture behavior;
```

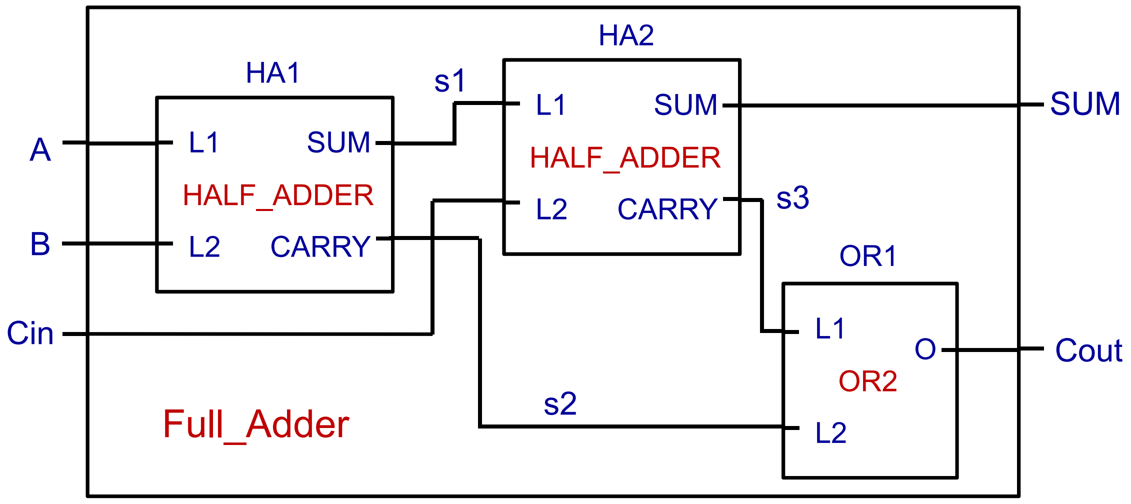
Modeling Structure

Component declaration: a component instance needs to be bound to an entity



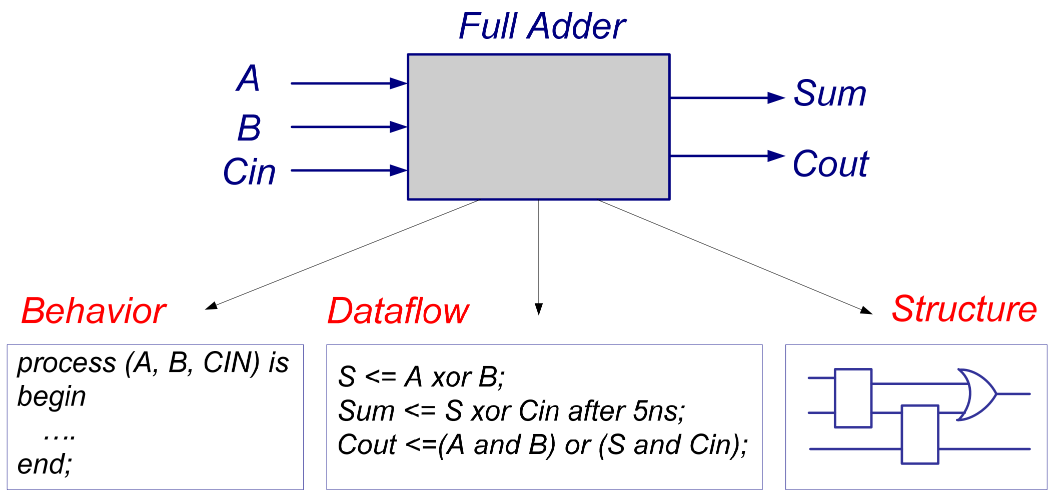
Component instantiation

```
entity NAND2 is
     port (A, B: in  bit; C : out  bit );
end entity NAND2;

architecture STRUCTURE  of  NAND2 is
     component AND2 is
          port (L1, L2 : in bit;
                L3      : out bit);
     end component AND2;
     component INV is
          port (L1 : in bit;
               L2: out bit);
     end component INV;
     signal S : BIT;
begin
     A1: AND2 port map (L1 => A, L2 => B, L3 => S);
     A2: INV port map (L1=>S, L2=> C);
end architecture STRUCTURE;
```



```
architecture STRUCTURE  of Full_Adder is
     -- Declare components and intermediate signal here.
begin
    HA1: HALF_ADDER port map (L1=>A, L2=>B, SUM=>s1, CARRY=>s2);
    HA2: HALF_ADDER port map (L1=>s1, L2=>CIN, SUM=>SUM, CARRY=>s3);
    OR1: OR2       port map (L1=>s3, L2=>s2, O=>COUT);
 end architecture STRUCTURE;
```

Full Adder: Entity and architecture

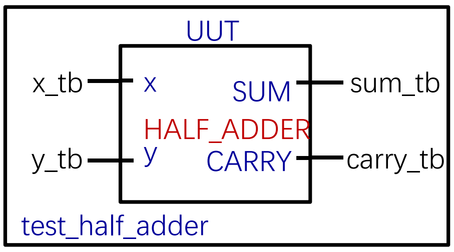


```
process (A, B, CIN) is
begin
   ….
end;
```

```
S <= A xor B;
Sum <= S xor Cin after 5ns;
Cout <=(A and B) or (S and Cin);
```

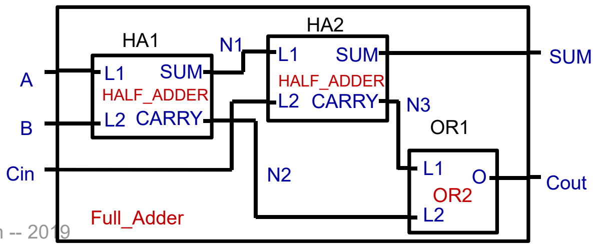
Test benches that compute stimulus and expected results

