CE233(CAD), Lecture 12:

Synthesis

Mehdi Modarressi

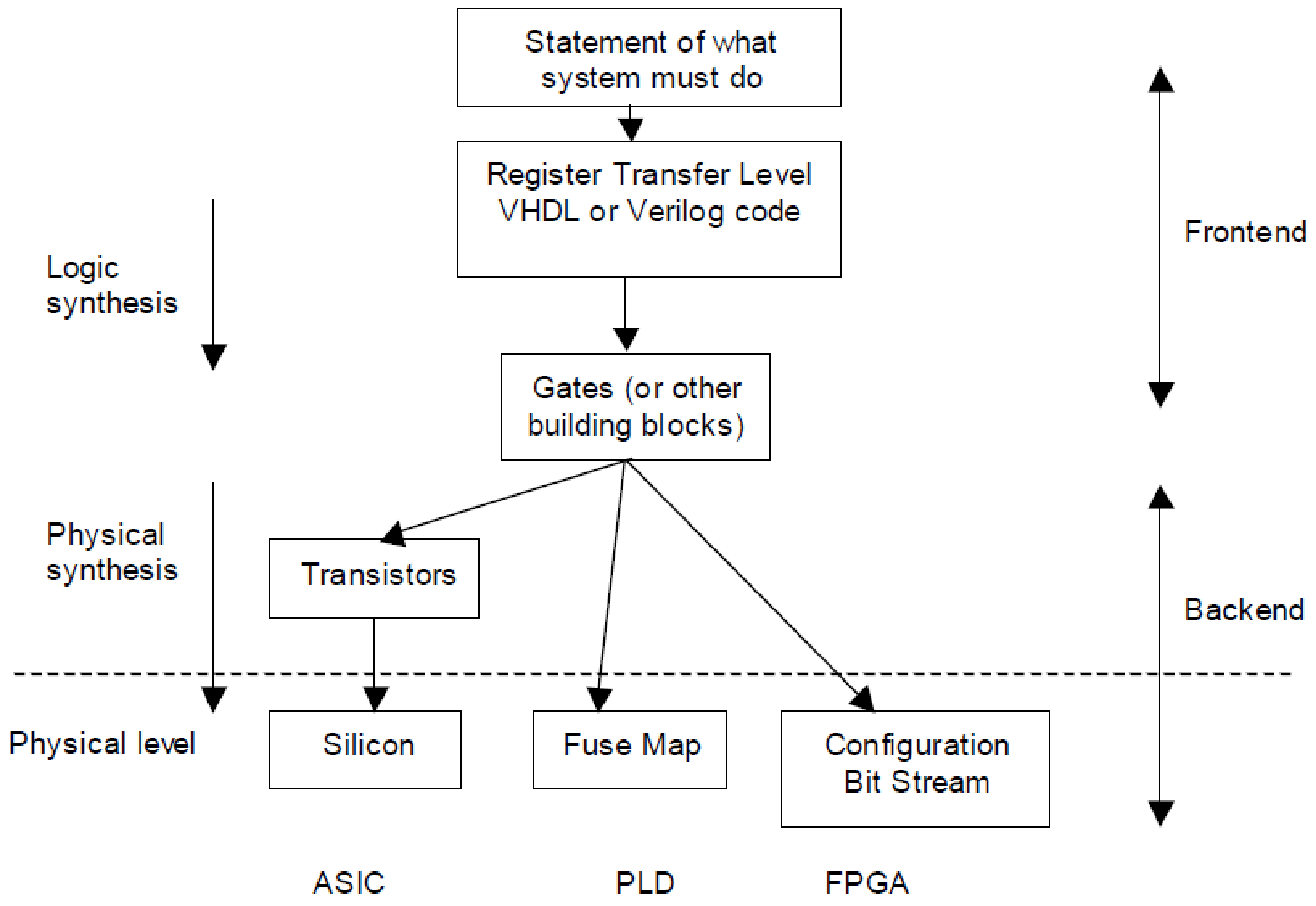
Department of Electrical and Computer Engineering,

University of Tehran

Reference

- See XST User Guide for a comprehensive list of synthesis guidelines for Xilinx FPGAs
- In particular study “XST HDL coding techniques” section

Digital systems design flow



source: EE3A1, *mishima.uwcs.co.uk/*

Digital systems design flow

- Once a hardware is described by VHDL, it is converted to a netlist (i.e. an interconnection) of basic building blocks (e.g. logic gates) by a synthesis tool
- Netlists are normally expressed in *EDIF* (Electronic Design Interchange Format)
  - An industry standard language
  - Used to easily port designs from one CAD tool to another
- See http://www.iue.tuwien.ac.at/phd/minixhofer/node53.html for a good reference for the EDIF format
- EDIF file format will be introduced more in the Lab. sessions

Implementation

- After synthesis, you run design implementation, which comprises the following steps:
  - **Translate -** merges the incoming netlists and constraints into a Xilinx design file.
  - **Map -** fits the design into the available resources on the target device based on the design and constraints
  - **Place and Route -** places and routes the design to the timing constraints.
  - **Generate Programming File -** creates a bitstream file that can be downloaded to the device.

