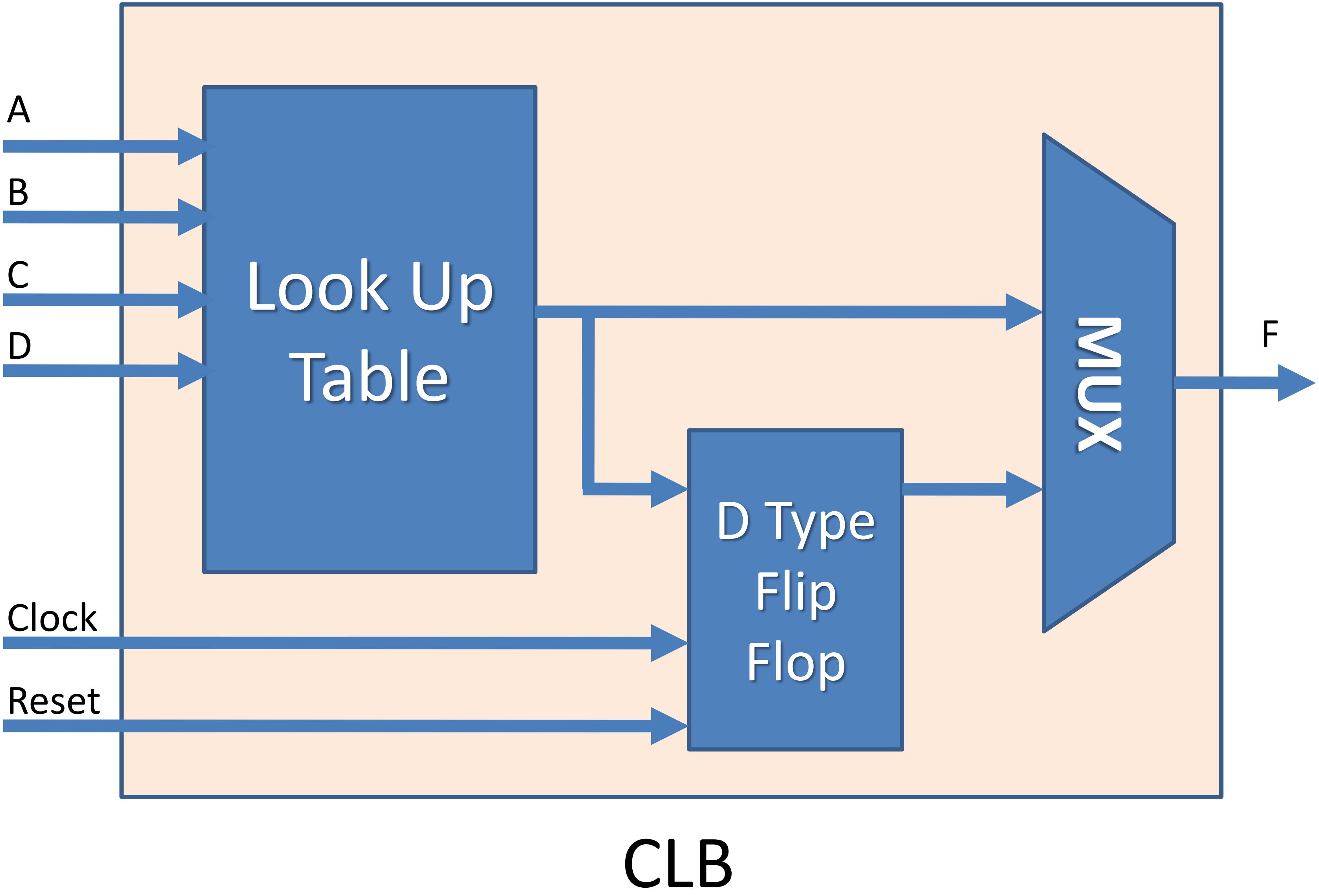
# FPGA Fundamentals

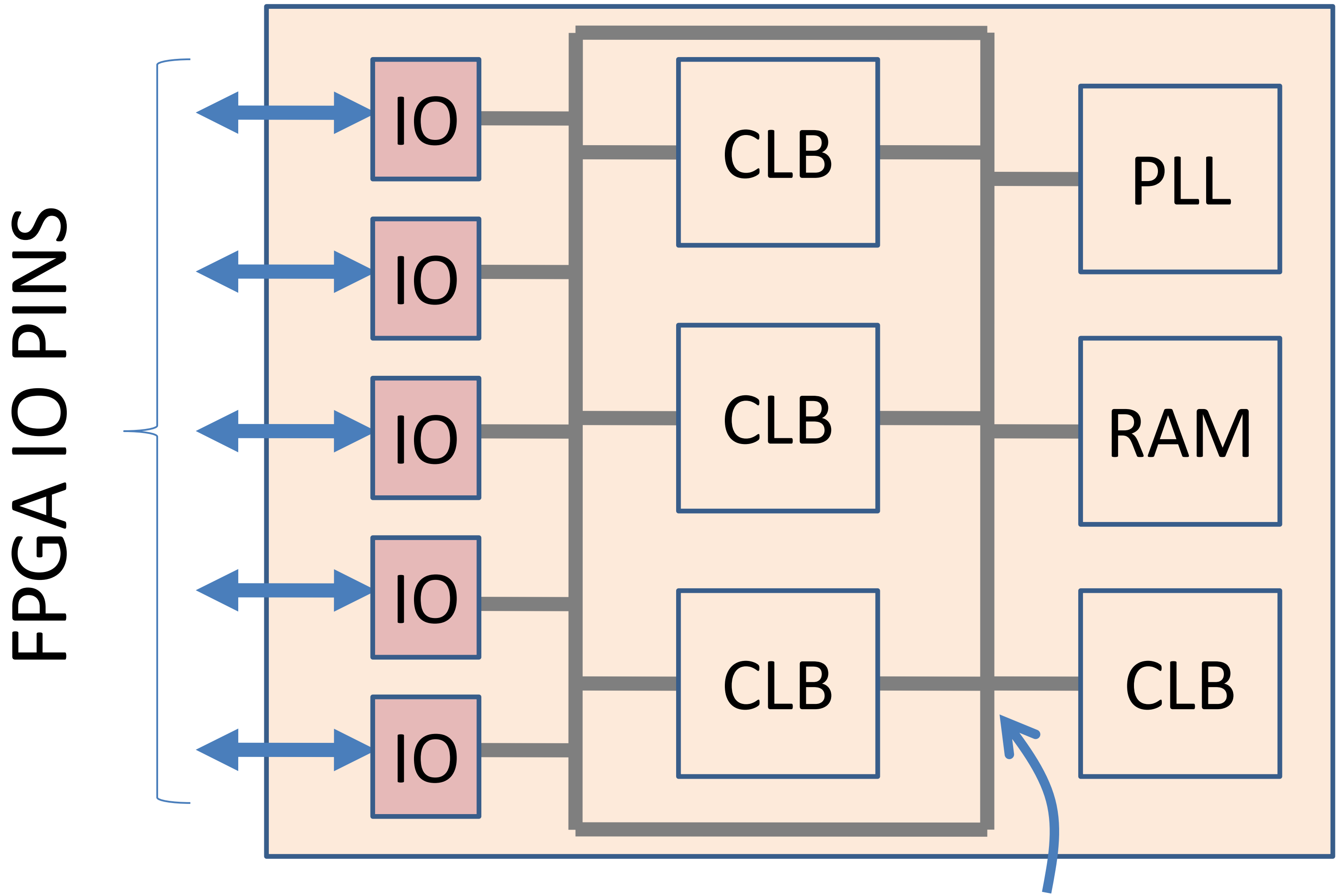
# What is an FPGA?

- Field Programmable Gate Array.
- Contains:
  - Logic gates
  - Registers
  - Configurable Routing
  - Memory (RAMs, ROMs)
  - PLLs
  - Arithmetic & DSP
  - Hard IP Blocks
- Fully programmable.

# Configurable Logic Block (CLB)



# Simplified FPGA Architecture



Programmable Routing

# Why Use An FPGA?

- Used for implementing Digital Systems.
- Extremely Flexible.
  - No limit to timers, counters, peripherals etc.
  - Does not have a fixed hardware structure.
  - Change whole design without changing layout.
- Very fast.
- Parallel processing (Doesn’t execute sequentially).
- Large bandwidth (Data width configurable).
- Intensive data processing (Video, DSP, FFT etc)

# When Not To Use An FPGA

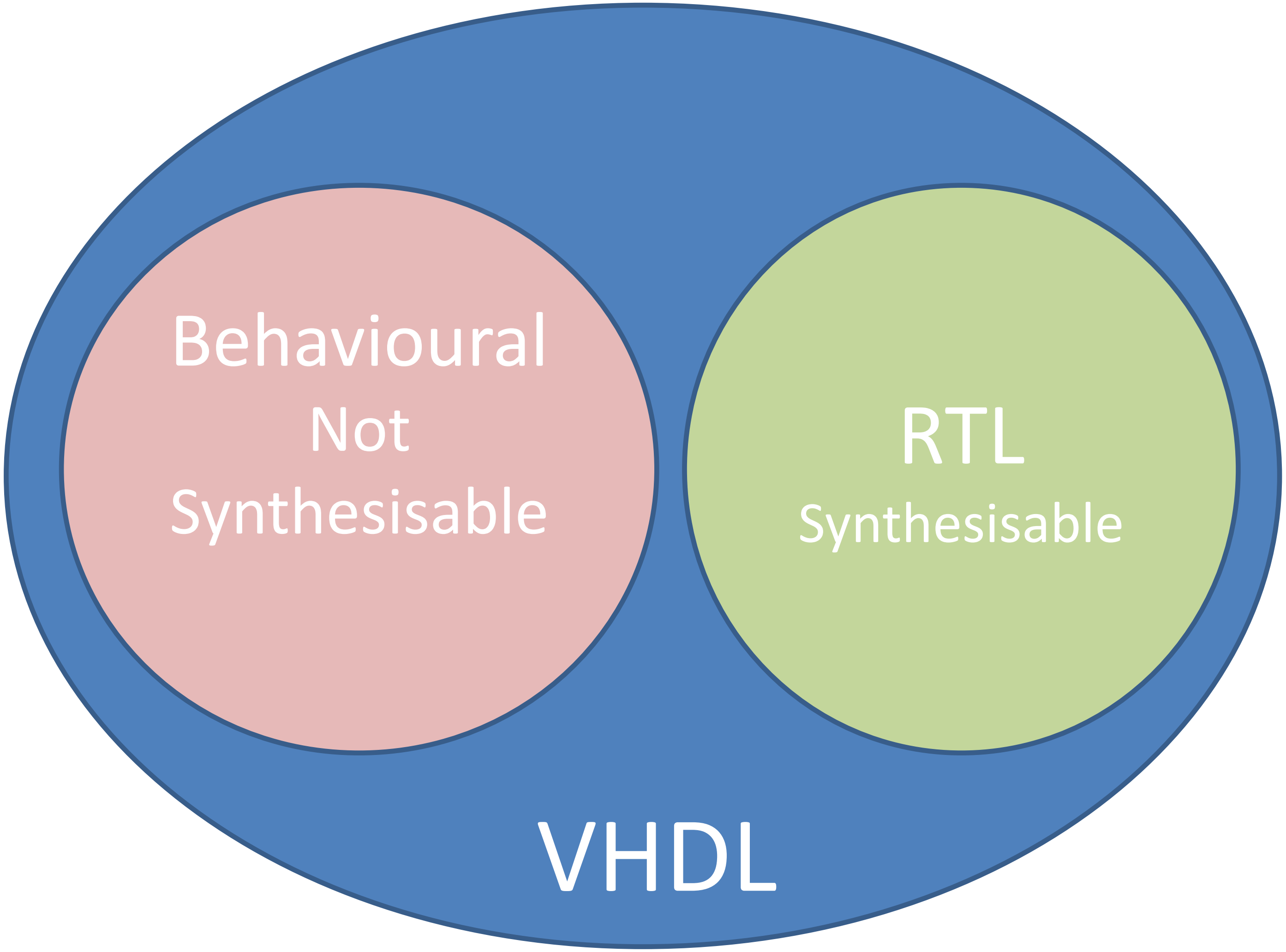
- In cost sensitive applications.
- Carrying out infrequent tasks that are sequential in nature (eg : search / sort algorithms).
- When speed and latency are not that important.

# FPGA Programming

- Hardware Descriptive Languages (HDL)
- C/C++ using HLS (High Level Synthesis)
- Popular HDL Languages :

| VHDL | Verilog |
|---|---|
| Strongly Typed | Weakly Typed |
| Deterministic | Only Deterministic If you Follow Rules Carefully. |
| Verbose | Concise |
| Non C Like Syntax | More C Like Syntax |

# Behavioural Vs RTL



# RTL (Register Transfer Level)

- Can be simulated, synthesised & implemented on an FPGA.
- Describes hardware in terms of registers and combinational logic that sits between them.
- Need to have an idea of the digital circuit.
- No explicit delays.
- All timing described in terms of clock edges.
- Used as input to a synthesis tool.

# Behavioural

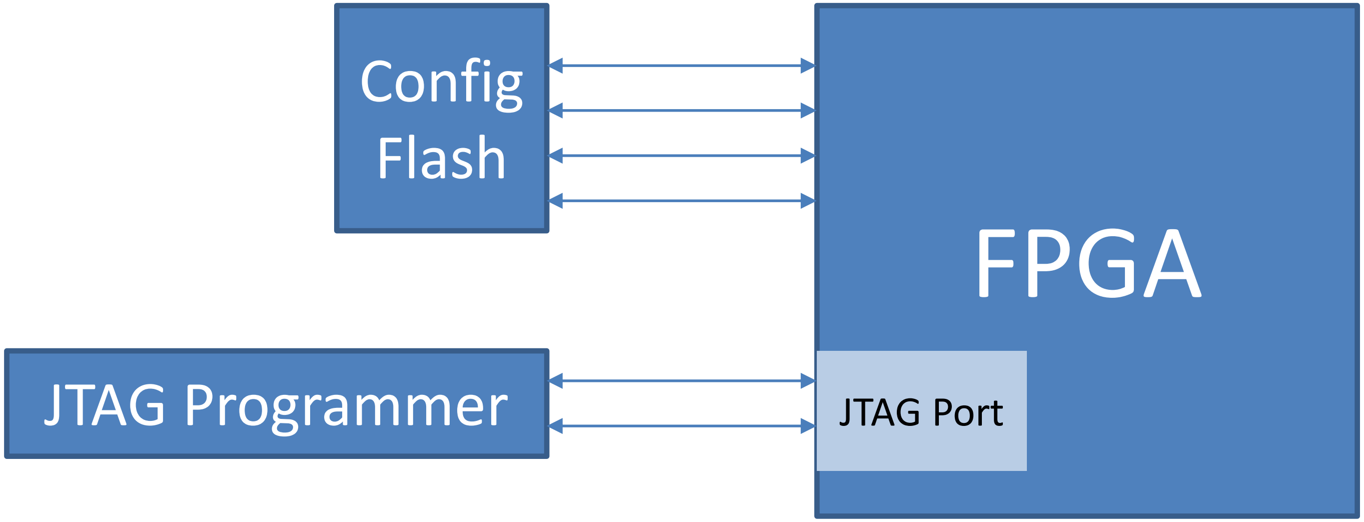
- Can only be simulated. Cannot be synthesised.
- Can have explicit delays,
- Uses an algorithm to describe functionality.
- Describes behaviour at a higher level.
- Used to build a verification model to simulate the behaviour of a system or module.
- Used for simulation purposes for example in writing test benches.

# FPGA Design Flow

- Start from the **Design Specification**
- Use VHDL Editor To :
  - Write VHDL (RTL)
  - Write **Test-Bench** (RTL + Behavioural)
- **Simulate** to ensure correct behaviour.
- Write **Timing Constraints.**
- **Synthesis** tool to generate a netlist.
- **Implementation** tool to take netlist and perform placement, routing and optimisation.
- Make sure designs meets **Timing**. If not go back to VHDL code.
- Implementation creates a **binary file** used to configure the FPGA.

# FPGA Design Flow

- Program binary file into FPGA configuration flash (using JTAG for example)



# Chapter Summary

- Configurable Logic Block (CLB)
- FPGA Architecture
- When to use an FPGA?
- What are FPGAs good at?
- When might you not use an FPGA?
- Overview of FPGA programming languages.
- Typical FPGA design flow.

# Signals & Data Types I

# Some Basics

- VHDL is NOT case sensitive.
    - OBJECT_NAME = object_name
- Reserved words – Keywords.
- Identifiers
    - Used to name objects in VHDL.
    - Cannot be a reserved word.

# Signals

**Signal Declaration Syntax :**

```
signal signal_name : data_type;
signal signal_name : data_type := initial_value;
```

**Signal Assignment Operator : <=**

**Example Signal Assignment:**

```
A <= B;
```

# Data Types – Std_logic

- Represents the value of a single data bit.
- Package - Std_Logic_1164.
- Can take the following values :
  - ‘1’ or ‘0’ – Logic high or low.
  - ‘Z’ – Tri-stated.
  - ‘H’ or ‘L’ – Weak high or low. (e.g: pull-up)
  - ‘W’ – Weak unknown.
  - ‘U’ – Un initialised.
  - ‘X’ – Unknown or cannot be determined.
  - ‘-’ – Don’t care.

# Data Types – Std_logic

- Examples:
  - Input or Output of a logic gate.
  - Input or Output of a register / flip-flops.
  - A clock or reset signal.

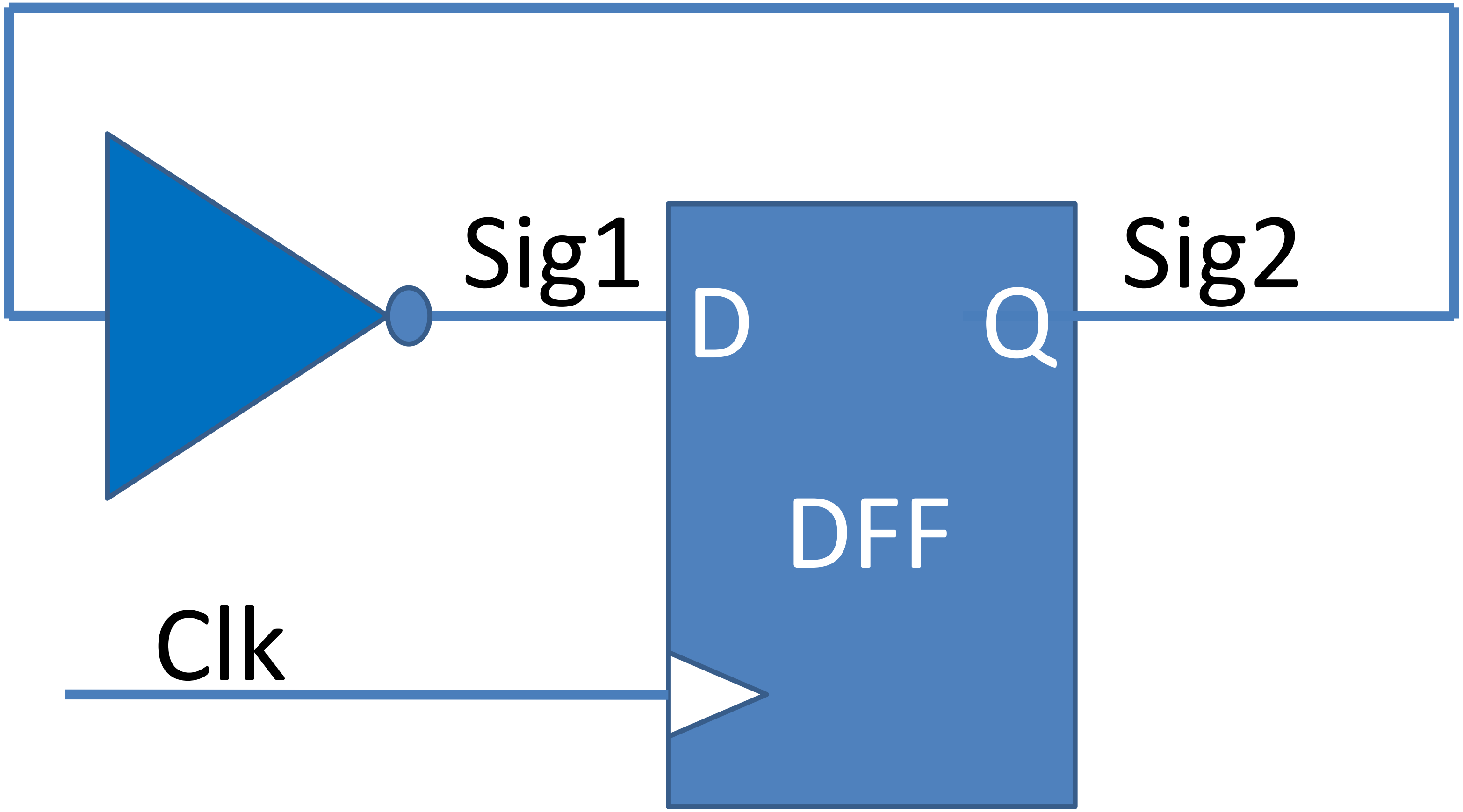
**Declaration :**

```
signal A : std_logic;
signal B : std_logic := ‘0’;
```

**Assignments :**

```
A <= B;
```

# Data Types – Std_logic



Sig1, Sig2 and Clk are **std_logic** signals used here to connect the logic devices together.

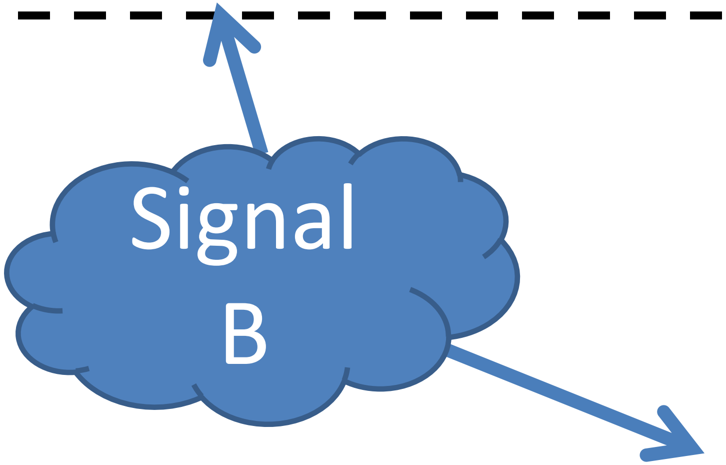
# Data Types – Std_logic_vector

- Is an array of **std_logic** signals.
- Similar to an array of bits.
- Package - Std_Logic_1164.
- Examples:
  - Address or  Data busses
  - Counter / Timer
  - Register

# Data Types – Std_logic_vector

**Declaration :**

```
signal A : std_logic_vector(7 downto 0);
signal B : std_logic_vector(7 downto 0) := x”A1”;
```



Signal B

| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |

# Data Types – Std_logic_vector

```
signal A : std_logic_vector(7 downto 0) ;
signal B : std_logic_vector(7 downto 0);
signal C : std_logic;
```

| Assignments |
|---|
| A <= B; |
| A(7 downto 6) <= B(3 downto 2); |
| A(4) <= B(0); |
| C  <= A(6); |

# Array Attributes

```
signal A : std_logic_vector(7 downto 0) ;
```

| Array Attributes | Returned Value |
|---|---|
| A’left | 7 |
| A’right | 0 |
| A’length | 8 |
| A’range | 7 downto 0 |

# Array Attributes

```
signal A : std_logic_vector(7 downto 0) ;
signal F : std_logic;
```

| Using Attributes | Without Using Attributes |
|---|---|
| signal B : std_logic_vector(A’range) ; | signal B : std_logic_vector(7 downto 0) ; |
| signal C : std_logic_vector(A’left – 1 downto 0); | signal C : std_logic_vector(6 downto 0); |
| F <= A(A’right); | F <= A(0); |
| F <= A(A’left-2); | F <= A(5); |

# Data Types – Unsigned & Signed

- Represents a signed or unsigned value.
- A vector of bits similar to std_logic_vector.
- Takes the same values as the std_logic type.
- Package – **numeric_std**.
- Examples :
  - Counters.
  - Used for arithmetic operations.

# Data Types – Unsigned & Signed

**Declaration :**

```
signal A : unsigned(7 downto 0);
signal B : signed(7 downto 0) := x”A1”;
```

- Assignments:
  - Must be same type (i.e both unsigned or both signed).
  - Must be the same size.

**Assignments :**

```
Y <= X;
```

# Data Types – integer

- Represents an integer value.
- Built into VHDL so no Package needed.
- At least covers the range : $-2^{31}$ to $+ 2^{31}-1$
- Examples :
  - Counters
  - Indices

# Data Types – integer

**Declaration :**

```
signal A : integer;
signal B : integer := 2;
```

Not recommended!

```
signal C : integer range 0 to 255;
signal D : integer range -128 to 127 := 35;
```

**Assignments :**

```
Y <= X;
```

# Data Types – Boolean

- Represents a value that can be true or false.
- Built into VHDL language.

**Declaration :**

```
signal A : boolean;
signal B : boolean := true;
```

# Enumerated Types

Allowed Values : Red, Amber and Green

## VHDL Type Declaration

```
type TrafficLight is (red, amber, green);
```

## VHDL Signal Declaration

```
signal my_signal : TrafficLight;
```

## Signal Assignment

```
my_signal <= red;
```

# Enumerated Types - Encoding

00 = Red

01 = Amber

10 = Green

11 = Unused

# Enumerated Types

- VHDL supports enumerated types.
- Allow us to create our own data types.
- Defined by listing all possible values.
- Example :
  - Std_logic
  - Used for state machine state encoding.

# Enumerated Types

## Declare New Enumerated Type

```
type SMType is (IDLE, DO_WORK, ASSERT_DONE);
```

## Declare State Machine Signal

```
signal SMState : SMType;
```

## Example Assignments

```
SMState <= IDLE;
SMState <= DO_WORK;
```

# Enumerated Types

- SMState signal will be implemented using 2 bits on the FPGA :

    00 = IDLE<br>
    01 = DO_WORK<br>
    10 = ASSERT_DONE<br>
    11 = Unused

# Arrays

- Array = Collection of signals of the same type.
- std_logic_vector is an array of std_logic elements.

- We can define our own array types.

```
type myArrayType is array (0 to 3) of integer;
```

# Arrays

```
signal myArray : myArrayType ;

myArray(0) <= 13;

myArray(1) <= 14

myArray(2) <= 15;

myArray(3) <= 16
```

| Index 0 | Index 1 | Index 2 | Index 3 |
|---|---|---|---|
| 13 | 14 | 15 | 16 |

# Unconstrained Array Type

- Unconstrained array type DOES NOT specify number of elements.
- Number of elements must be declared when creating the signal!

Array range is not defined.

```
type ABC is array (integer range <>) of integer;

signal Sig1: ABC(0 to 7);
signal Sig2: ABC(1 to 16);
```

Array index has an integer type

# Summary

- Signals are like wires.
- Used to connect things together.
- Data types specify what values a signal can carry.
- Define our own data types.
- Define our own array types.

# Signals & Data Types II

# More On Enumerated Types

**Defined In Package Standard:**

```
type bit is (‘0’, ‘1’);

type boolean is (false, true);

type CHARACTER is (all values in the ascii table!);

type SEVERITY_LEVEL is (NOTE, WARNING, ERROR, FAILURE);
```

**Defined In Package STD_LOGIC_1164:**

```
type std_ulogic is (‘U’, ‘X’, ‘0’, ‘1’, ‘Z’, ‘W’, ‘L’, ‘H’, ‘-’);
```

!!! Note : std_logic is a subtype of std_ulogic !!!

# Subtypes

- VHDL subtypes are subsets of the base type.

**Examples From Standard Package:**

```
subtype NATURAL is INTEGER range 0 to INTEGER'HIGH;

subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;
```

**Declaring Signals Using Subtypes:**

```
signal NaturalSignal : natural;

signal PositiveSignal : positive;
```

# Subtypes

- VHDL subtypes are subsets of the base type.

**User Defined Subtypes:**

```
subtype uint8_t is INTEGER range 0 to 255;
subtype MyVector8 is std_logic_vector(7 downto 0);
```

**Declaring Signals Using Subtypes:**

```
signal RxChar   : uint8_t;
signal DataBus : MyVector8;
```

# More On Array Types

## Examples of Array Types in the Standard Package:

```
type STRING is array (POSITIVE range <>) of CHARACTER;

type BOOLEAN_VECTOR is array (NATURAL range <>) of BOOLEAN;

type BIT_VECTOR is array (NATURAL range <>) of BIT;

type INTEGER_VECTOR is array (NATURAL range <>) of INTEGER;
```

Using a Natural array index means that the element indices cannot be negative.

## Signal Declarations

```
signal MyCharArray : string(1 to 10);

signal MyBoolArray : boolean_vector(7 downto 0);

signal MyBitArray    : bit_vector(15 downto 8);
```

# Creating Your Own Array Types

## User Defined Data Type:

```
type TLightType is (RED, AMBER, GREEN);
```

## User Defined Array Type:

```
type TLightArray is array (NATURAL range <>) of TLightType ;
```

## Signal Declarations

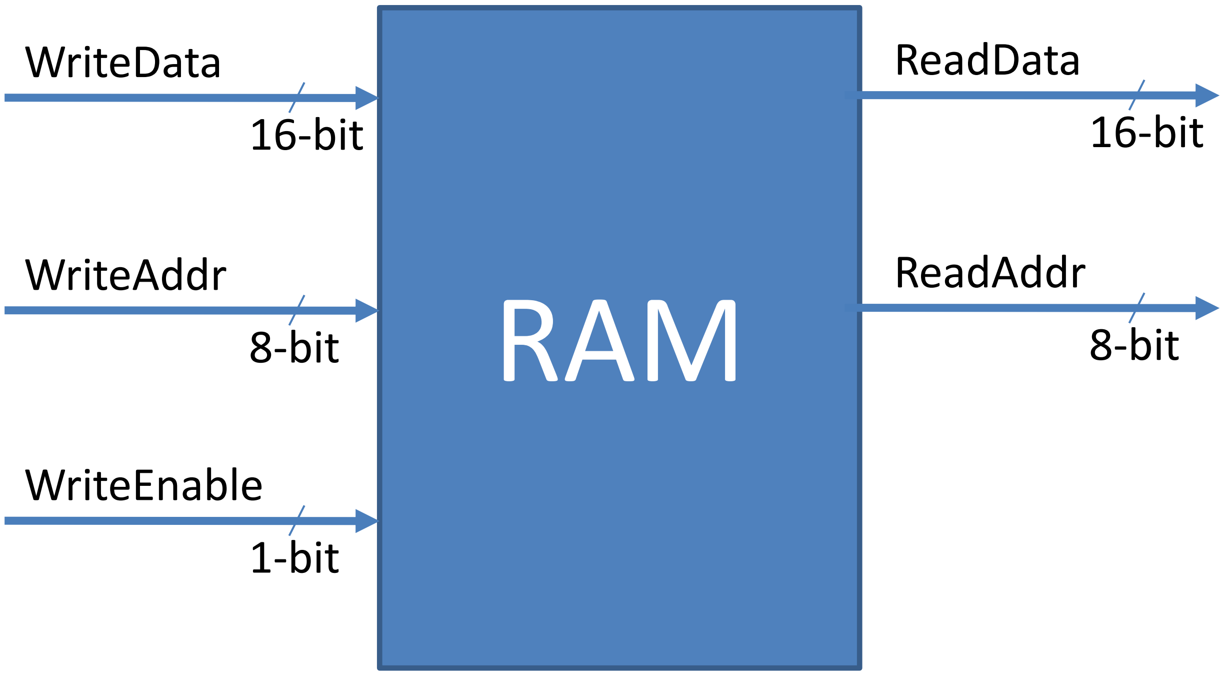
```
signal MyTrafficLight: TLightArray(1 to 10);
```

## Signal Assignments

```
MyTrafficLight(1) <= red;

MyTrafficLight(10) <= green;
```

# Record Type

- Record Type = A type that is a collection of elements (of any data type).



# Record Type

```
signal  WriteData     : std_logic_vector(15 downto 0);
signal  ReadData      : std_logic_vector(15 downto 0);
signal  WriteAddr     : std_logic_vector(7 downto 0);
signal  ReadAddr      : std_logic_vector(7 downto 0);
signal  WriteEnable   : std_logic;
```

# Records

**Step 1 : Create the new record type :**

```
type MyRAMType is record
      signal   WriteData      : std_logic_vector(15 downto 0);
      signal   ReadData       : std_logic_vector(15 downto 0);
      signal   WriteAddr      : std_logic_vector(7 downto 0);
      signal   ReadAddr       : std_logic_vector(7 downto 0);
      signal   WriteEnable    : std_logic;
end record;
```

# Records

**Step 2 : Declare our new signal**

```
signal   RAM_IO : MyRAMType;
```

**Examples of Signal Assignments**

```
RAM_IO.WriteData <= x”8000”;
RAM_IO.WriteAddr <= x”01”;
RAM_IO.WriteEnable<= ‘1’;
```

# Summary

- Enumerated types
- Subtypes
- Array types
- Record Types

# Constants

# Constants

**Constant Declarations:**

```
constant name : type := initial_value;
```

**Example Signal Assignment:**

```
MySignal <= MyConstant;
```

# Summary

- Constants must be initialised during declaration.
- Once declared, its value cannot be changed.
- Use constants as much as possible:
  - Makes code more readable.
  - Easier to maintain.

# Operators

# VHDL Operators

| | |
|---|---|
| Not | - inversion |
| And | - and function |
| Nand | - not and function |
| Or | - or function |
| Nor | - not-or function |
| Xor | - exclusive or function |
| Xnor | - exclusive nor function |
| = | - equality |
| /= | - inequality |
| >= | - greater than or equal |
| > | - greater than |
| <= | - less than or equal |
| < | - less than |
| Sll | - shift left logical |
| Srl | - shift right logical |
| Rol | - rotate left |
| Ror | - rotate right |
| Sla | - shift left arithmetic |
| Sra | - shift right arithmetic |
| + | - addition |
| - | - subtraction |
| + | - plus sign |
| - | - minus sign |
| * | - multiplication |
| / | - division |
| Mod | - modulo arithmetic |
| Rem | - remainder |
| ** | - exponential |
| Abs | - absolute value |
| & | - concatenation |

# Boolean Operators

| Operator | Operation |
|---|---|
| Not | Invert |
| And | And Gate |
| Nand | Inverted And Gate |
| Or | Or Gate |
| Nor | Inverted Or Gate |
| Xor | Exclusive or Gate |
| Xnor | Inverted exclusive or gate |

# Boolean Operators

**Libraries:**

- Std_Logic_1164

**Can Be Used With :**

- Bit
- Boolean
- Arrays of the above types.

# Boolean Operators

**Examples :**

```
C <= A and B; -- All signals std_logic type

Z <= X xor Y;   -- All signals Boolean type

if Sig1 = “1101” or Sig2 = true then
-- Code to execute when conditions are true
end if;
```

# Boolean Operators

```
signal P : std_logic_vector(3 downto 0);
signal Q : std_logic_vector(3 downto 0);
signal R : std_logic_vector(3 downto 0);

R <= P or Q;
```

# Boolean Operators

```
signal P : std_logic_vector(3 downto 0);
signal Q : std_logic_vector(3 downto 0);
signal R : std_logic_vector(3 downto 0);
```

```
R <= P or Q;
```

```
R(0) <= P(0) or Q(0);
R(1) <= P(1) or Q(1);
R(2) <= P(2) or Q(2);
R(3) <= P(3) or Q(3);
```

# Comparison Operators

| Operator | Operation |
|---|---|
| = | Equality |
| /= | Inequality |
| >= | Greater than or equal |
| <= | Less than or equal |
| > | Greater than |
| < | Less than |

# Comparison Operators

**Notes:**

- Used to compare signals.
- Data types of the compared signals must match.
- Comparison operators return a Boolean result.

**Libraries :**

- Std_logic_1164

**Can Be Used With**

- Most data types support all 6 operators.
- All types support the = and /= operators

# Comparison Operators

**Examples :**

```
Y <= (X > “0011”); -- Y is Boolean, X is a 4-bit vector
```

```
if Sig1 > 128 then – Sig1 is an integer type
-- Code to execute when condition is true
end if;
```

# Shift Operators

| Operator | Operation | Notes |
|---|---|---|
| SLL | Shift Left Logical | Shifts the array discarding bits that are shifted off one end and fills the other end with ‘0’. |
| SRL | Shift Right Logical | |

**Libraries:**

- Std_logic_1164

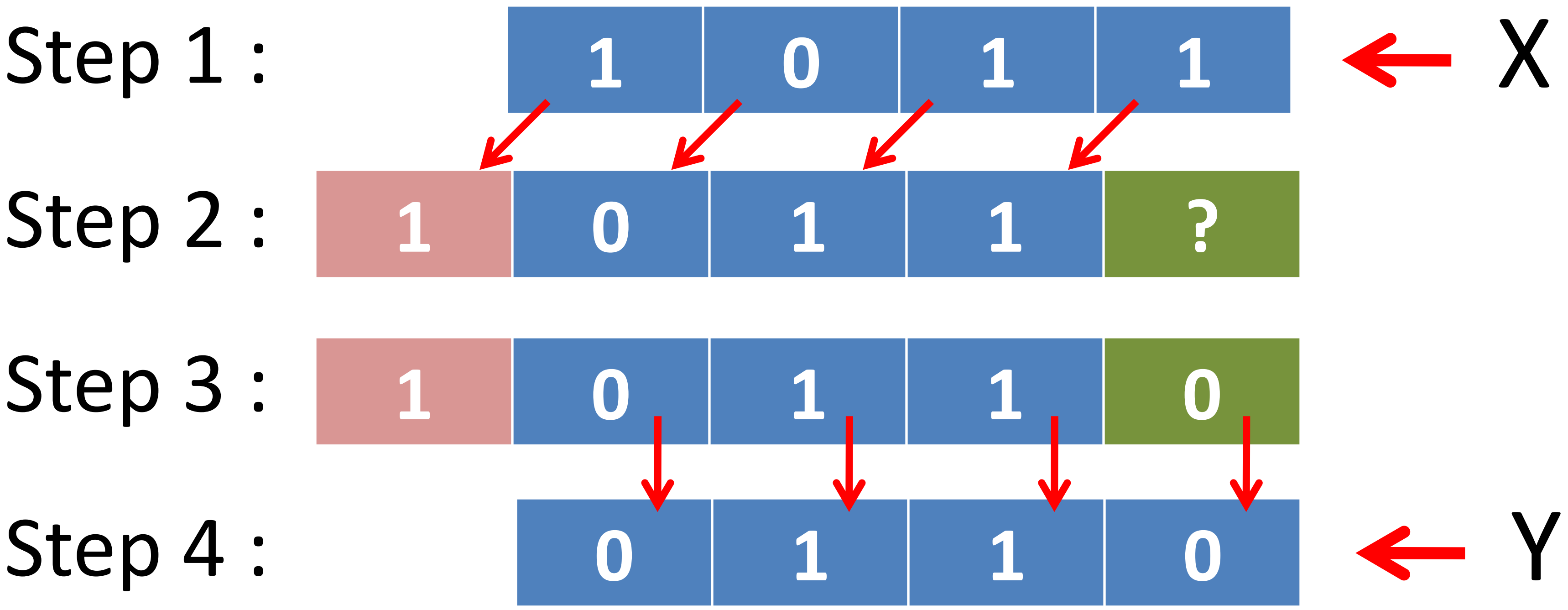
**Can Be Used With:**

- std_logic_vector and bit_vectors.

# SLL Example

```
signal X : std_logic_vector(3 downto 0) := “1011”;
signal Y : std_logic_vector(3 downto 0);

Y <= X sll 1;
```

Number of Shifts



Step 1 : 1 0 1 1 ← X

Step 2 : 1 0 1 1 ?