```
Begin
  tb: process is -- define a process to apply input stimulus and verify outputs.
    constant PERIOD: time := 20 ns;
    constant n : integer := 2;
  begin -- apply every possible input combination
    for i in 0 to 2**n - 1 loop
      (x_tb, y_tb) <= to_unsigned(i, n);
      wait for PERIOD;
      assert ( (sum_tb = (x_tb xor y_tb)) and (carry_tb = (x_tb and y_tb)) )
      report "Test failed" severity ERROR;
    end loop;
    wait;
  end process;
 UUT: half_adder port map (x =>x_tb, y => y_tb, sum =>  sum_tb,  carry => carry_tb);
end architecture;
```



```
architecture STRUCTURE  of Full_Adder is
    component HALF_ADDER is
    port (L1, L2 : in bit;   SUM, CARRY : out bit);
    end component HALF_ADDER;
    component OR_GATE is
    port (L1, L2 : in bit;    O  : out bit);
    end component OR_GATE;
    for HA1: HALF_ADDER use entity HALF_ADDER(BEHAVIOR);
    for HA2: HALF_ADDER use entity HALF_ADDER(STRUCTURE);
    signal N1, N2, N3 : BIT;
begin
   OR1: OR2 port map (L1=>N3, L2=>N2, O=>COUT);
   HA1: HALF_ADDER port map (L1=>A, L2=>B, SUM=>N1,CARRY=>N2);
   HA2: HALF_ADDER port map (L1=>N1, L2=>CIN, SUM=>SUM,
            CARRY=>N3);
end architecture STRUCTURE;
```

Configuration specifications

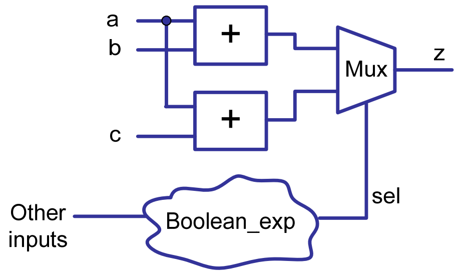


Operator sharing

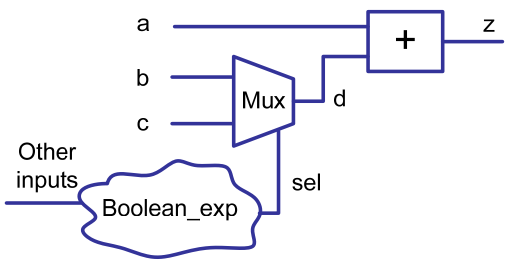
when a control signal is 1, z <= a + b;
Otherwise, z <= a + c;

- 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;
```

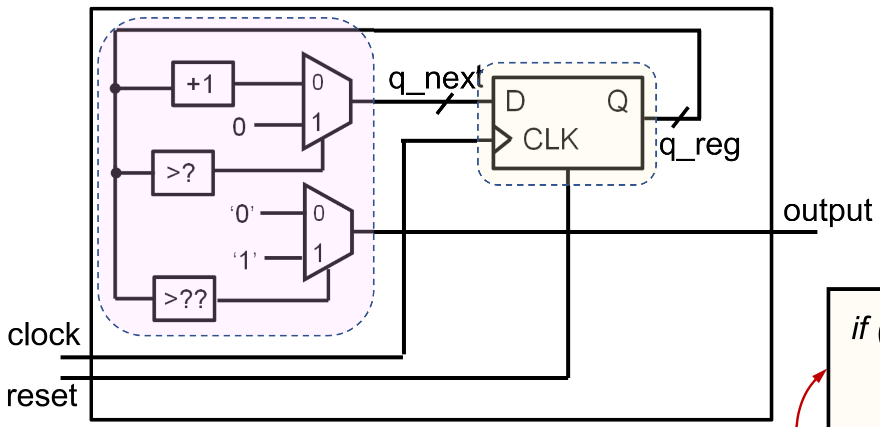


```
begin
    au <= a;
    bv0 <= (others => b(0));
    bv1 <= (others => b(1));
    bv2 <= (others => b(2));
    bv3 <= (others => b(3));
    bv4 <= (others => b(4));
    p0 <= "00000" & (bv0 and au);
    p1 <= "0000" & (bv1 and au) & '0';
    p2 <= "000" & (bv2 and au) & "00";
    p3 <= "00" & (bv3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

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

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

$$
\begin{array}{lcccccccccc}
 & & & & & & a_4 & a_3 & a_2 & a_1 & a_0 \\
\times & & & & & & b_4 & b_3 & b_2 & b_1 & b_0 \\
\hline
 & & & & & & a_4b_0 & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & & & & & a_4b_1 & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & \\
 & & & & a_4b_2 & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & \\
+ & & & a_4b_3 & a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & \\
 & & a_4b_4 & a_3b_4 & a_2b_4 & a_1b_4 & a_0b_4 & & & & \\
\hline
 & y_9 & y_8 & y_7 & y_6 & y_5 & y_4 & y_3 & y_2 & y_1 & y_0
\end{array}
$$



A synchronous section with asynchronous inputs

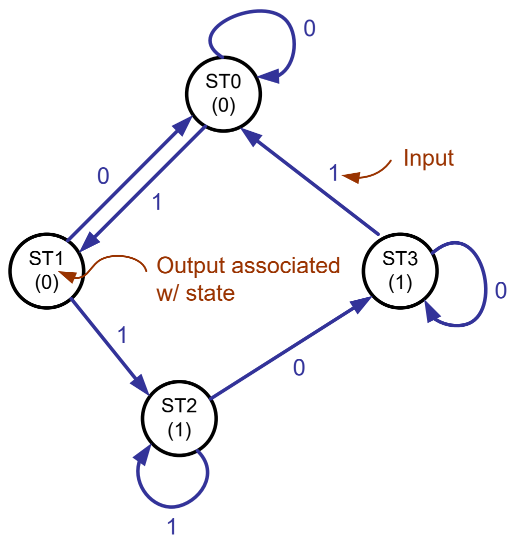
```
if (async_sig = '1') then
    q_reg <= '0';
    -- active high asynchronous reset
elsif (CLK's event AND CLK='1') then
    q_reg <= q_next;
    -- CLK is the clock input to reg Q
end if;
```

+

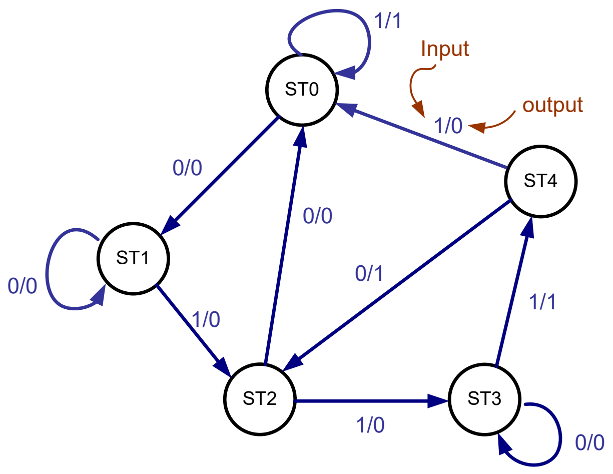
A combinational section

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

State transition diagrams



A Moore machine

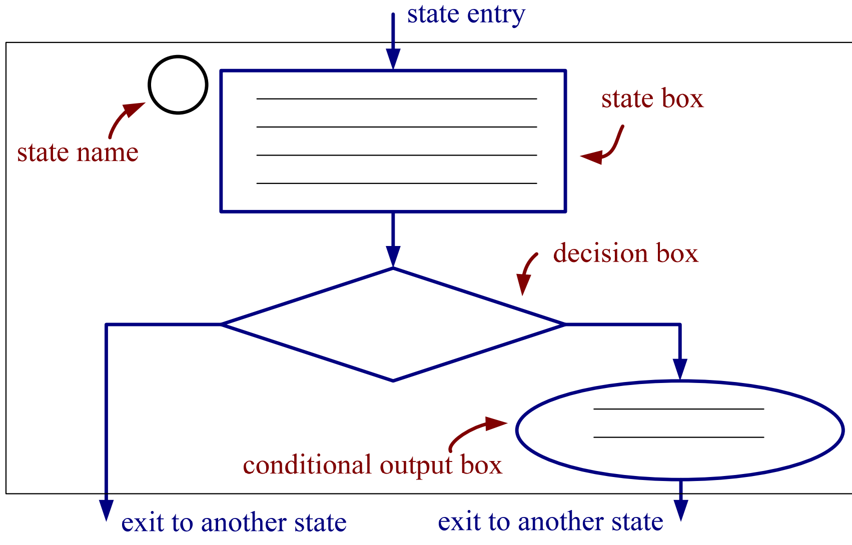


A Mealye machine

-- 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;
```