Implementation Steps of Xilinx

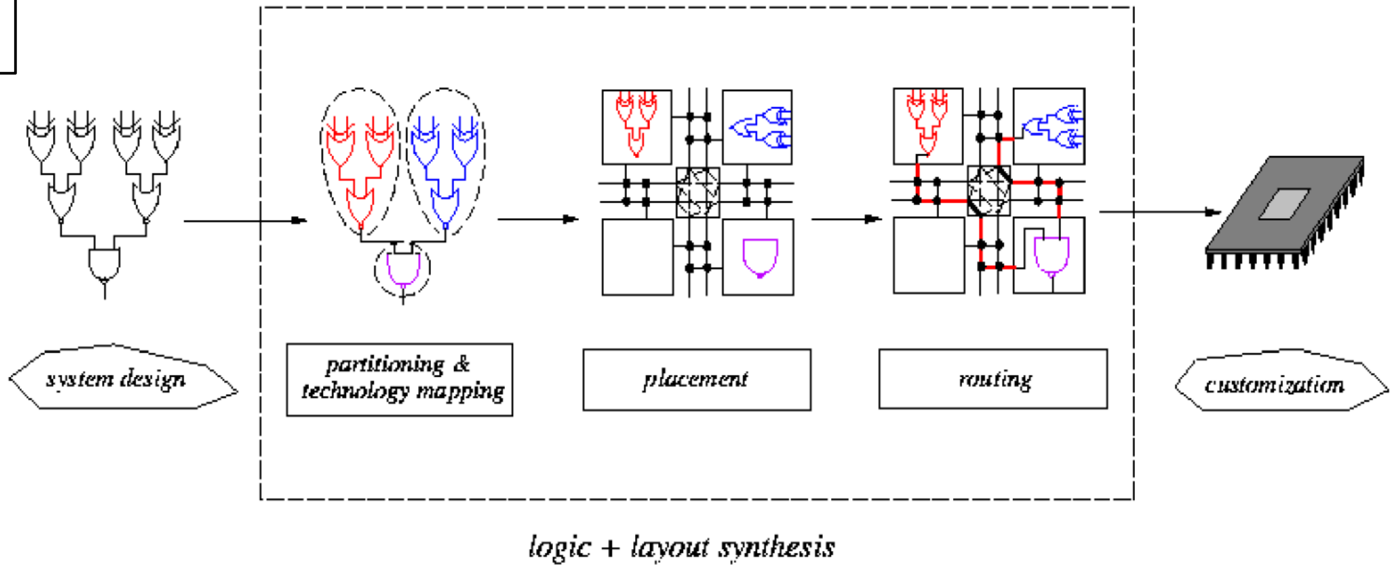
- **Synthesis:** converting a design written in a HDL into a netlist (EDIF)
- **Netlist Translation** (NGDBUILD): takes user constraint file (UCF) and EDIF and translates them into Xilinx Native Generic Database (NGD) file that contains user constraints and FPGA parts
- **MAP** (MAP tool): mapping of a design into Xilinx FPGA components (outputs NCD file)
- **Place&Route** (PAR tool): PAR outputs an NCD file that contains complete place and route information
- Bitstream generation

Logic Synthesis

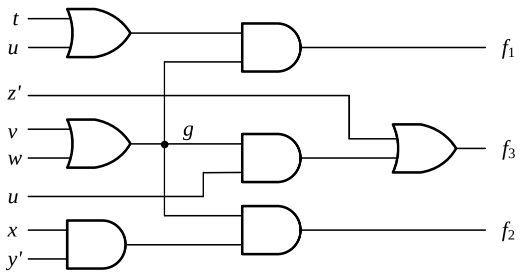
```
module x (a,b,c)
   ….
  Always@ (clk)
  ….
assign a=b &c;
……..
endmodule;
```

Logic synthesis

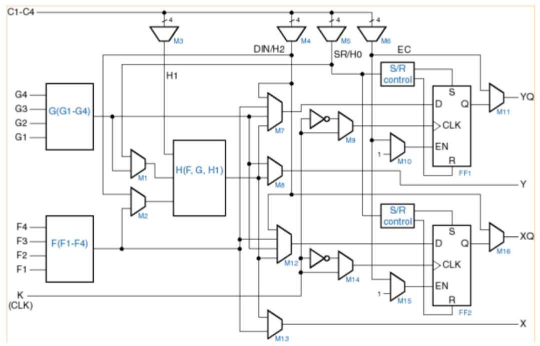
In this lecture, we focus on logic synthesis



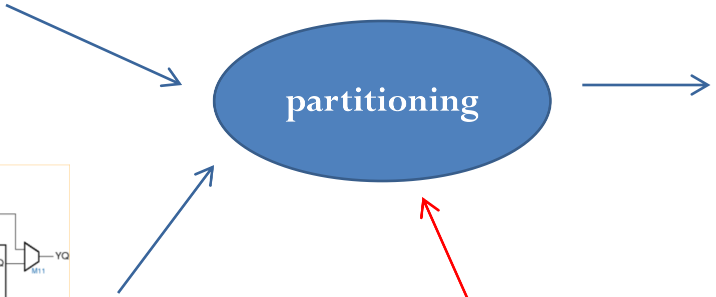
Logic Synthesis - Partitioning



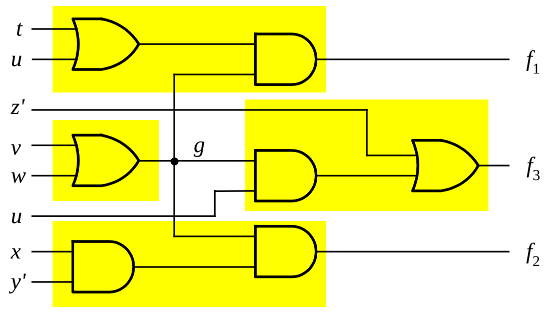
Gate-level description

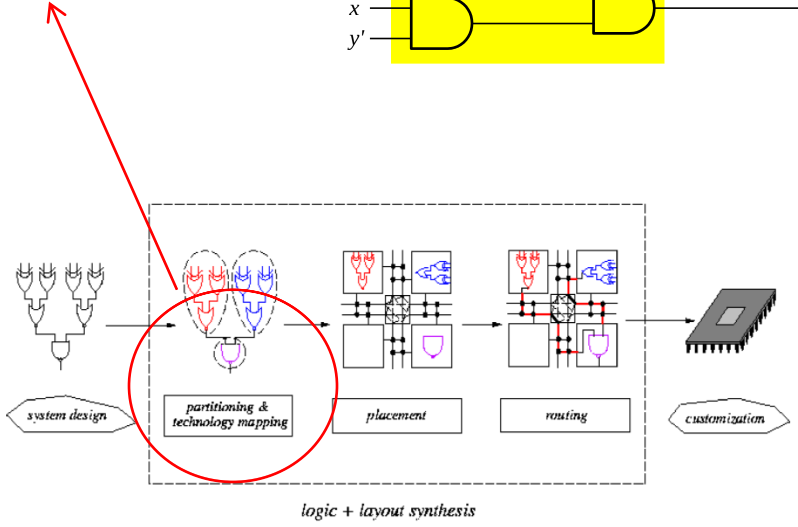


Technology Library (Target FPGA)
(just 2-input and 3-input LUTs)