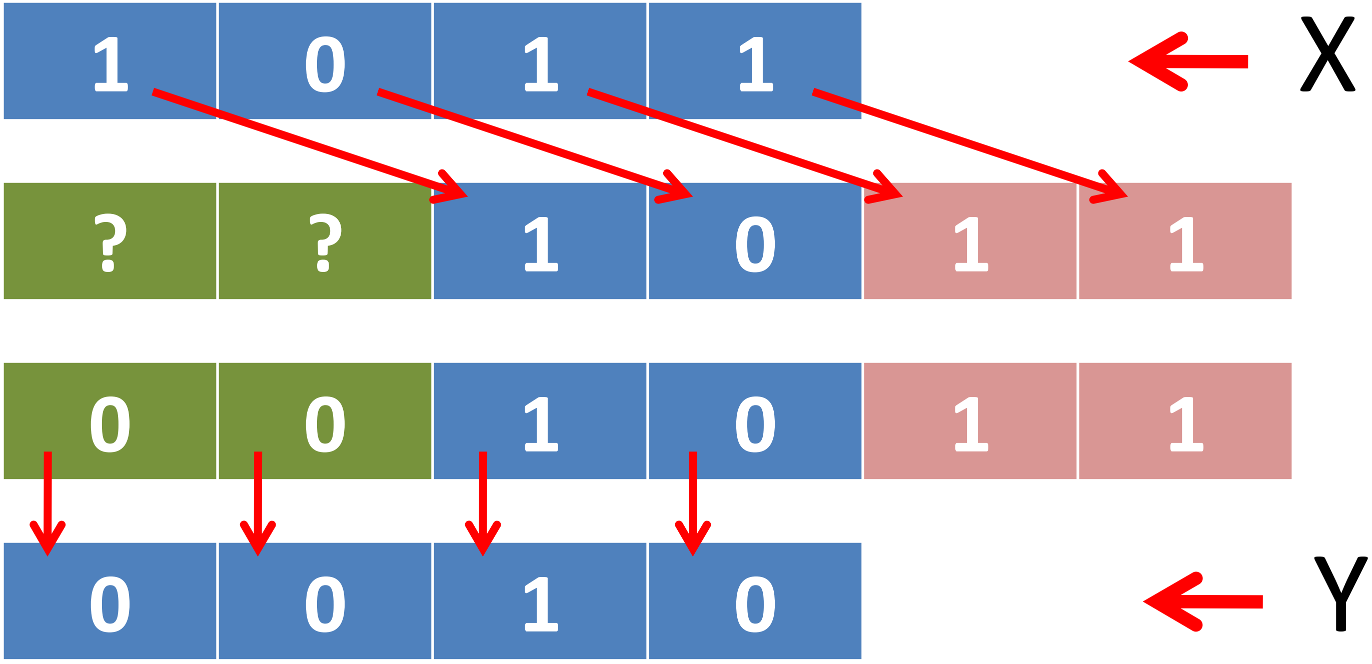
Step 3 : 1 0 1 1 0

Step 4 : 0 1 1 0 ← Y

# SRL Example

```
signal X : std_logic_vector(3 downto 0) := “1011”;
signal Y : std_logic_vector(3 downto 0);

Y <= X srl 2;
```

Number of Shifts



Step 1 : 1 0 1 1 ← X

Step 2 : ? ? 1 0 1 1

Step 3 : 0 0 1 0 1 1

Step 4 : 0 0 1 0 ← Y

# Rotate Operators

| Operator | Operation | Notes |
|---|---|---|
| ROL | Rotate Left | Take elements off one end of the array and shifts them in at the other end. |
| ROR | Rotate Right | |

**Libraries:**

- Std_logic_1164

**Can Be Used With:**

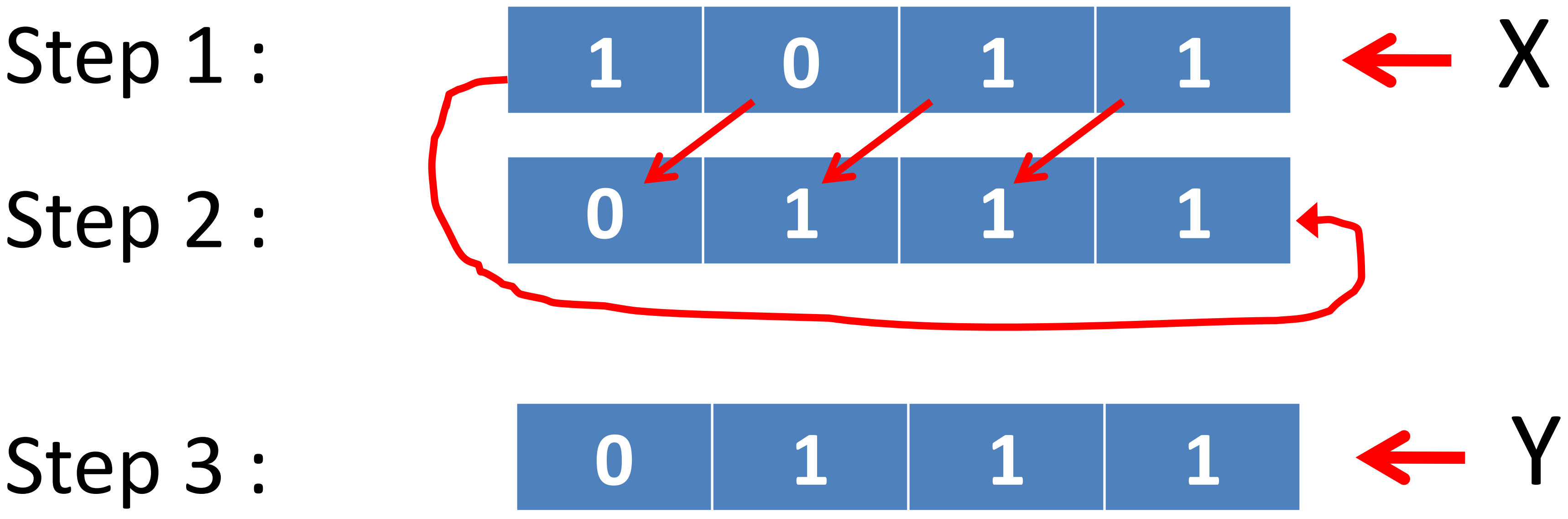
- std_logic_vector and bit_vectors.

# ROL Example

```
signal X : std_logic_vector(3 downto 0) := “1011”;
signal Y : std_logic_vector(3 downto 0);
```

Number of Shifts

```
Y <= X rol 1;
```



# Arithmetic Operators

| Operator | Meaning |
|---|---|
| + | Addition / Positive sign |
| - | Subtraction / Negative sign |
| * | Multiplication |
| / | Division |
| Mod | Modulo arithmetic |
| Rem | Remainder after divsion |
| ** | Exponential |
| Abs | Absolute value |

# Arithmetic Operators

**Libraries:**

- numeric_std

**Can be used with :**

- Unsigned Types
- Signed Types
- Integer Types

# Arithmetic Operators

1. C <= A + B; -- All signals are unsigned bit vectors
2. C <= A - B; -- All signals are integers
3. C <= 2 ** 8; -- All signals are integers
4. C <= (C+1) mod 16 -- All signals are integers

# Concatenation Operator

| Operator | Meaning |
|---|---|
| & | Concatenation Operator |

**Libraries:**

- Std_logic_1164
- Numeric_std

**Can Be Used With:**

- std_logic_vector and bit_vectors.
- Signed and unsigned.

# Concatenation Operators

```
Signal A : std_logic_vector(3 downto 0):= “0011”;

Signal B : std_logic_vector(3 downto 0):= “1111”;

Signal C : std_logic_vector(4 downto 0);

Signal D : std_logic_vector(7 downto 0);


C <= A & ‘0’;  -- C will be “00110”

C <= ‘1’ & A;  -- C will be “10011”

D <= A & B; -- D will be “00111111”
```

# Operator Precedence

| Operator |
|---|
| Abs, not, ** |
| Mod, rem, *, / |
| +, - (signs) |
| +, - , & |
| Sll, srl, rol, ror |
| And, or, nand, nor, xor, xnor |

# Operator Precedence

| VHDL Statement | Evaluated As |
|---|---|
| C <= 2 * 3 + 4; | C <= (2*3) + 4; |
| C <= A Sll 2 + 4; | C <= A Sll (6); |
| C <= -A * B; | C <= -(A * B); |

# Operator Precedence

| Incorrect Statements | Corrected |
|---|---|
| M <= A and B or C; | M <= (A and B) or C;<br>M <= A and (B or C); |
| M <= A nand B nand C; | M <= (A nand B) nand C;<br>M <= A nand (B nand C); |

(A and B) or C /= A and (B or C)

(A nand B) nand C /= A nand (B nand C)

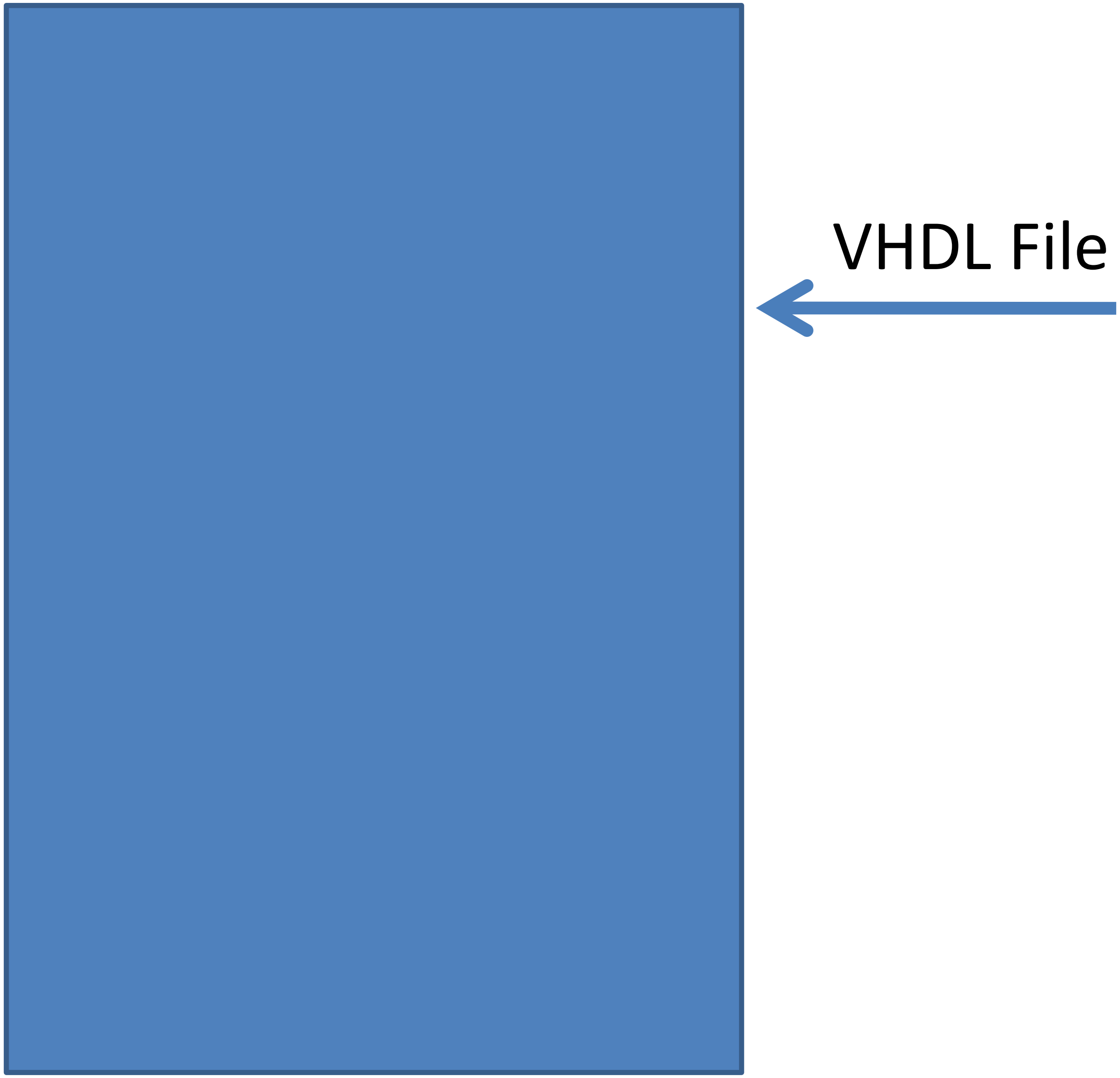
!! Use brackets as much as possible to avoid confusion !!

# Summary

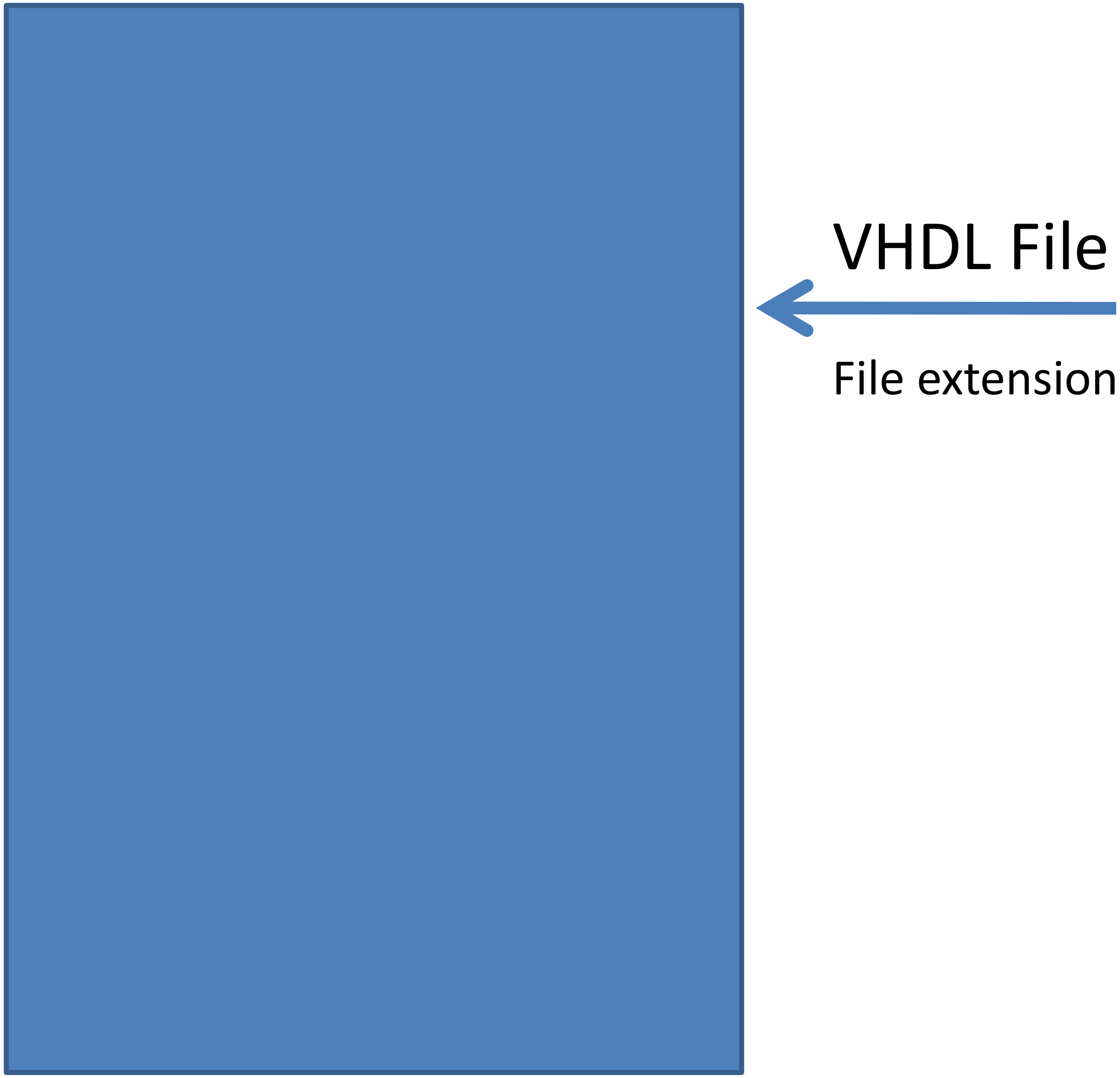
- Boolean operators
- Comparison operators
- Shift operators
- Arithmetic operators
- Concatenation operator

# Structure of a VHDL File

# VHDL File Structure



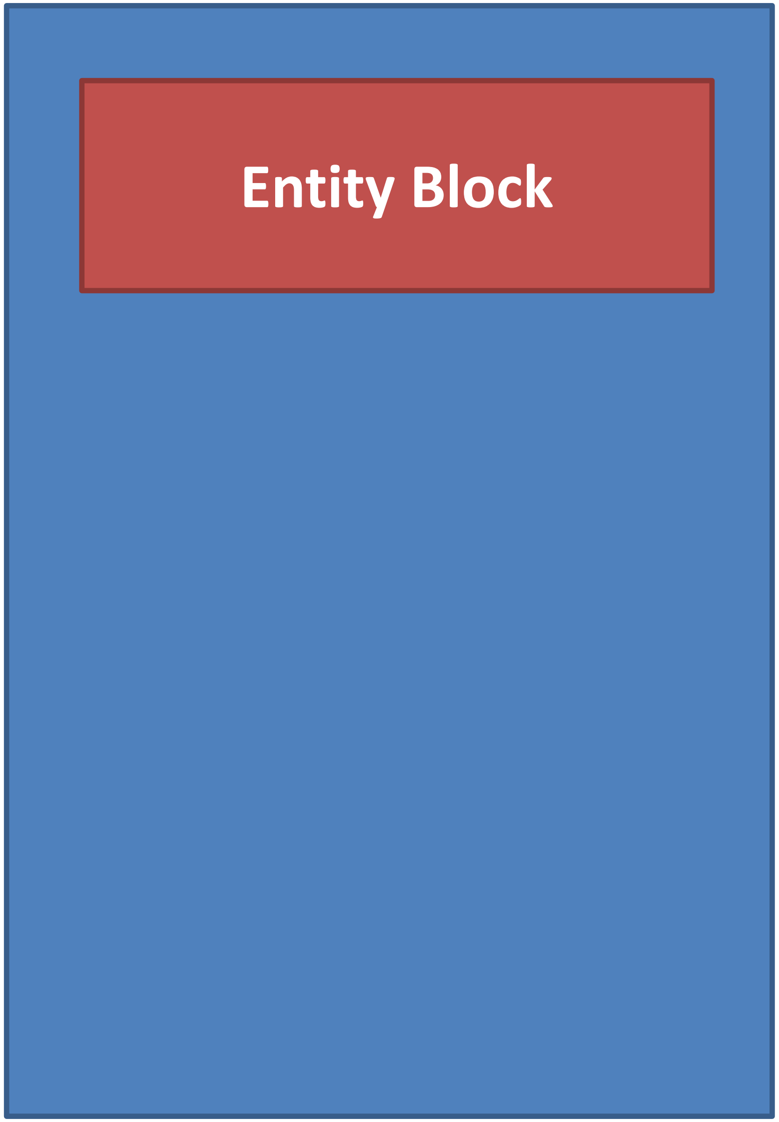
# VHDL File Structure



VHDL File

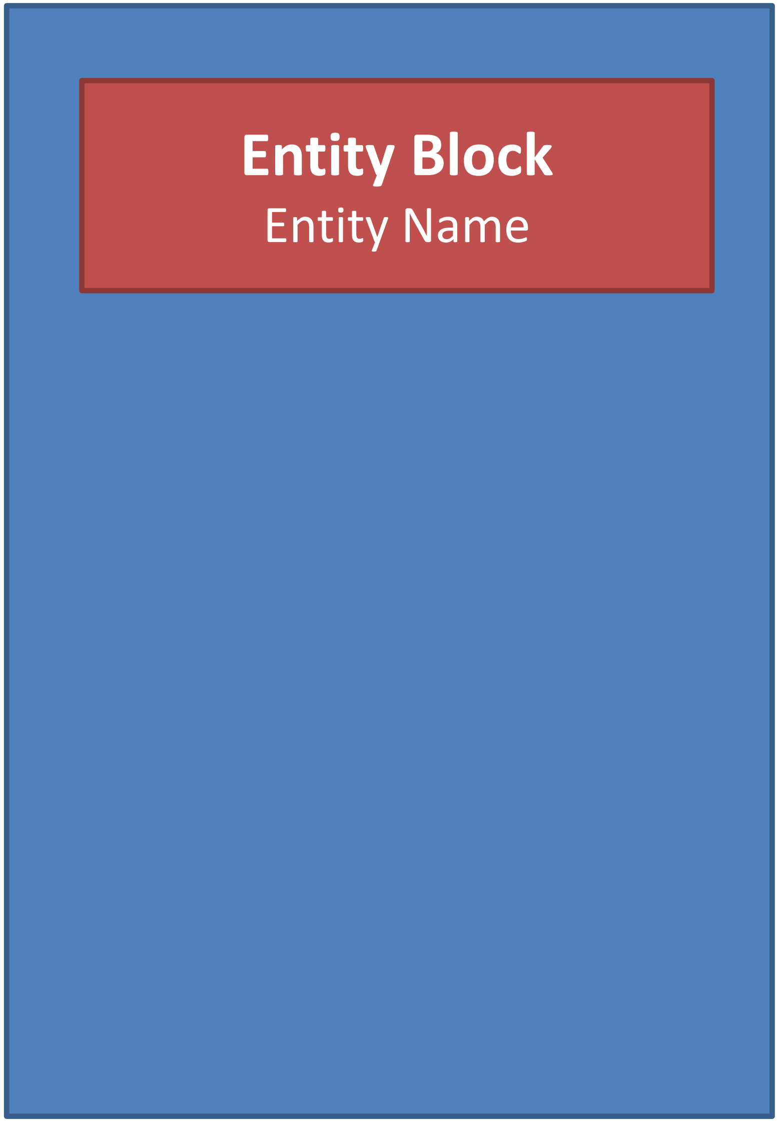
File extension is .vhd

# VHDL File Structure



Entity Block

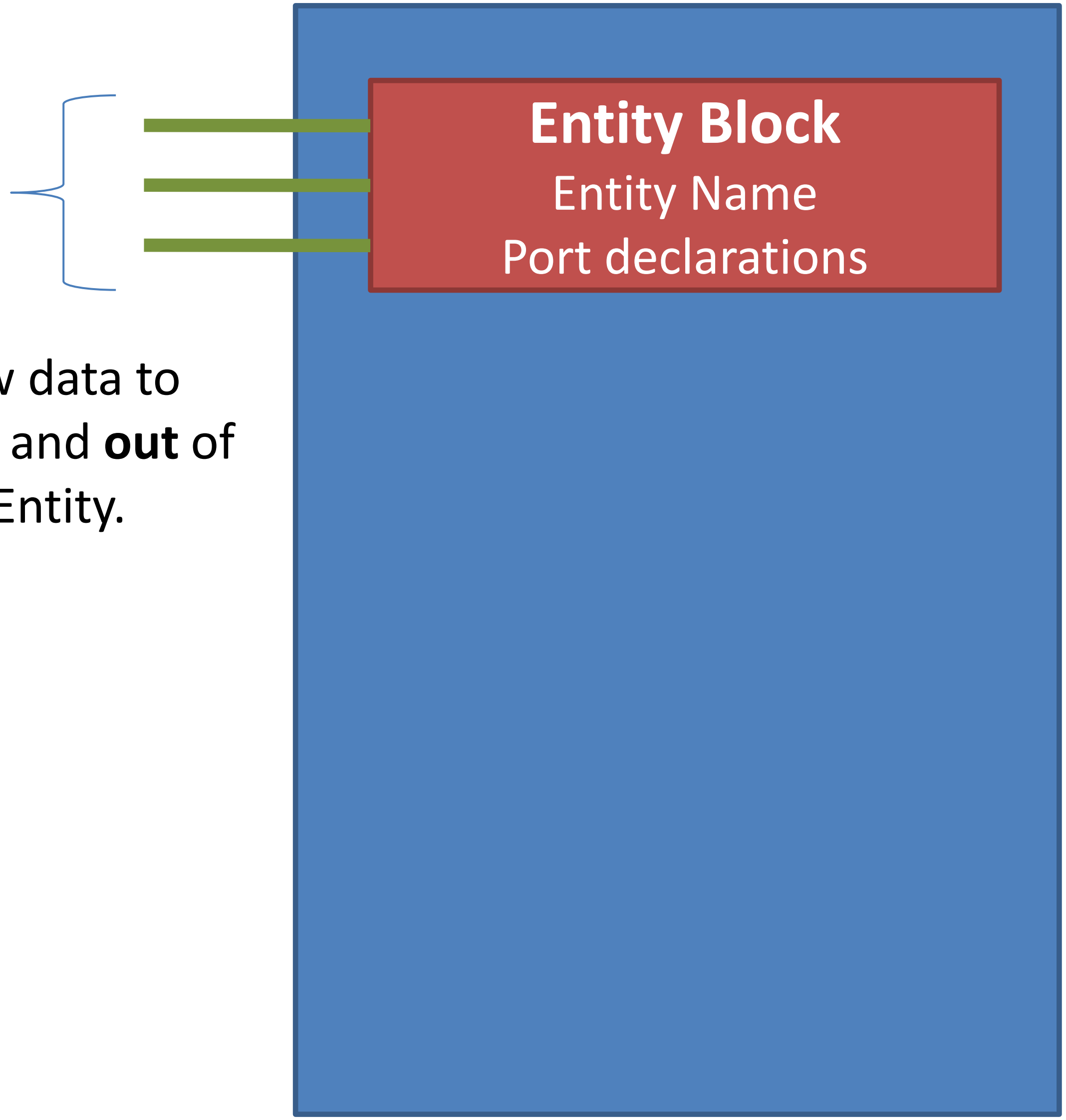
# VHDL File Structure



**Entity Block**

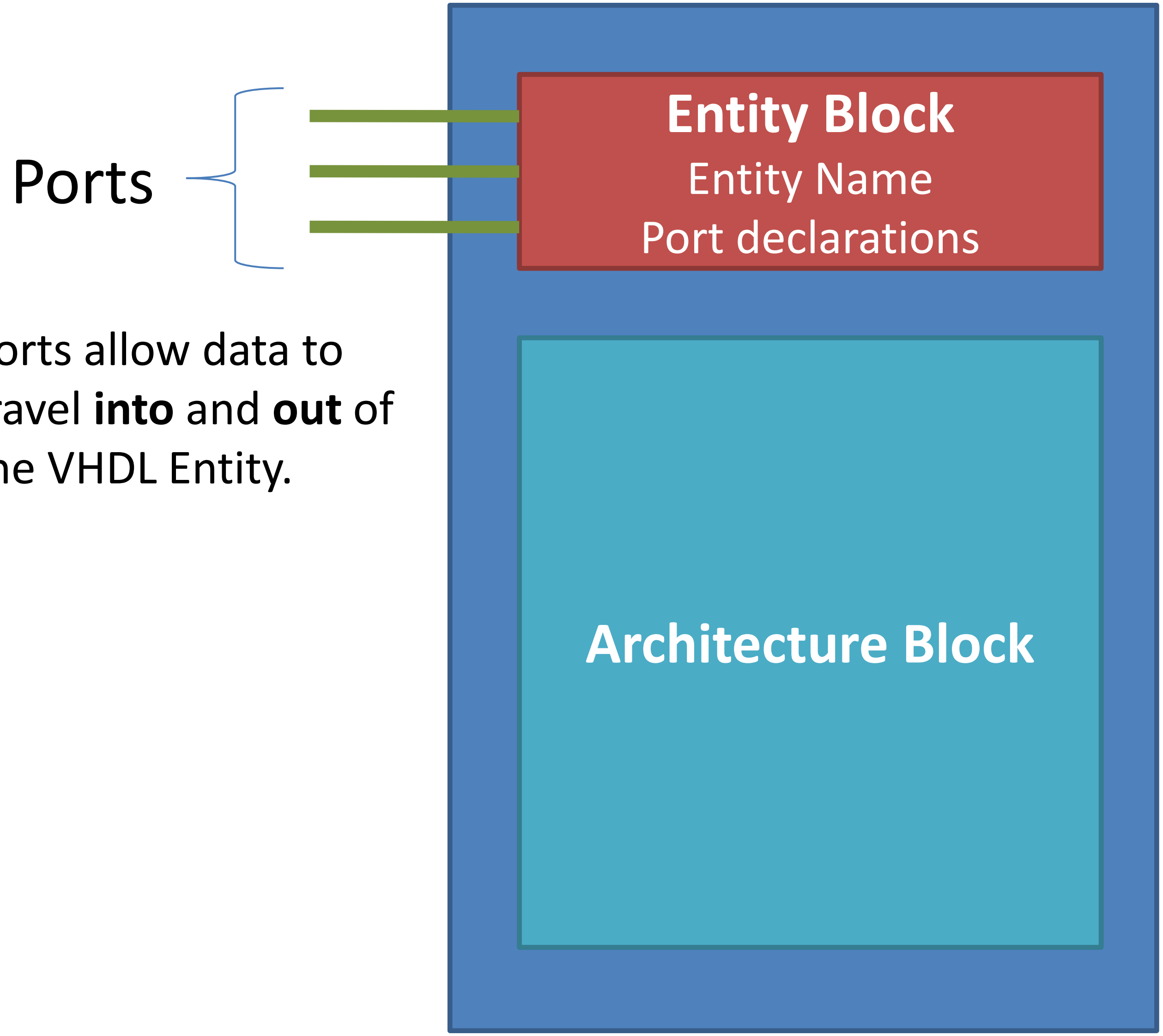
Entity Name

# VHDL File Structure



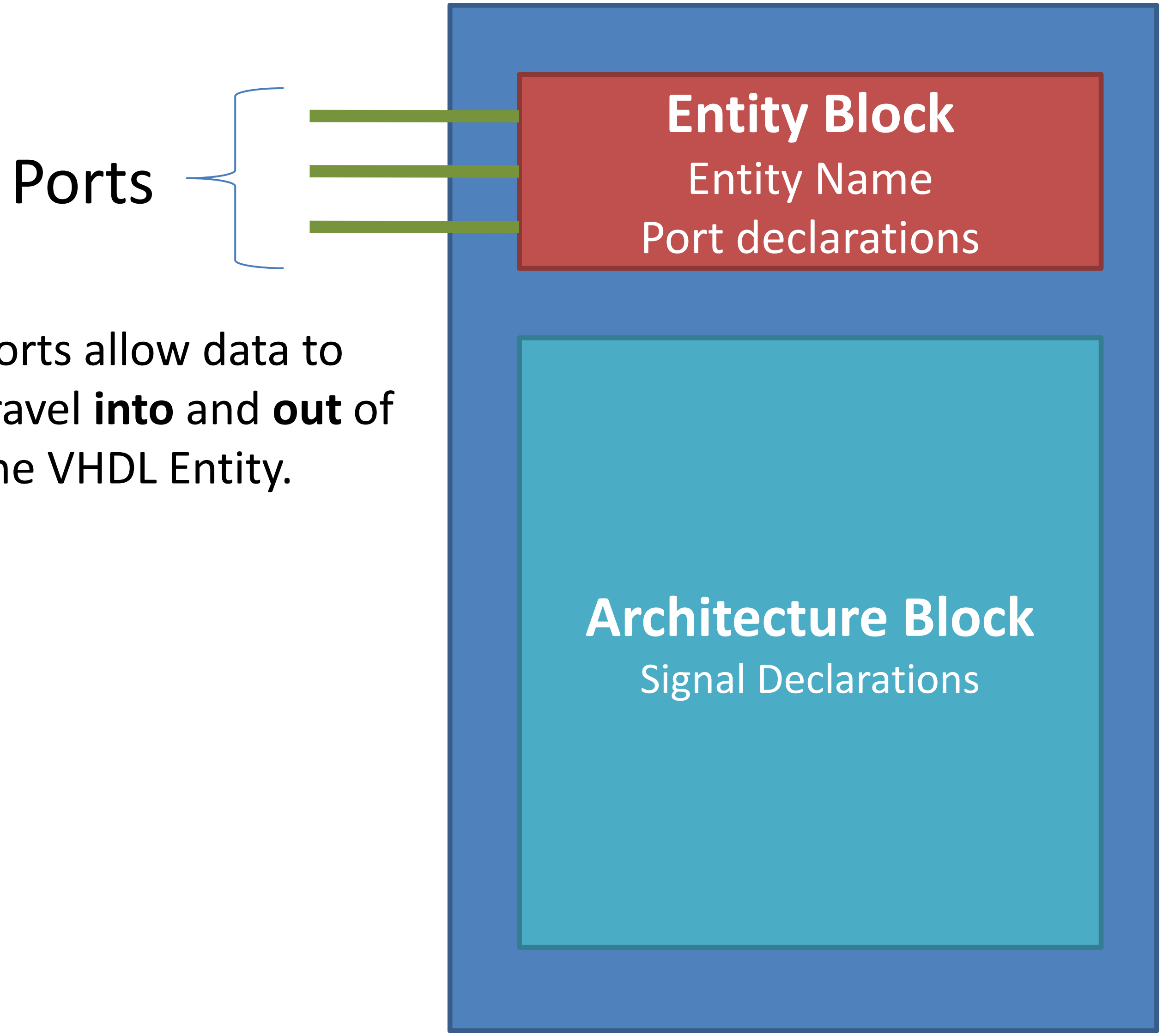
Ports allow data to travel **into** and **out** of the VHDL Entity.

# VHDL File Structure



Ports allow data to travel **into** and **out** of the VHDL Entity.

# VHDL File Structure



Ports

**Entity Block**

Entity Name

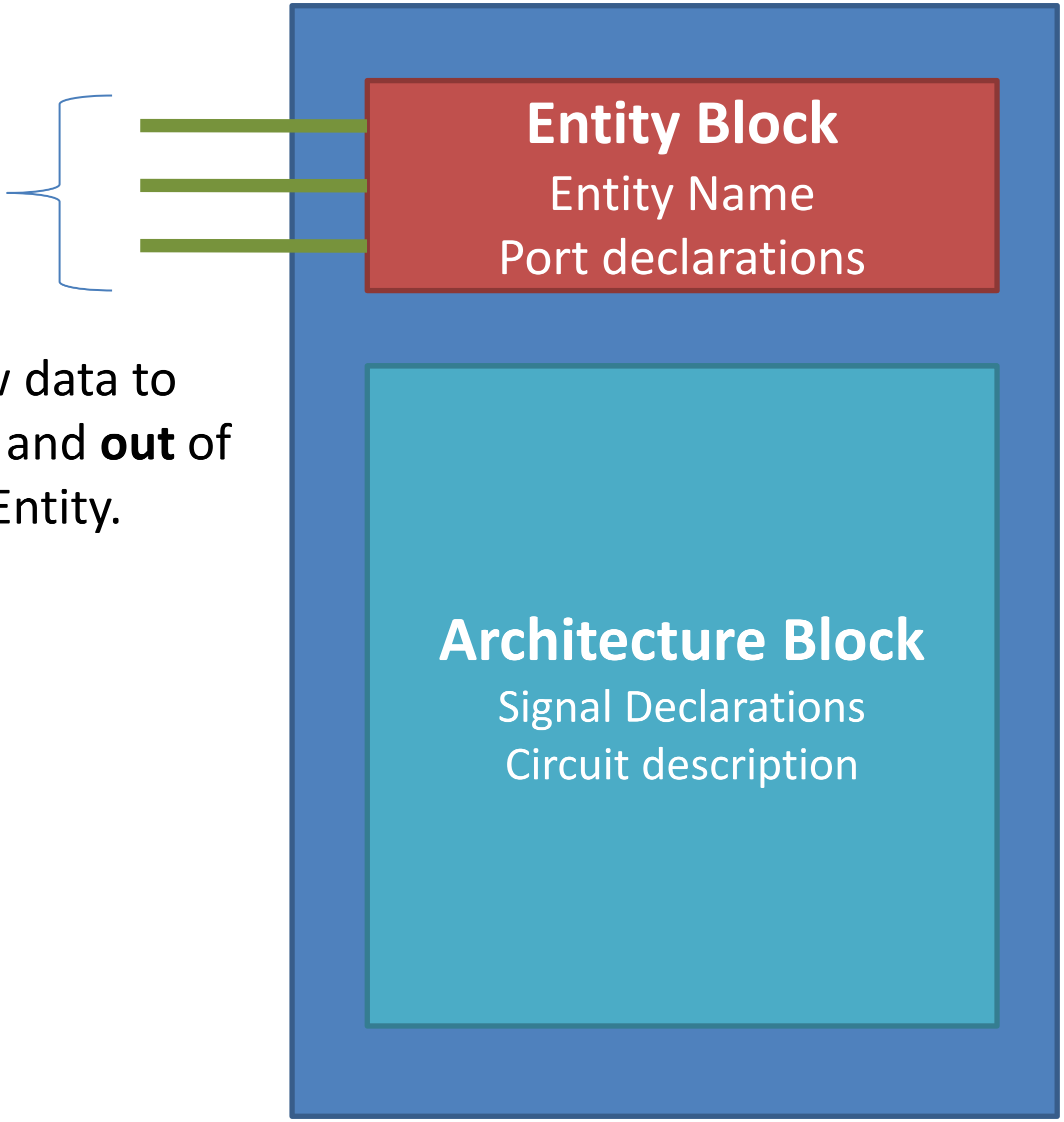
Port declarations

**Architecture Block**

Signal Declarations

Ports allow data to travel **into** and **out** of the VHDL Entity.

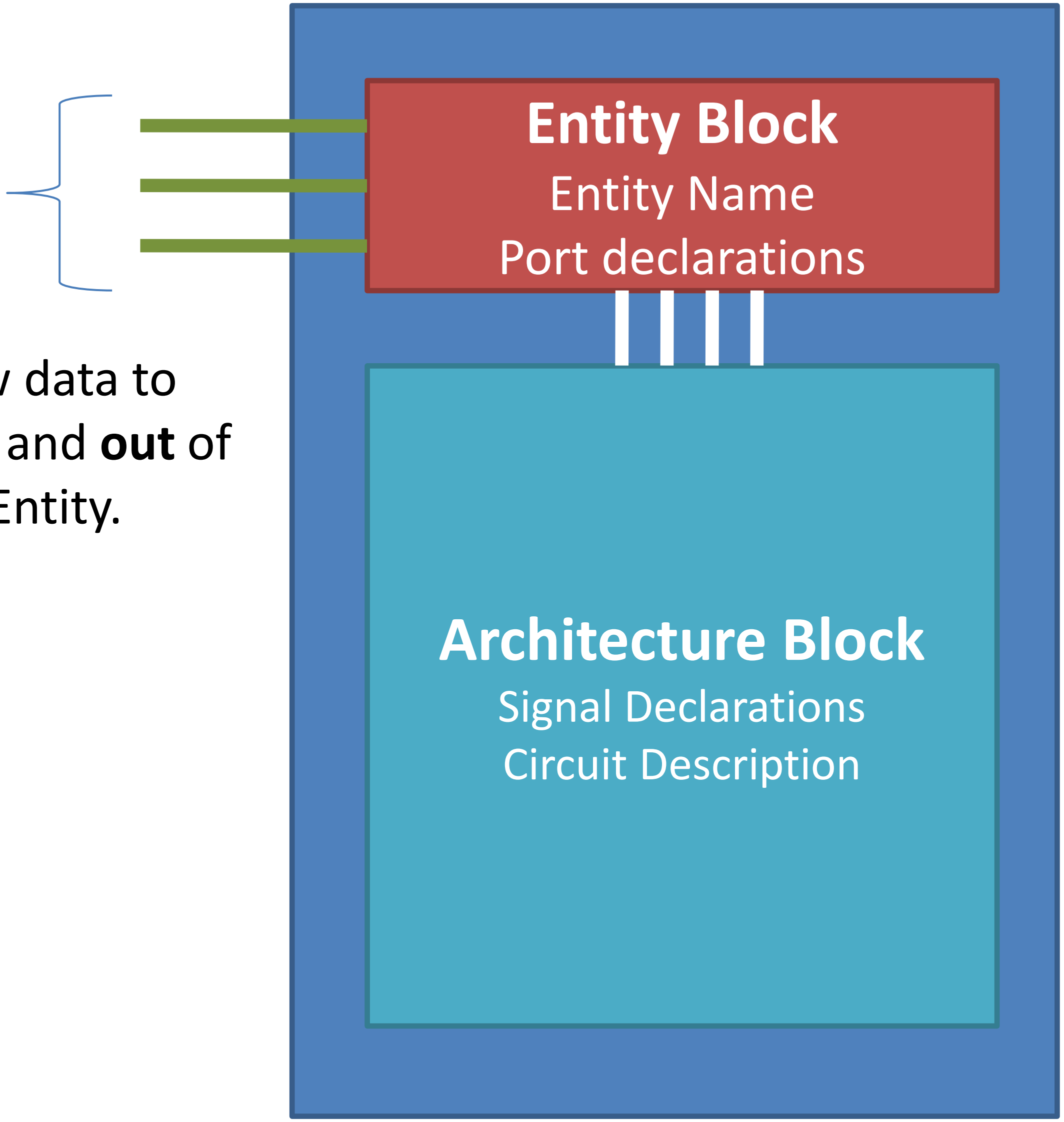
# VHDL File Structure



Ports allow data to travel **into** and **out** of the VHDL Entity.

# VHDL File Structure

Ports



**Entity Block**

Entity Name

Port declarations

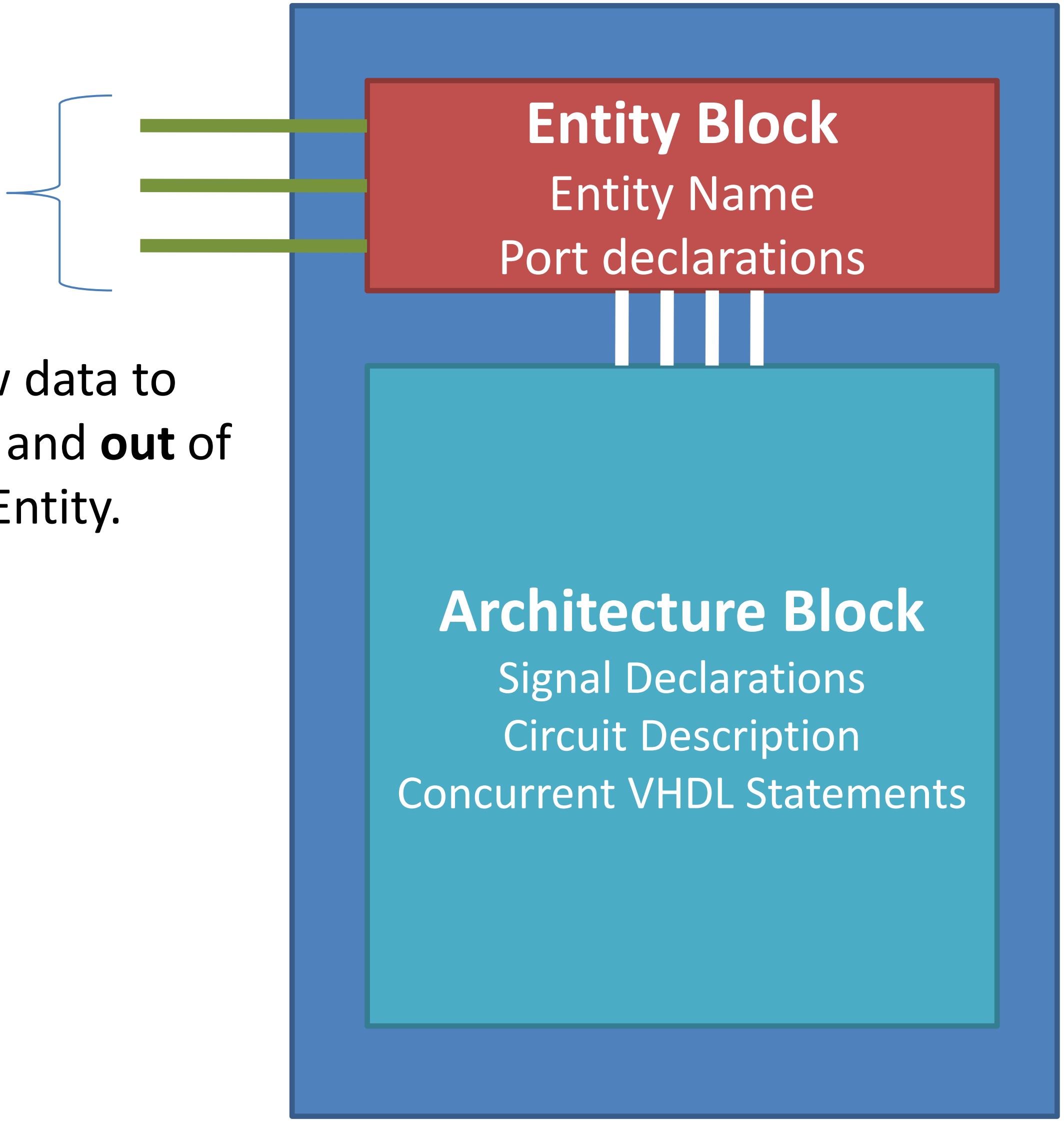
**Architecture Block**

Signal Declarations

Circuit Description

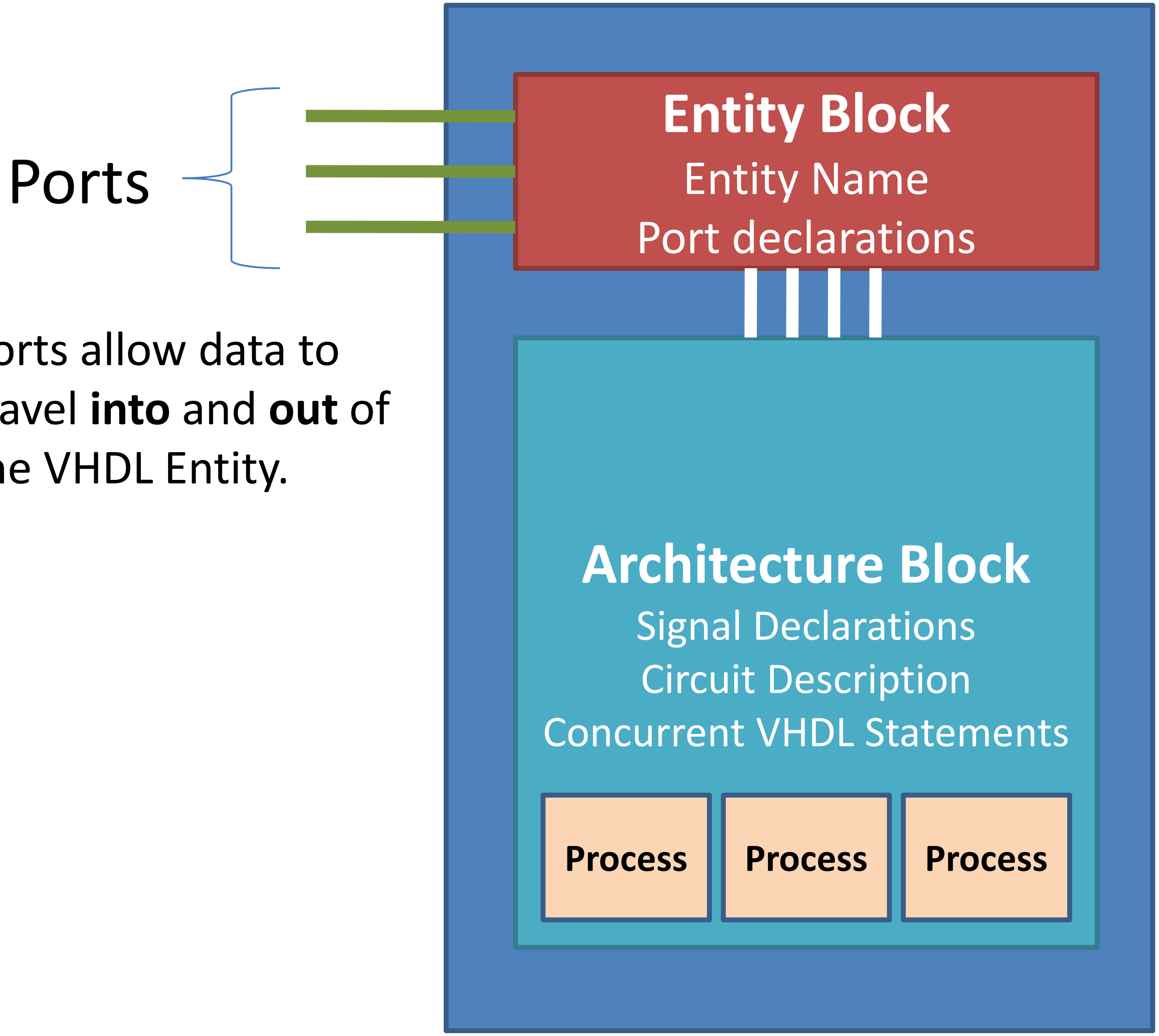
Ports allow data to travel **into** and **out** of the VHDL Entity.