ASM Chart Block



FSMD Design Procedure

Step 1: Defining the input and output signals

Step 2: Converting the algorithm to an ASM chart

Step 3: Constructing the FSMD

3.1 List all possible RT operations in the ASM chart.

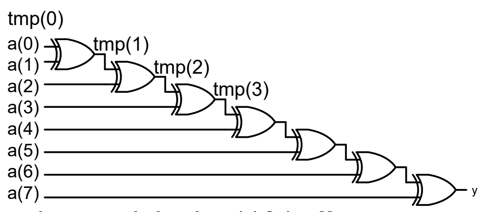
3.2 Group RT operations according to their destination registers.

3.3 Derive the circuit for each group RT operation.

3.4 Add the necessary circuits to generate the status signals.

Step 4: VHDL descriptions of FSMD

Parameterized Design: 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;
```

```
library ieee; use ieee.std_logic_1164.all;
entity generic_demo is
      port( a1: in std_logic_vector(3 downto 0);
              a2: in std_logic_vector(7 downto 0);
              y1, y2: out std_logic);
end entity generic_demo;
architecture arch of generic_demo is
   component reduced_xor is
      generic (WIDTH: natural); -- generic declaration
      port(a: in std_logic_vector(WIDTH–1 downto 0);
             y: out std_logic);
   end component reduced_xor;
begin  four_bit: reduced_xor
                  generic map (WIDTH => 4)
                  port map (a => a1, y => y1);
          eight_bit: reduced_xor
                  generic map (WIDTH => 8)
                  port map (a => a1, y => y1);
end architecture arch;
```

Parameterized Design: Array Attribute

```
library ieee;  use ieee.std_logic_1164.all;
entity reduced_xor is
      port( a: in std_logic_vector;
             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;
```

```
library ieee; use ieee.std_logic_1164.all;
entity generic_demo is
      port( a1: in std_logic_vector(3 downto 0);
             a2: in std_logic_vector(7 downto 0);
             y1, y2: out std_logic);
end entity generic_demo;
architecture arch of generic_demo is
   component reduced_xor is
      port(a: in std_logic_vector;
            y: out std_logic);
   end component reduced_xor;
begin  four_bit: reduced_xor
                 port map (a => a1, y => y1);
          eight_bit: reduced_xor
                 port map (a => a1, y => y1);
end architecture arch;
```

Generate Statement