Partitioned circuit

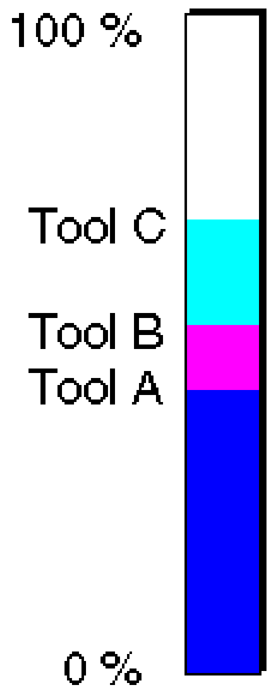




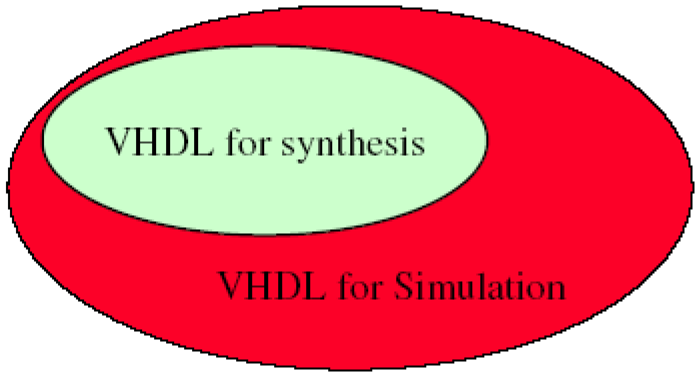
Logic synthesis

- Behavioral coding used to be used for rapid prototyping
  - Now, we can get hardware out of it by synthesis tools
- Assembly coding vs. C/Java programming
- Gate-level modeling vs. behavioral modeling

Synthesizability



- The VHDL subset that is synthesizable is tool specific
- Do not expect that your VHDL description is synthesizable with another tool!



Un-Synthesizable VHDL Subset

- Not supported: Ignored
  - No error, but ignored by the synthesis tool
  - Examples:
    - Initialization of variables or signals
    - Assert statements
    - Physical Type: Time
    - The synthesizer ignores the AFTER clause in expressions
      - Example: x <= y AFTER 100ns
- Not Supported: Illegal
  - Data types
    - Files, generic that are not integers

Synthesis

- General rules
  - General rules that determine how each piece of code is converted to hardware
- Hardware inferring
  - Specific ways of describing hardware in order to get a required component
  - Codes may be synthesized to unwanted hardware if the rules are not followed

Synthesis- General rules

- How to make simple hardware from VHDL structures
  - Operators
  - Signals and variables
  - Concurrent structures
  - Process statements
  - ….

VHDL -Hardware Correspondence

VHDL Construct	Hardware
Variables and Signals	Flip-flops, latches, wires
Arithmetic operators (+,-,*)	Adder, ALU, multiplier..
Logic operators	Gates
Relational operators	Comparator, ALU
Control Constructs (for,if-then, case,…..)	Decoders, Multiplexers, priority encoder, ..
Hierarchy Description	Hierarchical Hardware
Resolution Functions	Tri-state logic, wire-and, wire-or

Synthesis- Concurrent statements

- Concurrent statements are synthesized into gates and operators
- Only a subset of VHDL operators supported by synthesis tools
  - Supported operations: usually +, -, =, >, <, <>, abs, *
  - Generally, these operations are not supported for the real type, gust for integer
- The designer must write VHDL description (behavioral or structural) of all un-supported operators: / , **
  - Many tools also accept very simple division operations
  - In Xilinx, you can generate a divider core by CORE Generator
- Specifying a range of integer is important for efficient synthesis
- In case of an integer operation, all hardware will be 32-bit wide

Efficient integer implementation

For Unsigned numbers:

USE ieee.std_logic_unsigned.all
and
signals (inputs/outputs) of the type
STD_LOGIC_VECTOR

OR

USE ieee.std_logic_arith.all
and
signals (inputs/outputs) of the type
UNSIGNED

For Signed numbers:

USE ieee.std_logic_signed.all
and
signals (inputs/outputs) of the type
STD_LOGIC_VECTOR

OR

USE ieee.std_logic_arith.all
and
signals (inputs/outputs) of the type
SIGNED

std_logic_arith

- The IEEE standard arithmetic package is an important package for arithmetic and logical functions
- The *std_logic_arith d*efines two types: *SIGNED* and *UNSIGNED* unconstrained arrays of *std_logic*
- It overloads all arithmetic and relational operators of VHDL, for *SIGNED* or *UNSIGNED* and INTEGER
- With this overloading, we can use "+" for adding a signed or an unsigned *SIGNED* or *UNSIGNED* with an integer

Addition of Signed Numbers

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

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

ARCHITECTURE Behavior OF adder16 IS
        SIGNAL Sum : INTEGER;
BEGIN
        Sum<=X+Y;
        …
END Behavior ;
```

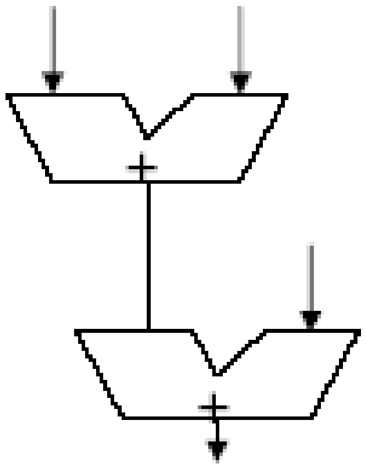
The (UN)SIGNED Package

- The use of std_logic_arith requires specification of all objects as SIGNED or UNSIGNED
  - Conversion to std_logic becomes difficult
- The std_logic_unsigned package assumes all std_logic_vector declarations are unsigned and overloads all arithmetic and relational operators for them
- The std_logic_signed package assumes all std_logic_vector declarations are signed
- If a design requires both signed and unsigned arithmetic, the *std_logic_arith* or *NUMERIC_STD* should be used

Synthesis- process

- The statements within a process are executed sequentially

```
process(a,b,c,y)
begin
          a <= b +c;
          p <= a + y;
end;
```



Process- General rule

- Sensitivity list is usually ignored during synthesis
- Equivalent behavior of simulation model and hardware
  - All signals which are read are entered into the sensitivity list
    - If SEL is missing in the sensitivity list, what will the simulation be?