# VHDL File Structure



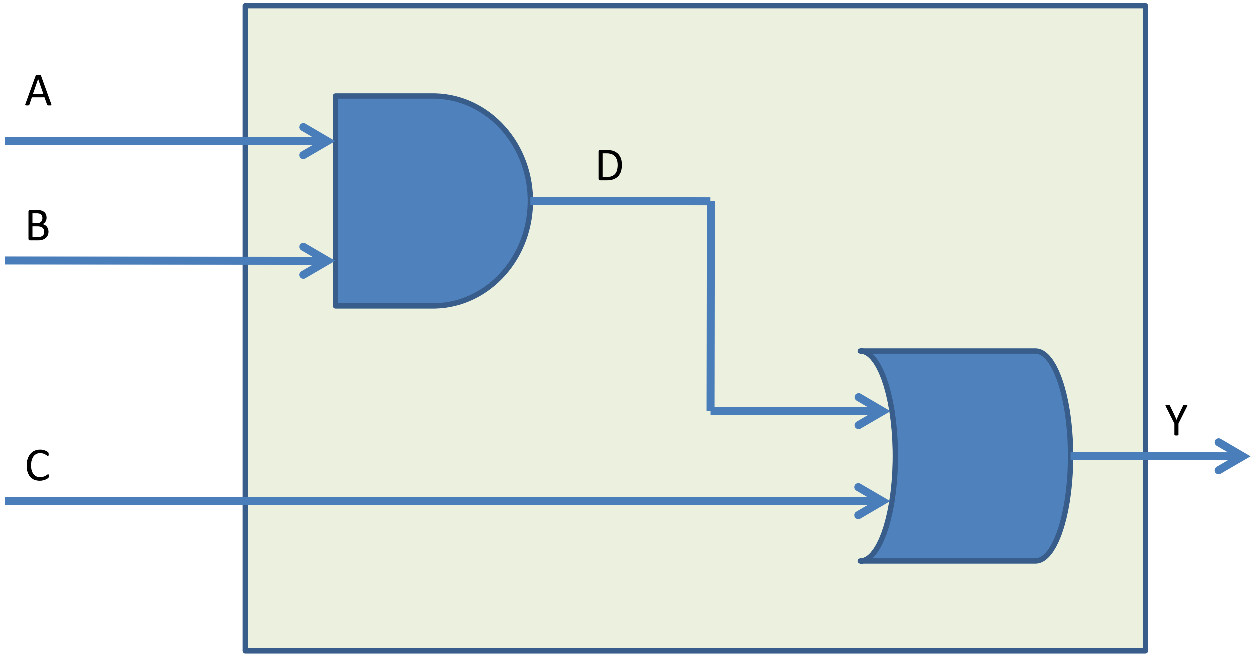
Ports allow data to travel **into** and **out** of the VHDL Entity.

# VHDL File Structure

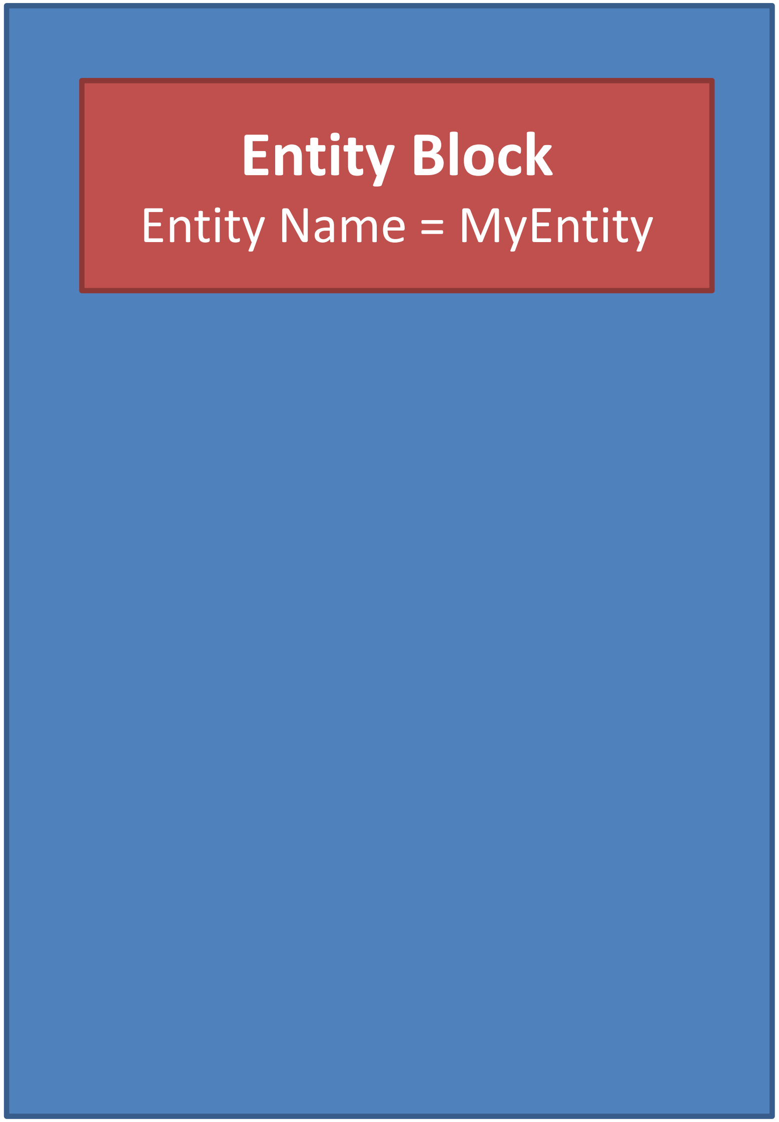


Ports allow data to travel **into** and **out** of the VHDL Entity.

# VHDL File Structure



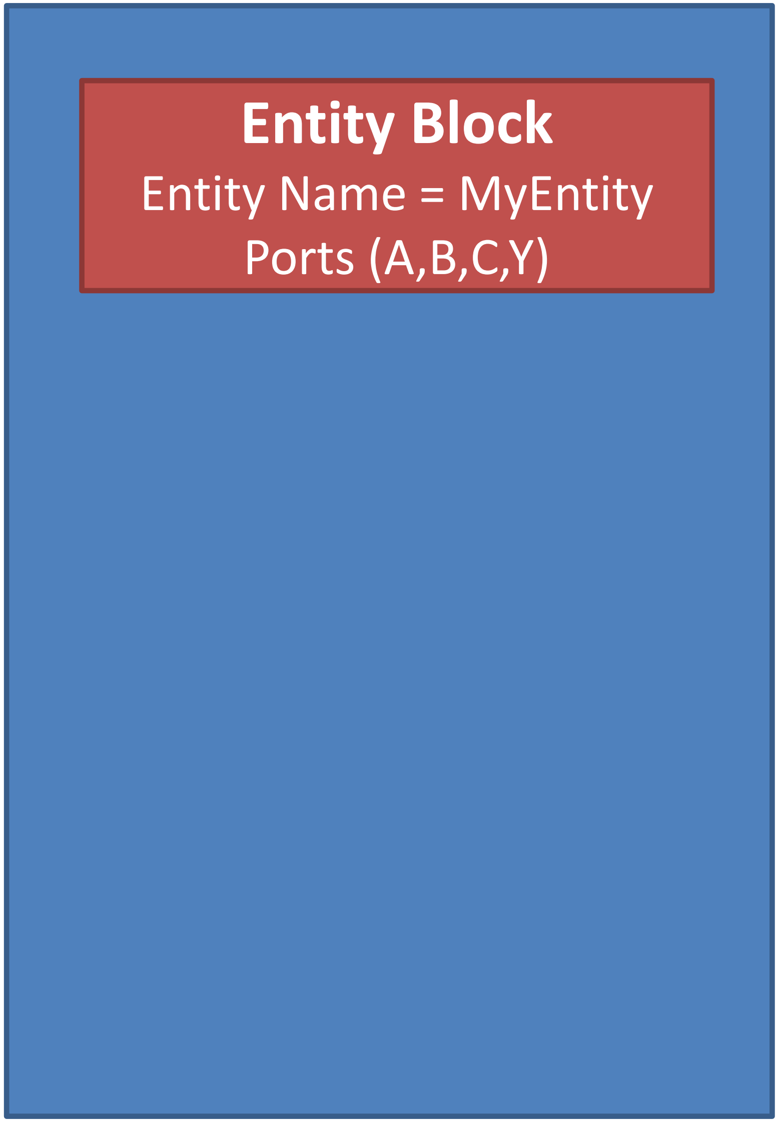
# VHDL File Structure



**Entity Block**

Entity Name = MyEntity

# VHDL File Structure

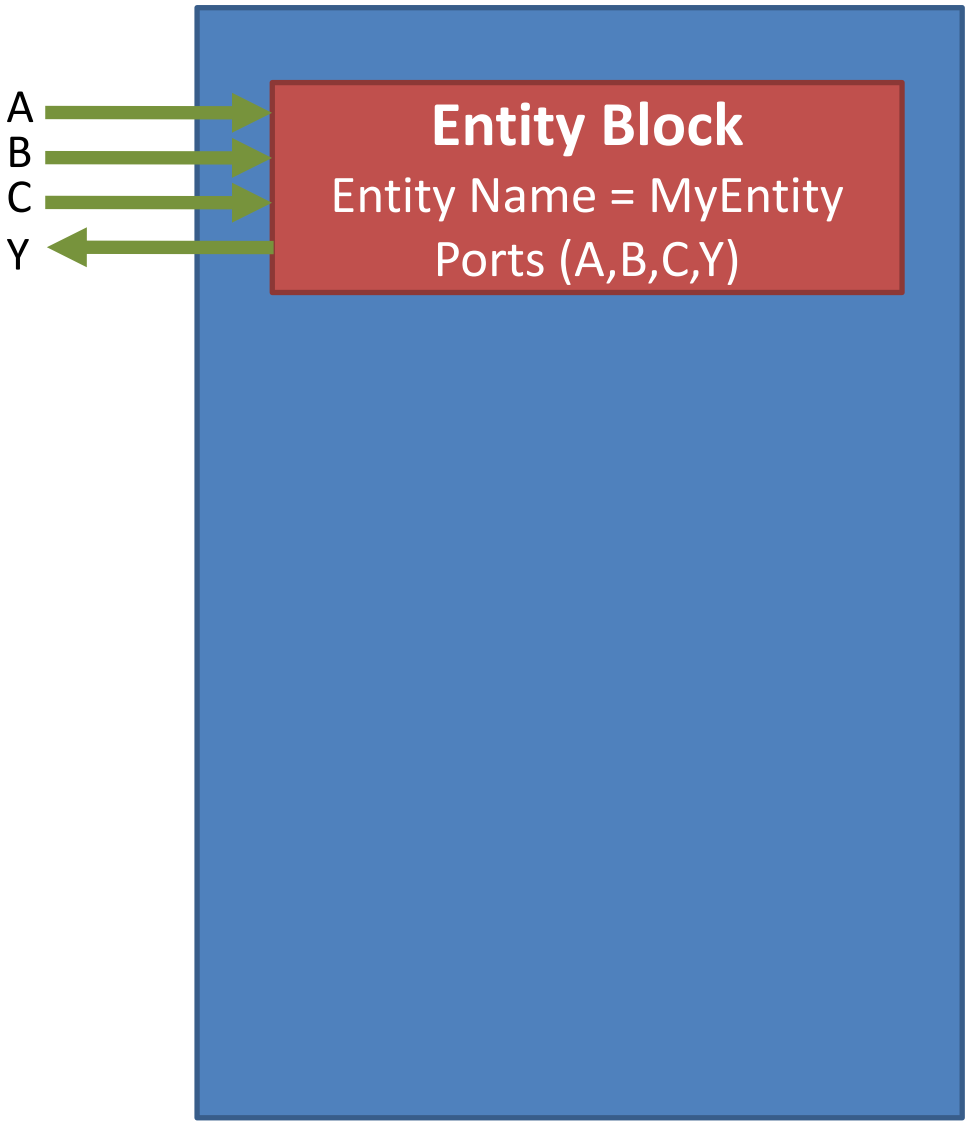


**Entity Block**

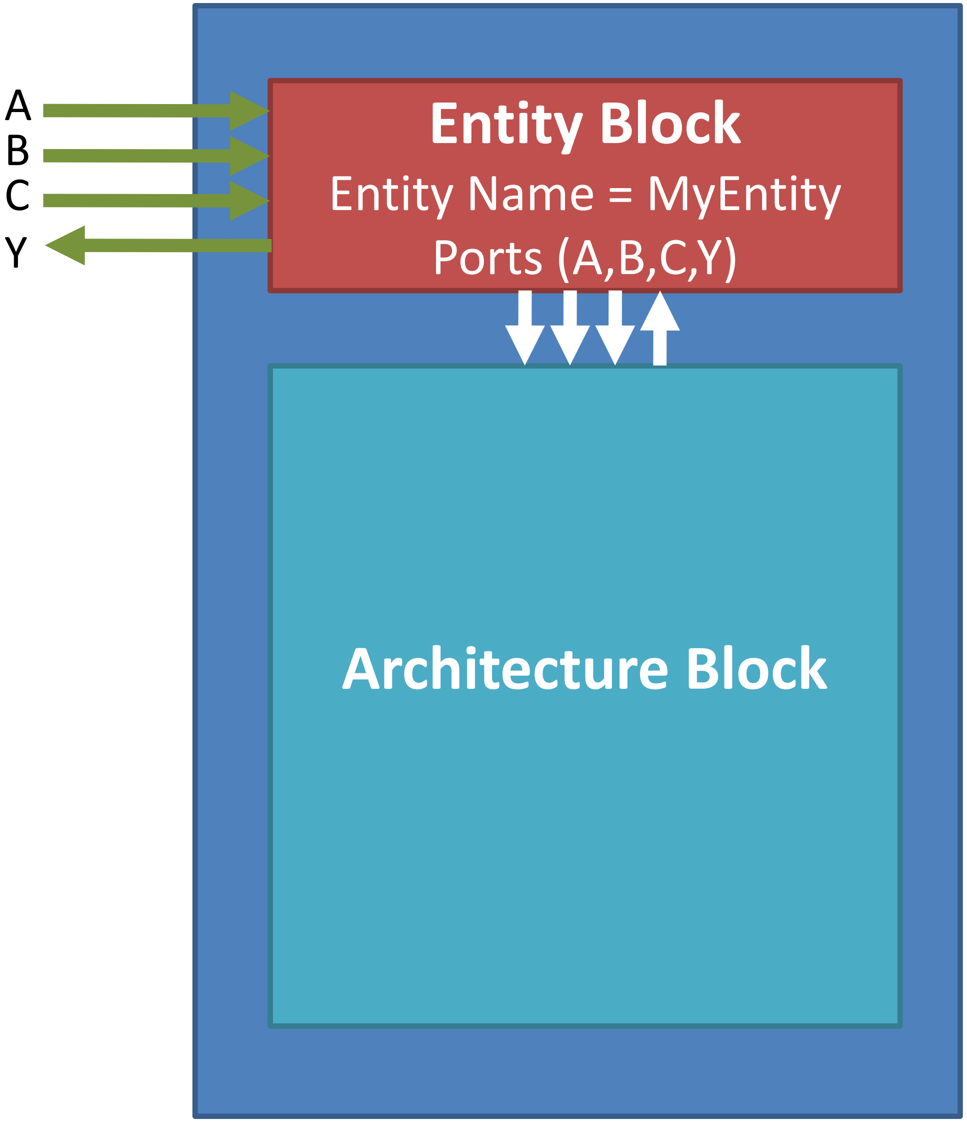
Entity Name = MyEntity

Ports (A,B,C,Y)

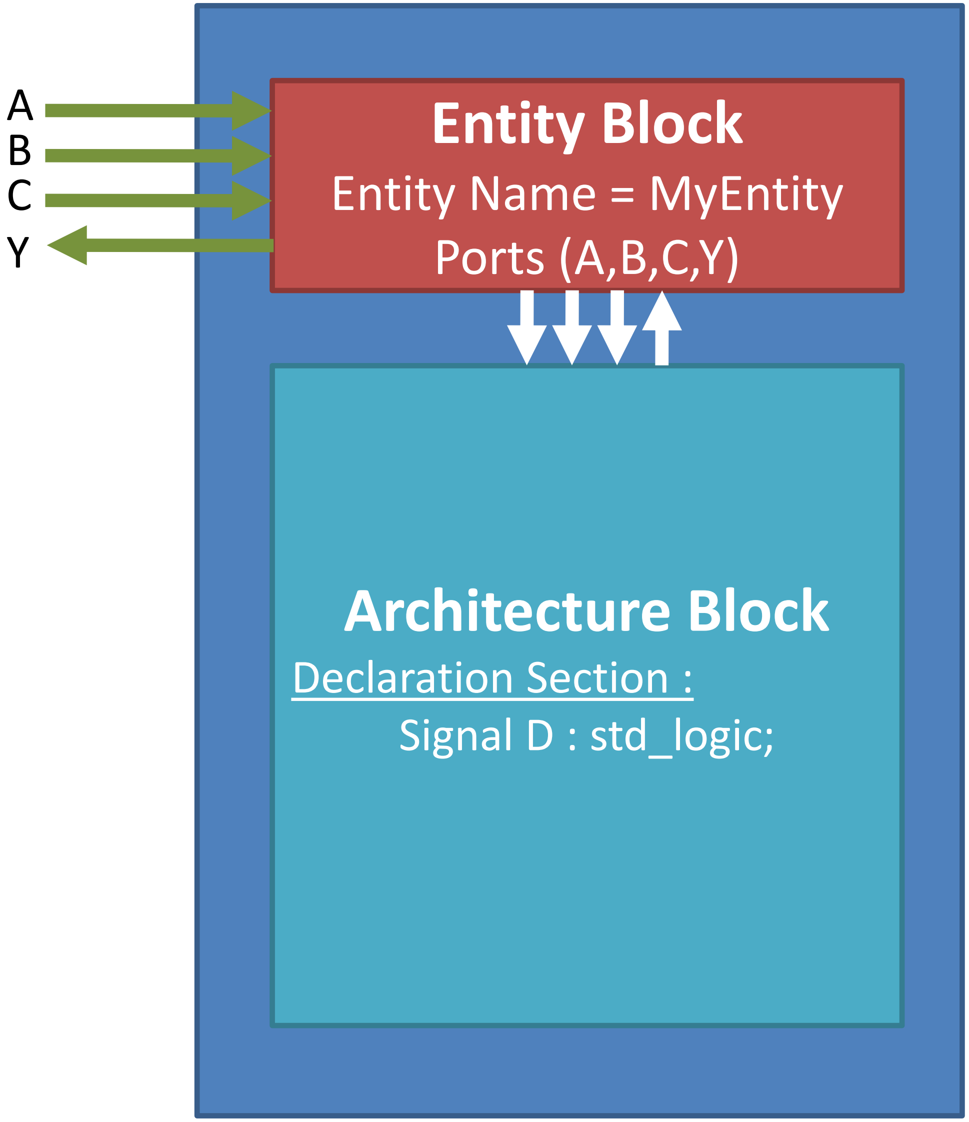
# VHDL File Structure



# VHDL File Structure



# VHDL File Structure



**Entity Block**

Entity Name = MyEntity

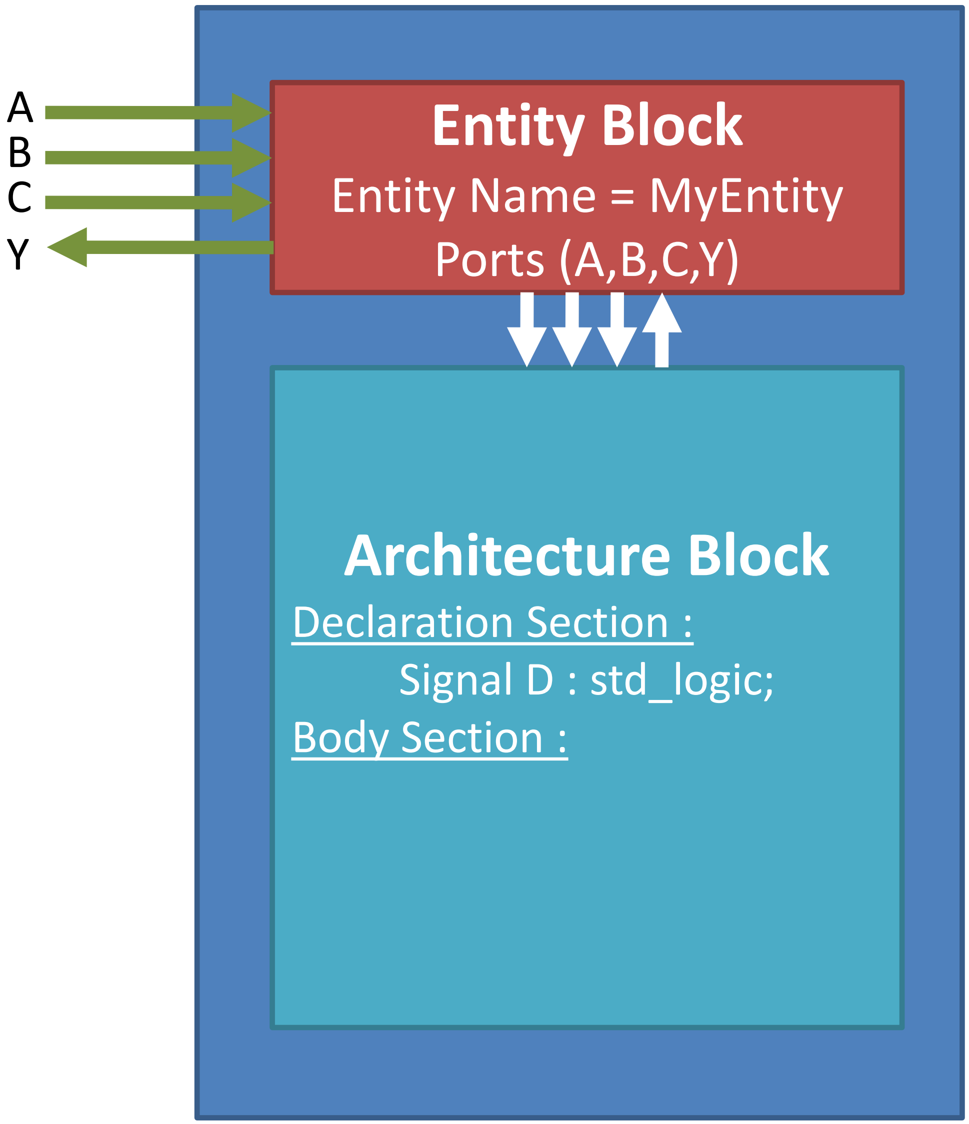
Ports (A,B,C,Y)

**Architecture Block**

Declaration Section :

Signal D : std_logic;

# VHDL File Structure



A

B

C

Y

**Entity Block**

Entity Name = MyEntity

Ports (A,B,C,Y)

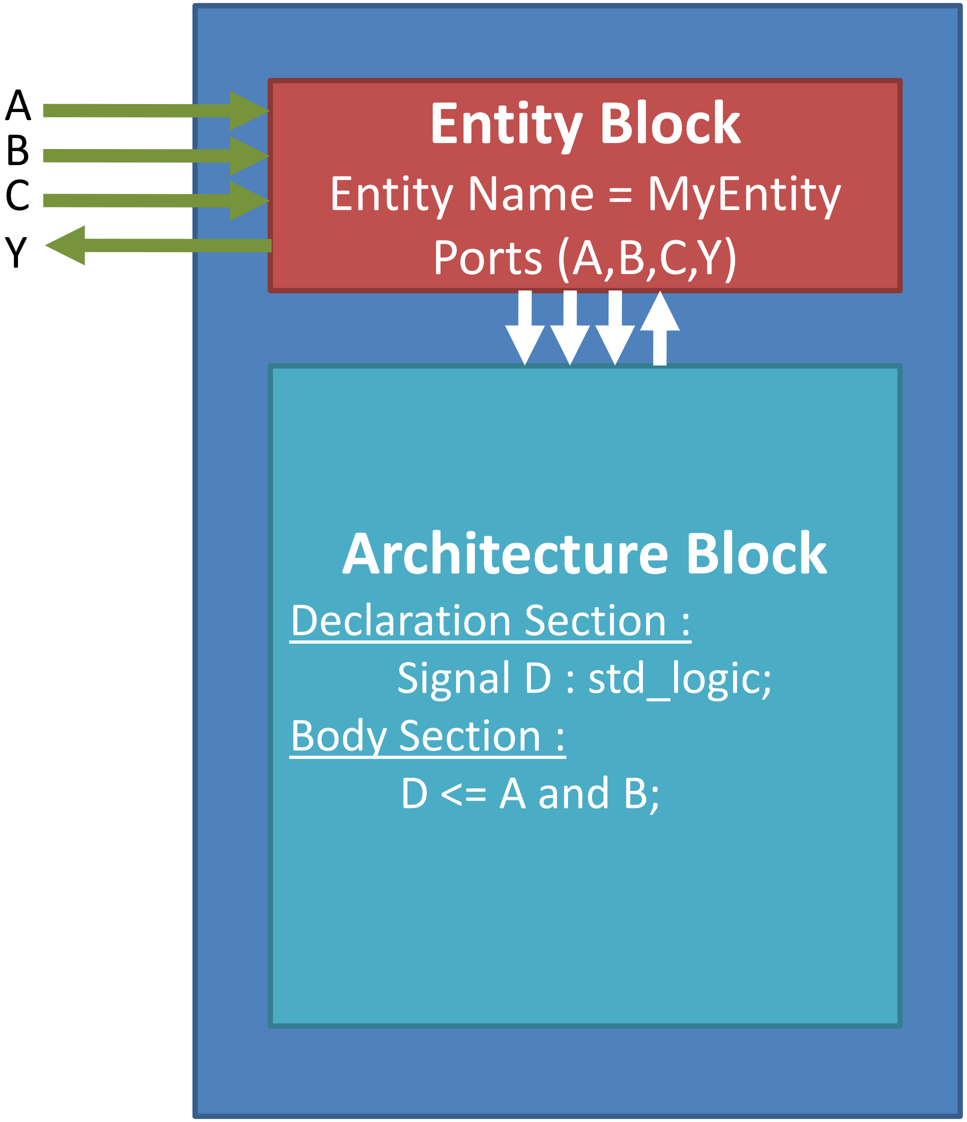
**Architecture Block**

Declaration Section :

Signal D : std_logic;

Body Section :

# VHDL File Structure



**Entity Block**

Entity Name = MyEntity

Ports (A,B,C,Y)

**Architecture Block**

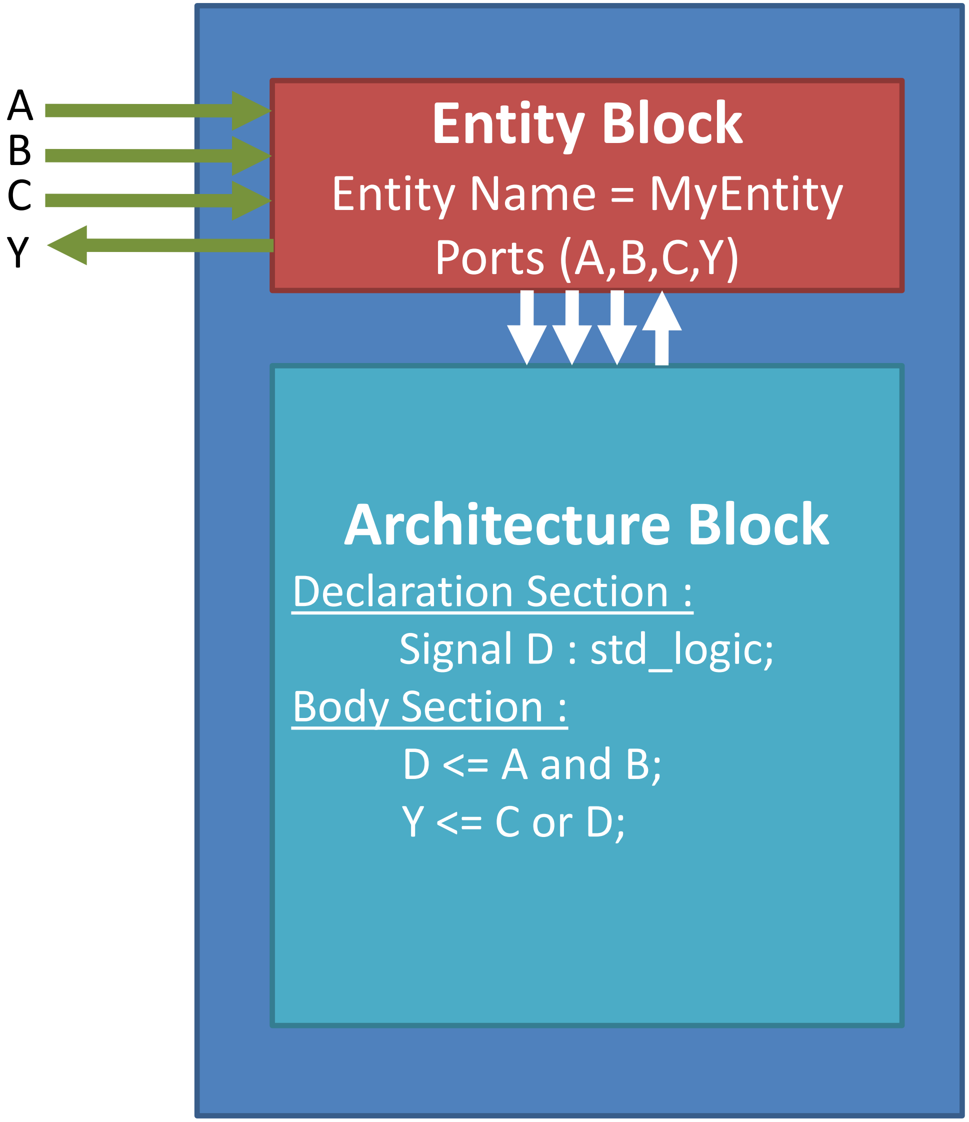
Declaration Section :

Signal D : std_logic;

Body Section :

D <= A and B;

# VHDL File Structure



**Entity Block**

Entity Name = MyEntity

Ports (A,B,C,Y)

**Architecture Block**

Declaration Section :

```
        Signal D : std_logic;
```

Body Section :

```
        D <= A and B;
        Y <= C or D;
```

# Entity Code

```
entity MyEntity is



end entity;
```

# Entity Code

```vhdl
entity MyEntity is
    port (



    );
end entity;
```

# Entity Code

```
entity MyEntity is
    port (
            A : in std_logic;
            B : in std_logic;
            C : in std_logic;
            Y : out std_logic
    );
end entity;
```

# Entity Code

```
entity MyEntity is
    port (
        A : in std_logic;
        B : in std_logic;
        C : in std_logic;
        Y : out std_logic
    );
end entity;
```

# Architecture Code

```
architecture MyArchitecture of MyEntity is

        -- Section 1

begin

        -- Section 2



end architecture;
```

# Architecture Code

```
architecture MyArchitecture of MyEntity is

    signal D  : std_logic;

begin




end architecture;
```

# Architecture Code

```
architecture MyArchitecture of MyEntity is

        signal D  : std_logic;

begin

        D <= A and B;



end architecture;
```

# Architecture Code

```
architecture MyArchitecture of MyEntity is
        signal D  : std_logic;
begin
        D <= A and B;
        Y <= C or D;
end architecture;
```

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

entity MyEntity is
    port (
          A : in std_logic;
                B : in std_logic;
                C : in std_logic;
                Y : out std_logic
    );
end entity;

architecture MyArchitecture of MyEntity is
        signal D  : std_logic;
begin
        D <= A and B;
        Y <= C or D;
end architecture;
```

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

entity MyEntity is
    port (
           A : in std_logic;
                  B : in std_logic;
                  C : in std_logic;
                  Y : out std_logic
    );
end entity;

architecture MyArchitecture of MyEntity is
         signal D  : std_logic;
begin
         D <= A and B;
         Y <= C or D;
end architecture;
```

# Summary

- Overview of a VHDL file
- Entity Declaration
    - Define Entity name
    - Port Declaration
- Architecture Declaration
    - Declarative Part Of Architecture.
    - Body of Architecture.

# Summary

- Overview of a VHDL file
- Entity Declaration
    - Define Entity name
    - Port Declaration
- Architecture Declaration
    - Declarative Part Of Architecture.
    - Body of Architecture.

# The VHDL Process Block

# Process Block

- Written inside the Architecture Body
- Can have multiple process blocks

# Process Block Syntax

```
process_name : process (sensitivity_list)
        -- Declarative Section
begin
        -- Body Section
end process;
```

# Process Block Syntax

```
process_name : process (sensitivity_list)
        -- Declarative Section
begin
        -- Body Section
end process;
```

# Process Block – Sensitivity List

- Is a list of signals

# Process Block – Sensitivity List

- Is a list of signals
- The process waits for one of the signals in this list to CHANGE state before running.

# Process Block – Sensitivity List

- Is a list of signals
- The process waits for one of the signals in this list to CHANGE state before running.
- Process Block runs only when a signal in the sensitivity list changes state.

# Process Block – Sensitivity List

- Is a list of signals
- The process waits for one of the signals in this list to CHANGE state before running.
- Process Block runs only when a signal in the sensitivity list changes state.
- The process block is run once at time 0.

# Process - Example

```
TestProcess1: process(Sig1)
begin
    Sig2 <= Sig1;
end process;
```

# Process - Example

```
AND_GATE : process(A, B)
begin
   C <= A and B;
end process;
```

# Process - Example

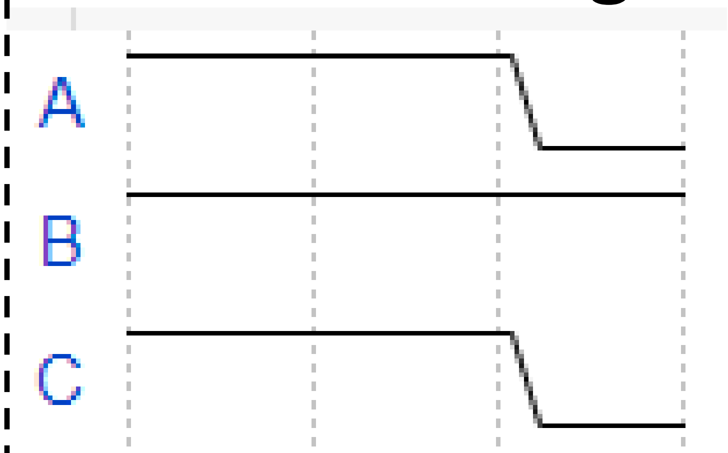
```
AND_GATE : process(A, B)
begin
   C <= A and B;
end process;
```

# Incomplete Sensitivity List

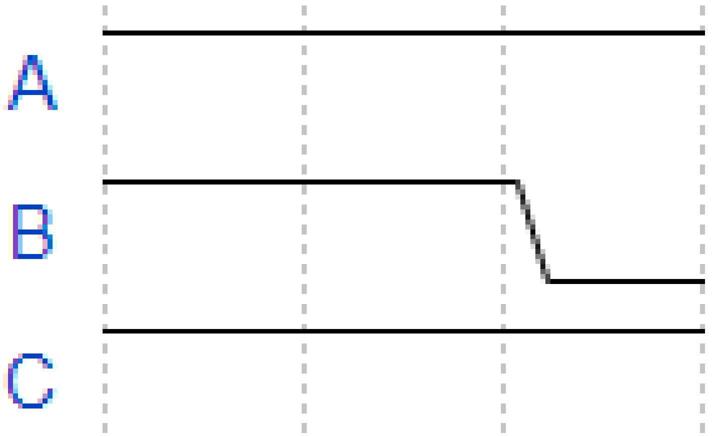
```
AND_GATE : process(A)
begin
   C <= A and B;
end process;
```

Missing signal B

When A Changes



When B Changes

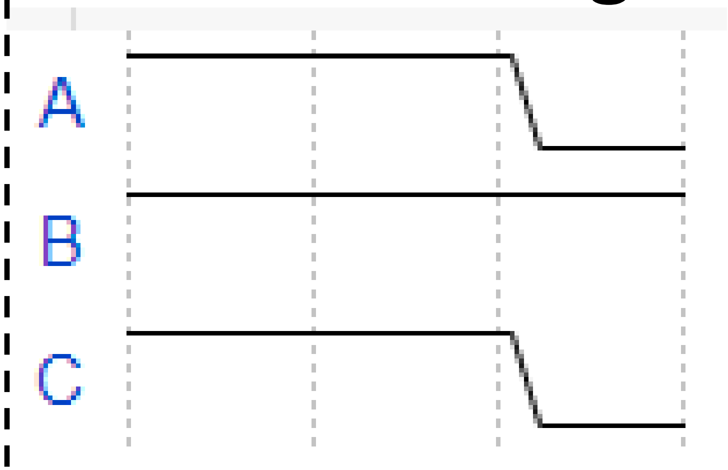


# Incomplete Sensitivity List

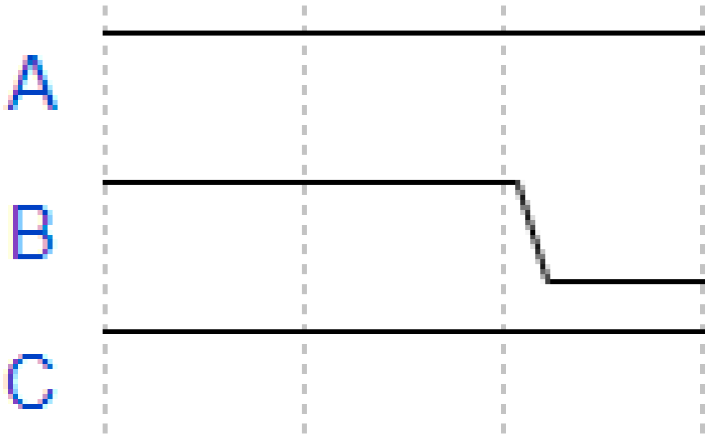
```
AND_GATE : process(A)
begin
   C <= A and B;
end process;
```

Missing signal B

When A Changes



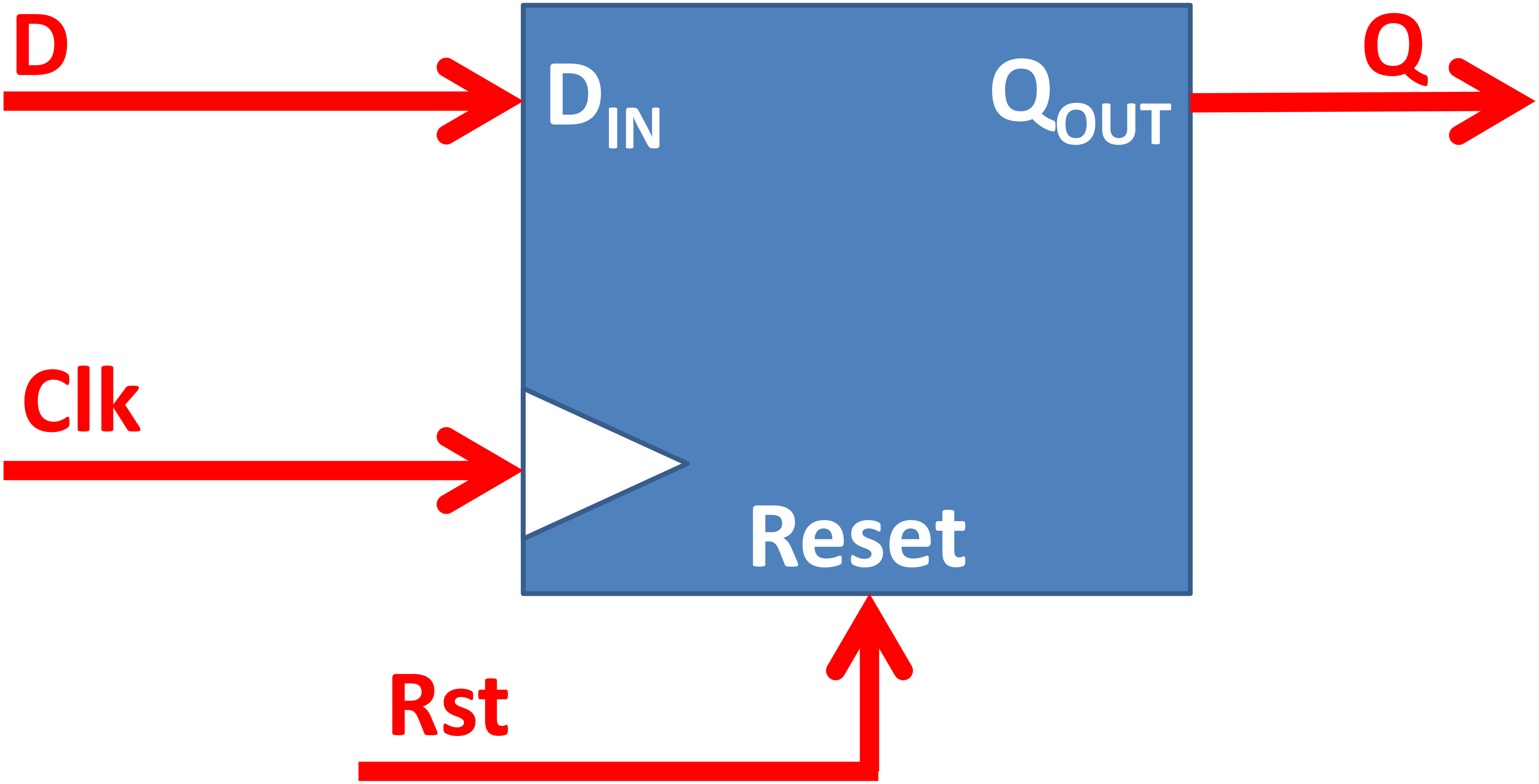
When B Changes



# Registered Process

- Used to implement registers (or flip-flops).
- Sensitivity list can **only** contain :
  - A Clock signal (Compulsory)
  - A Reset signal (Optional)
- Checks for the clock rising or falling edge.