```
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 generate_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 generate_arch;
```

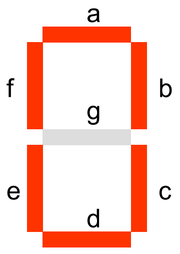
```
library ieee; use ieee.std_logic_1164.all;
entity generic_demo is
      port( a1: in std_logic_vector(3 downto 0);
             a2: in std_logic_vector(7 downto 0);
             y1, y2: out std_logic);
end entity generic_demo;
architecture arch of generic_demo is
   component reduced_xor is
      generic (WIDTH: natural); -- generic declaration
      port(a: in std_logic_vector(WIDTH–1 downto 0);
            y: out std_logic);
   end component reduced_xor;
begin  four_bit: reduced_xor
                 generic map (WIDTH => 4)
                 port map (a => a1, y => y1);
          eight_bit: reduced_xor
                 generic map (WIDTH => 8)
                 port map (a => a1, y => y1);
end architecture arch;
```

Generate Statement

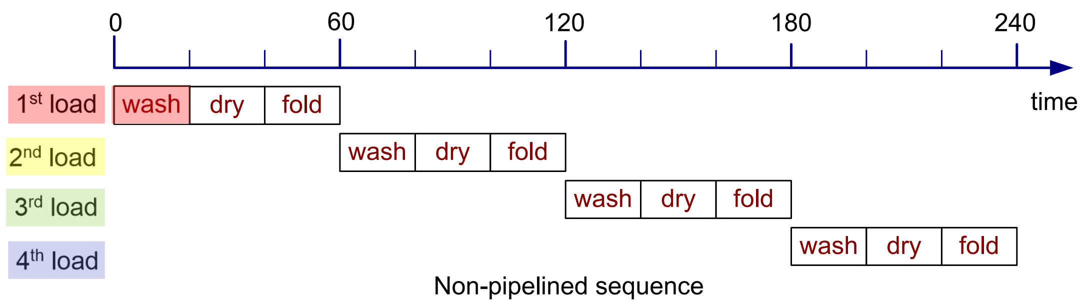
```
library ieee;  use ieee.std_logic_1164.all;
entity reduced_xor is
     generic (WIDTH: natural); -- generic declaration
     port( a: in std_logic_vector(WIDTH–1 downto 0);
           y: out std_logic);
end entity reduced_xor;