```
process (A, B, SEL)
begin
  if SEL = `1` then
    Z <= A;
  else
    Z <= B;
  end if;
end process;
```

```
process (A, B, C)
begin
          D <= (A AND B) OR C;
end process;
```

```
process (A, B)
begin
          D <= (A AND B) OR C;
end process;
```

Variables

- Variables are only used inside PROCCESS statements
- Variables generate FFs, when they are read before they are assigned
- If they are assigned first and then read, they generate wires

```
signal input_foo, output_foo, clk : bit ;
...
process (clk)
      variable a, b : bit ;
begin
      if (clk'event and clk='1') then
            output_foo <= b ;
            b := a ;
            a := input_foo ;
      end if ;
end process ;
```

Shift register

```
signal input_foo, output_foo, clk : bit ;
...
process (clk)
      variable a, b : bit ;
begin
      if (clk'event and clk='1') then
            a := input_foo ;
            b := a ;
            output_foo <= b ;
      end if ;
end process ;
```

Wire

Source: Leonardo Spec synthesis manual

Hardware inferring- Common Macros

- Synthesis tool can infer some common macros from the code
  - Register
  - Latch
  - Multiplexer
  - Adder
  - Decoder
  - ….
- Some manufacture-specific directives to force synthesis tools to extract a macro
  - Example: MUX_EXTRACT constraint in Xilinx that tells the tool if it should use a MUX macro for each identified multiplexer in the VHDL code

Flip-flops

Only Clock
in sensitivity list (and all the other asynchronous Signals like RESET)

Instead of using:

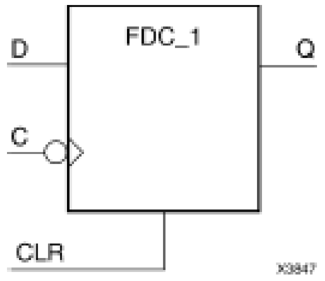
if (C'event and C='1')

We can also use:

if (rising_edge(C))

- Defined in std_logic_1164 package

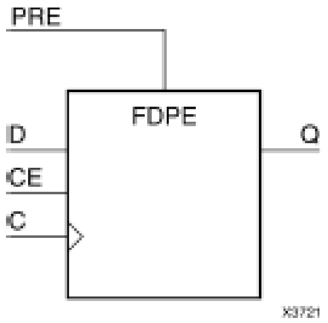
```
library ieee;
use ieee.std_logic_1164.all;

entity registers_1 is
    port(C, D : in std_logic;
         Q    : out std_logic);
end registers_1;

architecture archi of registers_1 is
begin

    process (C)
    begin
        if (C'event and C='1') then
            Q <= D;
        end if;
    end process;

end archi;
```



Register

4-Bit Register With Positive-Edge Clock, Asynchronous Set, and Clock Enable

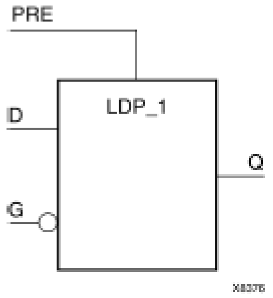
```
library ieee;
use ieee.std_logic_1164.all;

entity registers_5 is
    port(C, CE, PRE : in std_logic;
         D          : in std_logic_vector (3 downto 0);
         Q          : out std_logic_vector (3 downto 0));
end registers_5;

architecture archi of registers_5 is
begin

    process (C, PRE)
    begin
        if (PRE='1') then
            Q <= "1111";
        elsif (C'event and C='1')then
            if (CE='1') then
                Q <= D;
            end if;
        end if;
    end process;

end archi;
```



Latch

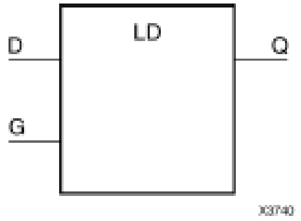
4-Bit Latch With Inverted Gate and Asynchronous Set

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

entity latches_3 is
    port(D      : in std_logic_vector(3 downto 0);
         G, PRE : in std_logic;
         Q      : out std_logic_vector(3 downto 0));
end latches_3;

architecture archi of latches_3 is
begin
    process (PRE, G, D)
    begin
        if (PRE='1') then
            Q <= "1111";
        elsif (G='0') then
            Q <= D;
        end if;
    end process;
end archi;
```



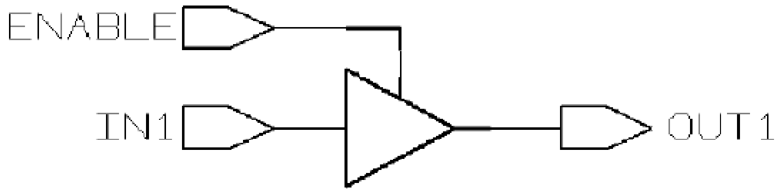
A simple latch

```
library ieee;
use ieee.std_logic_1164.all;
entity latches_1 is
 port(G, D : in std_logic;
  Q : out std_logic);
end latches_1;

architecture archi of latches_1 is
begin
process (G, D)
begin
 if (G='1') then
  Q <= D;
  end if;
 end process;
end archi;
```



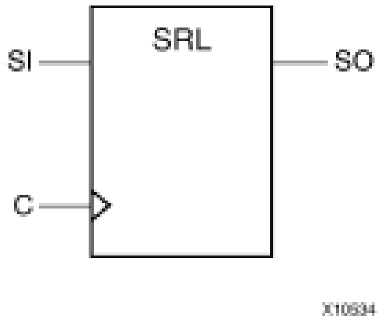
Tri-state gate

```
entity three_state is
   port(IN1, ENABLE : in std_logic;
        OUT1 : out std_logic );
   end;
architecture rtl of three_state is
begin
   process (IN1, ENABLE) begin
     if (ENABLE = '1') then
       OUT1 <= IN1;
     else
       OUT1 <= 'Z'; -- assigns high-impedance state
     end if;
   end process;
end rtl;
```