# Flip Flop Circuit



# Registered Process

```
TestProcess : process(Rst, Clk)
begin






end process;
```

# Registered Process

```
TestProcess : process(Rst, Clk)
begin
    if Rst = ‘1’ then





end process;
```

# Registered Process

```
TestProcess : process(Rst, Clk)
begin
   if Rst = ‘1’ then
         Q <= ‘0’;




end process;
```

# Registered Process

```
TestProcess : process(Rst, Clk)
begin
    if Rst = ‘1’ then
        Q <= ‘0’;
    elsif rising_edge(Clk) then



end process;
```

# Registered Process

```
TestProcess : process(Rst, Clk)
begin
    if Rst = ‘1’ then
        Q <= ‘0’;
    elsif rising_edge(Clk) then
        Q <= D;

end process;
```

# Registered Process

```
TestProcess : process(Rst, Clk)
begin
    if Rst = ‘1’ then
        Q <= ‘0’;
    elsif rising_edge(Clk) then
        Q <= D;
    end if
end process;
```

# More on Registered Processes

- Signals driven by a registered process are always outputs of flip flops.

# More on Registered Processes

- Signals driven by a registered process are always outputs of flip flops.

```
TestProcess2 : process(Reset, Clock)
begin
    if Reset = ‘1’ then
       Out1 <= ‘0’;
       Out2 <= ‘0’;
       Out3 <= ‘0’;
   elsif rising_edge(Clock) then
       Out1 <= A;
       Out2 <= A and B;
       Out3 <= A or B;
    end if;
end process;
```

Clocked Section

A signal assigned here will create a flip flop.

# More on Registered Processes

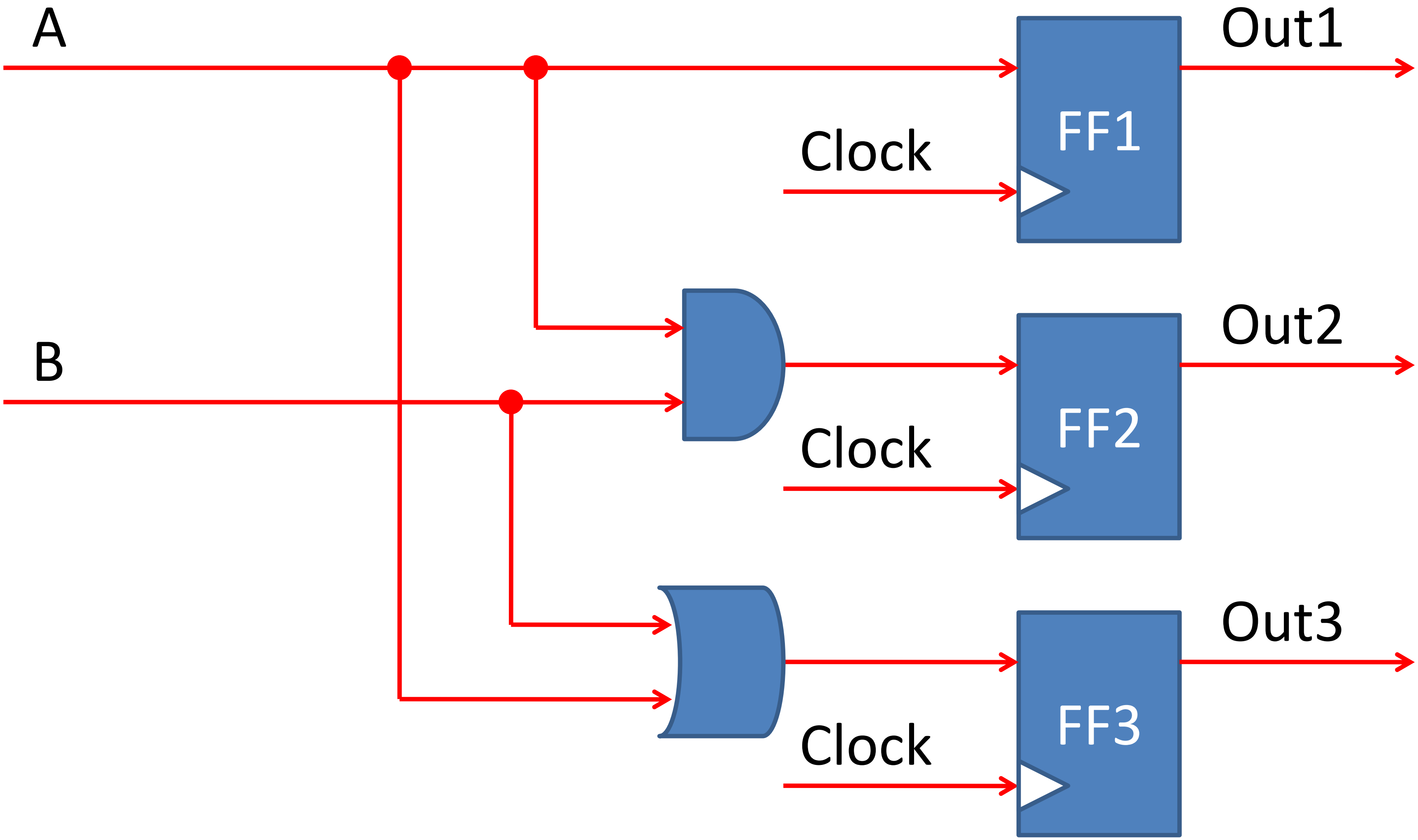
- Signals driven by a registered process are always outputs of flip flops.

```
TestProcess2 : process(Reset, Clock)
begin
    if Reset = ‘1’ then
       Out1 <= ‘0’;
       Out2 <= ‘0’;
       Out3 <= ‘0’;
   elsif rising_edge(Clock) then
       Out1 <= A;
       Out2 <= A and B;
       Out3 <= A or B;
    end if;
end process;
```

<u>Clocked Section</u>

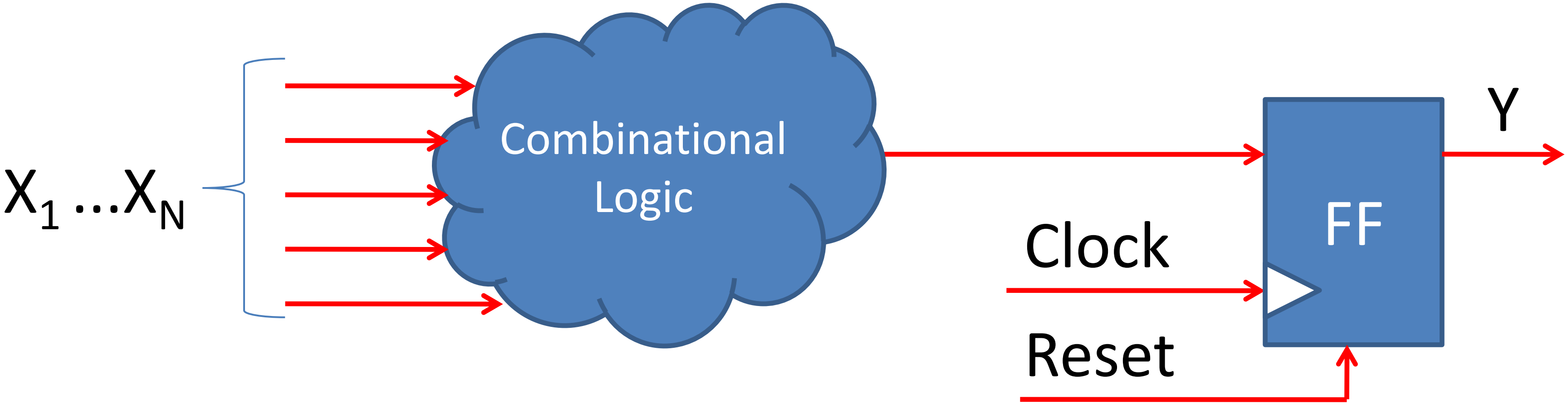
A signal assigned here will create a flip flop.

# More on Registered Processes



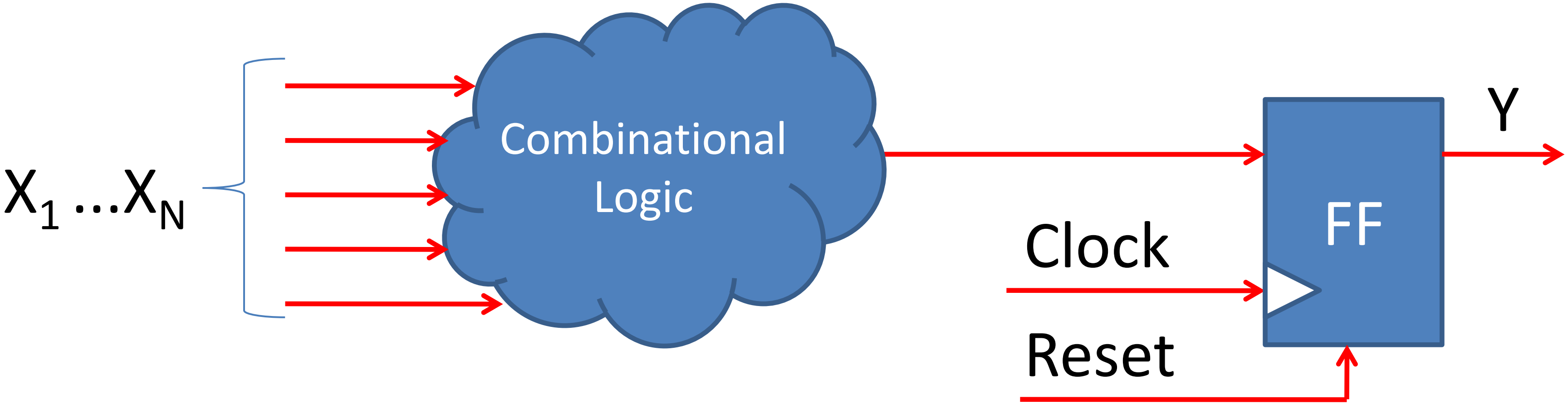
# More on Registered Processes

```
TestProcessGeneral : process(Reset, Clock)
begin
    if Reset = ‘1’ then
        Y <= ‘0’;
elsif rising_edge(Clock) then
        Y <= Function(X1, X2, X3,… XN);
    end if;
end process;
```



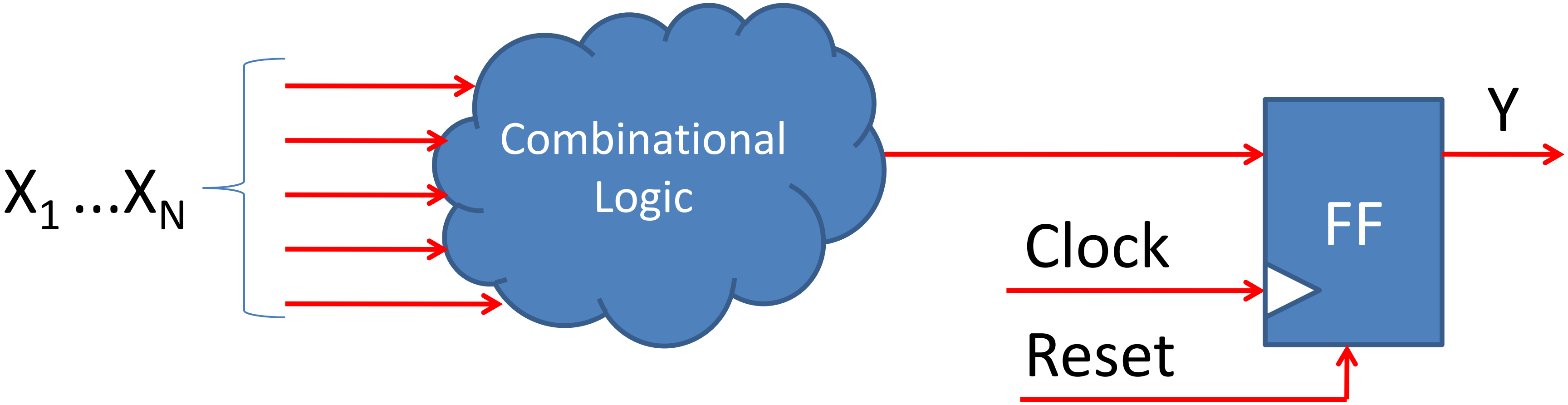
# More on Registered Processes

```
TestProcessGeneral : process(Reset, Clock)
begin
    if Reset = ‘1’ then
       Y <= ‘0’;
elsif rising_edge(Clock) then
       Y <= Function(X1, X2, X3,... XN);
    end if;
end process;
```



# More on Registered Processes

```
TestProcessGeneral : process(Reset, Clock)
begin
    if Reset = ‘1’ then
        Y <= ‘0’;
elsif rising_edge(Clock) then
        Y <= Function(X1, X2, X3,… XN);
    end if;
end process;
```



# Comments

- **rising_edge()** -> Is a function found in the std_logic_1164 Package.

- **falling_edge()**

# Comments

```
if Condition1  then
    -- Statement Group 1
elsif Condition2 then
    -- Statement Group 2
else
    -- Statement Group 3
end if;
```

# More on the IF Statement

- If statements must be written inside process block

```
if Condition1  then
    -- Statement Group 1
else
    -- Statement Group 2
end if;
```

```
if Condition2  then
    -- Statement Group 3
end if;
```

# Sensitivity List

- What goes in a sensitivity list?

| Combinational Process | Registered Process |
|---|---|
| All signals read within the process. | Only Clock and reset. |

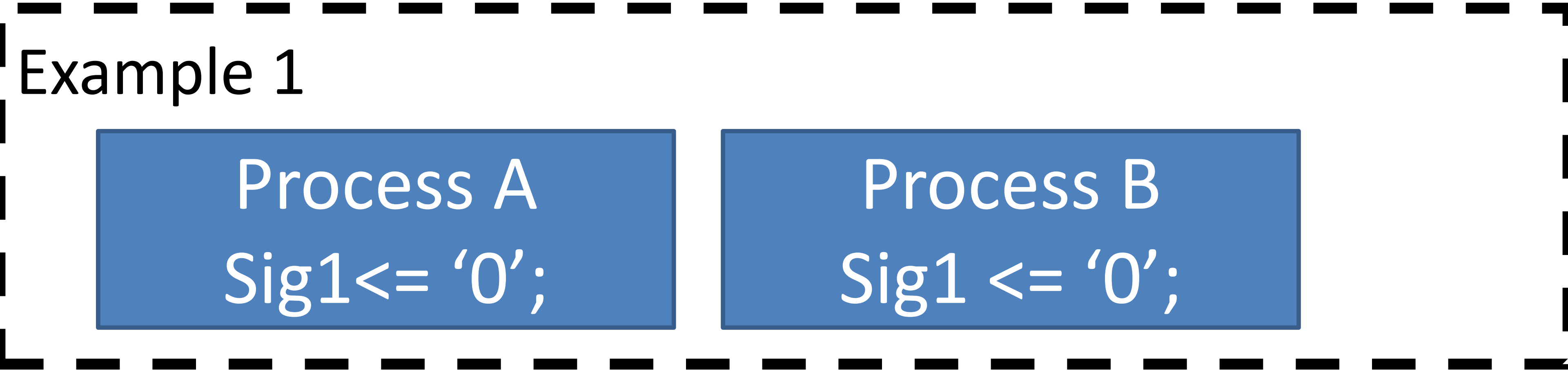
# Sensitivity List

- What goes in a sensitivity list?

| Combinational Process | Registered Process |
|---|---|
| All signals read within the process. | Only Clock and reset. |

- Sensitivity list can be empty:
  - Only used in simulation!
  - Then must have wait statements.
  - Not synthesisable.

# More on Processes



Example 1

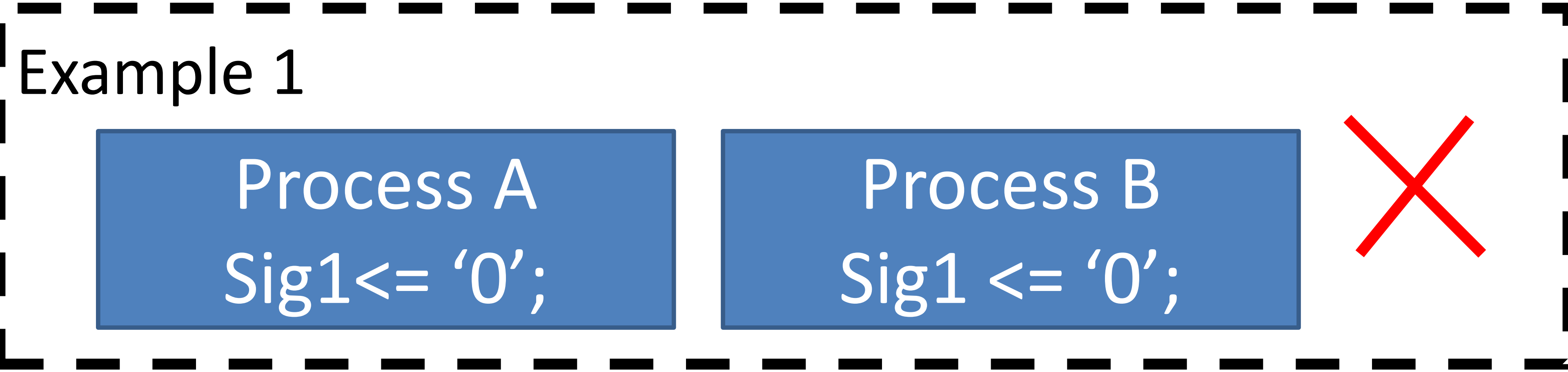
Process A

Sig1<= ‘0’;

Process B

Sig1 <= ‘0’;

# More on Processes



Example 1

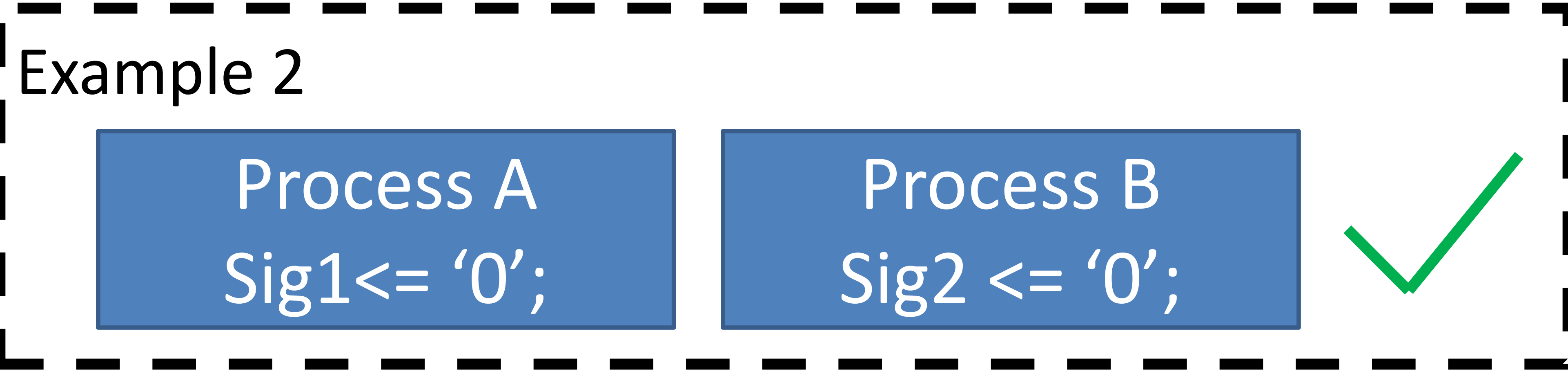
Process A

Sig1<= ‘0’;

Process B

Sig1 <= ‘0’;

# More on Processes



Example 2

Process A

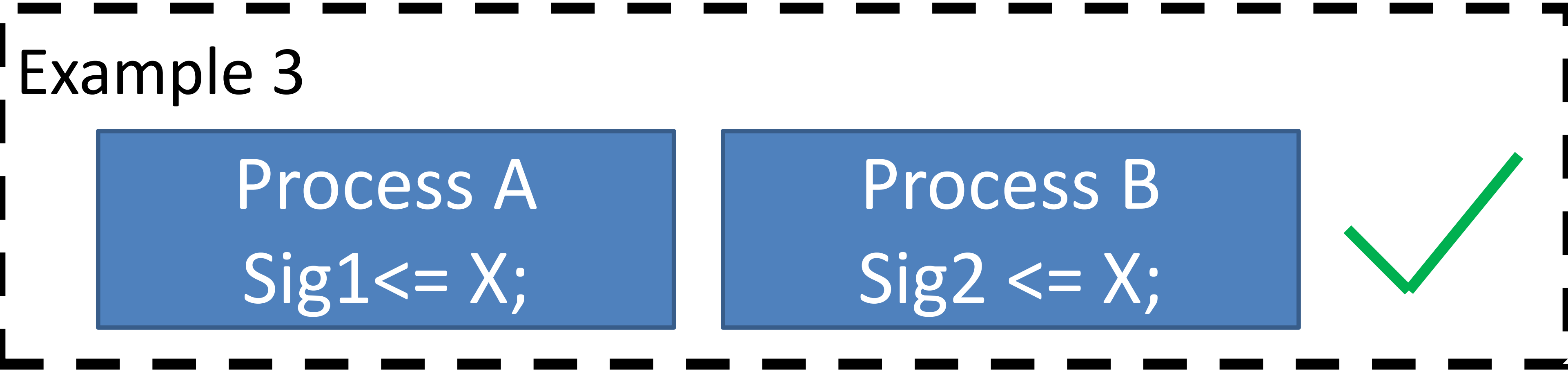
Sig1<= ‘0’;

Process B

Sig2 <= ‘0’;

✓

# More on Processes



Example 3

Process A
Sig1<= X;

Process B
Sig2 <= X;

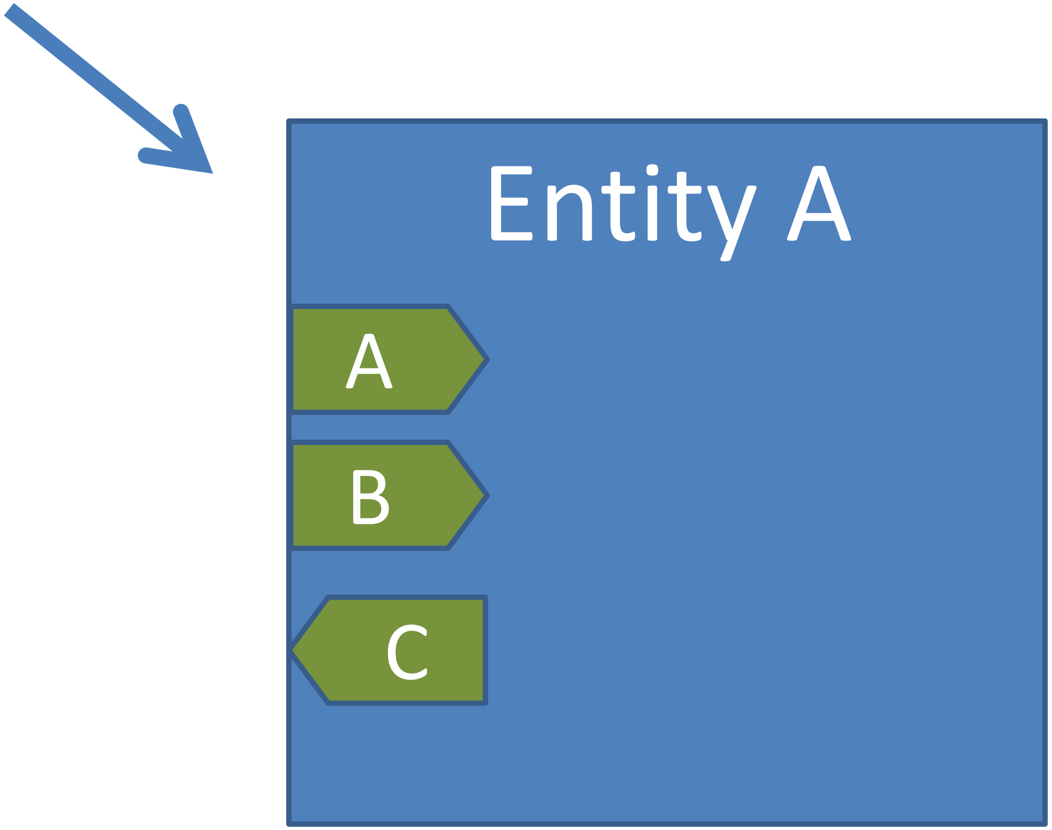
✓

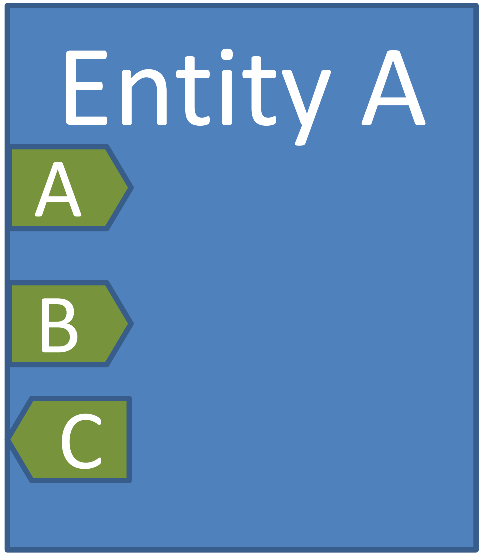
# Summary

- What is a process?
- Combinational & Registered Processes.
- Writing a basic process in VHDL.
- Sensitivity list.
- If statement
- rising_edge() & falling_edge() functions.
- Hardware visualisation of a process.