architecture generate_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
     left_gen: if i = 1 generate -- leftmost stage
               tmp(i) <= a(i) xor a(0); end generate;
     middle_gen: if (i>1) and (i<(WIDTH –1)) generate
               tmp(i) <= a(i) xor tmp(i–1); end generate;
     right_gen: if i = (WIDTH –1) generate  -- rightmost stage
               y <= a(i) xor tmp(i–1); end generate;
   end generate;
end architecture generate_arch;
```

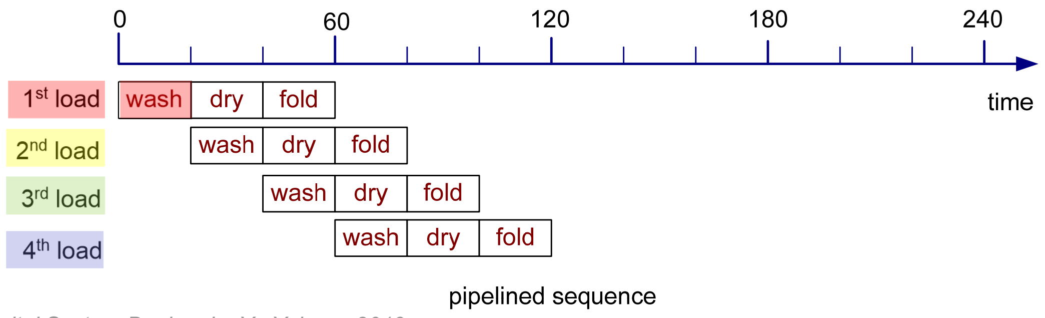
```
library ieee; use ieee.std_logic_1164.all;
entity generic_demo is
     port( a1: in std_logic_vector(3 downto 0);
           a2: in std_logic_vector(7 downto 0);
           y1, y2: out std_logic);
end entity generic_demo;
architecture arch of generic_demo is
   component reduced_xor is
     generic (WIDTH: natural); -- generic declaration
     port(a: in std_logic_vector(WIDTH–1 downto 0);
          y: out std_logic);
   end component reduced_xor;
begin  four_bit: reduced_xor
              generic map (WIDTH => 4)
              port map (a => a1, y => y1);