Shift register

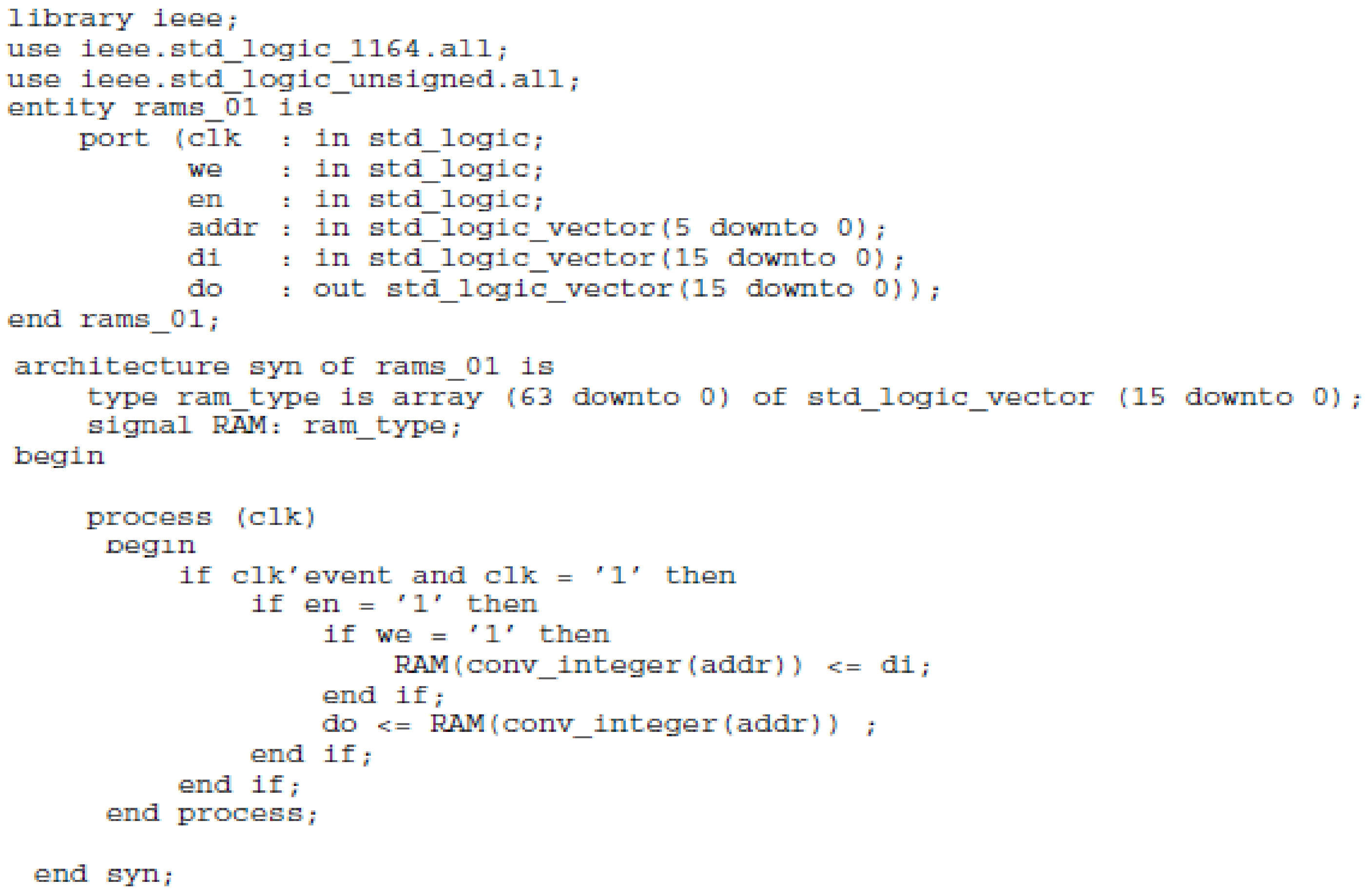
8-Bit Shift-Left Register With Positive-Edge Clock, Serial In and Serial Out



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

entity shift_registers_1 is
    port(C, SI : in std_logic;
         SO : out std_logic);
end shift_registers_1;

architecture archi of shift_registers_1 is
    signal tmp: std_logic_vector(7 downto 0);
begin

    process (C)
    begin
        if (C'event and C='1') then
            for i in 0 to 6 loop
                tmp(i+1) <= tmp(i);
            end loop;
            tmp(0) <= SI;
        end if;
    end process;

    SO <= tmp(7);

end archi;
```

RAM

- If a given template can be implemented using Block and Distributed RAM, Xilinx tools implement BLOCK ones



```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity rams_01 is
    port (clk  : in std_logic;
          we   : in std_logic;
          en   : in std_logic;
          addr : in std_logic_vector(5 downto 0);
          di   : in std_logic_vector(15 downto 0);
          do   : out std_logic_vector(15 downto 0));
end rams_01;

architecture syn of rams_01 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM: ram_type;
begin

    process (clk)
     begin
        if clk'event and clk = '1' then
            if en = '1' then
                if we = '1' then
                    RAM(conv_integer(addr)) <= di;
                end if;
                do <= RAM(conv_integer(addr)) ;
            end if;
        end if;
     end process;

 end syn;
```

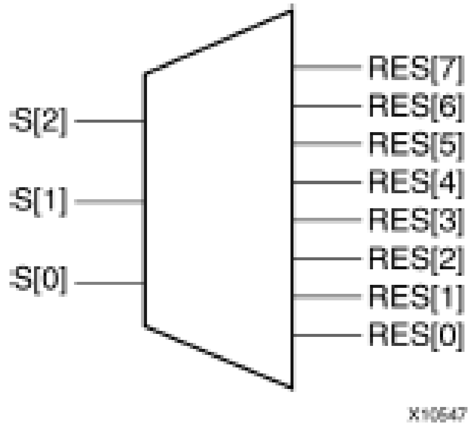
ROM

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

entity rams_21a is
    port (clk  : in std_logic;
          en   : in std_logic;
          addr : in std_logic_vector(5 downto 0);
          data : out std_logic_vector(19 downto 0));
end rams_21a;

architecture syn of rams_21a is

    type rom_type is array (63 downto 0) of std_logic_vector (19 downto 0);
    signal ROM : rom_type:= (X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A",
                             X"00300", X"08602", X"02310", X"0203B", X"08300", X"04002",
                             X"08201", X"00500", X"04001", X"02500", X"00340", X"00241",
                             X"04002", X"08300", X"08201", X"00500", X"08101", X"00602",
                             X"04003", X"0241E", X"00301", X"00102", X"02122", X"02021",
                             X"00301", X"00102", X"02222", X"04001", X"00342", X"0232B",
                             X"00900", X"00302", X"00102", X"04002", X"00900", X"08201",
                             X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
                             X"00102", X"02137", X"02036", X"00301", X"00102", X"02237",
                             X"04004", X"00304", X"04040", X"02500", X"02500", X"02500",
                             X"0030D", X"02341", X"08201", X"0400D");

begin

    process (clk)
    begin
        if (clk'event and clk = '1') then
            if (en = '1') then
                data <= ROM(conv_integer(addr));
            end if;
        end if;
    end process;

end syn;
```

- Xilinx tools use block RAM resources to implement ROMs

Decoder

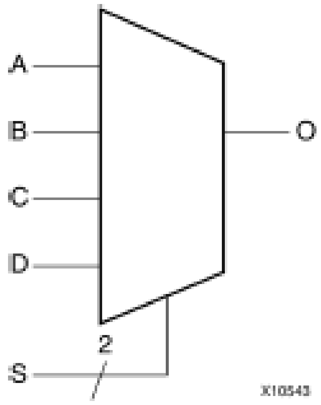
- 1-of-8 Decoder (One-Hot) VHDL Coding Example
- No Decoder Inference if there are unused decoder outputs