# Component Instantiation

VHDL File



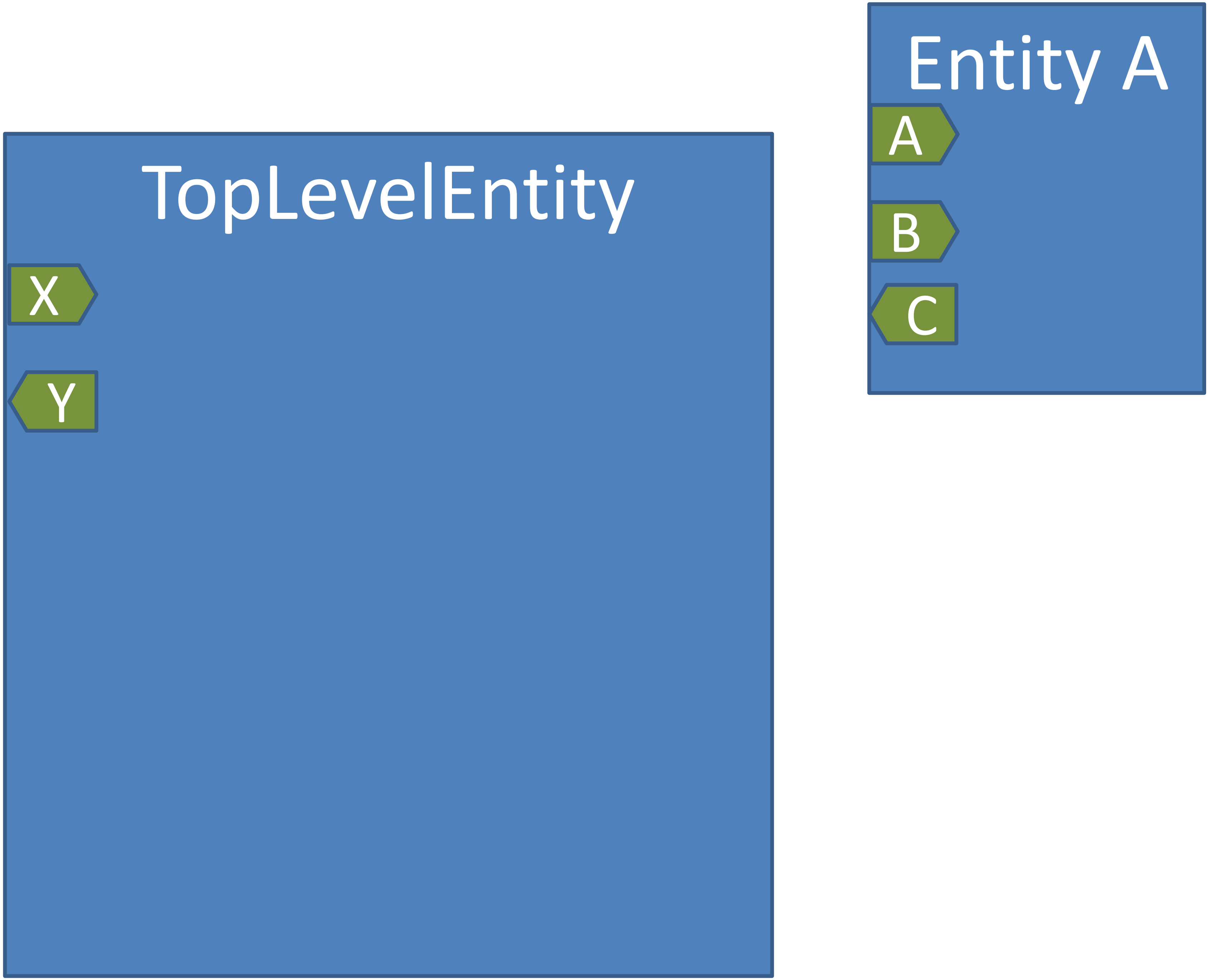


Entity A

A

B

C



TopLevelEntity

X

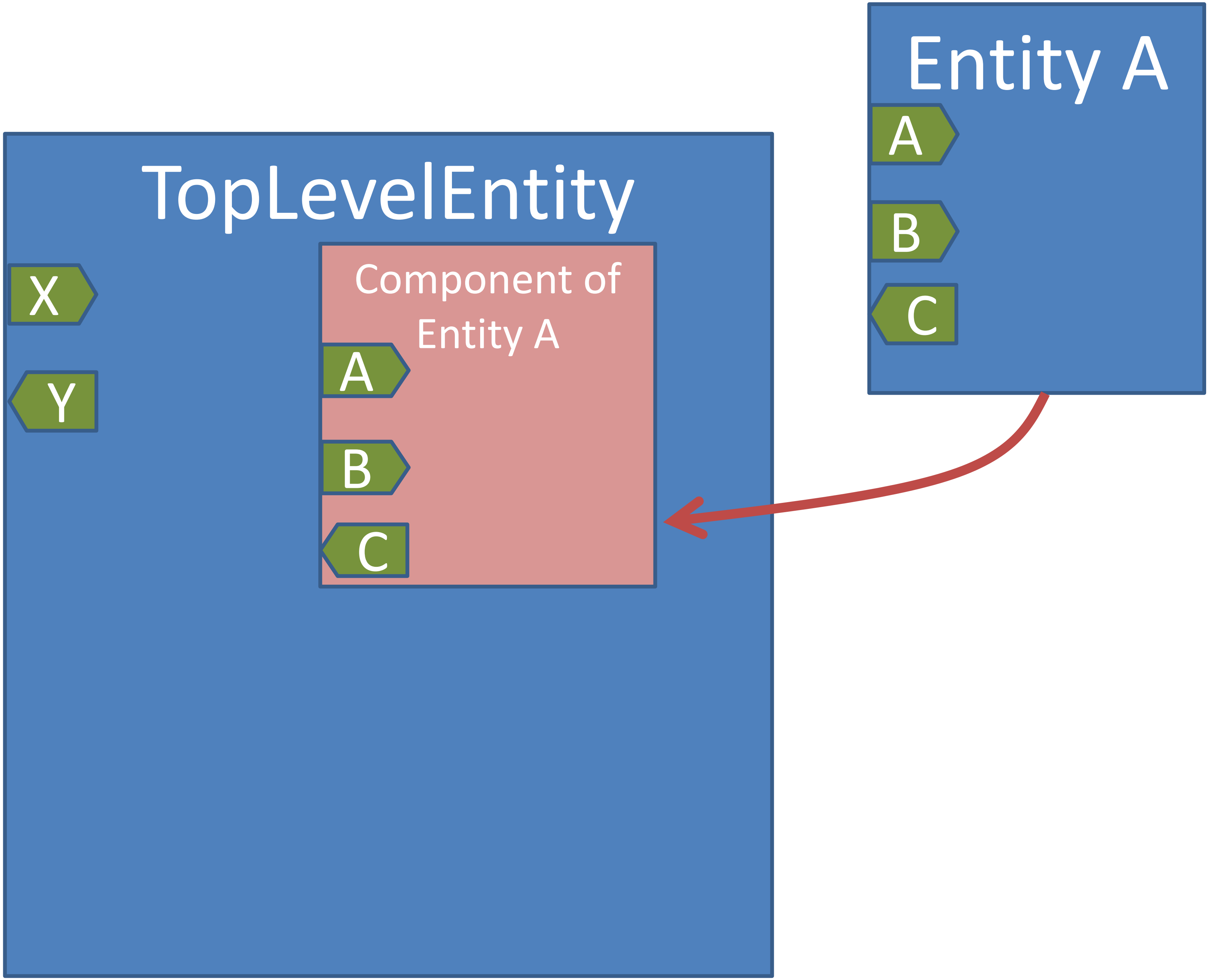
Y

Entity A

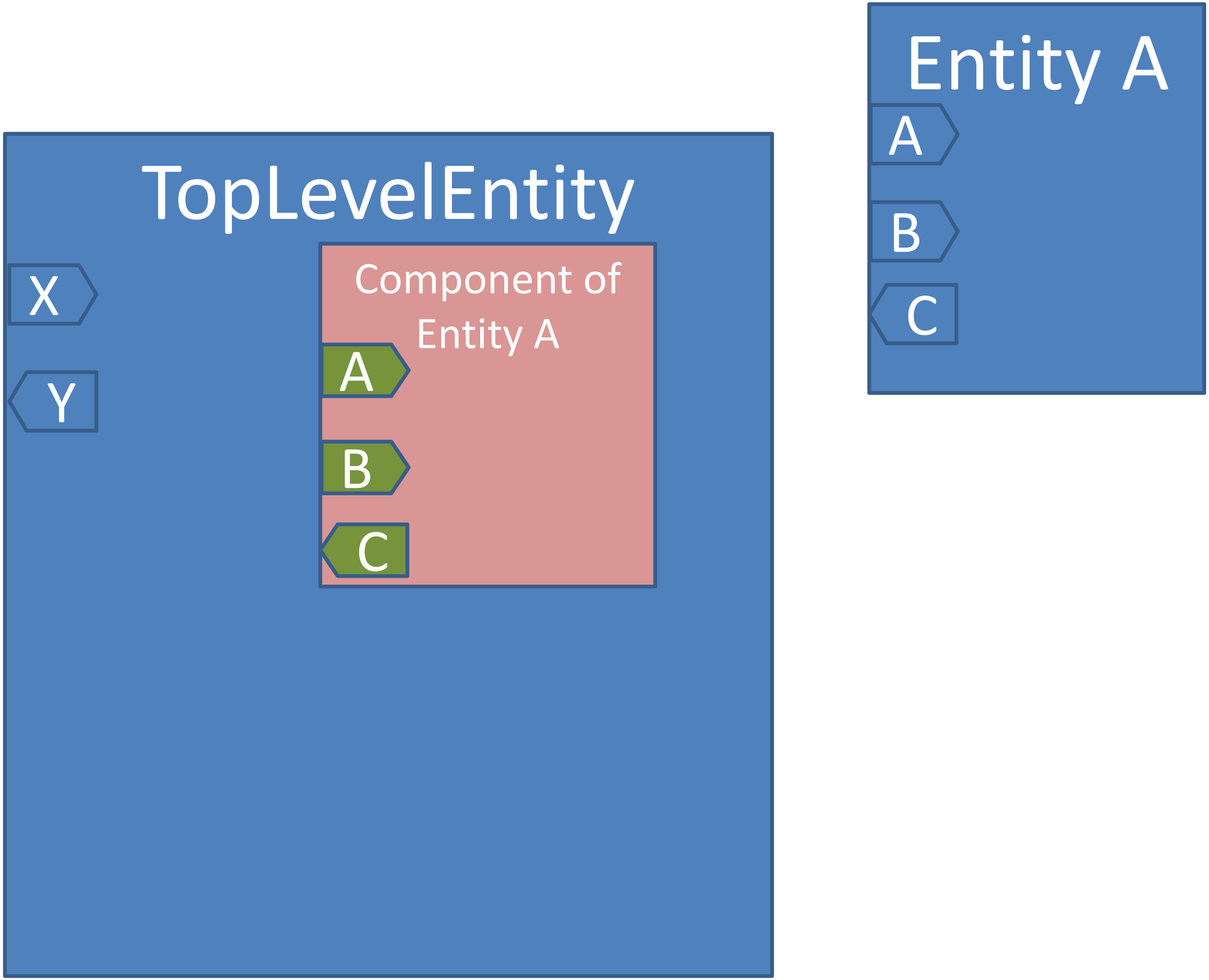
A

B

C



Entity A
A
B
C
TopLevelEntity
X
Y
Component of Entity A
A
B
C



TopLevelEntity

X

Y

Component of Entity A

A

B

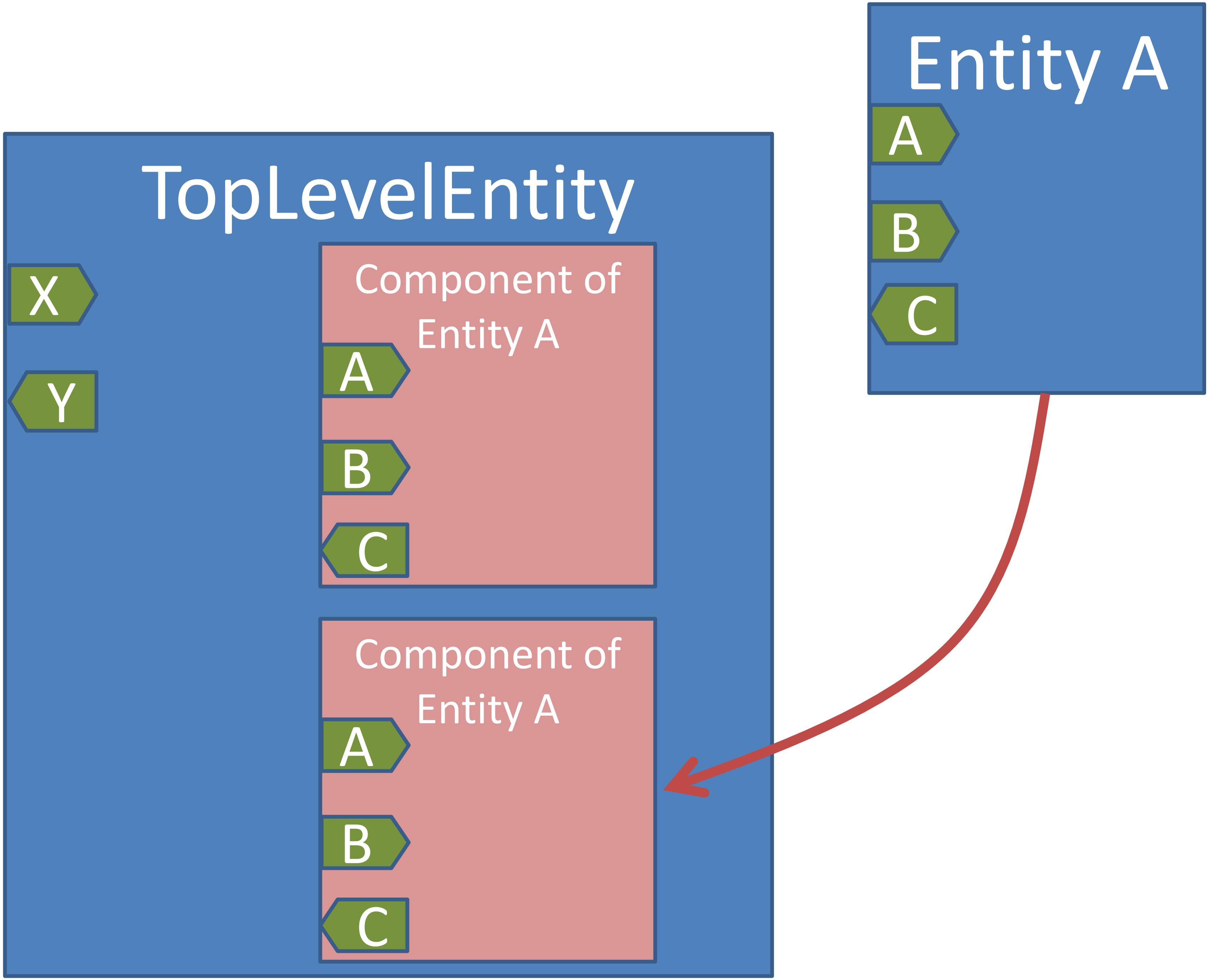
C

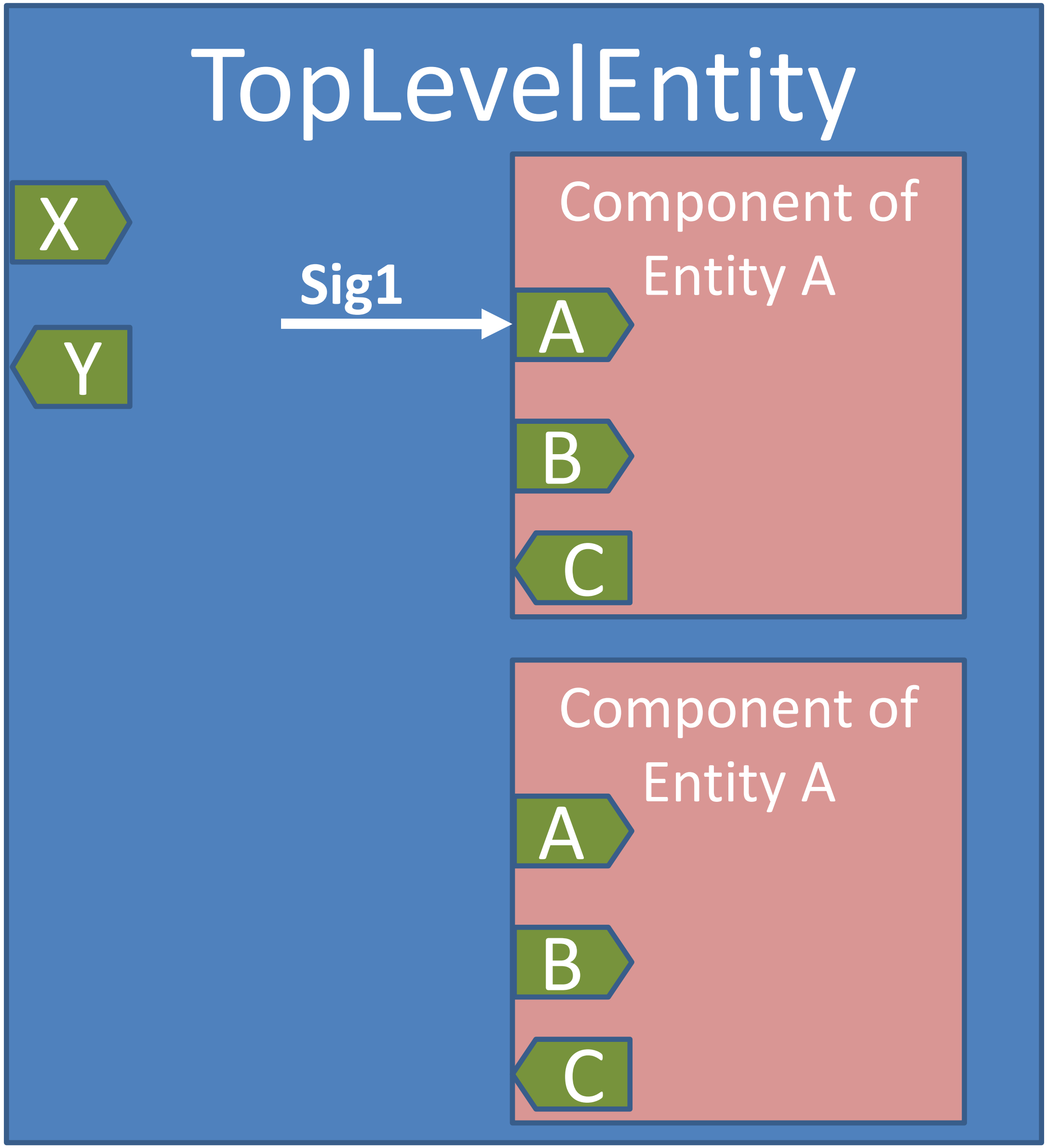
Entity A

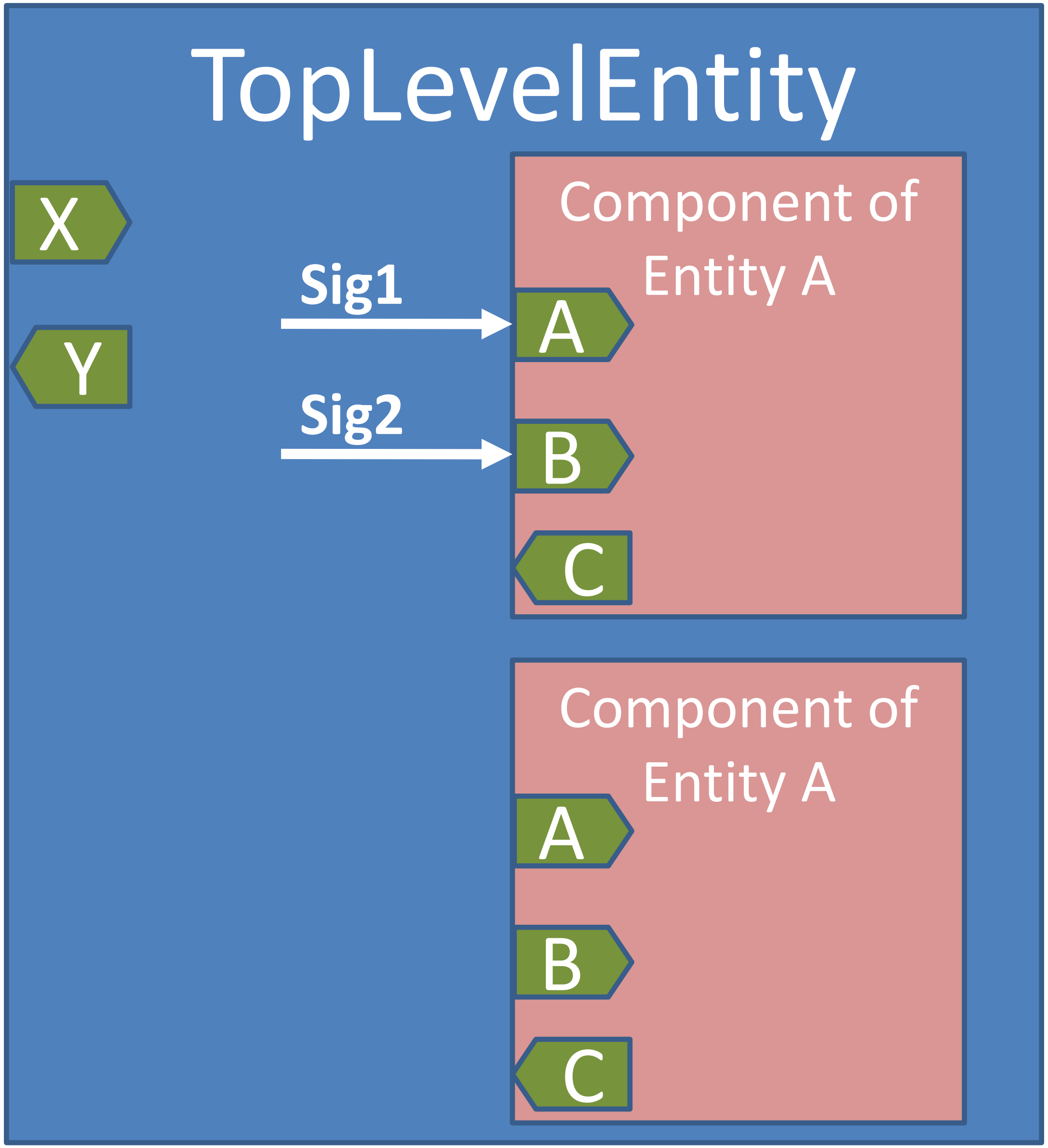
A

B

C


Entity A
A
B
C
TopLevelEntity
X
Y
Component of Entity A
A
B
C
Component of Entity A
A
B
C


TopLevelEntity
X
Y
Sig1
Component of Entity A
A
B
C
Component of Entity A
A
B
C



TopLevelEntity

X

Y

Sig1

Sig2

Component of Entity A

A

B

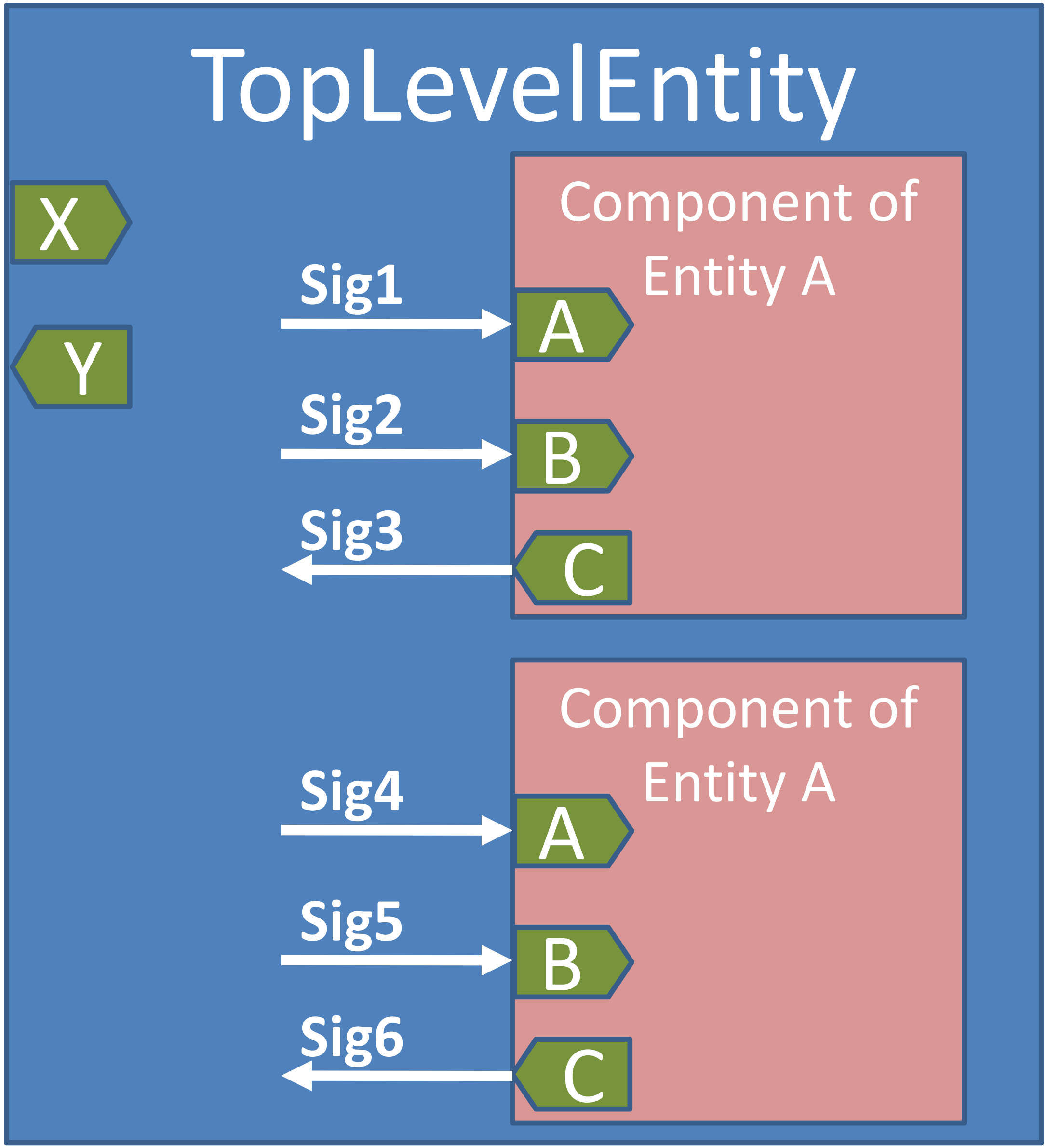
C

Component of Entity A

A

B

C


TopLevelEntity
X
Y
Component of Entity A
Sig1
A
Sig2
B
Sig3
C
Component of Entity A
Sig4
A
Sig5
B
Sig6
C

# VHDL File Template

```
-- Declare your VHDL libraries & packages up here.
entity EntityName is
    port
    (
        -- Declare your modules’s IO ports here!
    );
end entity;

architecture ArchitectureName of EntityName is
    -- Declare your internal signals & constants here
begin
    -- Define your module’s behaviour here!
end architecture;
```

# VHDL Entity A

# VHDL Entity A

```
entity EntityA is




end entity;
```

# VHDL Entity A

```
entity EntityA is
    port
    (
        A : in  integer range 0 to 255;
        B : in  integer range 0 to 255;
        C : out integer range 0 to 511
    );
end entity;
```

# VHDL Entity A

```vhdl
entity EntityA is
    port
    (
        A : in  integer range 0 to 255;
        B : in  integer range 0 to 255;
        C : out integer range 0 to 511
    );
end entity;

architecture rtl of EntityA is
```

# VHDL Entity A

```
entity EntityA is
    port
    (
        A : in  integer range 0 to 255;
        B : in  integer range 0 to 255;
        C : out integer range 0 to 511
    );
end entity;

architecture rtl of EntityA is
begin
```

# VHDL Entity A

```vhdl
entity EntityA is
    port
    (
        A : in   integer range 0 to 255;
        B : in   integer range 0 to 255;
        C : out integer range 0 to 511
    );
end entity;

architecture rtl of EntityA is
begin
        C <= A + B; -- Perform the addition operation.
end architecture;
```

```vhdl
entity TopLevelEntity is
   port
   (
      X : in   integer range 0 to 255;
      Y : out integer range 0 to 255
   );
end entity;
```

```
entity TopLevelEntity is
    port
    (
        X : in   integer range 0 to 255;
        Y : out integer range 0 to 255
    );
end entity;

architecture rtl of TopLevelEntity is
```

```vhdl
entity TopLevelEntity is
   port
   (
      X : in   integer range 0 to 255;
      Y : out integer range 0 to 255
   );
end entity;

architecture rtl of TopLevelEntity is

        signal Sig1, Sig2, Sig4, Sig5 : integer range 0 to 255;
        signal Sig3, Sig6 : integer range 0 to 511;
```

```vhdl
entity TopLevelEntity is
  port
  (
    X : in   integer range 0 to 255;
    Y : out integer range 0 to 255
  );
end entity;

architecture rtl of TopLevelEntity is

     signal Sig1, Sig2, Sig4, Sig5 : integer range 0 to 255;
      signal Sig3, Sig6 : integer range 0 to 511;

      component EntityA is
      port
      (
          A : in   integer range 0 to 255;
          B : in   integer range 0 to 255;
          C : out integer range 0 to 511
       );
      end component ;
```

```
entity TopLevelEntity is
   port
   (
      X : in   integer range 0 to 255;
      Y : out integer range 0 to 255
   );
end entity;

architecture rtl of TopLevelEntity is

        signal Sig1, Sig2, Sig4, Sig5 : integer range 0 to 255;
        signal Sig3, Sig6 : integer range 0 to 511;

         component EntityA is
         port
         (
               A : in   integer range 0 to 255;
               B : in   integer range 0 to 255;
               C : out integer range 0 to 511
          );
         end component ;

begin
```

```
entity TopLevelEntity is
   port
   (
      X : in   integer range 0 to 255;
      Y : out integer range 0 to 255
   );
end entity;

architecture rtl of TopLevelEntity is

        signal Sig1, Sig2, Sig4, Sig5 : integer range 0 to 255;
        signal Sig3, Sig6 : integer range 0 to 511;

         component EntityA is
         port
         (
               A : in   integer range 0 to 255;
               B : in   integer range 0 to 255;
               C : out integer range 0 to 511
          );
         end component ;

begin
         Instance1
```

```
entity TopLevelEntity is
   port
   (
      X : in   integer range 0 to 255;
      Y : out integer range 0 to 255
   );
end entity;

architecture rtl of TopLevelEntity is

        signal Sig1, Sig2, Sig4, Sig5 : integer range 0 to 255;
        signal Sig3, Sig6 : integer range 0 to 511;

         component EntityA is
         port
         (
               A : in   integer range 0 to 255;
               B : in   integer range 0 to 255;
               C : out integer range 0 to 511
          );
         end component ;

begin
         Instance1 : EntityA
```

```vhdl
entity TopLevelEntity is
   port
   (
      X : in   integer range 0 to 255;
      Y : out integer range 0 to 255
   );
end entity;

architecture rtl of TopLevelEntity is

        signal Sig1, Sig2, Sig4, Sig5 : integer range 0 to 255;
        signal Sig3, Sig6 : integer range 0 to 511;

         component EntityA is
         port
         (
               A : in   integer range 0 to 255;
               B : in   integer range 0 to 255;
               C : out integer range 0 to 511
          );
         end component ;

begin
         Instance1 : EntityA
         port map
```

```
entity TopLevelEntity is
   port
   (
      X : in   integer range 0 to 255;
      Y : out integer range 0 to 255
   );
end entity;

architecture rtl of TopLevelEntity is

        signal Sig1, Sig2, Sig4, Sig5 : integer range 0 to 255;
        signal Sig3, Sig6 : integer range 0 to 511;

         component EntityA is
         port
         (
               A : in   integer range 0 to 255;
               B : in   integer range 0 to 255;
               C : out integer range 0 to 511
          );
         end component ;

begin
         Instance1 : EntityA
         port map
         (
               A
               B
               C
         );
```

```
entity TopLevelEntity is
   port
   (
      X : in   integer range 0 to 255;
      Y : out integer range 0 to 255
   );
end entity;

architecture rtl of TopLevelEntity is

        signal Sig1, Sig2, Sig4, Sig5 : integer range 0 to 255;
        signal Sig3, Sig6 : integer range 0 to 511;

         component EntityA is
         port
         (
               A : in   integer range 0 to 255;
               B : in   integer range 0 to 255;
               C : out integer range 0 to 511
          );
         end component ;

begin
         Instance1 : EntityA
         port map
         (
               A => Sig1,
               B => Sig2,
               C => Sig3
         );
```

```vhdl
entity TopLevelEntity is
   port
   (
      X : in   integer range 0 to 255;
      Y : out integer range 0 to 255
   );
end entity;

architecture rtl of TopLevelEntity is

        signal Sig1, Sig2, Sig4, Sig5 : integer range 0 to 255;
        signal Sig3, Sig6 : integer range 0 to 511;

         component EntityA is
         port
         (
               A : in   integer range 0 to 255;
               B : in   integer range 0 to 255;
               C : out integer range 0 to 511
          );
         end component ;

begin
         Instance1 : EntityA
         port map
         (
               A => Sig1,
               B => Sig2,
               C => Sig3
         );

         Instance2 : EntityA
         port map
         (
               A => Sig4,
               B => Sig5,
               C => Sig6
         );

end architecture;
```

```
entity TopLevelEntity is
   port
   (
      X : in   integer range 0 to 255;
      Y : out integer range 0 to 255
   );
end entity;

architecture rtl of TopLevelEntity is

        signal Sig1, Sig2, Sig4, Sig5 : integer range 0 to 255;
        signal Sig3, Sig6 : integer range 0 to 511;

         component EntityA is
         port
         (
               A : in   integer range 0 to 255;
               B : in   integer range 0 to 255;
               C : out integer range 0 to 511
          );
         end component ;

begin
         Instance1 : EntityA
         port map
         (
               A => Sig1,
               B => Sig2,
               C => Sig3
         );

         Instance2 : EntityA
         port map
         (
               A => Sig4,
               B => Sig5,
               C => Sig6
         );

end architecture;
```

# VHDL File Template

```
-- Declare your VHDL libraries & packages up here.
entity EntityName is
    port
    (
        -- Declare your modules’s IO ports here!
    );
end entity;

architecture ArchitectureName of EntityName is
    -- Declare your internal signals & constants here
    -- Declare components
begin
     -- Define your module’s behaviour here!
end architecture;
```

# VHDL Concurrent Statements

# VHDL Statements

Concurrent VHDL Statements :

- Can only be used inside the **Architecture** Block.

Sequential VHDL Statements :

- Can only be used inside the **Process** Block.

# Concurrent VHDL Statements

- Process Block
- Component Instantiation
- Concurrent Signal Assignments (using <=)
- Conditional Signal Assignment (When – Else)
- Selected Signal Assignment (With – Select)
- Generate

# Concurrent VHDL Statements

- Process Block
- Component Instantiation
- Concurrent Signal Assignments (using <=)
- Conditional Signal Assignment (When – Else)
- Selected Signal Assignment (With – Select)
- Generate

```
A <= B;
```

# Concurrent VHDL Statements

- Process Block
- Component Instantiation
- Concurrent Signal Assignments
- Conditional Signal Assignment (When – Else)
- Selected Signal Assignment (With – Select)
- Generate

# When-Else

```
architecture RTL of MyEntity is

    A <= "1000" when B = "00" else
         "0100" when B = "01" else
         "0010" when B = "10" else
         "0001“;

end architecture;
```

# With-Select

```
architecture RTL of MyEntity is

        with B select
        A <= "1000"  when  "00",
             "0100"  when  "01",
              "0010" when  "10",
              "0001" when others;

end architecture;
```

# Generate Statement

- Replicate hardware (for generate)

# Generate Statement

- Replicate hardware (for generate)
- Conditional hardware (if generate)

# Generate Statement

- Can be used to create :
  - Component instances
  - Signal Assignments
  - Process Blocks

# For Generate Syntax

```
Label: for parameter in range generate
        -- Hardware to Generate
end generate;
```

# For Generate Syntax

```
Label: for parameter in range generate
        -- Hardware to Generate
end generate;
```

# For Generate Example 1

```
Signal InputSignal    : std_logic_vector(15 downto 0);
```

# For Generate Example 1

```
Signal InputSignal   : std_logic_vector(15 downto 0);
Signal EvenBits      : std_logic_vector(7 downto 0);
Signal OddBits       : std_logic_vector(7 downto 0);
```

# For Generate Example 1

```
Signal InputSignal    : std_logic_vector(15 downto 0);
Signal EvenBits       : std_logic_vector(7 downto 0);
Signal OddBits        : std_logic_vector(7 downto 0);
```

---

```
EvenBits(0) <= InputSignal(0);
EvenBits(1) <= InputSignal(2);
EvenBits(2) <= InputSignal(4);

.......
```

# For Generate Example 1

```
Signal InputSignal    : std_logic_vector(15 downto 0);
Signal EvenBits       : std_logic_vector(7 downto 0);
Signal OddBits        : std_logic_vector(7 downto 0);
```

```
OddBits(0) <= InputSignal(1);
OddBits(1) <= InputSignal(3);
OddBits(2) <= InputSignal(5);
.......
```

# For Generate Example 1

```
Signal InputSignal    : std_logic_vector(15 downto 0);
Signal EvenBits       : std_logic_vector(7 downto 0);
Signal OddBits        : std_logic_vector(7 downto 0);
```

---

```
Example1: for  count  in 0 to 7 generate
```

# For Generate Example 1

```
Signal InputSignal    : std_logic_vector(15 downto 0);
Signal EvenBits       : std_logic_vector(7 downto 0);
Signal OddBits        : std_logic_vector(7 downto 0);
```

```
Example1: for  count  in 0 to 7 generate
        EvenBits (count) <= InputSignal(2*count);
```

# For Generate Example 1

```
Signal InputSignal    : std_logic_vector(15 downto 0);
Signal EvenBits       : std_logic_vector(7 downto 0);
Signal OddBits        : std_logic_vector(7 downto 0);
```

---

```
Example1: for  count  in 0 to 7 generate
        EvenBits (count) <= InputSignal(2*count);
        OddBits (count) <= InputSignal(2*count+1);
```

# For Generate Example 1

```
Signal InputSignal    : std_logic_vector(15 downto 0);
Signal EvenBits       : std_logic_vector(7 downto 0);
Signal OddBits        : std_logic_vector(7 downto 0);
```

```
Example1: for  count  in 0 to 7 generate
        EvenBits (count) <= InputSignal(2*count);
        OddBits (count) <= InputSignal(2*count+1);
end generate;
```

# For Generate Example 2

# For Generate Example 2

```
Example2: for n  in  0  to 19 generate




end generate;
```

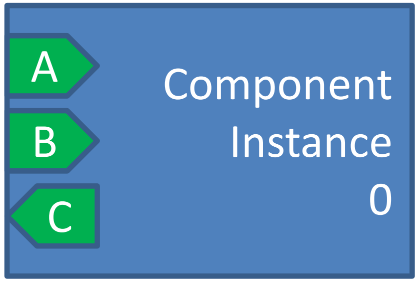
# For Generate Example 2

```
Example2: for n  in  0  to 19 generate

   MyComponent  :  EntityA
   port map
   (
         A  =>  Signal1,
         B  =>  Signal2,
         C  =>  Sum
   );

end generate;
```

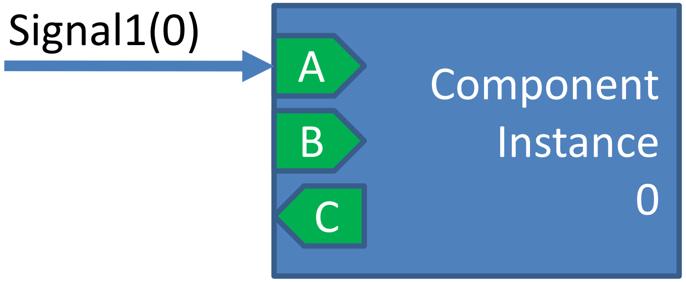
# For Generate Example 2

```
Example2: for n  in  0  to 19 generate

   MyComponent  :  EntityA
   port map
   (
        A  =>  Signal1(n),
        B  =>  Signal2(n),
        C  =>  Sum(n)
   );

end generate;
```

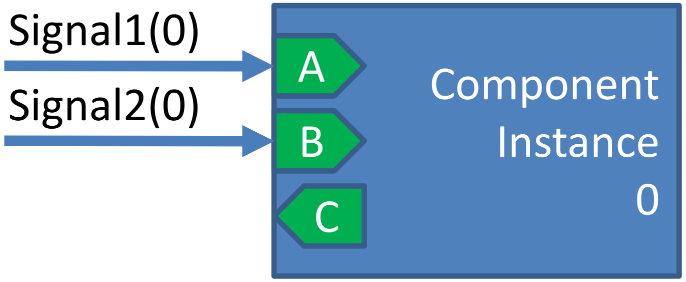
# For Generate Example 2



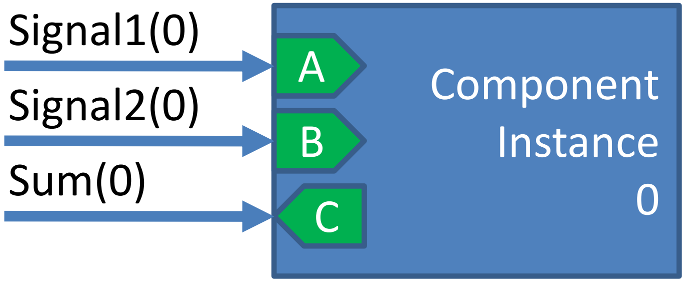
# For Generate Example 2



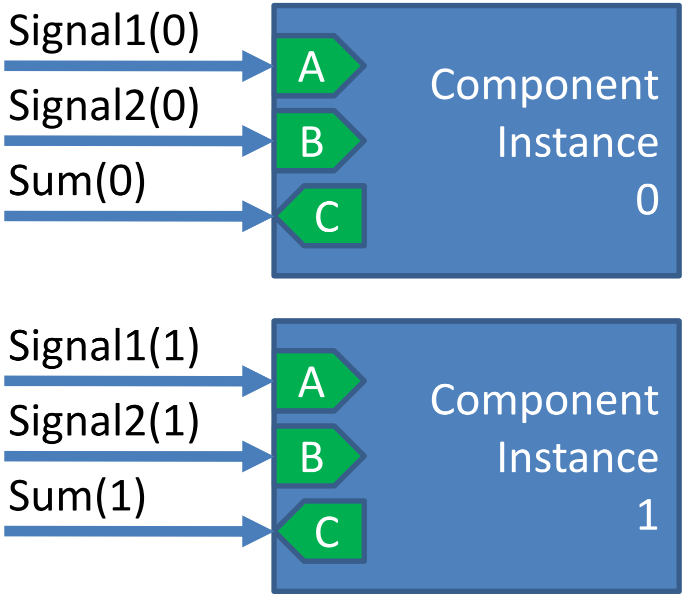
# For Generate Example 2



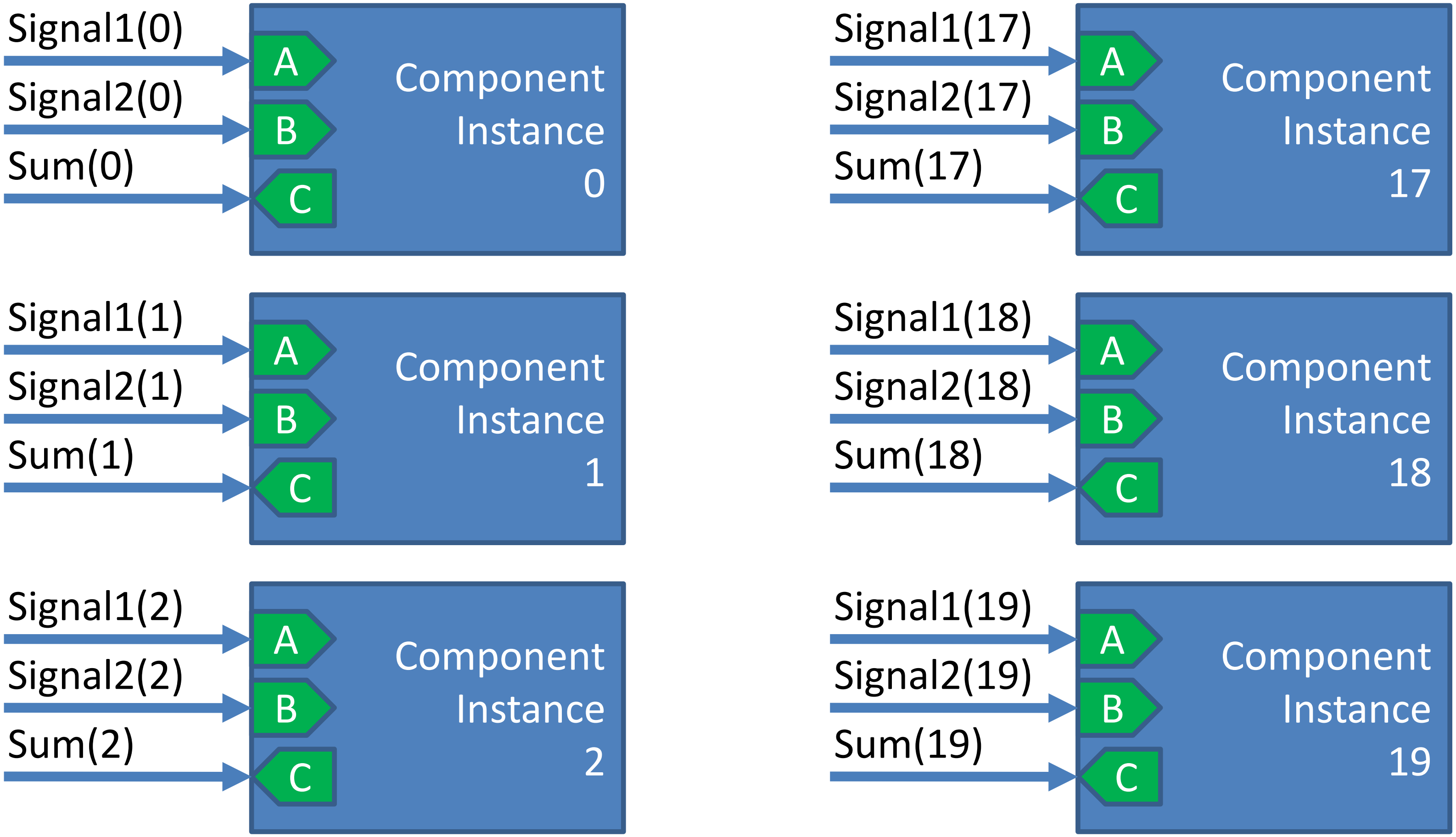
# For Generate Example 2



# For Generate Example 2



# For Generate Example 2



Signal1(0)
Signal2(0)
Sum(0)
A
B
C
Component
Instance
0

Signal1(1)
Signal2(1)
Sum(1)
A
B
C
Component
Instance
1

Signal1(2)
Signal2(2)
Sum(2)
A
B
C
Component
Instance
2

Signal1(17)
Signal2(17)
Sum(17)
A
B
C
Component
Instance
17

Signal1(18)
Signal2(18)
Sum(18)
A
B
C
Component
Instance
18

Signal1(19)
Signal2(19)
Sum(19)
A
B
C
Component
Instance
19

# For Generate Example 2

```
type Integer8x20 is array (0 to 19) of integer range 0 to 255;
```

# For Generate Example 2

```
type Integer8x20 is array (0 to 19) of integer range 0 to 255;
type Integer9x20 is array (0 to 19) of integer range 0 to 511;
```

# For Generate Example 2

```
type Integer8x20 is array (0 to 19) of integer range 0 to 255;
type Integer9x20 is array (0 to 19) of integer range 0 to 511;


Signal Signal1          : Integer8x20;
```

# For Generate Example 2

```
type Integer8x20 is array (0 to 19) of integer range 0 to 255;
type Integer9x20 is array (0 to 19) of integer range 0 to 511;


Signal Signal1          : Integer8x20;
Signal Signal2          : Integer8x20;
```

# For Generate Example 2

```
type Integer8x20 is array (0 to 19) of integer range 0 to 255;
type Integer9x20 is array (0 to 19) of integer range 0 to 511;


Signal Signal1          : Integer8x20;
Signal Signal2          : Integer8x20;
Signal Sum              : Integer9x20;
```

# If Generate

- Generates hardware IF given condition is met.
- Condition must evaluate to a constant
- Cannot use signals or variables in condition
- Can test for multiple conditions.

# If Generate Syntax

```
Label  :  if  condition1  generate
-- Hardware to generate on condition 1.
elsif  condition2  generate
-- Hardware to generate on condition 2.
else generate
-- Hardware to generate if all conditions false.
end generate;
```

# If Generate Example

```
Example   :   if    ASSIGN_TO_Y  generate
        Y <= X;
else generate
        Z <= X;
end generate;
```

# If Generate Example

```
Example  :  if  ENABLE_REGISTER  generate
    Reg : process (clk)
    begin
        if rising_edge(clk) then
            Y <= X;
        end if;
    end process;
else generate
    Y <= X;
end generate;
```

This section of code will be implemented if ENABLE_REGISTER is TRUE.

This section of code will be implemented if ENABLE_REGISTER is false.

# Concurrent VHDL Statements

- Process Block
- Component Instantiation
- Concurrent Signal Assignments (using <=)
- Conditional Signal Assignment (When – Else)
- Selected Signal Assignment (With – Select)
- Generate

# VHDL Statements

# Sequential Statements

- Can only be used inside **Process Blocks**.
- Sequential Assignment
- If statements
- Case statements
- For loops
- Wait

# Sequential Statements

- Can only be used inside **Process Blocks**.
- Sequential Assignment
- If statements
- Case statements
- For loops
- Wait

```
Y <= X;
```

# Sequential Statements

- Can only be used inside **Process Blocks**.
- Sequential Assignment
- If statements
- Case statements
- For loops
- Wait

# If Statement - Template

```
if Condition1 then
        -- Code Section 1
elsif Condition2 then
        -- Code Section 2
elsif Condition3 then
        -- Code Section 3
else
        -- Code Section 4
end if;
```

# If Statement - Combinational

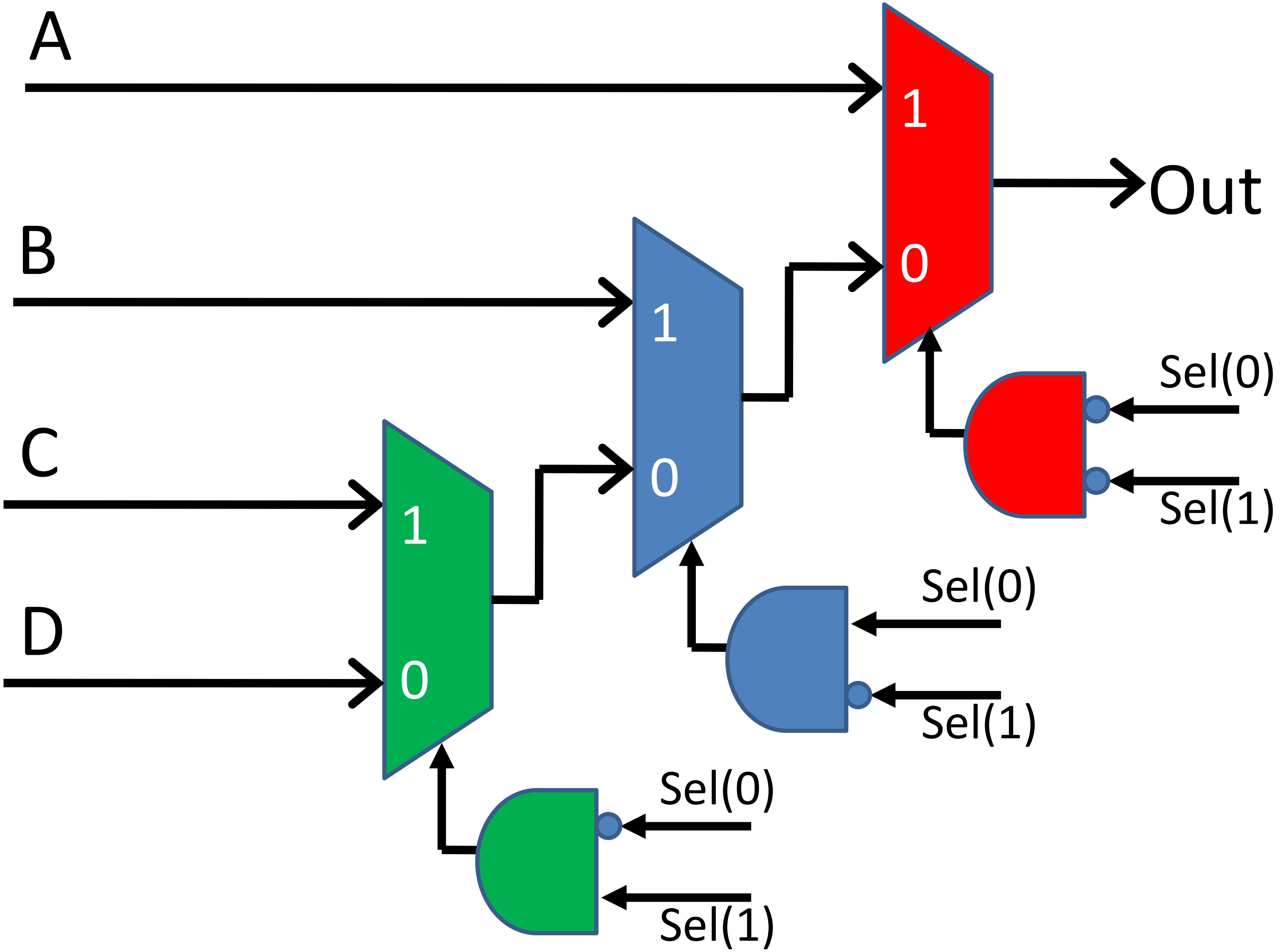
```
MyProcess : process(Sel, A, B, C, D)
begin
    if Sel = “00” then
        Out <= A;
    elsif Sel = “10” then
        Out <= B;
    elsif Sel = “10” then
        Out <= C;
    else
        Out <= D;
    end if;
end process;
```