       eight_bit: reduced_xor
              generic map (WIDTH => 8)
              port map (a => a1, y => y1);
end architecture arch;
```

0000

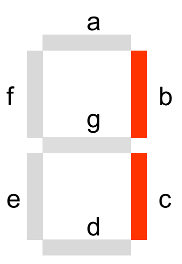


An Example of non-pipelined laundry and pipelined laundry

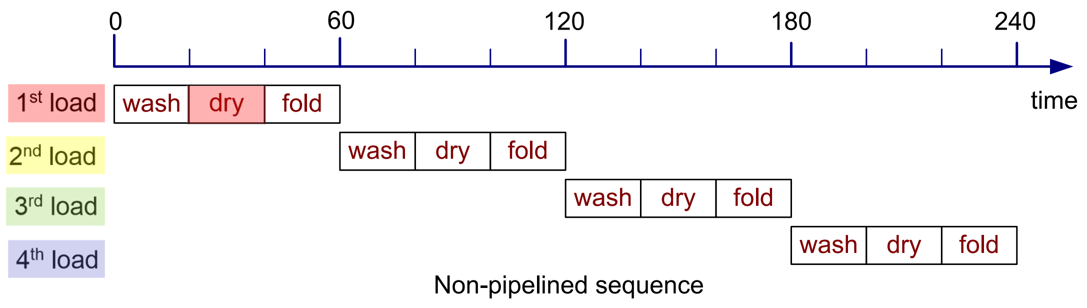




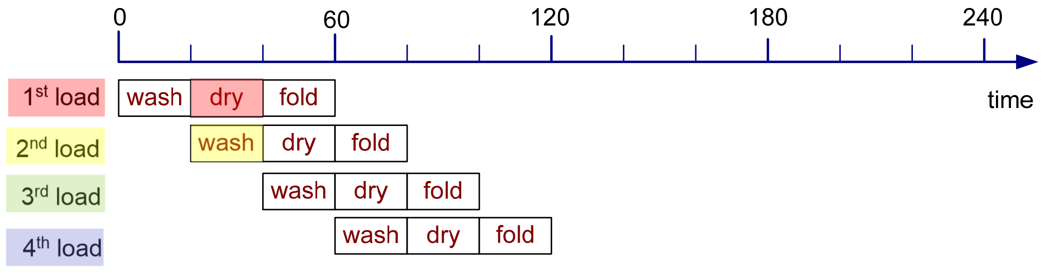
0001



An Example of non-pipelined laundry and pipelined laundry

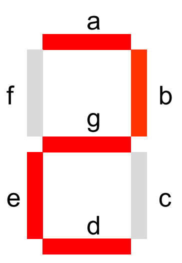


Non-pipelined sequence

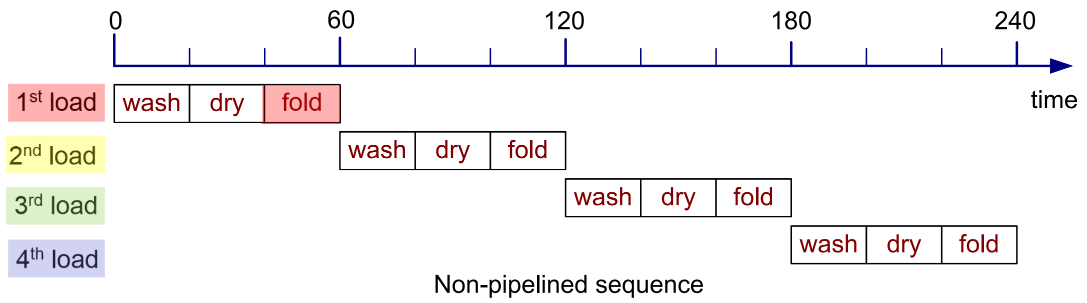


pipelined sequence

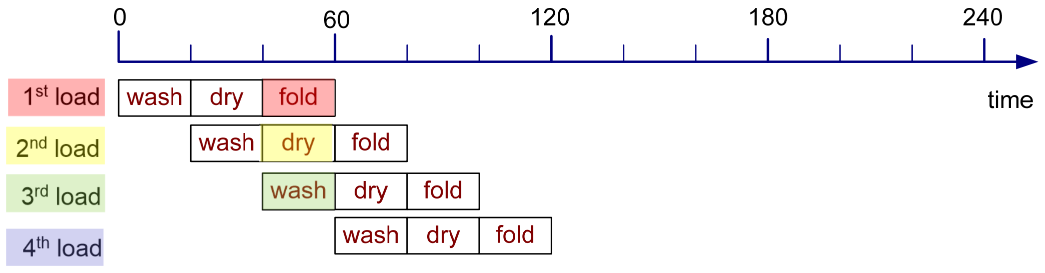
0010



An Example of non-pipelined laundry and pipelined laundry

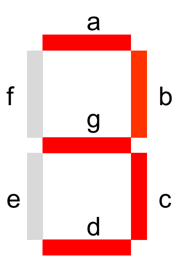


Non-pipelined sequence

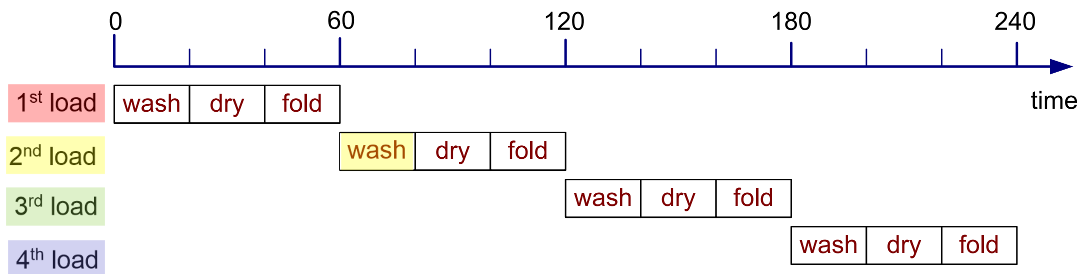


pipelined sequence

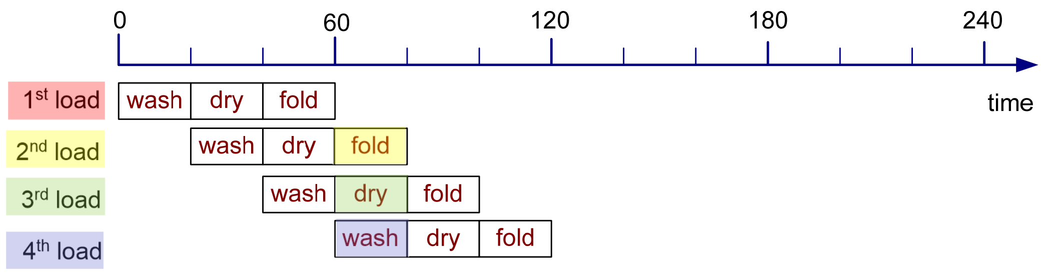
0011



An Example of non-pipelined laundry and pipelined laundry

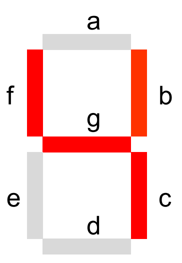


Non-pipelined sequence

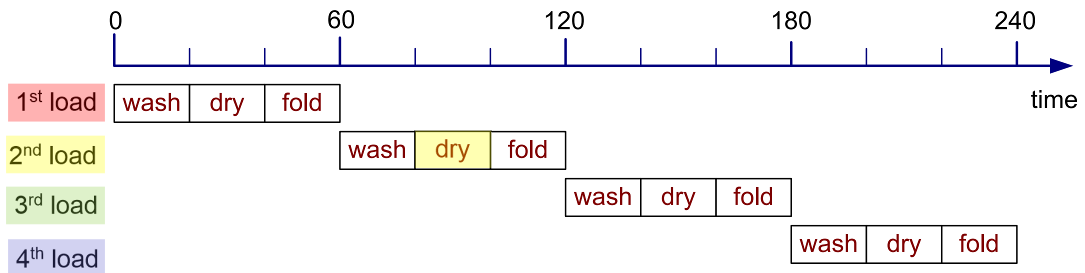


pipelined sequence

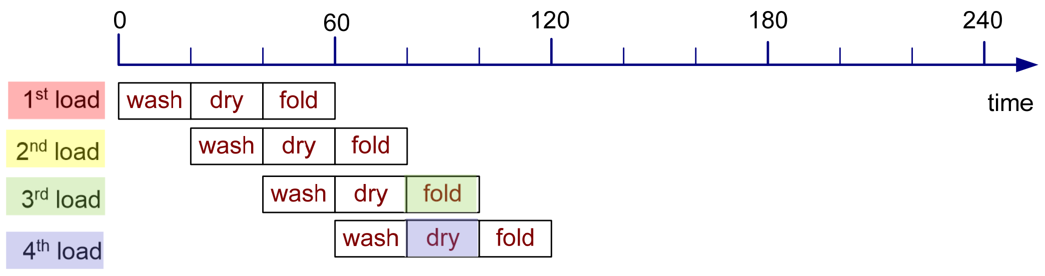
0100