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

entity decoders_1 is
    port (sel: in std_logic_vector (2 downto 0);
          res: out std_logic_vector (7 downto 0));
    end decoders_1;

architecture archi of decoders_1 is
begin
    res <= "00000001" when sel = "000" else
           "00000010" when sel = "001" else
           "00000100" when sel = "010" else
           "00001000" when sel = "011" else
           "00010000" when sel = "100" else
           "00100000" when sel = "101" else
           "01000000" when sel = "110" else
           "10000000";
end archi;
```

Multiplexer

- 4-to-1 1-Bit MUX Using Case Statement

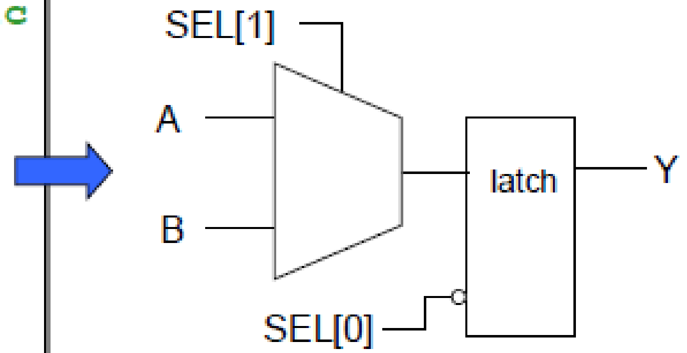


```
library ieee;
use ieee.std_logic_1164.all;

entity multiplexers_2 is
    port (a, b, c, d : in std_logic;
          s : in std_logic_vector (1 downto 0);
          o : out std_logic);
end multiplexers_2;

architecture archi of multiplexers_2 is
begin
    process (a, b, c, d, s)
    begin
        case s is
            when "00" => o <= a;
            when "01" => o <= b;
            when "10" => o <= c;
            when others => o <= d;
        end case;
    end process;
end archi;
```

Latch in *Case* statements

- *CASE* statement should be mutually exclusive and exhaustive in order for synthesis tools to efficiently derive a single MUX equivalent circuit
- Unless an unwanted latch will be synthesized

```
-- sel, A, B are std_logic
process (SEL, A, B)
begin
case SEL is
  when "00" => Y <= A;
  when "10" => Y <= B;
  when others => null;
end case;
end process;
```

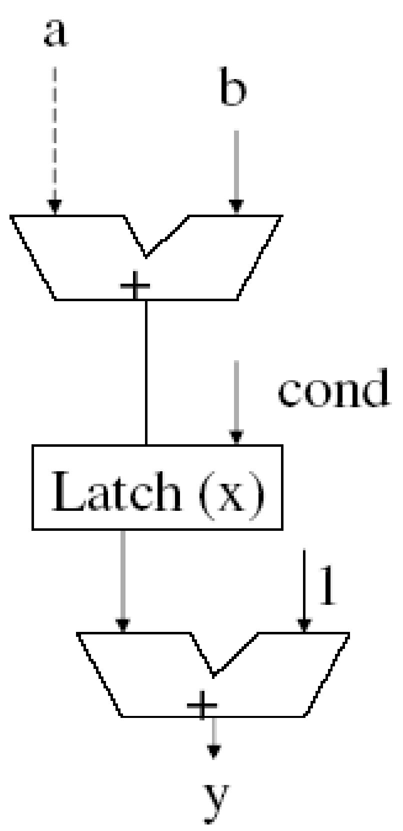


To avoid unwanted latches, actually define Y for the "others" condition, for example:

```
when others => Y <= '0';
```

Latch in *If* statements

```
process(a,b)
   variable x,y,z;
begin
   ...
   if cond = '1' then
      x := a + b;
   end if;
   y := x + 1;
   ...
end process;
```



Latch elimination in *If* statements

- Two ways to remove latches:
  - Using default value
  - Covering all cases by *ELSE*

Using ELSE

```
IF  A = B  THEN
        AeqB <= '1' ;
ELSE
        AeqB <= '0' ;
```

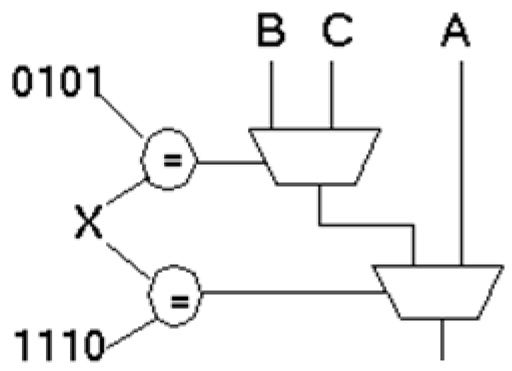
Using default values

```
AeqB <= '0' ;
IF A = B THEN
        AeqB <= '1' ;
```

If statement

```
Library IEEE;
use IEEE.Std_Logic_1164.all;
entity IF_EXAMPLE is
port (A, B, C, X : in std_ulogic_vector(3 downto 0);
        Z              : out std_ulogic_vector(3 downto 0));
end IF_EXAMPLE;

architecture A of IF_EXAMPLE is
begin
    process (A, B, C, X)
    begin
      if ( X = "1110" ) then
        Z <= A;
      elsif (X = "0101") then
        Z <= B;
      else
        Z <= C;
      end if;
    end process;
end A;
```

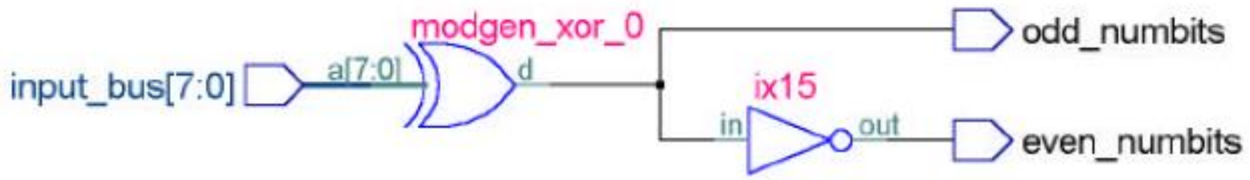
Hardware realisation



For statement

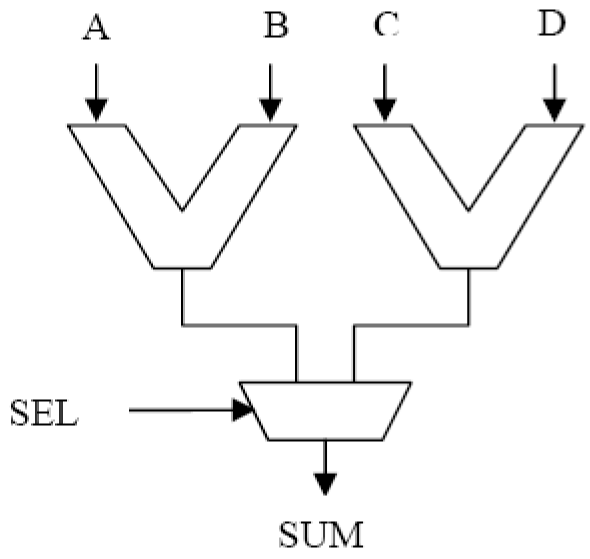
```
-- lokale Variablen :- für "Loop-Enrolling"
library ieee;