First condition

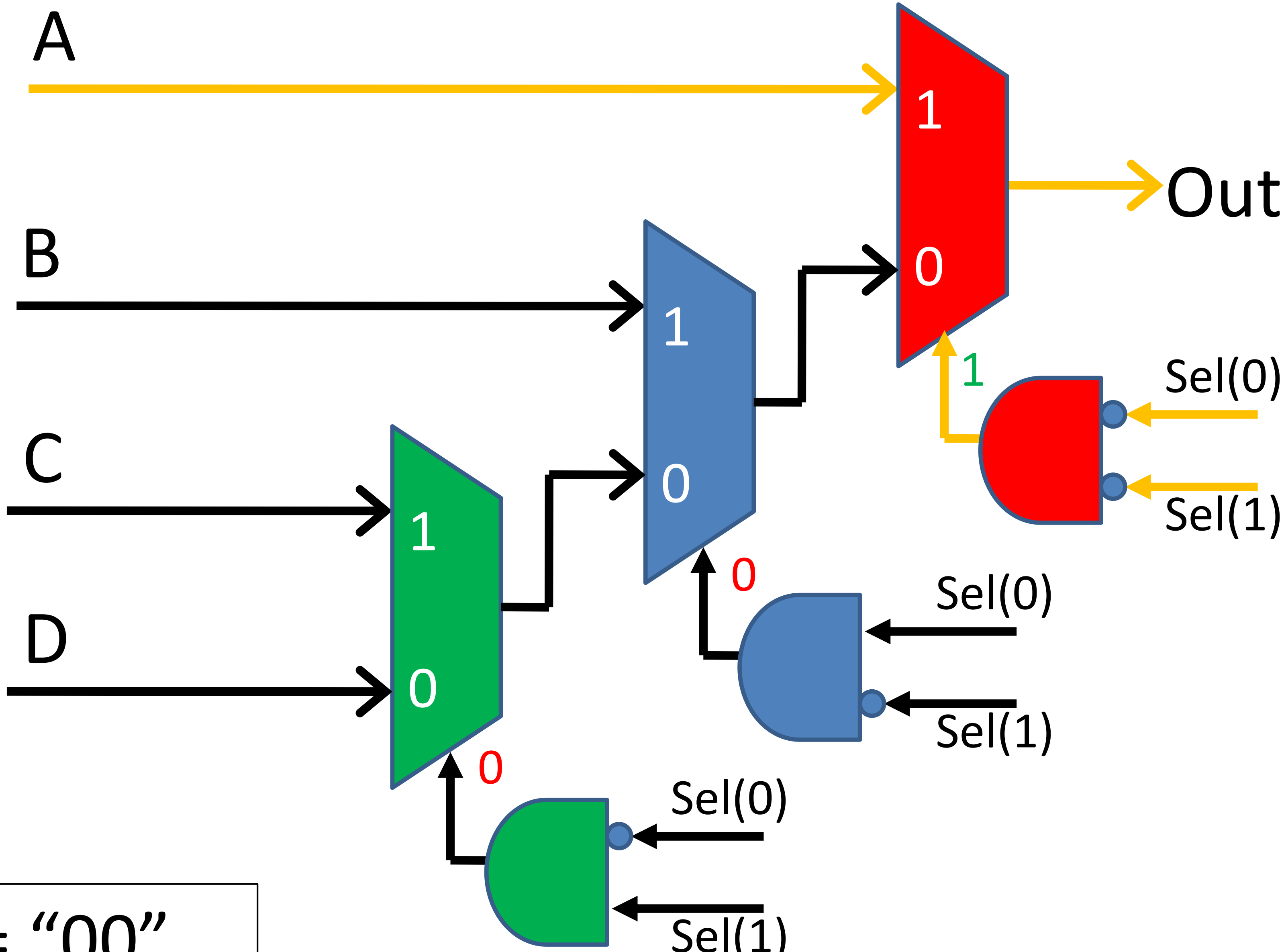
Second condition

Third condition

# If Statement - Hardware

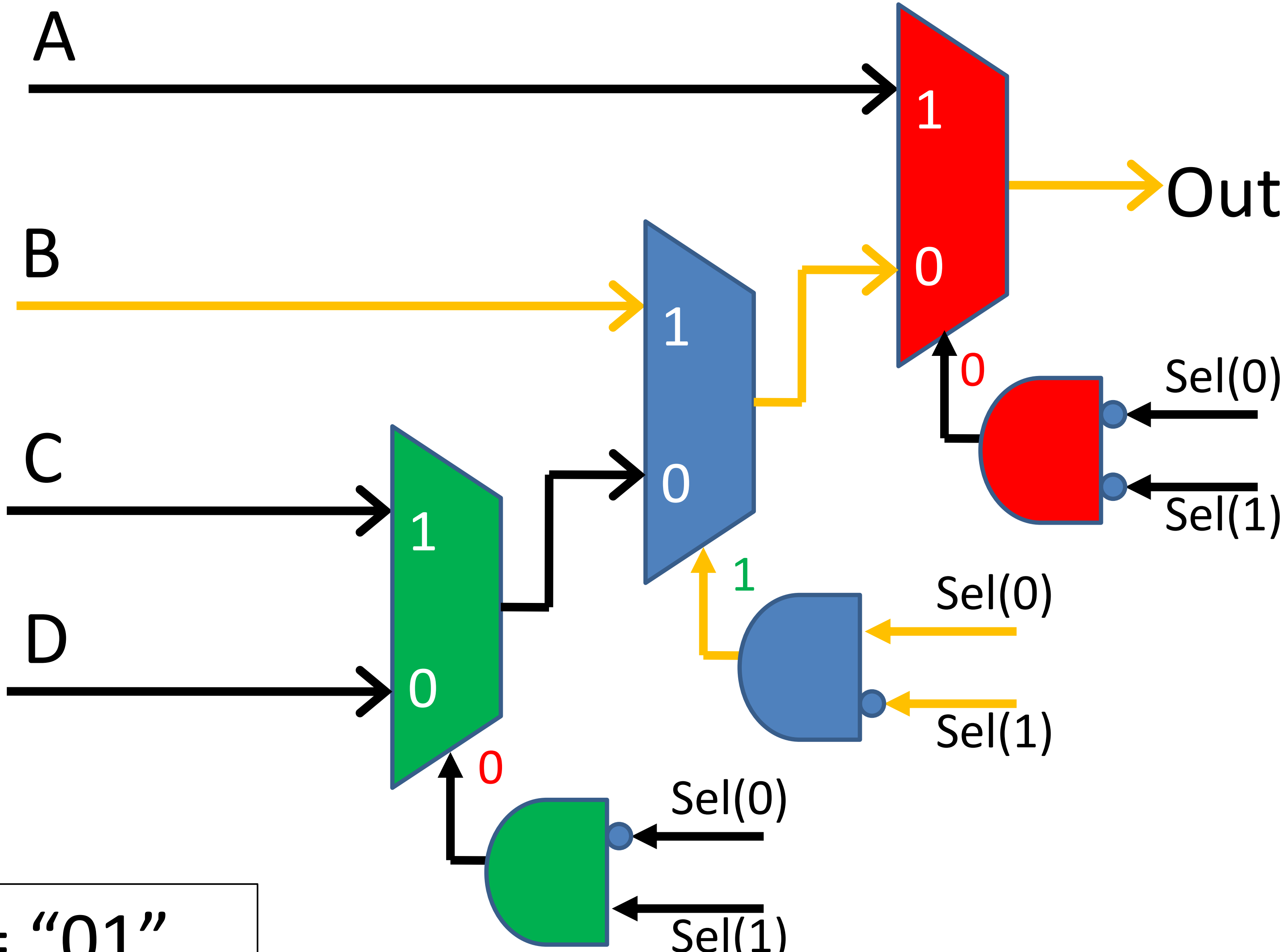


# If Statement – 1st Condition



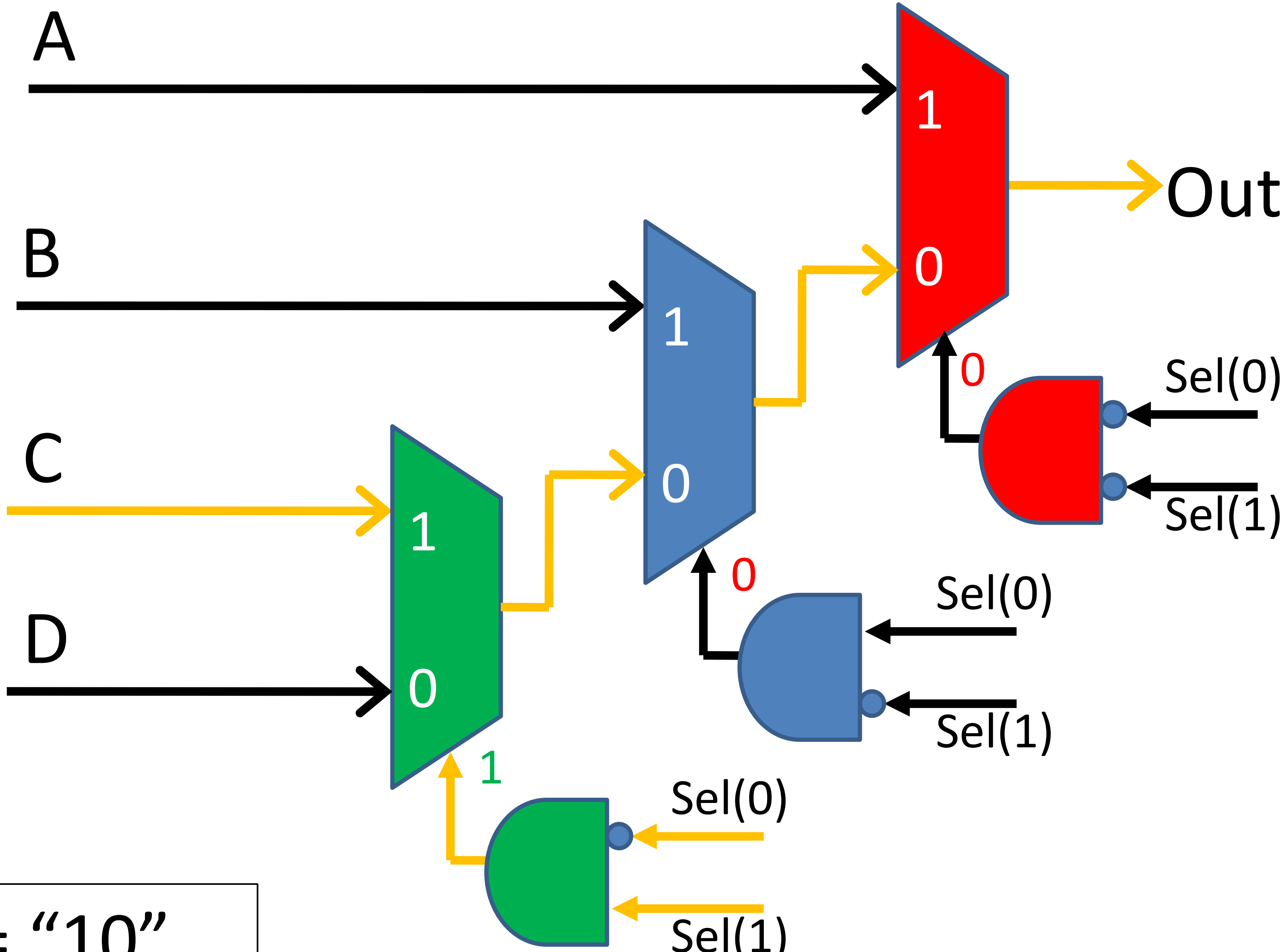
Sel = “00”

# If Statement – $2^{nd}$ Condition



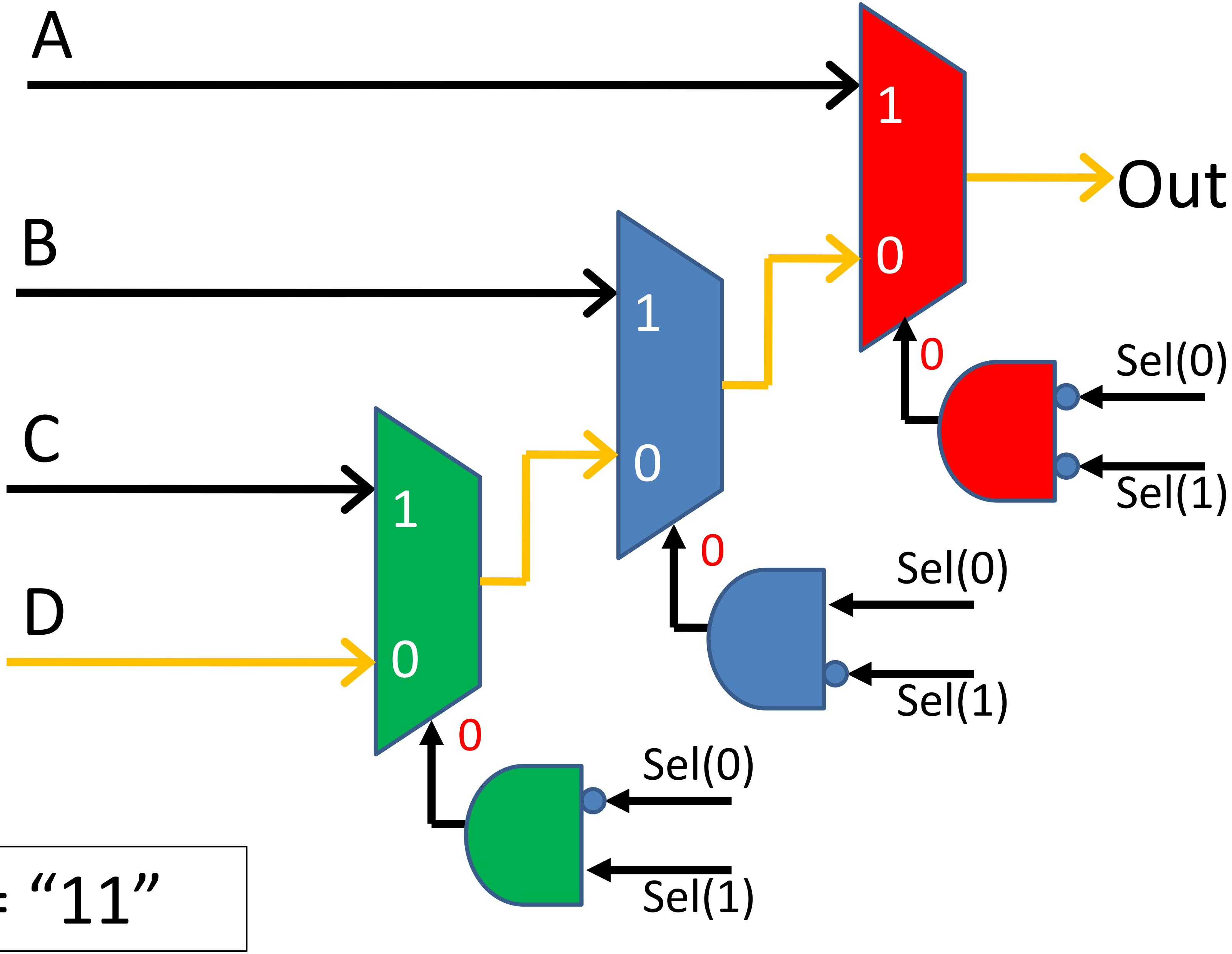
Sel = “01”

# If Statement – 3rd Condition



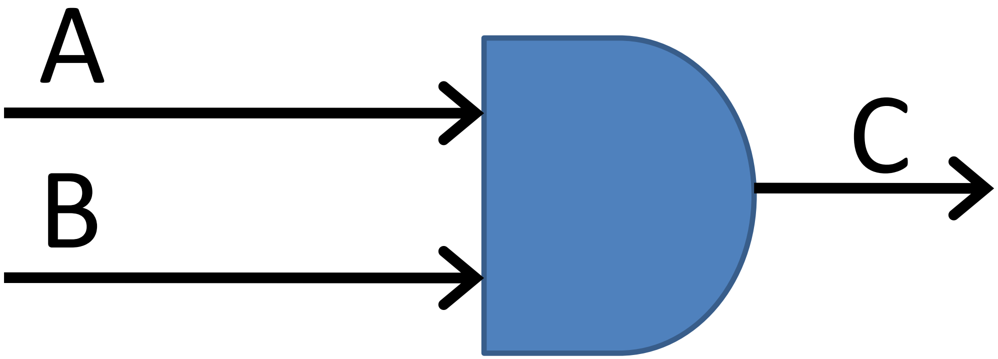
Sel = “10”

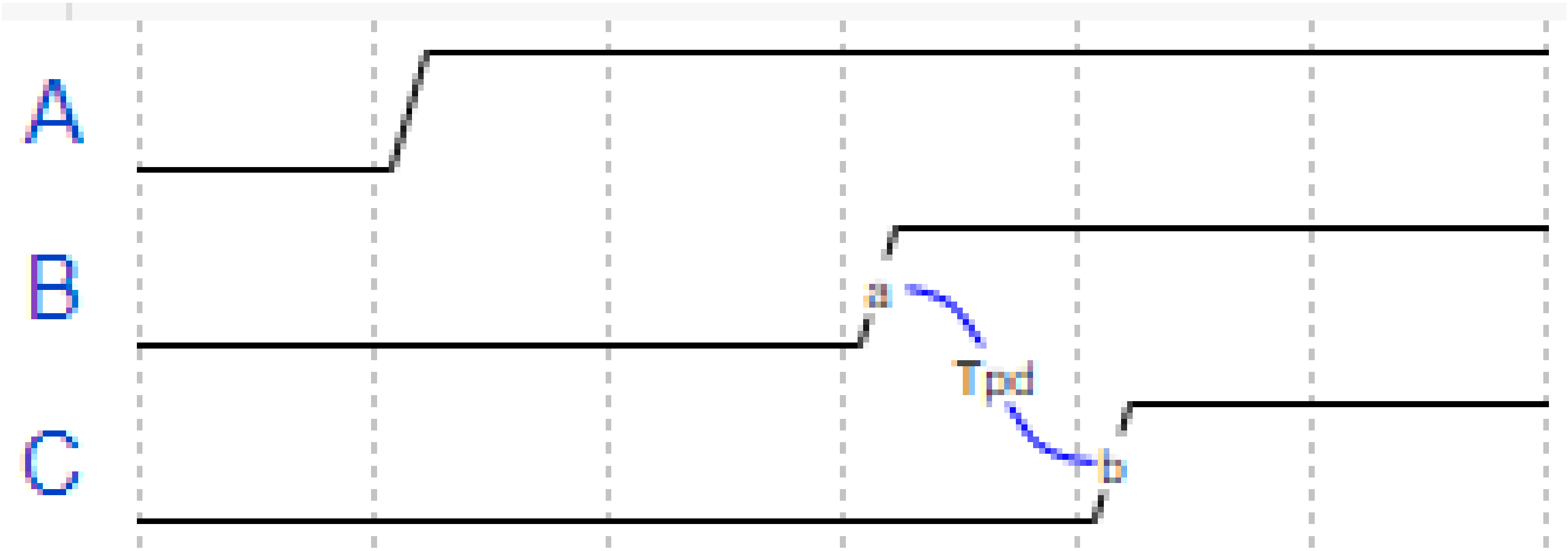
# If Statement – Default



Sel = “11”

# Propagation Delay





# If Statement – Notes

- The conditions must return a Boolean (true or false) result.
- If statements having more than one condition will create cascaded multiplexers.
- Lowest priority path has to propagate through all multiplexers to get to output.
- This is bad for timing closure.

# If Statement – Notes

- The conditions must return a Boolean (true or false) result.
- If statements having more than one condition will create cascaded multiplexers.
- Lowest priority path has to propagate through all multiplexers to get to output.
- This is bad for timing closure.

# Case Statement - Template

```
case (Expression) is
    when Choice1=>
        -- Code Section 1
    when Choice2=>
        -- Code Section 2
    when others =>
        -- Code Section 3
end case;
```

# Case Statement - Template

```
case (Expression) is
    when Choice1=>
        -- Code Section 1
    when Choice2=>
        -- Code Section 2
    when others =>
        -- Code Section 3
end case;
```

# Case Statement - Template

```
case (Expression) is
    when Choice1=>
        -- Code Section 1
    when Choice2=>
        -- Code Section 2
    when others =>
        -- Code Section 3
end case;
```

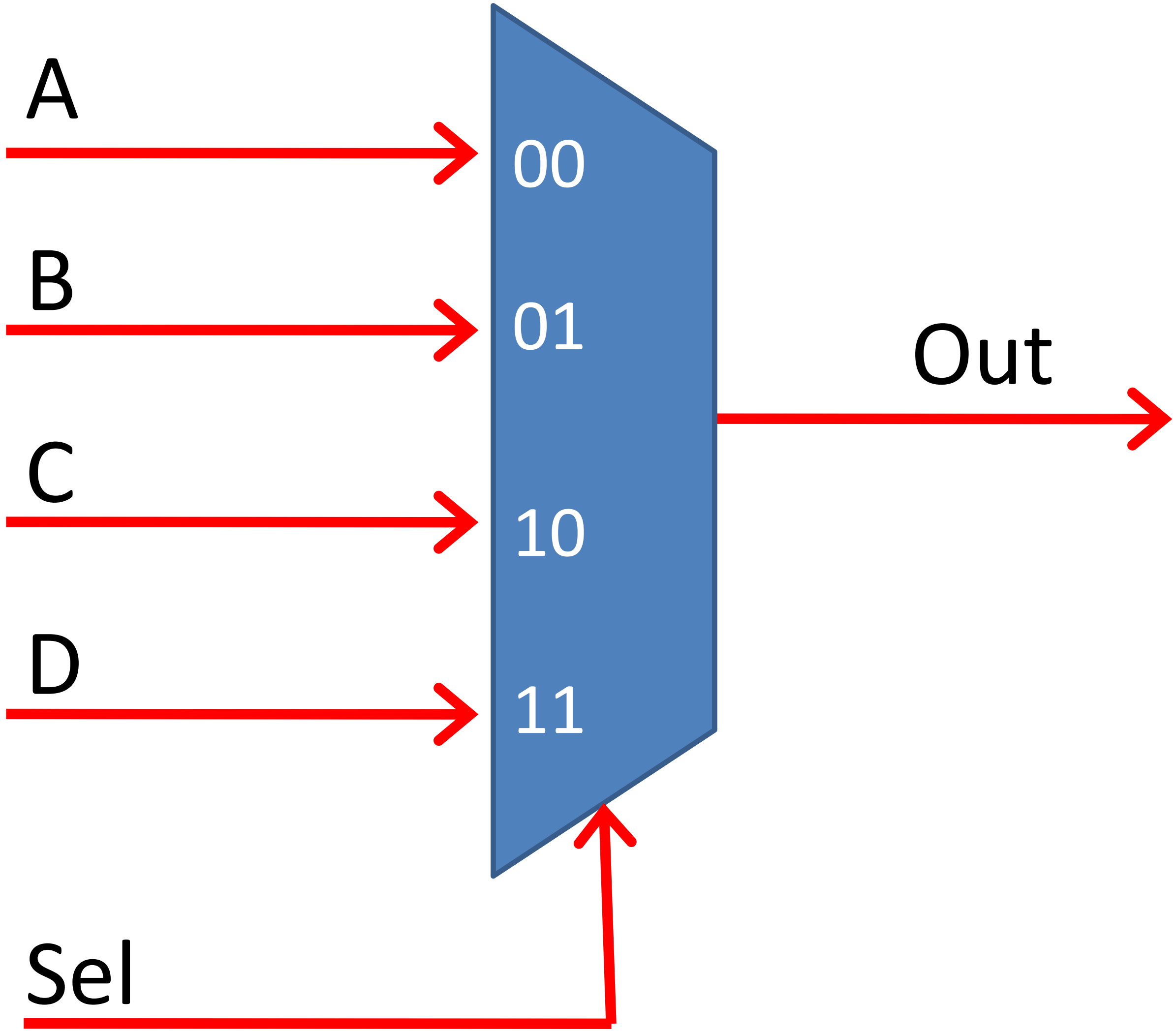
# Case Statement – Combinational Process

```
MyProcess : process(Sel, A, B, C, D)
begin
    case (Sel) is
           when “00” => Out <= A;
           when “01” => Out <= B;
           when “10” => Out <= C;
           when others => Out <= D;
    end case;
end process;
```

# Case Statement – Combinational Process

```
MyProcess : process(Sel, A, B, C, D)
begin
    case (Sel) is
            when “00” => Out <= A;
            when “01” => Out <= B;
            when “10” => Out <= C;
            when others => Out <= D;
    end case;
end process;
```

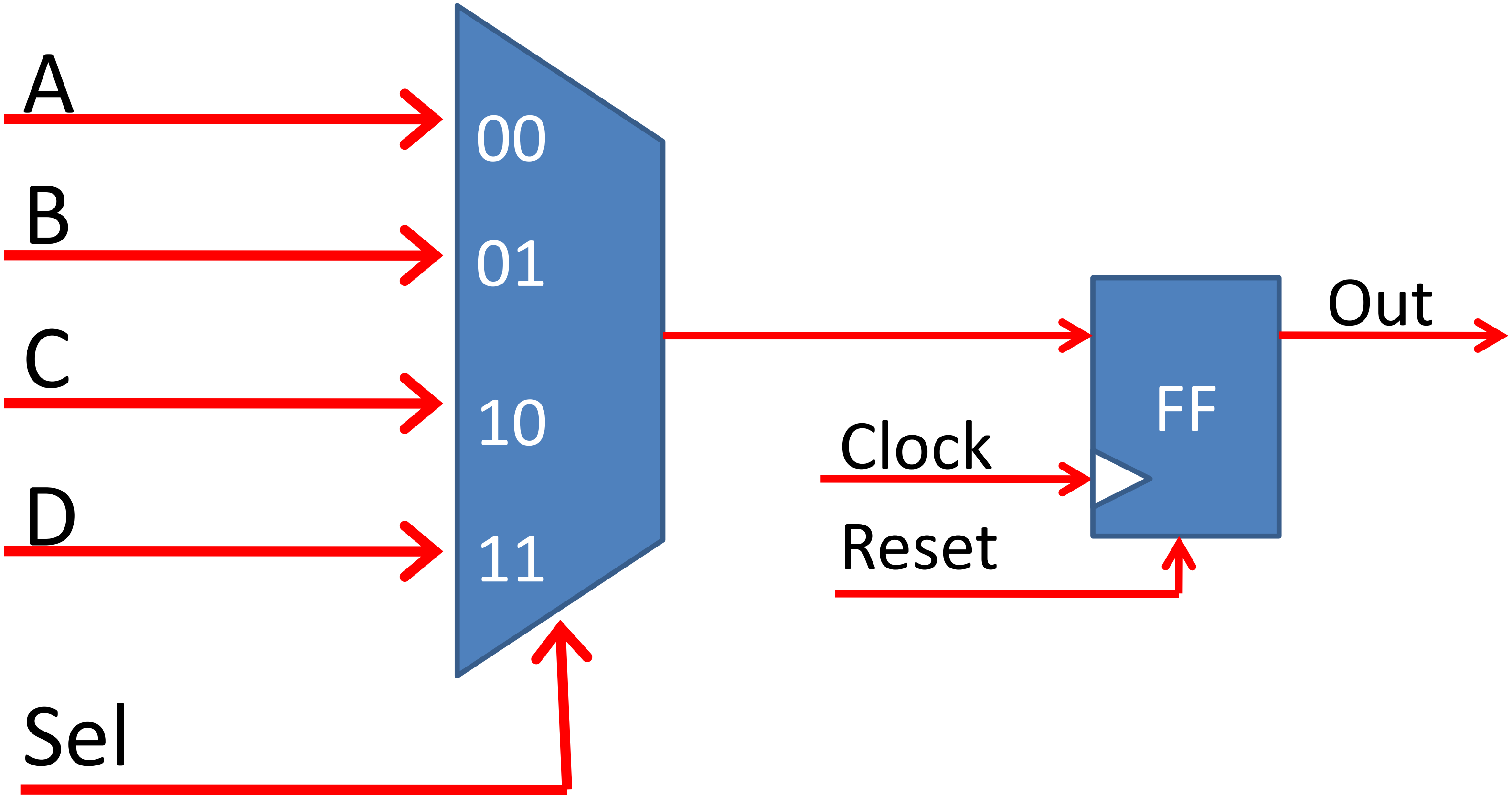
# Case Statement – Combinational Process



# Case Statement – Registered Process

```
MyProcess : process(Reset, Clock)
Begin
    if Reset = ‘1’ then
           Out <= ‘0’;
    elsif rising_edge(Clock) then
          case (Sel) is
                 when “00” => Out <= A;
                 when “01” => Out <= B;
                 when “10” => Out <= C;
                 when others => Out <= D;
          end case;
    end if;
end process;
```

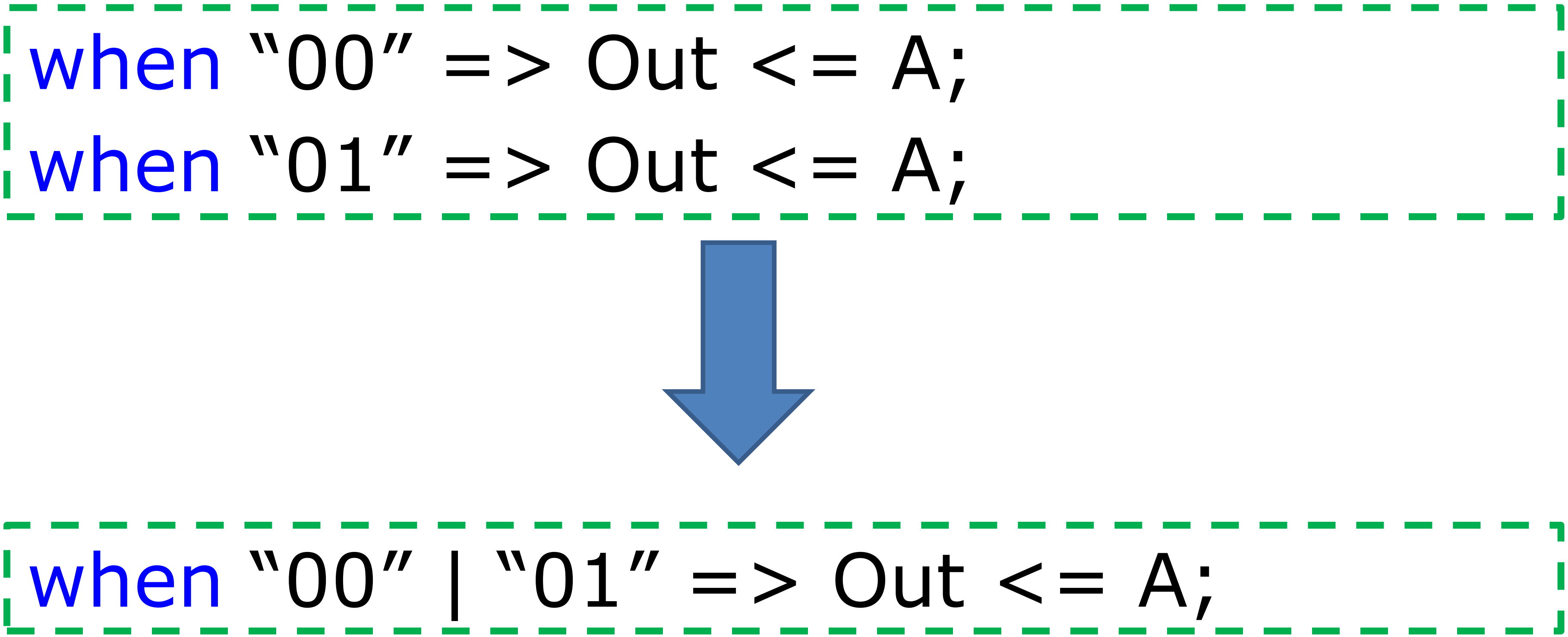
# Case Statement – Registered Process



# Case Statement Notes

```
when “00” => Out <= A;
when “01” => Out <= A;
```

# Case Statement Notes



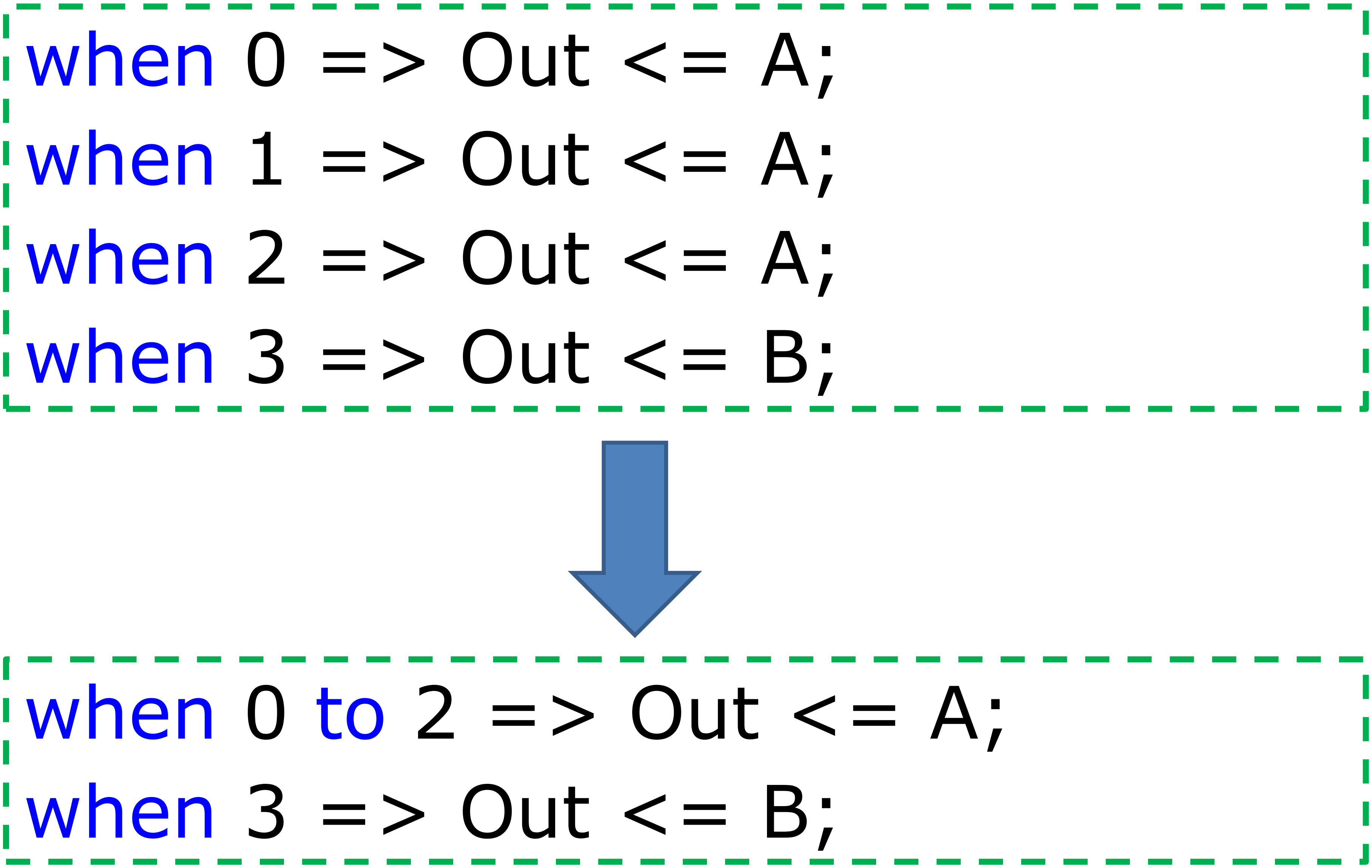
```
when “00” => Out <= A;
when “01” => Out <= A;
```

```
when “00” | “01” => Out <= A;
```

# Case Statement Notes

```
when 0 => Out <= A;
when 1 => Out <= A;
when 2 => Out <= A;
when 3 => Out <= B;
```

# Case Statement Notes



```
when 0 => Out <= A;
when 1 => Out <= A;
when 2 => Out <= A;
when 3 => Out <= B;
```

```
when 0 to 2 => Out <= A;
when 3 => Out <= B;
```

# Case Statement Notes

```
when “00” | “01” => Out <= A;
when “01” | “10” => Out <= B;
```

Test conditions cannot overlap!!

```
when Sig1 => Out <= A;
```

Test conditions must be constants!

# For Loop - Template

```
for LoopVariableName in Range loop
   -- Code Section That Needs Repeating
end loop;
```

# For Loop - Template

```
for LoopVariableName in Range loop
   -- Code Section That Needs Repeating
end loop;
```

# For Loop - Template

```
for LoopVariableName in Range loop
  -- Code Section That Needs Repeating
end loop;
```

Range = 0 **to** 7

# For Loop – Combinational Process

```
MyProcess : process(A, B)
Begin
        for i in 0 to 2 loop
                Y(i) <= A(2-i) and B(i);
        end loop;
end process;
```

# For Loop – Combinational Process

```
MyProcess : process(A, B)
Begin
        for i in 0 to 2 loop
                Y(i) <= A(2-i) and B(i);
        end loop;
end process;
```

```
MyProcess : process(A, B)
Begin
        Y(0) <= A(2) and B(0);
        Y(1) <= A(1) and B(1);
        Y(2) <= A(0) and B(2);
end process;
```

# For Loop – Combinational Process

```
MyProcess : process(A, B)
Begin
        for i in 0 to 2 loop
                Y(i) <= A(2-i) and B(i);
        end loop;
end process;
```

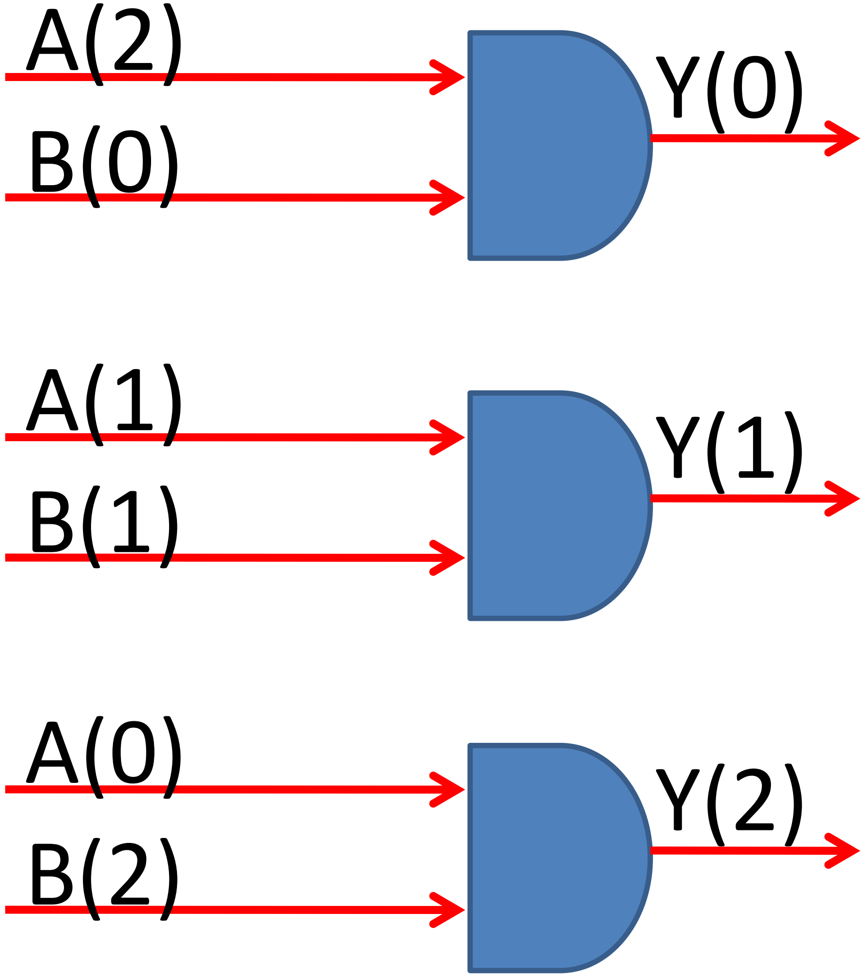
```
MyProcess : process(A, B)
Begin
        Y(0) <= A(2) and B(0);
        Y(1) <= A(1) and B(1);
        Y(2) <= A(0) and B(2);
end process;
```

# For Loop – Combinational Process

```
MyProcess : process(A, B)
Begin
        for i in 0 to 2 loop
                Y(i) <= A(2-i) and B(i);
        end loop;
end process;
```

```
MyProcess : process(A, B)
Begin
        Y(0) <= A(2) and B(0);
        Y(1) <= A(1) and B(1);
        Y(2) <= A(0) and B(2);
end process;
```

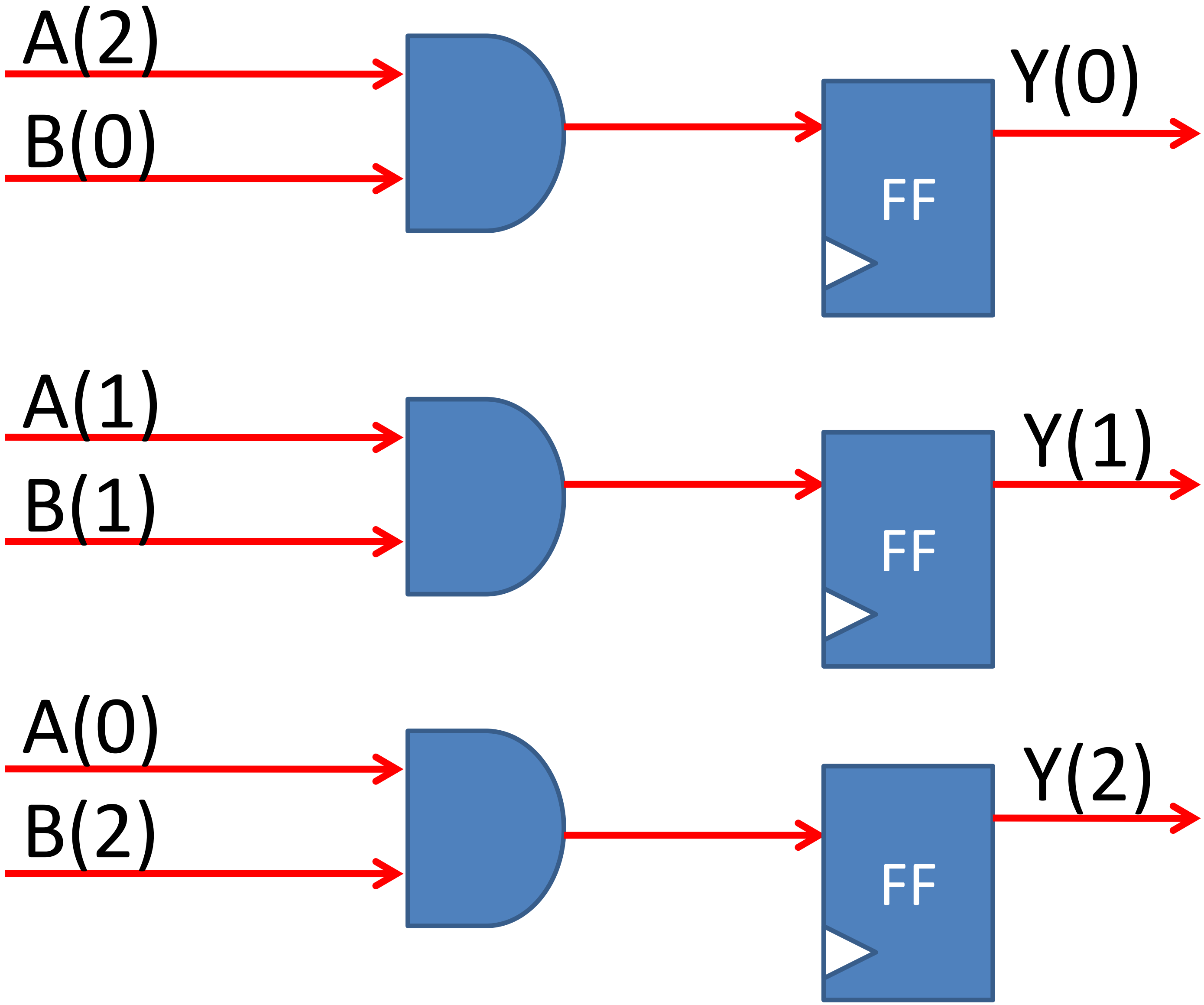
# For Loop – Combinational Process



# For Loop – Registered Process

```
MyProcess : process(Reset, Clock)
Begin
    if Reset = ‘1’ then
         Y <= “000”;
    elsif rising_edge(Clock) then
        for i in 0 to 2 loop
                Y(i) <= A(2-i) and B(i);
        end loop;
    end if;
end process;
```

# For Loop – Registered Process



# For Loop Notes

```
signal A : std_logic_vector(2 downto 0);
signal B : std_logic_vector(2 downto 0);
signal Y : std_logic_vector(2 downto 0);
for i in 0 to 2 loop
    Y(i) <= A(2-i) and B(i);
end loop;
```

# For Loop Notes

```
signal A : std_logic_vector(2 downto 0);
signal B : std_logic_vector(2 downto 0);
signal Y : std_logic_vector(2 downto 0);
for i in 0 to 2 loop
    Y(i) <= A(2-i) and B(i);
end loop;
```

Y’left

Y’right

```
for i in Y’right to Y’left loop
   Y(i) <= A(2-i) and B(i);
end loop;
```

# For Loop Notes

```
signal A : std_logic_vector(2 downto 0);
signal B : std_logic_vector(2 downto 0);
signal Y : std_logic_vector(2 downto 0);
for i in 0 to 2 loop
    Y(i) <= A(2-i) and B(i);
end loop;
```

```
for i in Y'right to Y'left loop
   Y(i) <= A(2-i) and B(i);
end loop;
```

```
for i in Y'range loop
   Y(i) <= A(2-i) and B(i);
end loop;
```

Y'range = 2 downto 0.
The loop variable, i, counts down from 2 to 0.