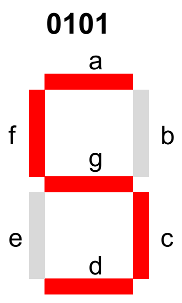
An Example of non-pipelined laundry and pipelined laundry



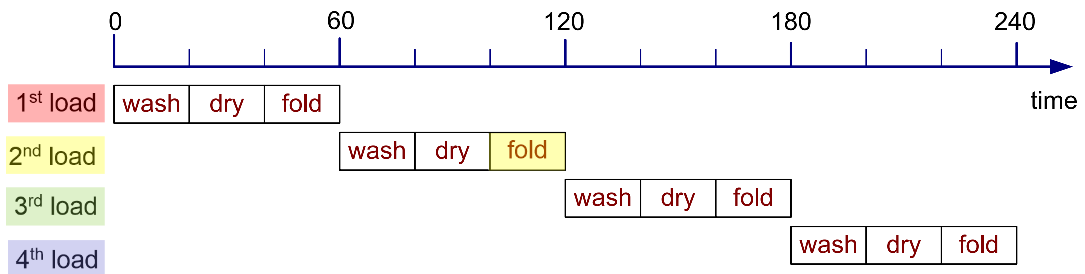
Non-pipelined sequence



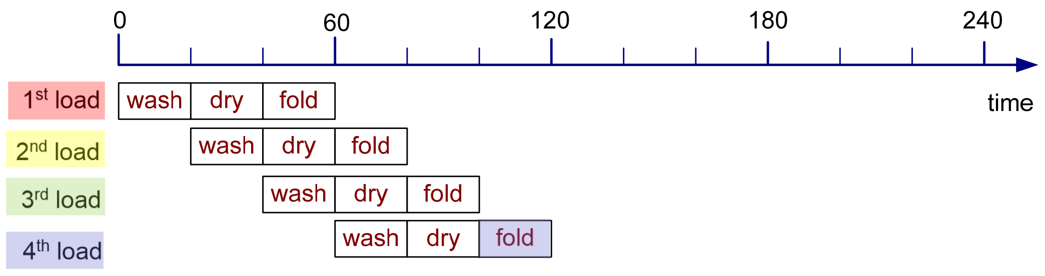
pipelined sequence



An Example of non-pipelined laundry and pipelined laundry



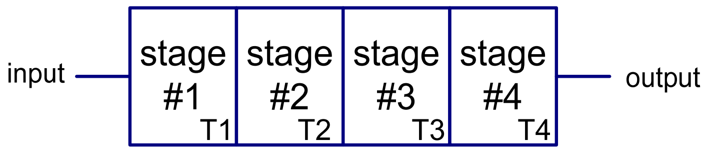
Non-pipelined sequence



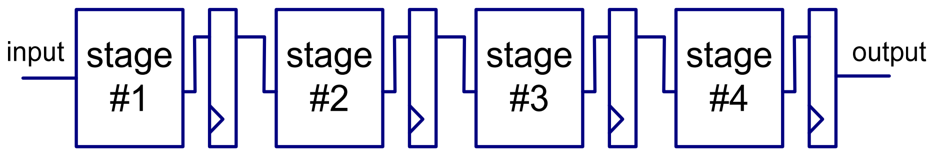
pipelined sequence

Non-pipelined: Delay: $T_{comb} = T1+T2+T3+T4$

Throughput: $1/\ T_{comb}$



Original combinational circuit



Pipelined circuit

Assume $T_{max} = \max(T1, T2, T3, T4)$; $T_c = T_{max} + T_r$;

Pipelined: Ideally, for an N-stage circuit, $T_{max} = T_{comb}/N$, and $T_r = 0$;

Delay: $T_{pipe} = NT_c = T_{max} = T_{comb}$

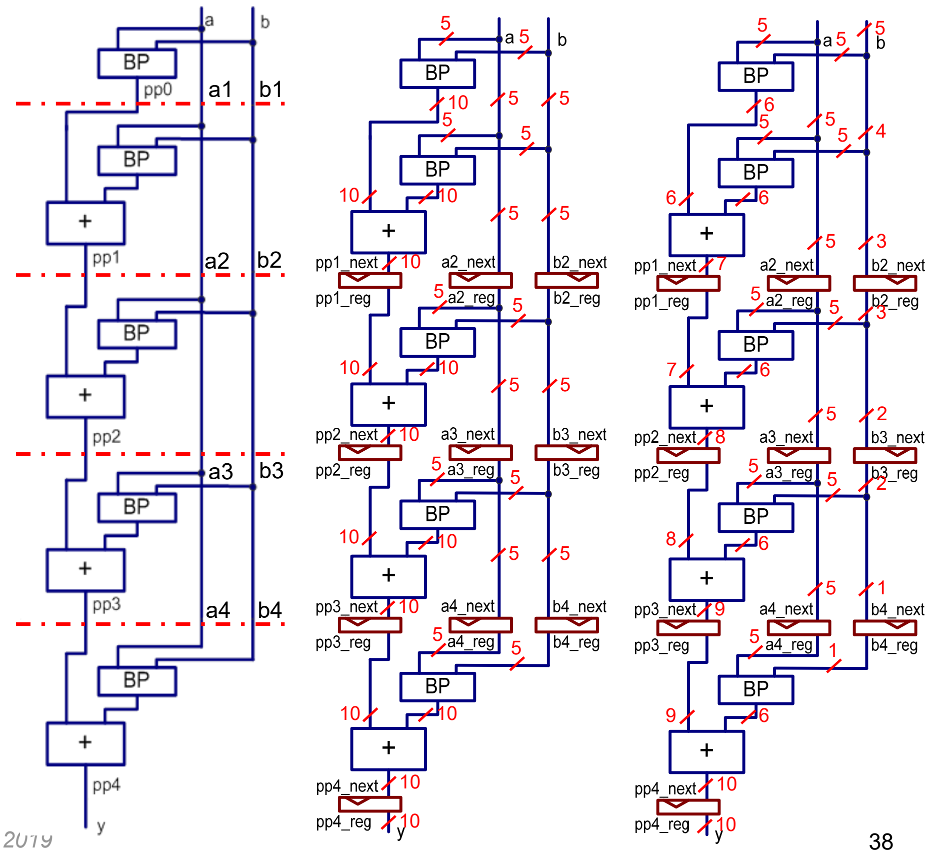
Throughput: $1/T_c = 1/T_{max} = N/T_{comb}$

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.

Example: Simple pipelined adder-based multiplier

						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



- Function Declaration: (in the declaration part of a package)