use ieee.std_logic_1164.all;
entity parity is
        generic (bus_size : integer := 8 );
        port (input_bus  : in std_logic_vector (bus_size-1 downto 0);
           even_numbits, odd_numbits : out std_logic ) ;
end parity ;

architecture behave of parity is
begin
 process (input_bus)
        variable temp: std_logic;
 begin
        temp := '0';
        for i in input_bus'low to input_bus'high loop
         temp := temp xor input_bus( i ) ;
        end loop ;
        odd_numbits <= temp ;
        even_numbits <= not temp;
 end process;
end behave;
```

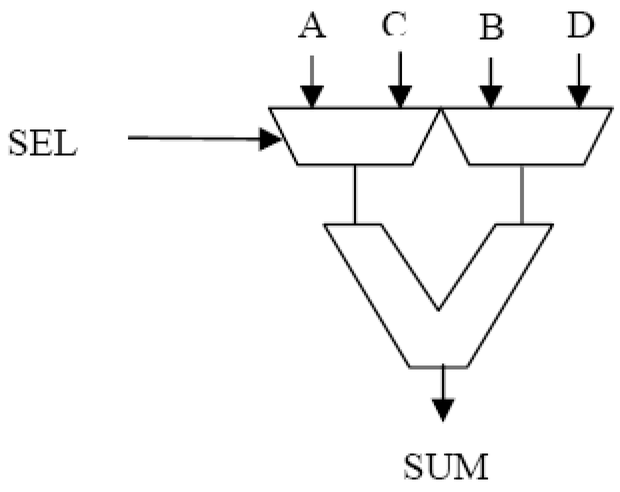
- Will be synthesized by replicating the corresponding hardware
  - By unrolling the loop
- For synthesis, the range specified for the loop variable must be a compile-time constant, otherwise the construct is not often synthesizable
- Synthesis in parallel or serial:
  - Example: parity generation



Resource sharing

```
if sel = '1' then
  sum := a +b;
else
  sum := c+d;
end if
```





Resource sharing: Sharing an operator under mutually exclusive conditions

Resource sharing- another example

```
signal a,b,c,d : integer range 0 to 255 ;
...
process (a,b,c,d) begin
   if ( a+b = c ) then     <statements>
   elsif ( a+b = d) then <more_statements>
   end if ;
end process ;
```

Requires two adders

```
process  (a,b.c.d)
         variable tmp : integer range 0 to 255 ;
begin
         tmp := a+b ;
         if ( tmp = c ) then  <statements>
         elsif ( tmp = d) then <more_statements>
         end if ;
end process ;
```

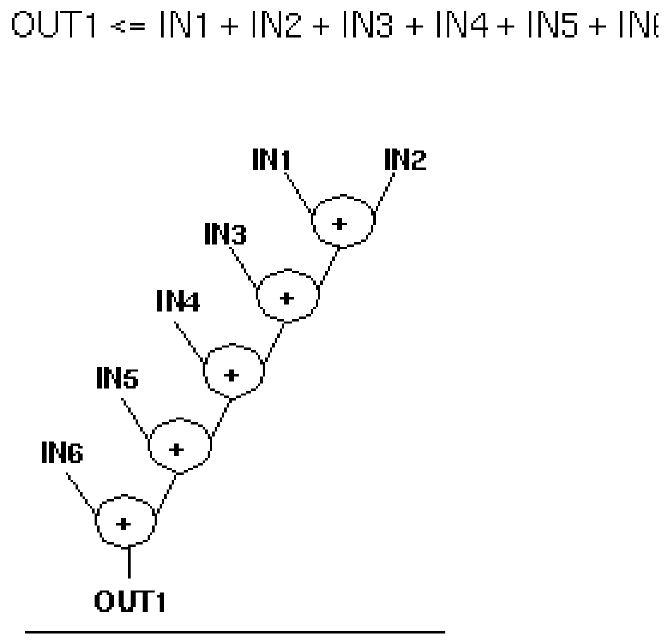
Requires one adder

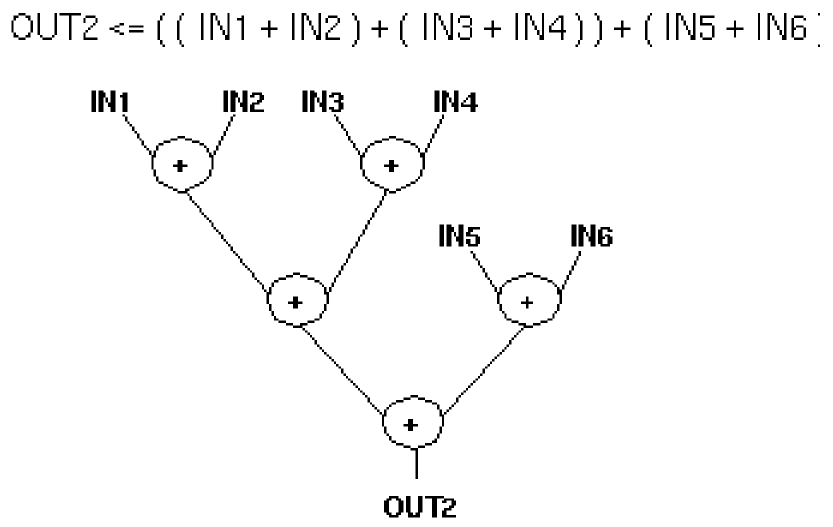
Resource sharing

- Synthesis tools often reduce resources by resource sharing automatically, often provided that:
  - Expressions drive the same signal
  - Operators are of the same type (e.g., both are adders) and have the same width

Source Code Optimization

- Use parenthesis to optimize the speed
- Often done automatically by synthesis tools


OUT1 <= IN1 + IN2 + IN3 + IN4 + IN5 + IN


OUT2 <= ((IN1 + IN2) + (IN3 + IN4)) + (IN5 + IN6

Source Code Optimization