# For Loop Notes

```
-- A is a std_logic_vector of length 8
for i in 0 to 7 loop
     A(i) <= ‘1’;
     if i = 3 then
          exit;
     end if;
end loop;
```

Can use “exit” to immediately exit the loop.

# For Loop Notes

```
-- A is a std_logic_vector of length 8
for i in 0 to 7 loop
    A(i) <= ‘1’;
    if i = 3 then
        exit;
    end if;
end loop;
```

Can use “exit” to immediately exit the loop.

```
for i in 0 to 7 loop
    A(i) <= ‘1’;
    exit when i = 3;
end loop;
```

# For Loop Notes

```
-- A is a std_logic_vector of length 8
for i in 0 to 7 loop
    A(i) <= ‘1’;
    i = 2;
end loop;
```

Cannot change loop variable.

# Wait Statement

- NOT Synthesisable!
- Used only in Test-Benches (or Models)
- Used to pause a simulation or model.
- Process must not have a sensitivity list
- Allows us to pause the simulation :
  - for a fixed period of time
  - until some event occurs

# Wait Statement

- NOT Synthesisable!
- Used only in Test-Benches (or Models)
- Used to pause a simulation or model.
- Process must not have a sensitivity list
- Allows us to pause the simulation :
  - for a fixed period of time
  - until some event occurs

# Wait

```
Test : process
begin
        Y <= X;
        wait;
        Z <= Y;
end process;
```

# Wait for

```
Test : process
begin
        Y <= X;
        wait for 100ns;
        Z <= Y;
end process;
```

# Clock Signal Using Wait for

```
Clk_Generator: process
begin
        Clock <= ‘0’;
        wait for 10 ns;
        Clock <= ‘1’;
        wait for 10 ns;
end process;
```

# Wait On

```
Test : process
begin
        Y <= X;
        wait on A;
        Z <= Y;
end process;
```

# Wait On

```
Test : process
begin
        Y <= X;
        wait on A;
        Z <= Y;
end process;
```

# Wait Until

```
Test : process
begin
        Y <= X;
        wait until A = ‘0’;
        Z <= Y;
end process;
```

# Wait Until

```
Test : process
begin
        Y <= X;
        wait until rising_edge(A);
        Z <= Y;
end process;
```

# Summary

- Can only be used inside process block
- If statements
- Case statements
- For loops
- Wait statements
- Hardware realisation

# Summary

- Can only be used inside process block
- If statements
- Case statements
- For loops
- Wait statements
- Hardware realisation

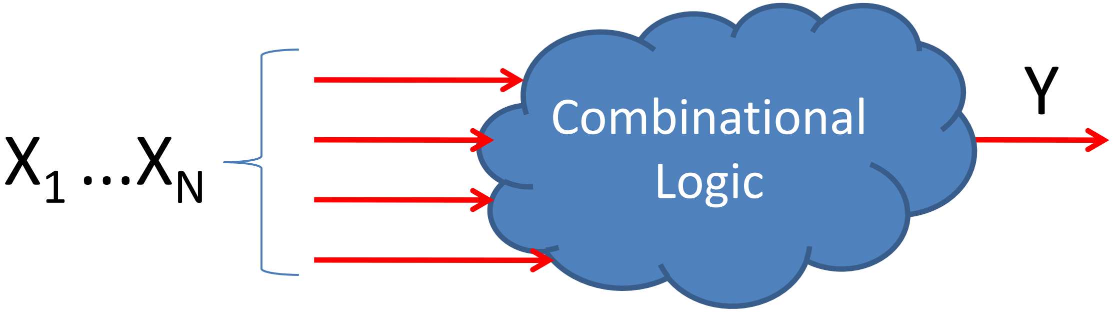
# Signal Assignments Inside The Process Block

Part I

# Signal Assignments Inside The Process Block

## Part I

# Combinational Process - Hardware



# Code For Example 1

```
TestProcess1 : process ($X_1$, $X_2$, $X_3$)
Begin




end process;
```

# Code For Example 1

```
TestProcess1 : process ($X_1$, $X_2$, $X_3$)
Begin
    if $X_1$ = '1' then

    else

    end if;
end process;
```

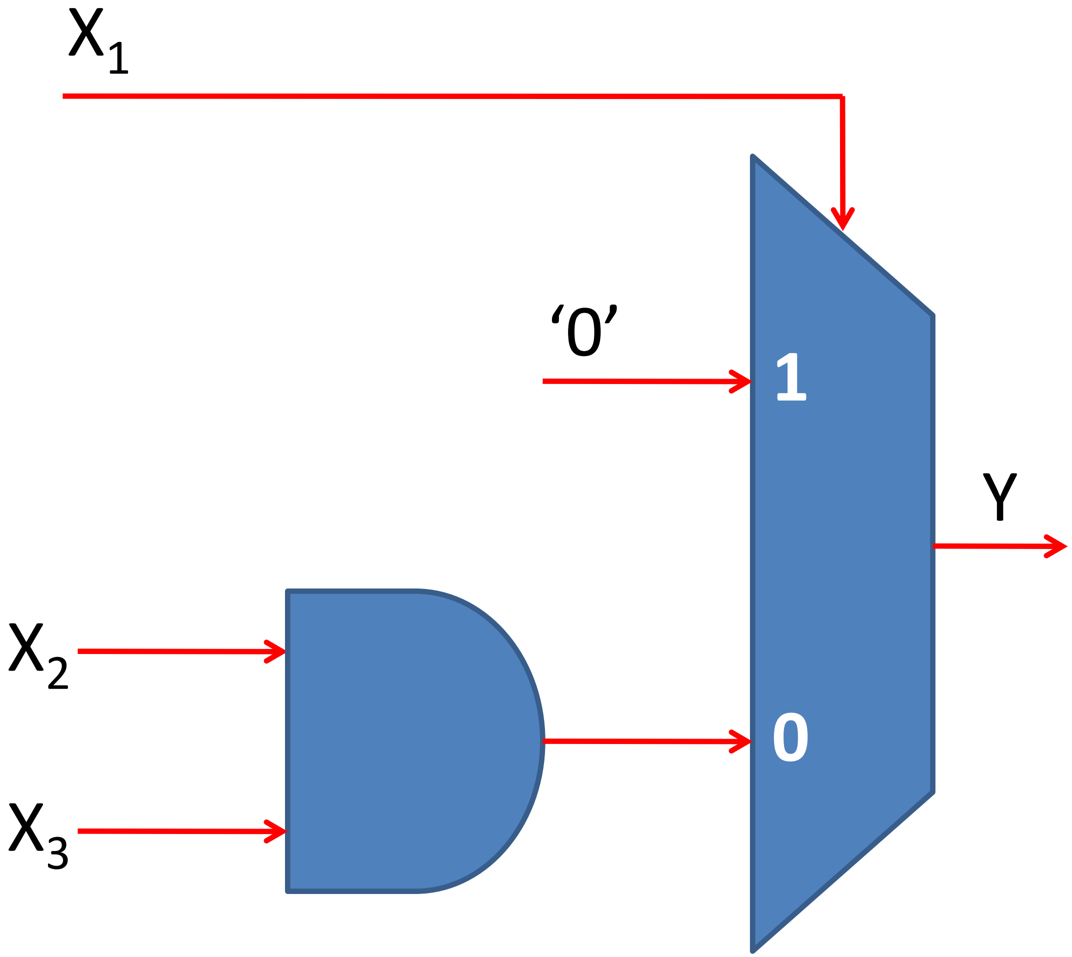
# Code For Example 1

```
TestProcess1 : process ($X_1$, $X_2$, $X_3$)
Begin
        if $X_1$ = ‘1’ then
            Y <= ‘0’;
        else


        end if;
end process;
```

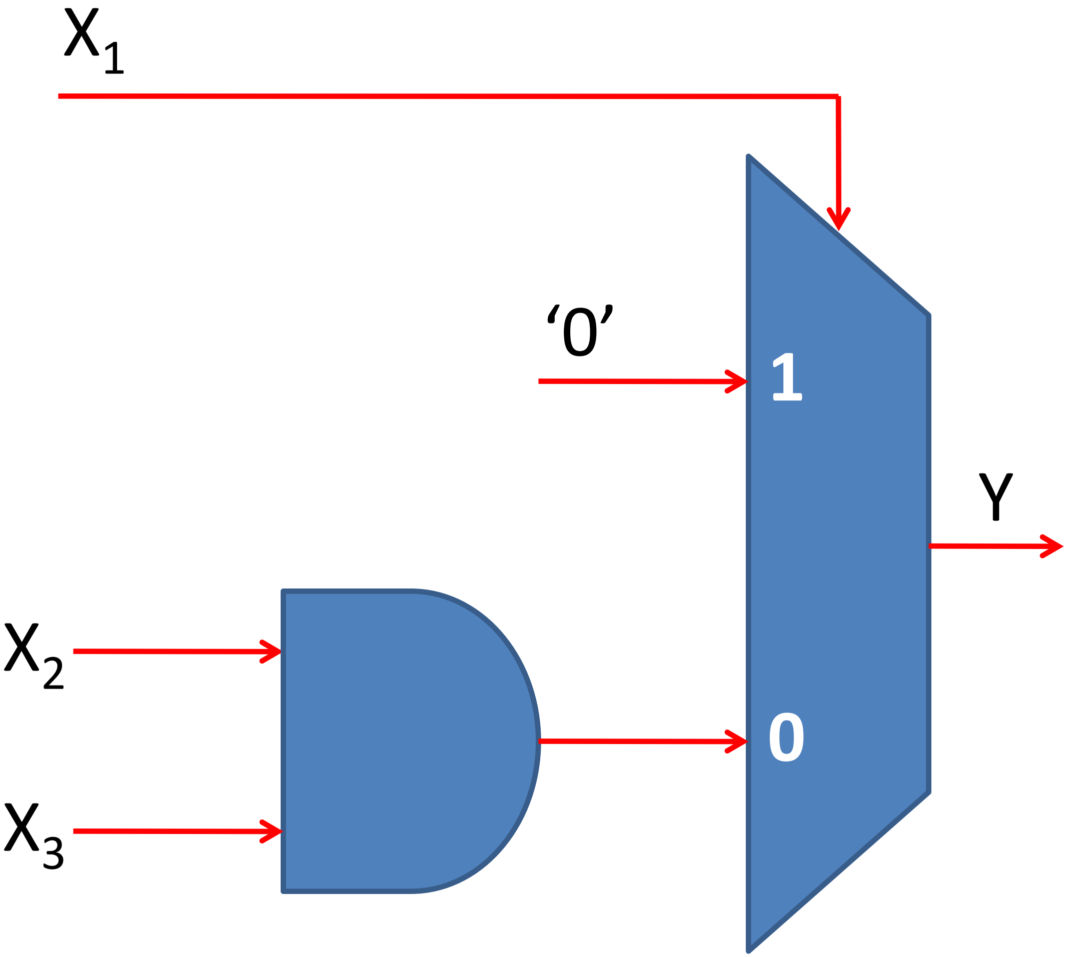
# Code For Example 1

```
TestProcess1 : process ($X_1$, $X_2$, $X_3$)
Begin
    if $X_1$ = ‘1’ then
        Y <= ‘0’;
    else
        Y <= $X_2$ and $X_3$;
    end if;
end process;
```

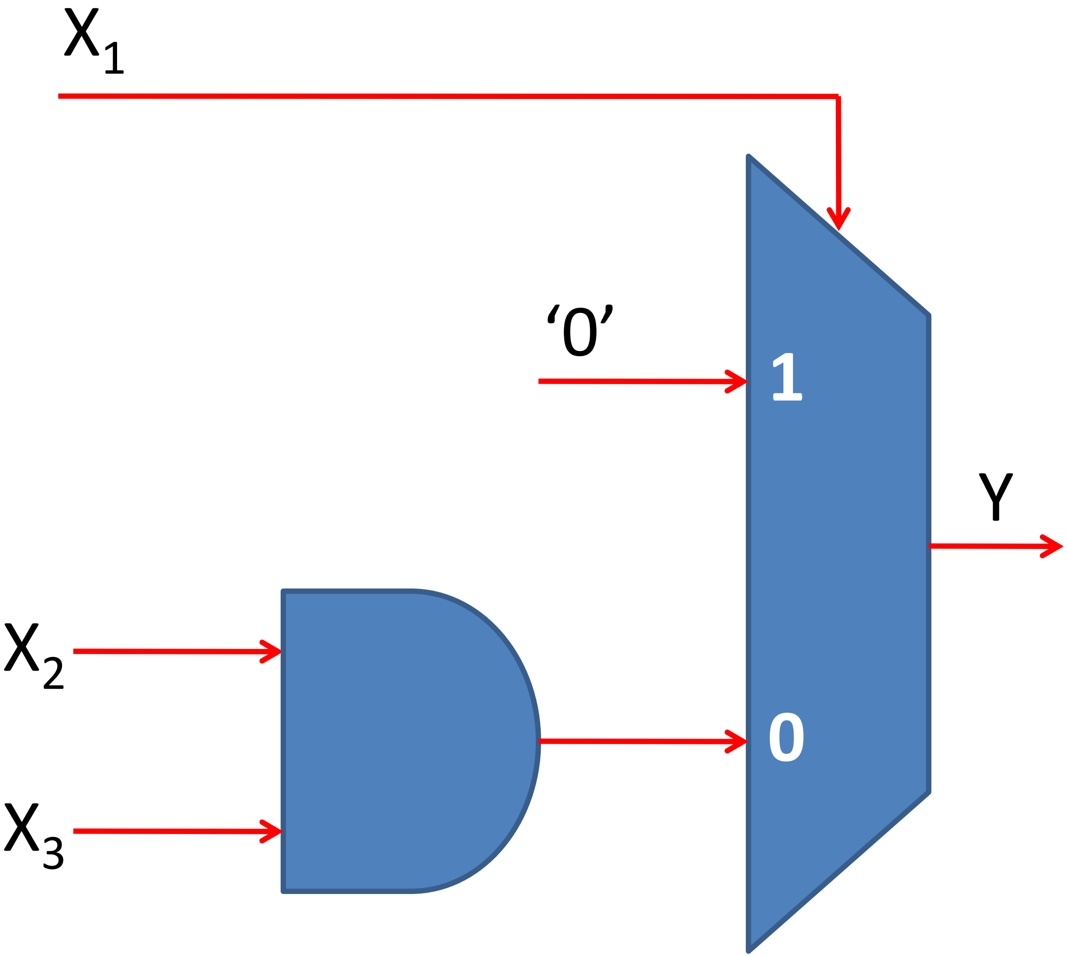
# Hardware For Example 1



# Hardware For Example 1



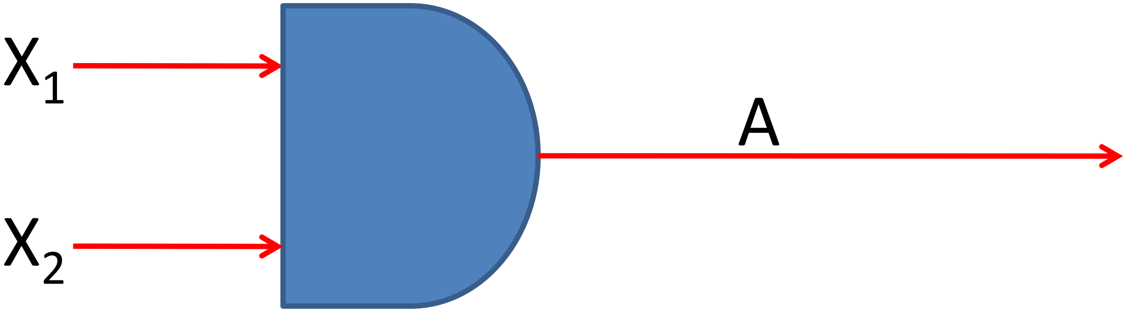
# Hardware For Example 1



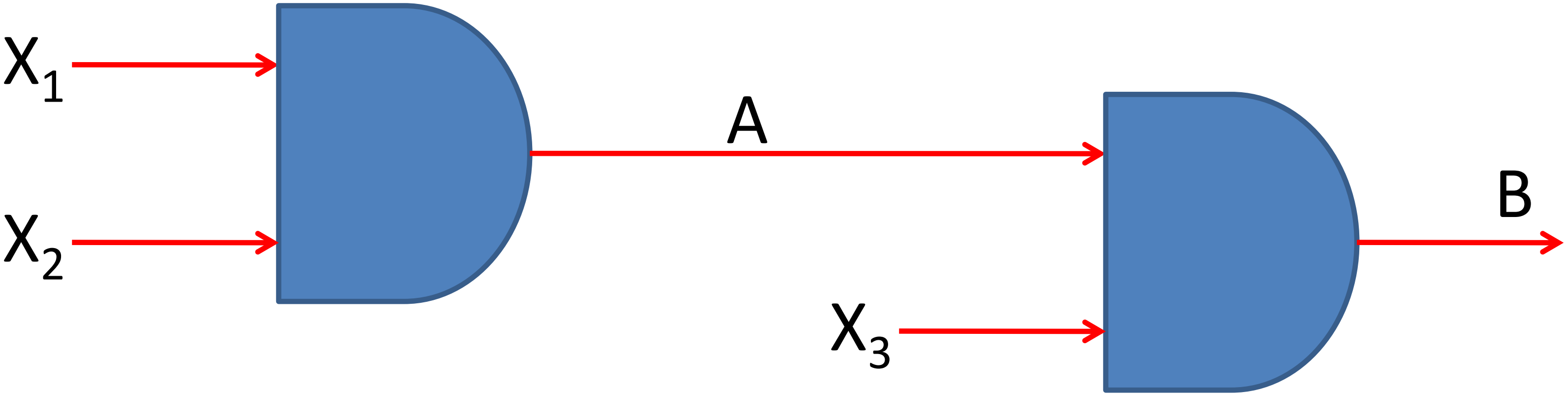
# Code For Example 2

```
TestProcess2A : process ($X_1$, $X_2$, $X_3$)
Begin
        A <= $X_1$ and $X_2$;
        B <= A and $X_3$;
end process;

TestProcess2B : process ($X_1$, $X_2$, $X_3$)
Begin
        B <= A and $X_3$;
        A <= $X_1$ and $X_2$;
end process;
```

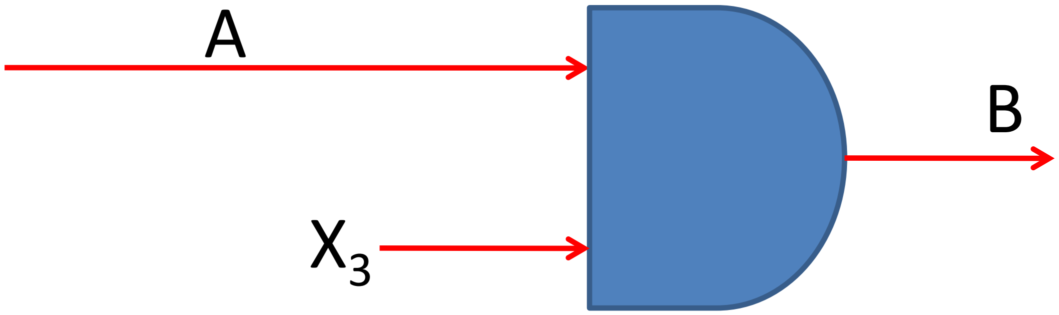
# Hardware For TestProcess2A



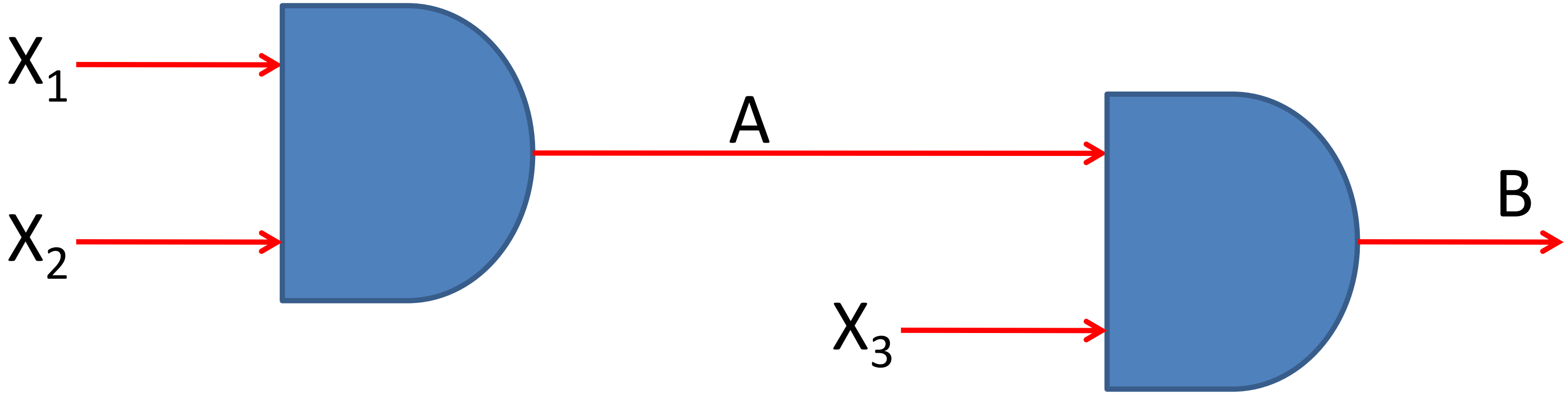
# Hardware For TestProcess2A



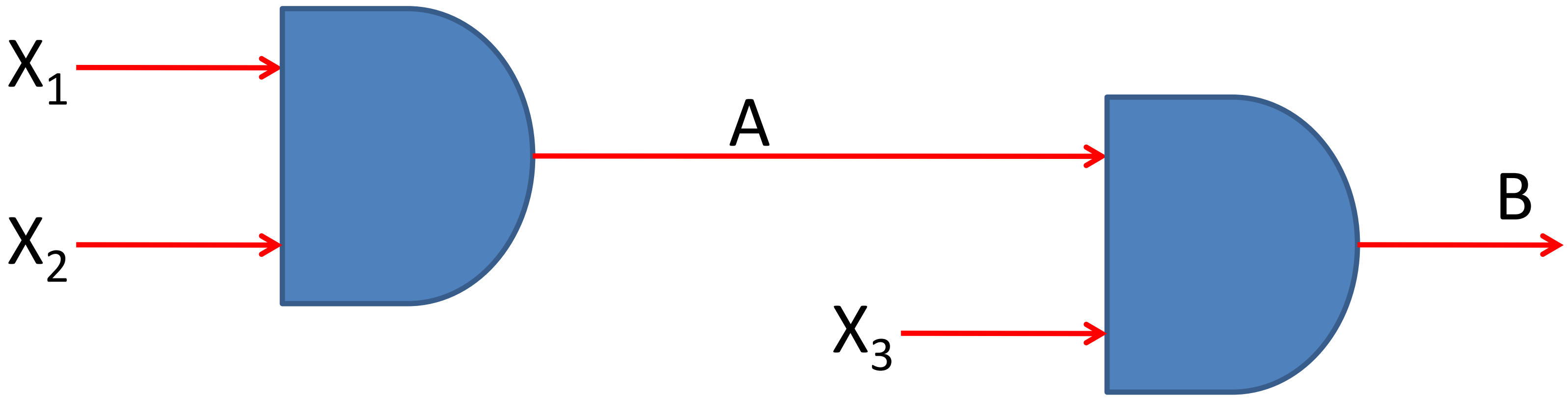
# Hardware For TestProcess2B



# Hardware For TestProcess2B



# Hardware For TestProcess2B



# Code For Example 3

```
TestProcess3A : process ($X_1$, $X_2$)
Begin
        Y <= $X_1$ or $X_2$;
        Y <= $X_1$ and $X_2$;
end process;

TestProcess3B : process ($X_1$, $X_2$)
Begin
        Y <= $X_1$ and $X_2$;
        Y <= $X_1$ or $X_2$;
end process;
```

# Code For Example 3

```
TestProcess3A : process ($X_1$, $X_2$)
Begin
        Y <= $X_1$ or $X_2$;
        Y <= $X_1$ and $X_2$;
end process;

TestProcess3B : process ($X_1$, $X_2$)
Begin
        Y <= $X_1$ and $X_2$;
        Y <= $X_1$ or $X_2$;
end process;
```

# Code For Example 3

```
TestProcess3A : process ($X_1$, $X_2$)
Begin
        Y <= $X_1$ or $X_2$; -- this is ignored !
        Y <= $X_1$ and $X_2$; -- this is implemented.
end process;

TestProcess3B : process ($X_1$, $X_2$)
Begin
        Y <= $X_1$ and $X_2$;
        Y <= $X_1$ or $X_2$;
end process;
```

# Code For Example 3

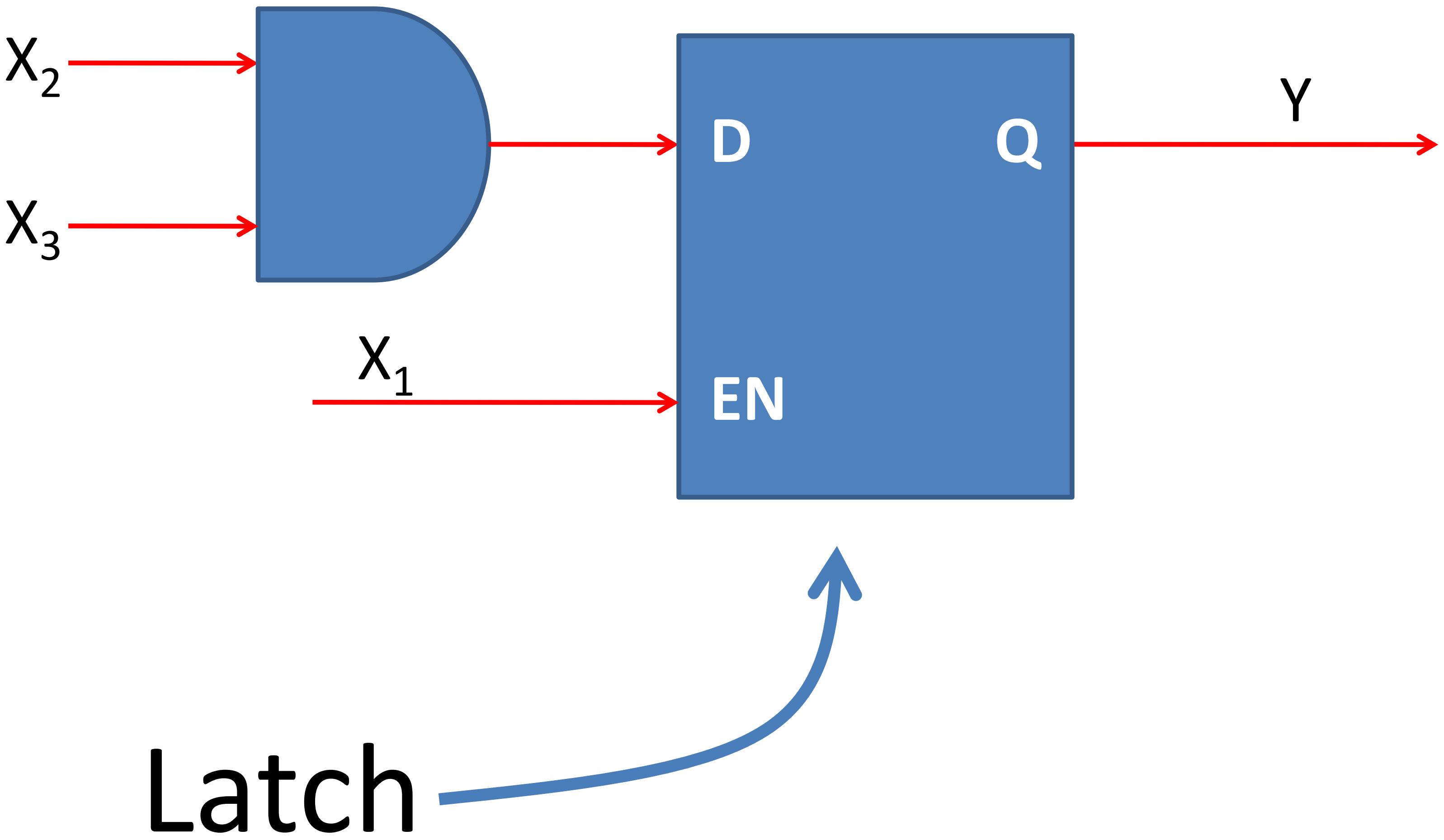
```
TestProcess3A : process ($X_1$, $X_2$)
Begin
        Y <= $X_1$ or $X_2$; -- this is ignored !
        Y <= $X_1$ and $X_2$; -- this is implemented.
end process;

TestProcess3B : process ($X_1$, $X_2$)
Begin
        Y <= $X_1$ and $X_2$; -- this is ignored !
        Y <= $X_1$ or $X_2$; -- this is implemented.
end process;
```

# Code For Example 4

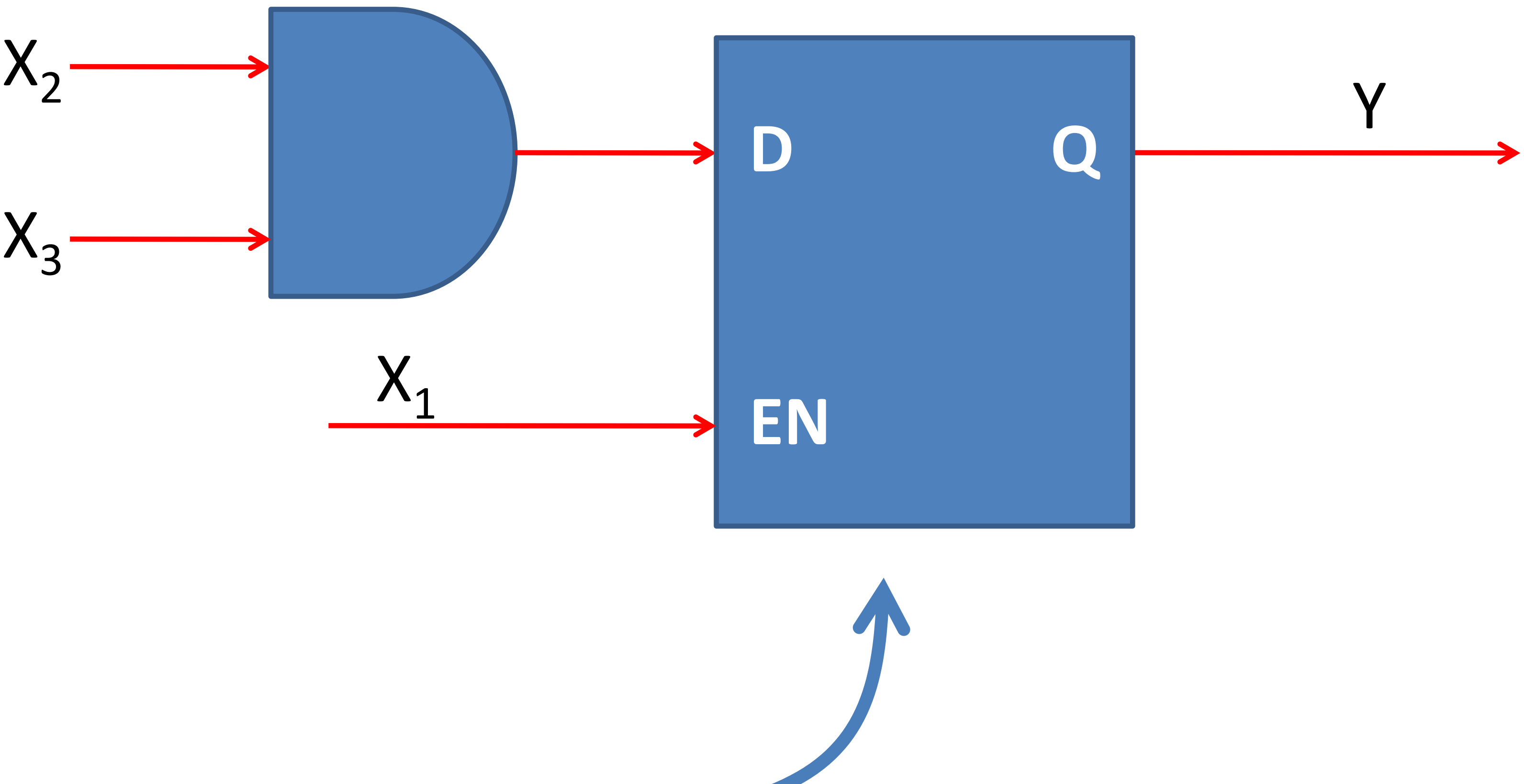
```
TestProcess4 : process ($X_1$, $X_2$, $X_3$)
Begin
    if $X_1$ = ‘1’ then
        Y <= $X_2$ and $X_3$;
    end if;
end process;
```

# Code For Example 4

```
TestProcess4 : process ($X_1$, $X_2$, $X_3$)
Begin
        if $X_1$ = ‘1’ then
                Y <= $X_2$ and $X_3$;
        end if;
end process;
```



Latch



Latch

# More On Latches

- Latches can be created in **Combinational** processes.
- Latches are created when the state of the output is **not** defined in all paths through the process.
- A process can have **multiple paths** due to **branch statements** (if statement, case statement).
- Make sure your outputs are assigned in **all** possible branches of the code (In combinational processes)

# More On Latches

- Latches can be created in **Combinational** processes.
- Latches are created when the state of the output is **not** defined in all paths through the process.
- A process can have **multiple paths** due to **branch statements** (if statement, case statement).
- Make sure your outputs are assigned in **all** possible branches of the code (In combinational processes)

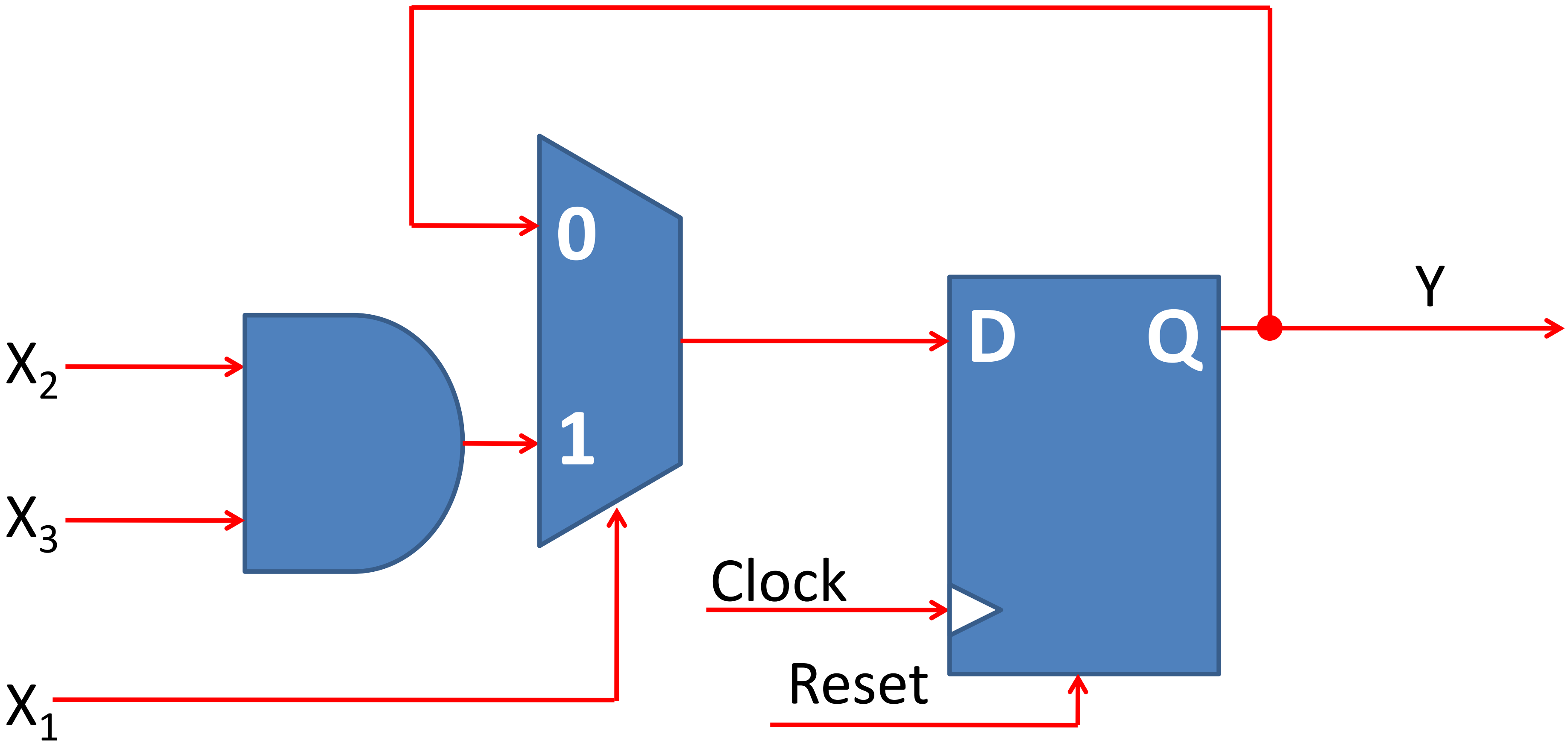
# Avoiding Latches

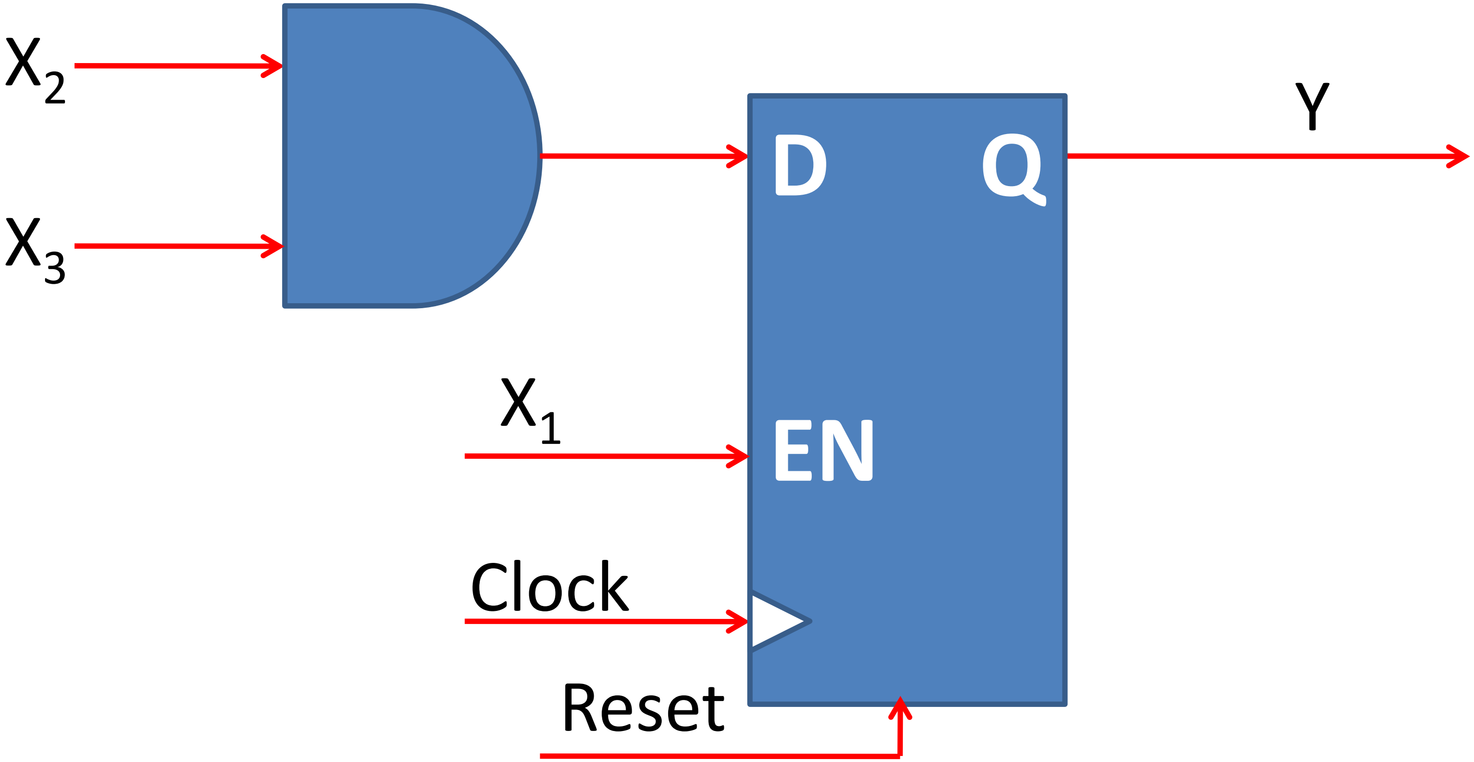
```
TestProcess : process ($X_1$, $X_2$, $X_3$)
Begin
        if $X_1$ = ‘1’ then
                Y <= $X_2$ and $X_3$;
        else
                Y <= ‘0’;
        end if;
end process;
```