```
    function rising_edge (clock: std_logic) return boolean;
```

- Function definition (in the declaration part of an architecture or the body of a package)

```
    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;
```

- Call function (in an architecture)

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

An example of a package declaration and its 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

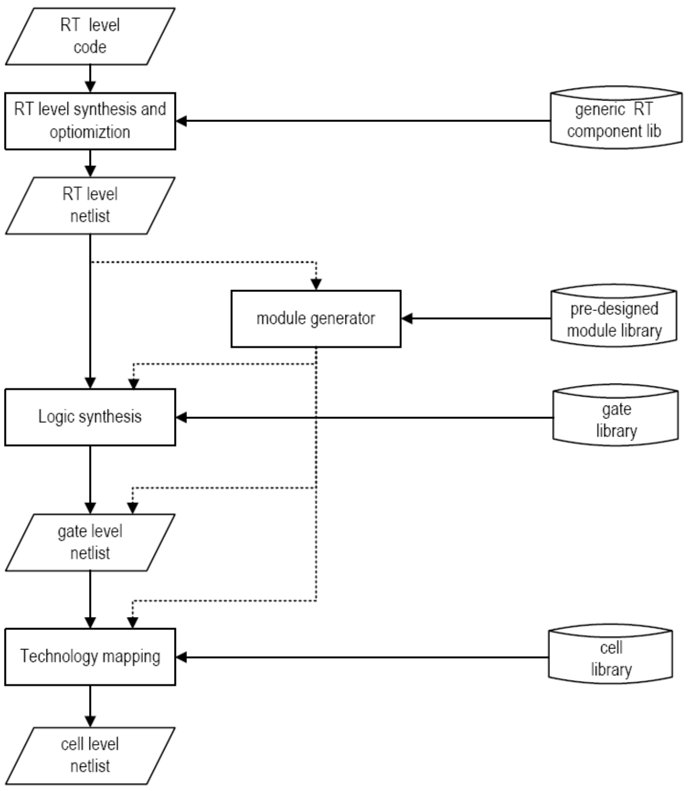
Computation complexity

- How fast an algorithm can run (or how good an algorithm is)? – time complexity
  - “Interferences” in measuring execution time:
    types of CPU, speed of CPU, compiler etc.
- Described by Big-*O* notation
  - The complexity is in the order of functions such as $1, \log_2 n, n, n\log_2 n, n^2, n^3, 2^n, etc.$
  - Algorithm with $O(2^n)$ is an intractable problem.
  - Frequently tractable algorithm for sub-optimal solution exists
- Many problem encountered in synthesis is intractable
  - Good VHDL code provides a good starting point for the local search

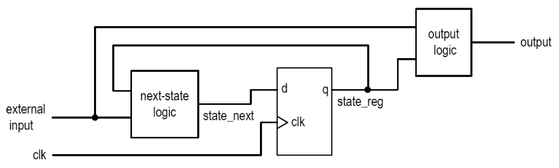
Synthesis Guidelines

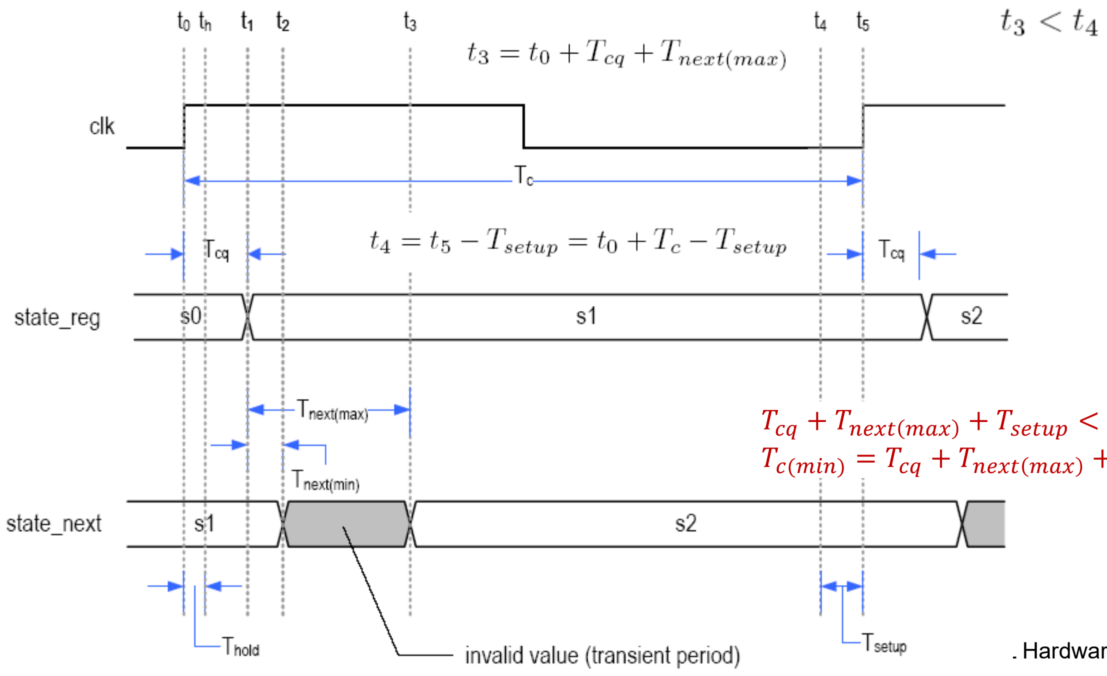
- Be aware of the theoretical limitation of synthesis software
- Be aware of hardware complexity of different VHDL operators
- Isolate tri-state buffers from other logic and code them in a separate segment
- Unless there is a compelling reason, use a multiplexer instead of an internal tri-state bus
- Avoid using the ‘-’ value of the std_logic data type as an input value

- In RT-level description, there is no effective way to eliminate glitches from a combinational circuit. We should deal with the glitches rather than attempting to derive a glitch-free combinational circuit.
- Do not use delay-sensitive design in RT-level description.



- Setup time violation and maximal clock rate





$t_3 < t_4$

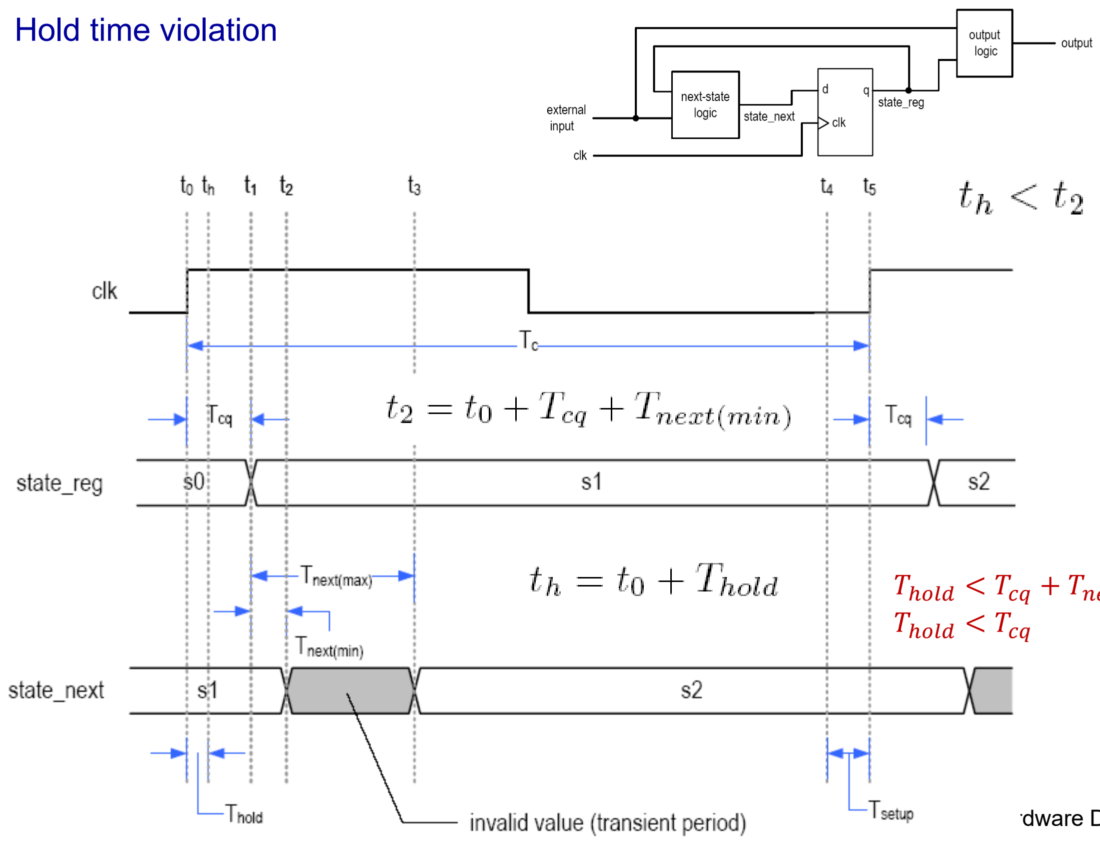
$t_3 = t_0 + T_{cq} + T_{next(max)}$

$t_4 = t_5 - T_{setup} = t_0 + T_c - T_{setup}$

$T_{cq} + T_{next(max)} + T_{setup} < T_c$

$T_{c(min)} = T_{cq} + T_{next(max)} + T_{setup}$

- Hold time violation



$$t_h < t_2$$

$$t_2 = t_0 + T_{cq} + T_{next(min)}$$

$$t_h = t_0 + T_{hold}$$

$$T_{hold} < T_{cq} + T_{next(min)}$$

$$T_{hold} < T_{cq}$$

Poor design practice and remedy

- **Misuse of asynchronous reset**
  - Glitches in output
  - Glitches in asynchronous reset signal
  - Difficulties in timing analysis
  - **load “0000” synchronously**
- **Misuse of gated clock**
  - Gated clock width can be narrow
  - Gated clock may pass glitches of ‘en’
  - Difficult to design the clock distribution network
  - **use a synchronous enable**
- **Misuse of derived clock**
  - Multiple clock distribution network
  - Difficulties in timing analysis
  - **use a synchronous one-clock enable pulse**

Course Learning Outcome

1. I have an ability to use the hardware description language VHDL and the development environment Vivado to document, to simulate and to synthesize digital systems.
2. I have an ability to design a digital circuit to realize specified combinational and squential functions.
3. I have an ability to evaluate and tradeoff the performance and cost of the designs.
4. I have an ability to design and implement digital system to solve reasonably complex practical problems.
5. I have an ability to communicate effectively with peer students and professors.
6. I have an ability to function effectively on a team to collaboratively establish goals, plan tasks, and meet objectives.