```
process begin
   wait until clk'event and clk='1' ;
      if (count = input_signal) then
         count <= 0 ;
      else
         count <= count + 1 ;
      end if ;
end process ;
```

- Generates a counter and a full comparator

```
process begin
   wait until clk'event and clk='1' ;
      if (count = 0) then
         count <= input_signal ;
      else
         count <= count - 1 ;
      end if ;
end process ;
```

- If the specifications allows reading the input at reset, simpler hardware can be obtained
- One counter and a comparator to zero that is very simpler than a full comparator

Bus generation

- Use std_logic vectors with multiple tri-state drivers

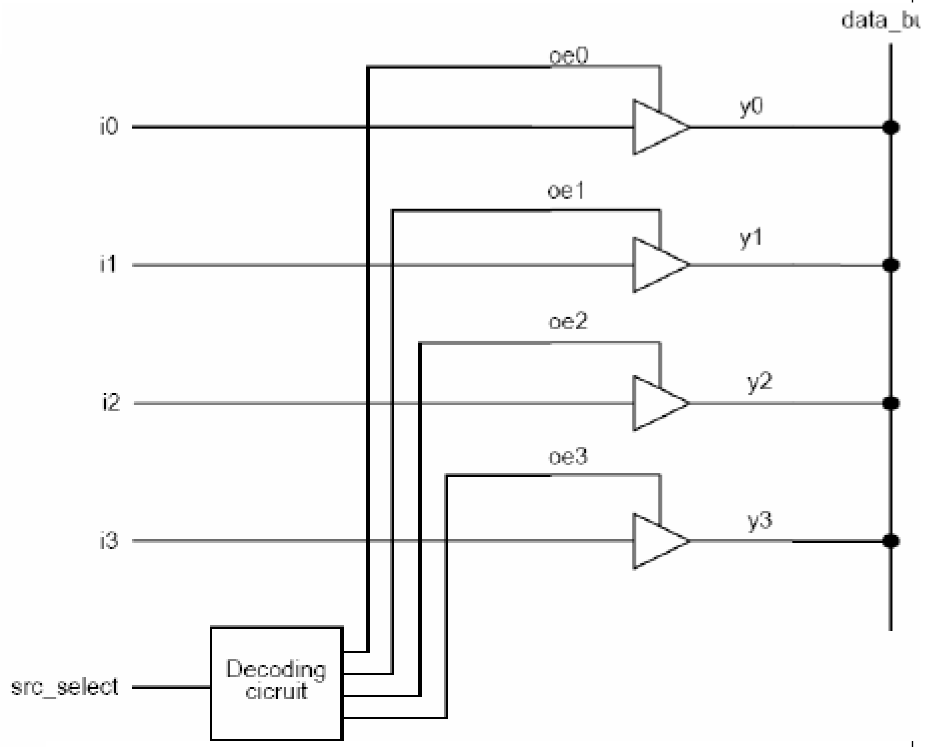
- Driver code:

```
entity three-state is
   port (   input_signal_1 : in std_logic_vector (7 downto 0);
            ena_1   : in std logic ;
            output_signal  : out std_logic_vector(7 downto 0)
            ) ;
end three-state ;

architecture rtl of three-state is
begin
   output_signal <= input_signal_1 when ena_1 = '1'
               else "ZZZZZZZZ" ;
end rtl ;
```

Tristate bus

- Code for a tri-state bus with control logic

```
-- binary decoder
with src_select select
   oe <= "0001" when "00",
         "0010" when "01",
         "0100" when "10",
         "1000" when others; -- "11
-- tri-state buffers
y0 <= i0 when oe(0)='1' else 'Z';
y1 <= i1 when oe(1)='1' else 'Z';
y2 <= i2 when oe(2)='1' else 'Z';
y3 <= i3 when oe(3)='1' else 'Z';
data_bus <= y0;
data_bus <= y1;
data_bus <= y2;
data_bus <= y3;
```

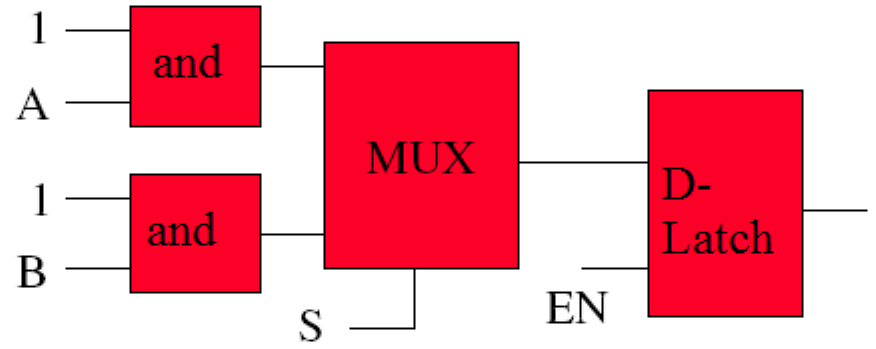


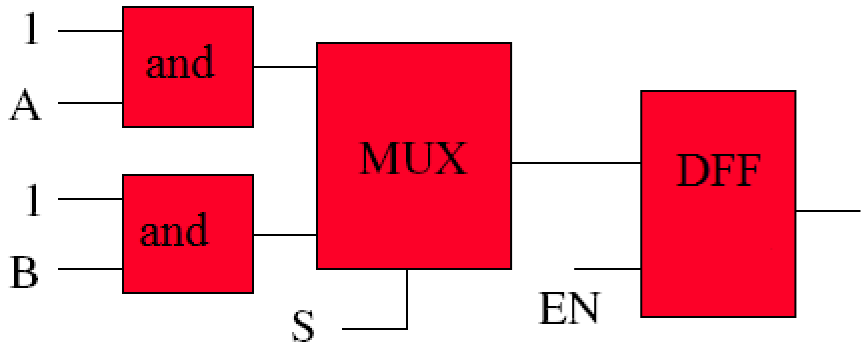
Sequential logic

- To build a sequential logic, we can divide it into combinational logic and memory elements (FF or Register)
  - The output of combinational logic is fed to a memory element and is given to the rest of system by clock
- Alternatively, we can move the entire code into a process that is sensitive to a clock signal

Exercise

Write a VHDL description that generates this implementation





Xilinx Naming Conventions

- In addition to naming rules we talked about before, Xilinx has some extra naming conventions for naming signals, variables, and instances of entities
- See in page 60 of "Xilinx - Synthesis and Simulation Design Guide"