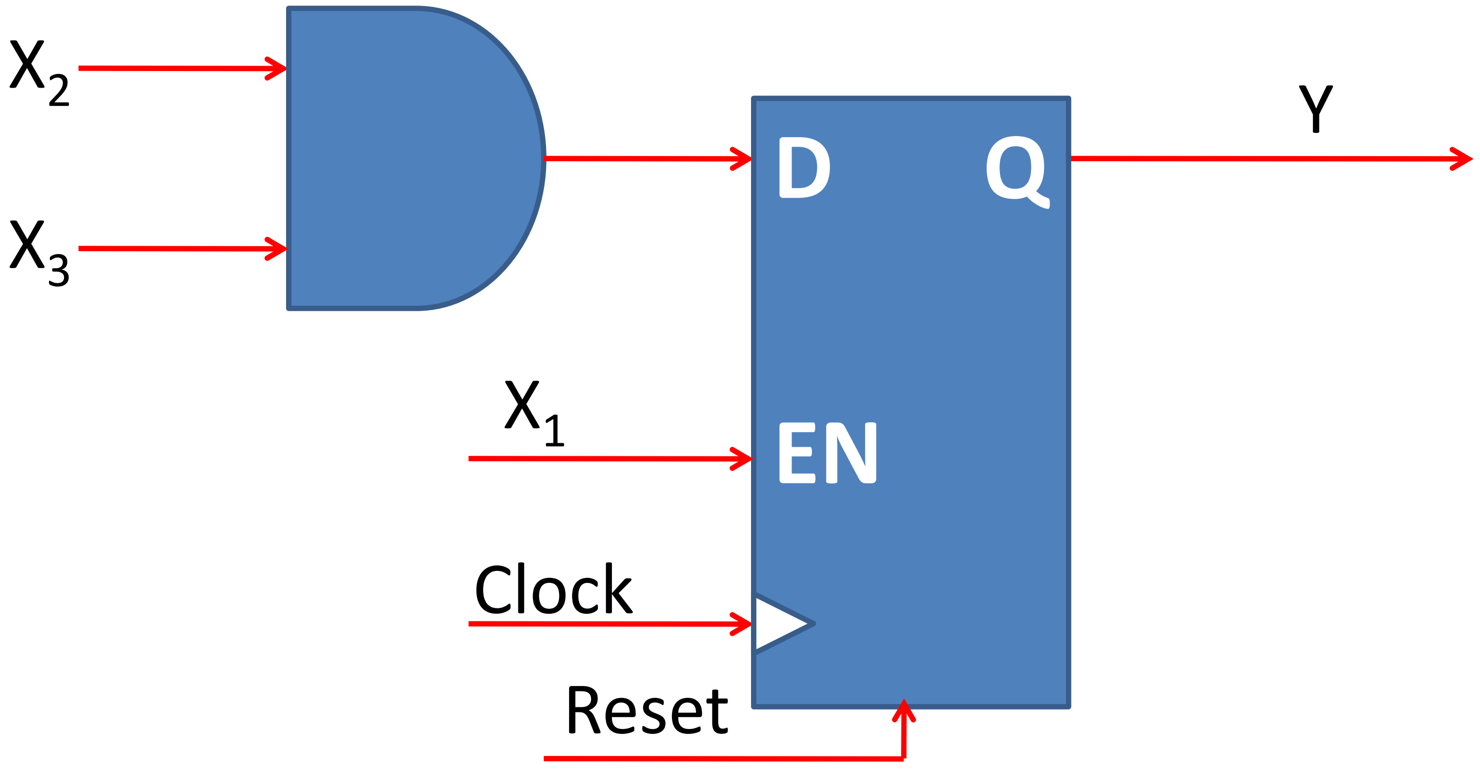
# Avoiding Latches

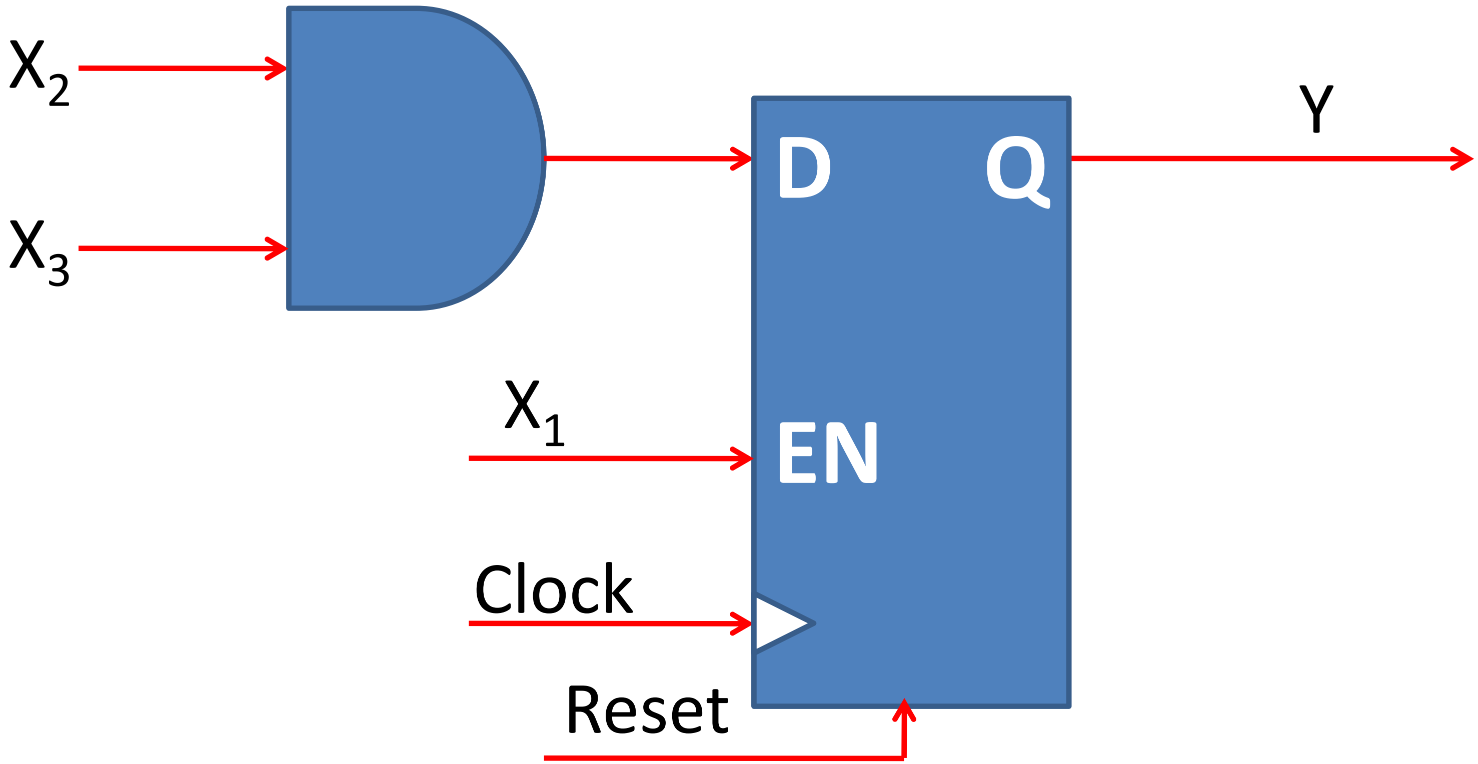
```
TestProcess : process ($X_1$, $X_2$, $X_3$)
Begin
        if $X_1$ = ‘1’ then
                Y <= $X_2$ and $X_3$;
        else
                Y <= ‘0’;
        end if;
end process;
```

```
TestProcess : process (Reset, Clock)
Begin
        if Reset = ‘1’ then
           Y <= ‘0’;
        elsif rising_edge(Clock)  then
            if $X_1$ = ‘1’ then
                Y <= $X_2$ and $X_3$;
            end if;
        end if;
end process;
```


0
1
D
Q
Y
$X_2$
$X_3$
$X_1$
Clock
Reset


$X_2$
$X_3$
D
Q
Y
$X_1$
EN
Clock
Reset


$X_2$
$X_3$
D
Q
Y
$X_1$
EN
Clock
Reset


$X_2$
$X_3$
D
Q
Y
$X_1$
EN
Clock
Reset

# Signal Assignments Inside The Process Block

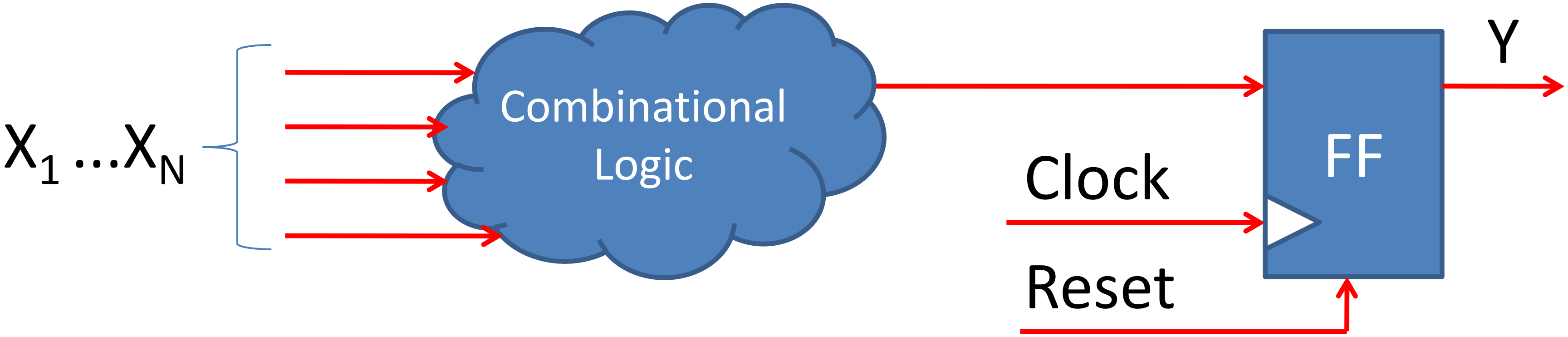
# Part II

```
TestProcess : process(Reset, Clock)
begin





end process;
```

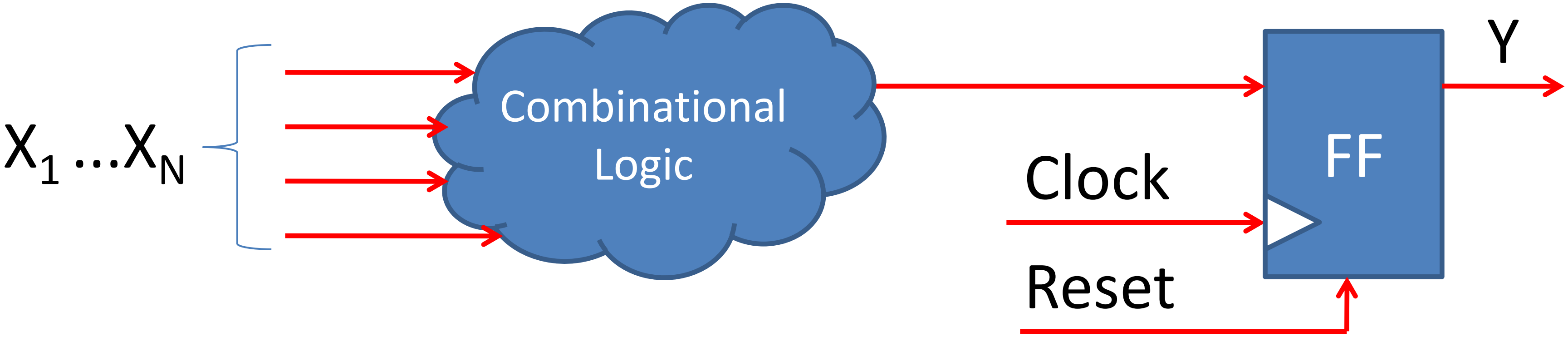
```
TestProcess : process(Reset, Clock)
begin
    if Reset = ‘1’ then
       Y <= ‘0’;




end process;
```

```
TestProcess : process(Reset, Clock)
begin
    if Reset = ‘1’ then
        Y <= ‘0’;
    elsif rising_edge(Clock) then
        Y <= Function(X1, X2, X3,... XN);
    end if;
end process;
```

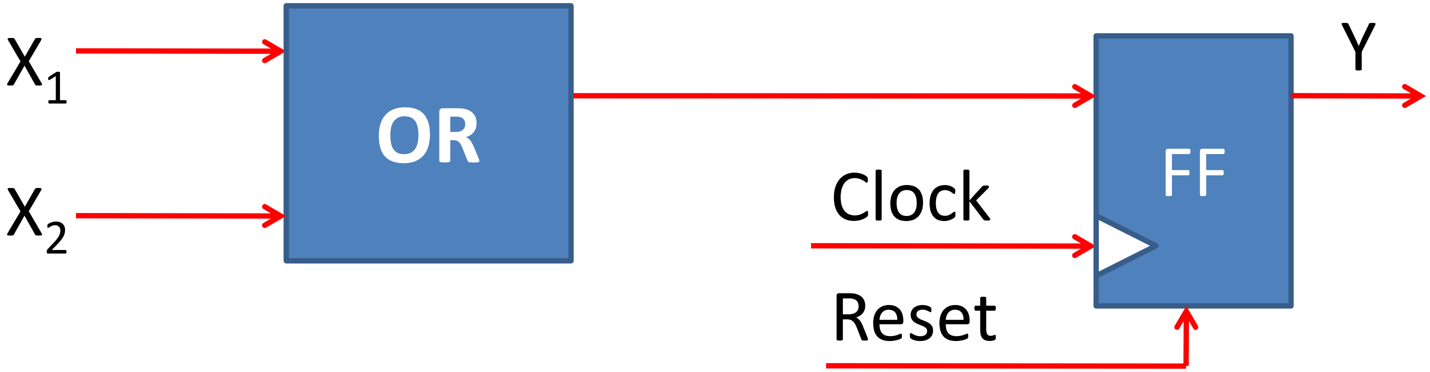
```
TestProcess : process(Reset, Clock)
begin
    if Reset = '1' then
       Y <= '0';
    elsif rising_edge(Clock) then
       Y <= Function(X1, X2, X3,... XN);
   end if;
end process;
```



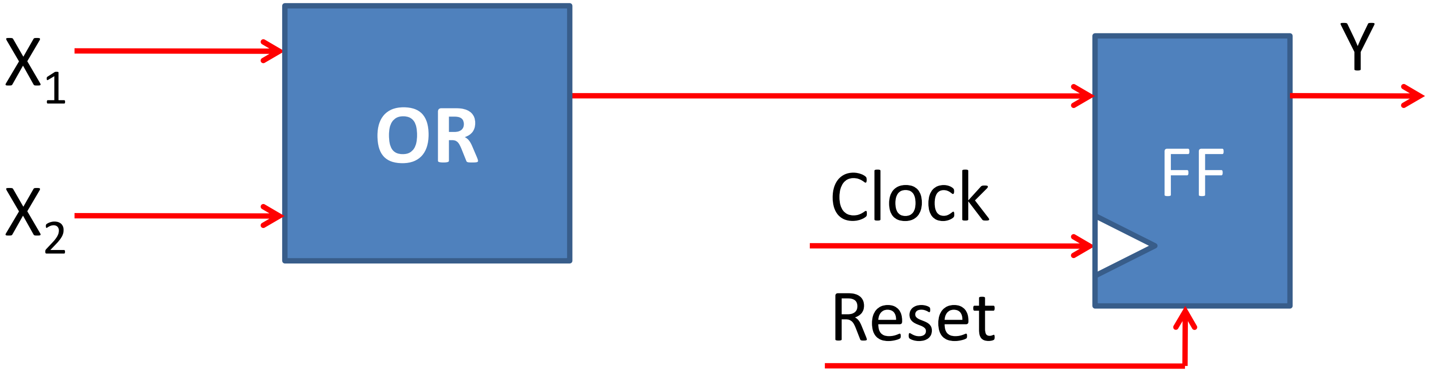
```
TestProcess : process(Reset, Clock)
begin
    if Reset = ‘1’ then
       Y <= ‘0’;
    elsif rising_edge(Clock) then
       Y <= Function(X1, X2, X3,... XN);
    end if;
end process;
```



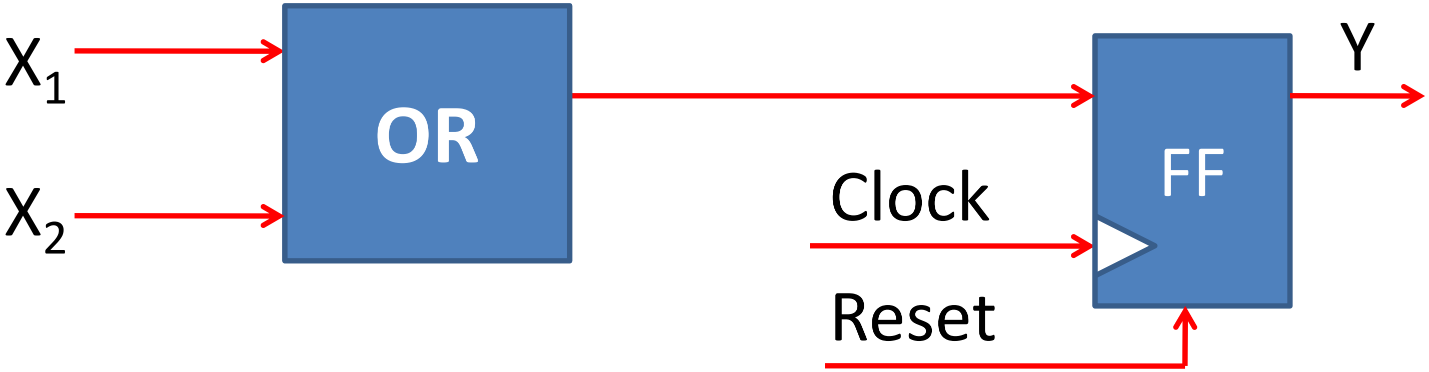
```
TestProcess : process(Reset, Clock)
begin
    if Reset = ‘1’ then
        Y <= ‘0’;
    elsif rising_edge(Clock) then
        Y <= X1 or X2;
    end if;
end process;
```



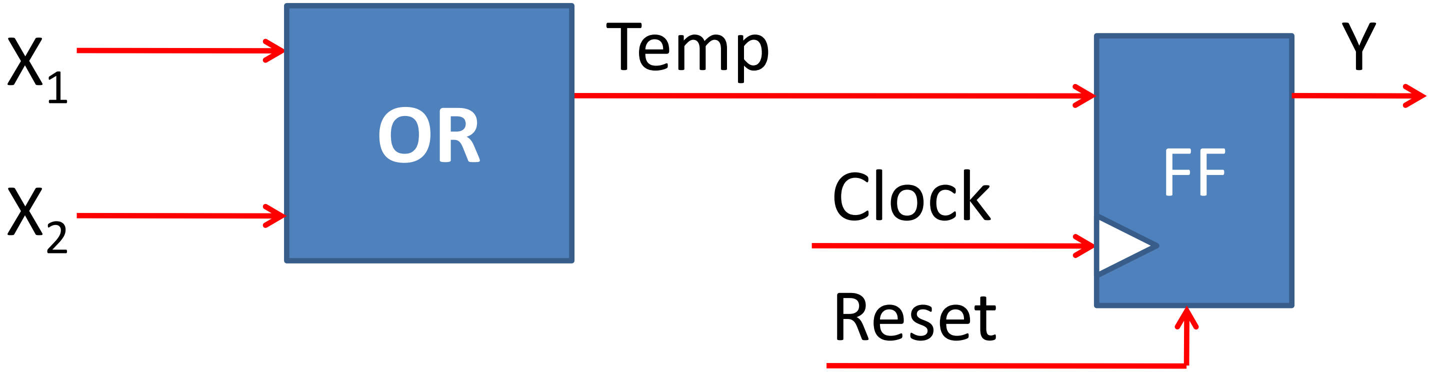
```
TestProcess : process(Reset, Clock)
begin
    if Reset = ‘1’ then
        Y <= ‘0’;
    elsif rising_edge(Clock) then
        Y <= X1 or X2;
    end if;
end process;
```

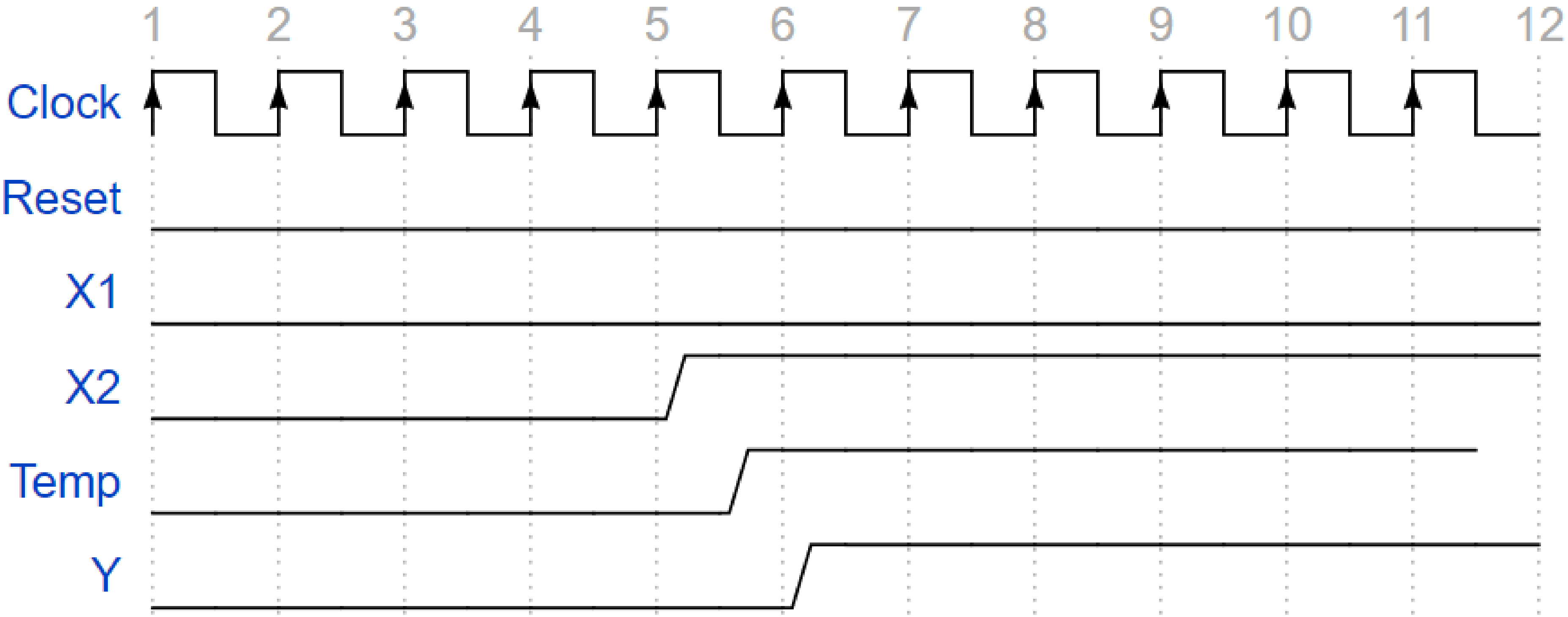


```
TestProcess : process(Reset, Clock)
begin
   if Reset = '1' then
      Y <= '0';
   elsif rising_edge(Clock) then
      Y <= X1 or X2;
   end if;
end process;
```

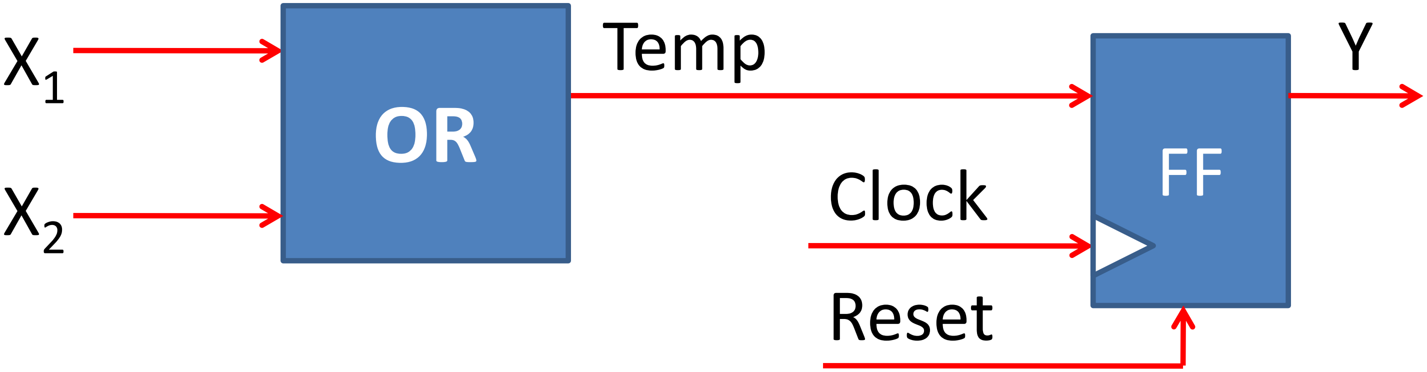


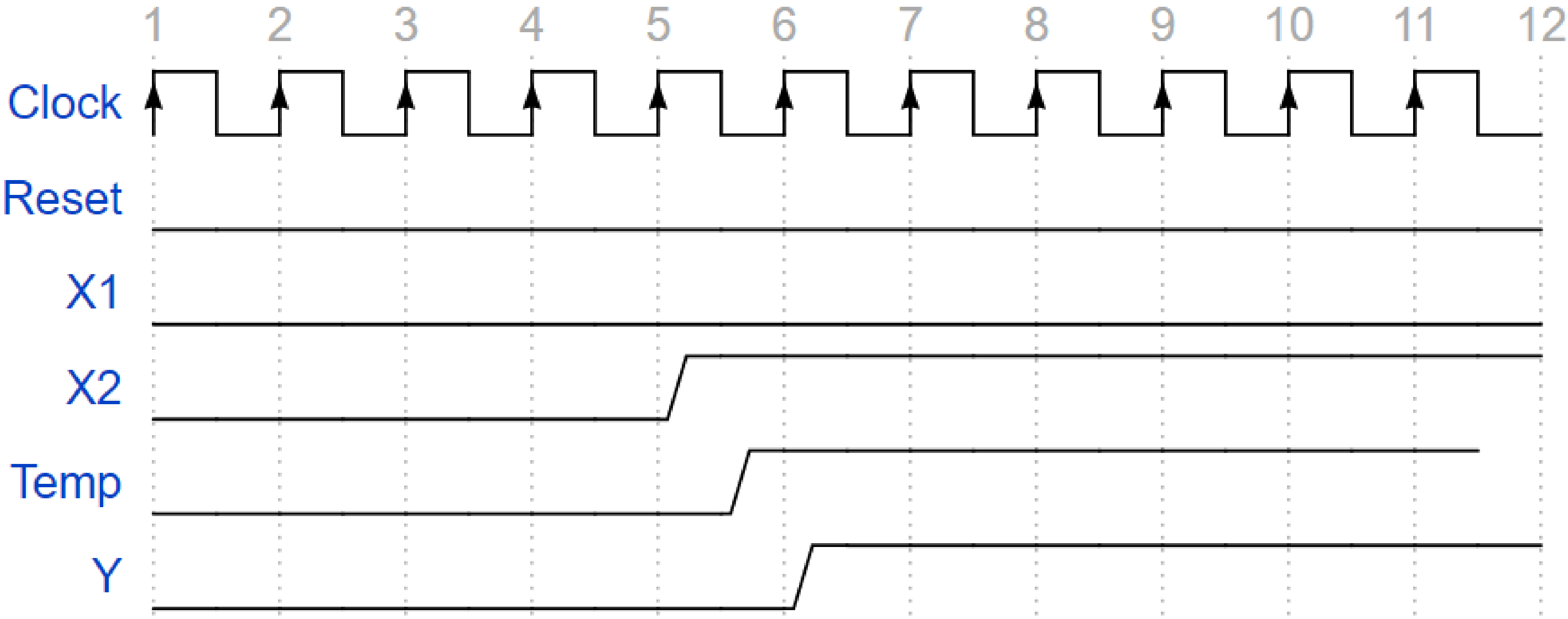
# Timing Of Signal Y



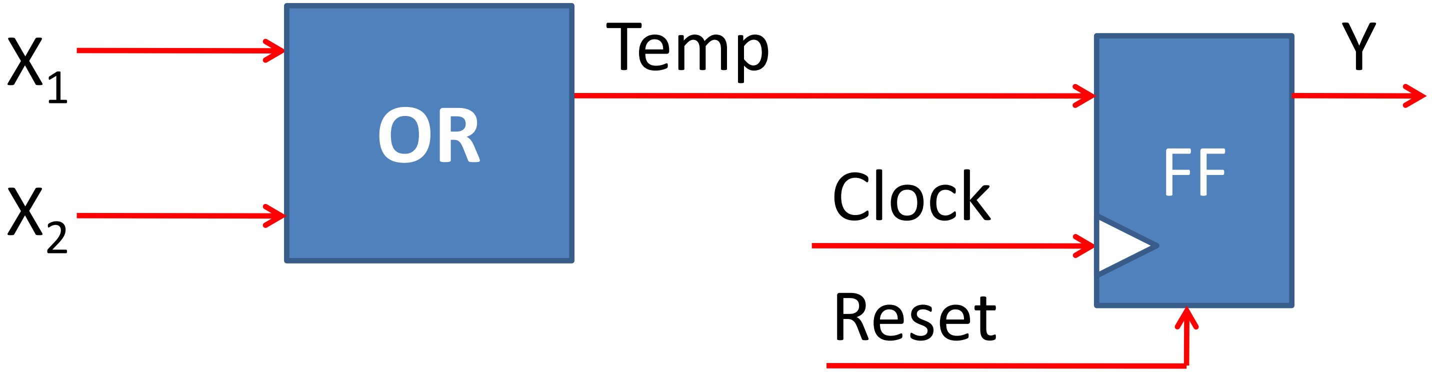


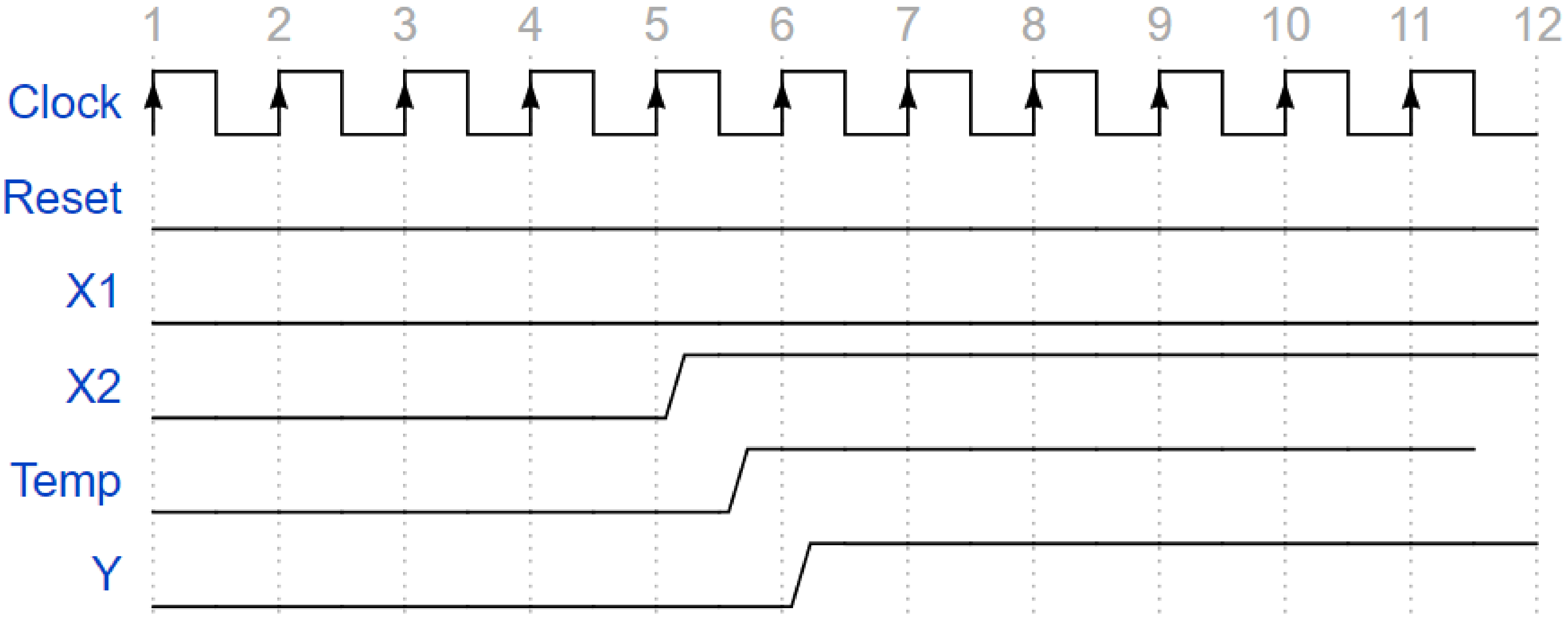
# Timing Of Signal Y



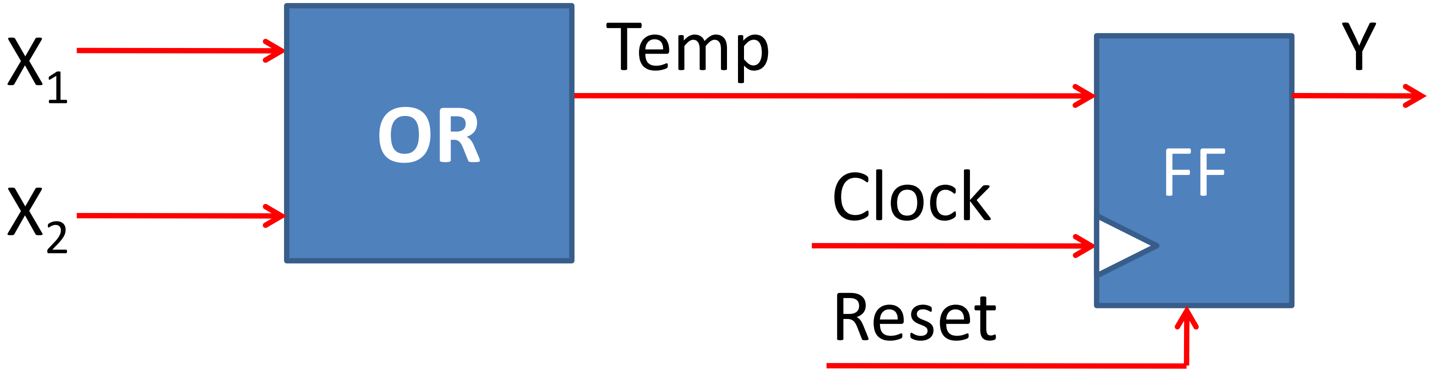


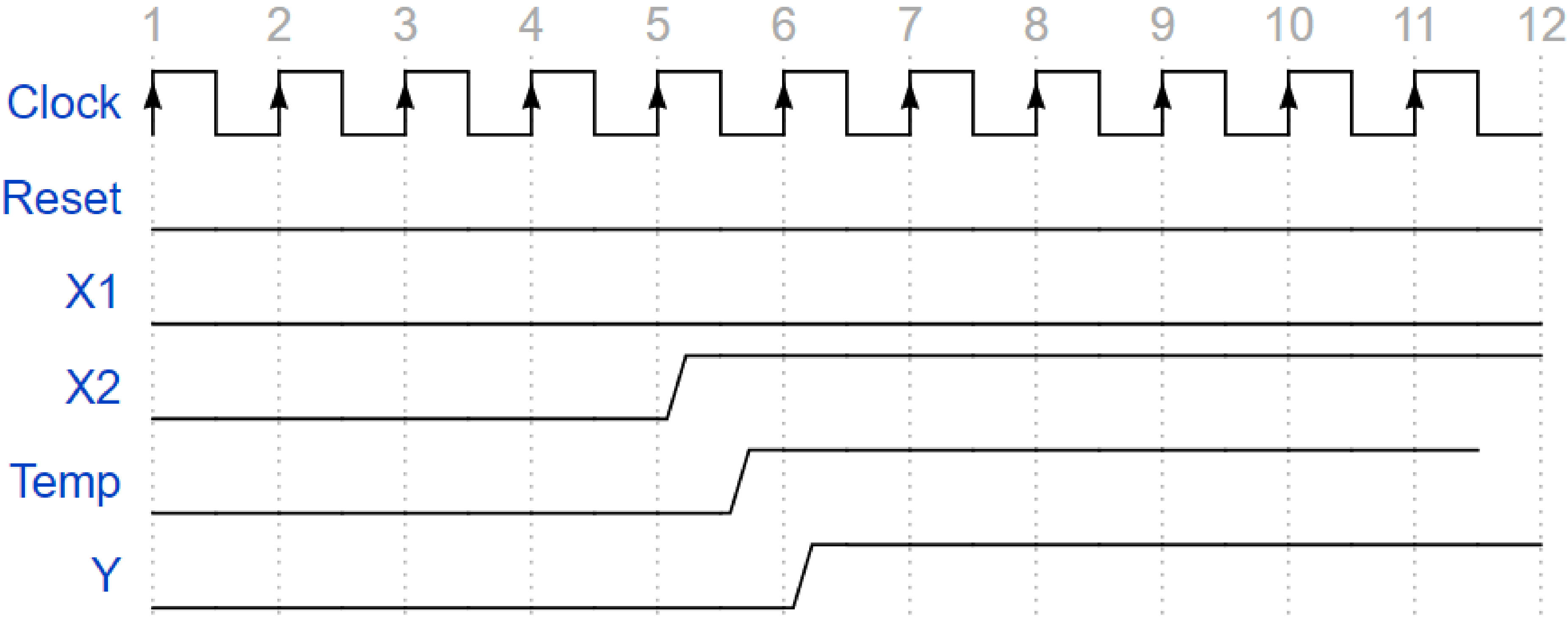
# Timing Of Signal Y





# Timing Of Signal Y



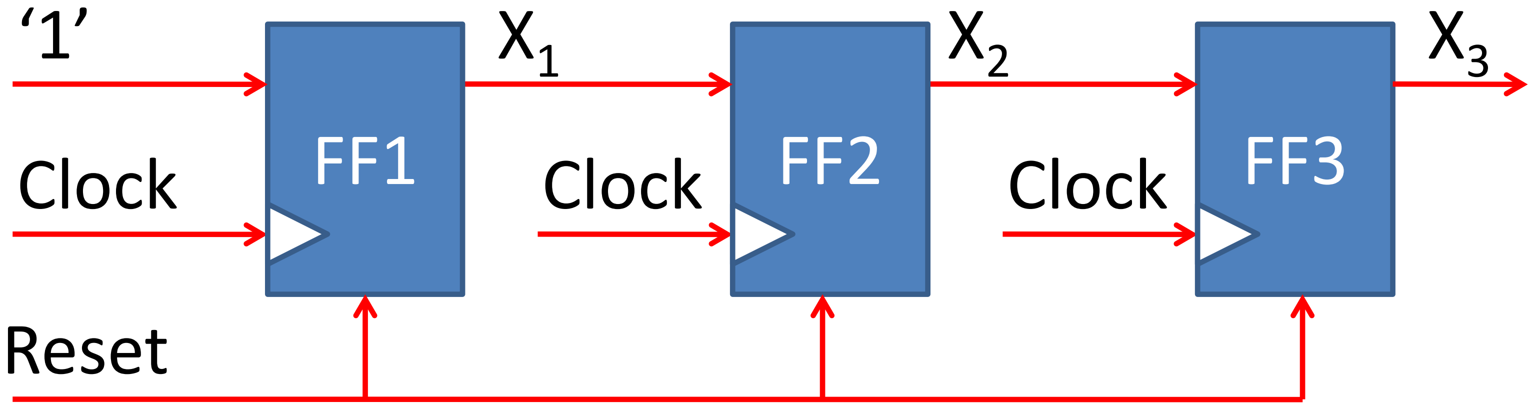


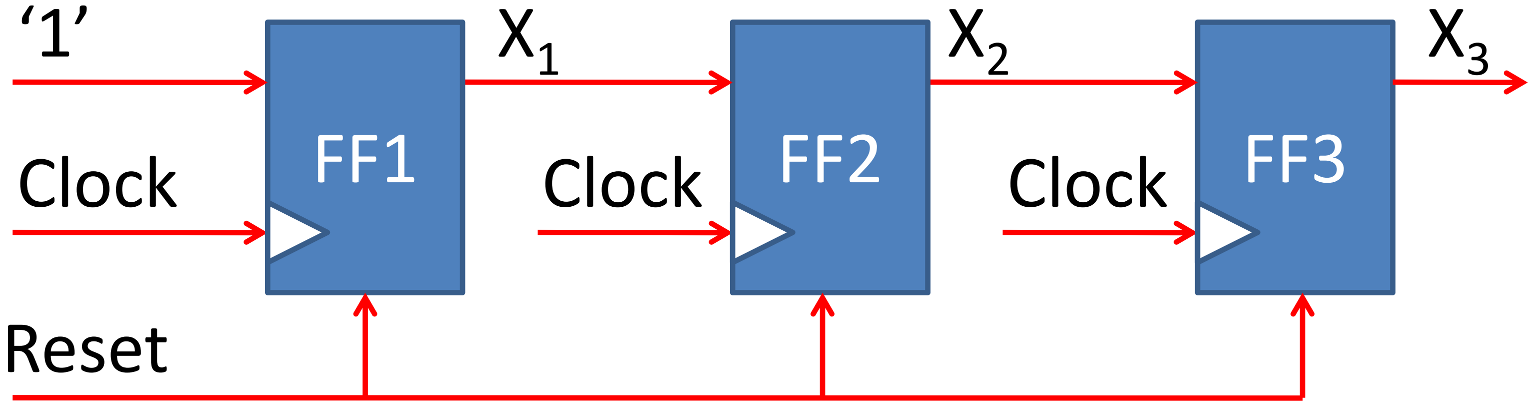
```
TestProcess : process(Reset, Clock)
begin
    if Reset = ‘1’ then
        X1 <= ‘0’;
        X2 <= ‘0’;
        X3 <= ‘0’;
    elsif rising_edge(Clock) then
         X1 <= ‘1’;
         X2 <= X1;
         X3 <= X2;
    end if;
end process;
```

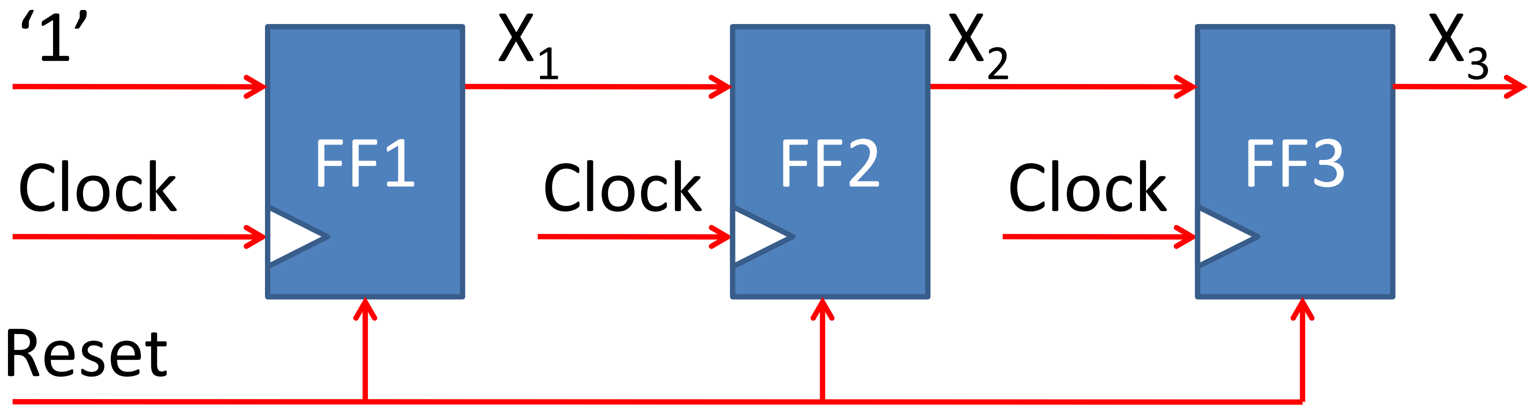
```
TestProcess : process(Reset, Clock)
begin
    if Reset = ‘1’ then
        X1 <= ‘0’;
        X2 <= ‘0’;
        X3 <= ‘0’;
    elsif rising_edge(Clock) then
         X1 <= ‘1’;
         X2 <= X1;
         X3 <= X2;
    end if;
end process;
```

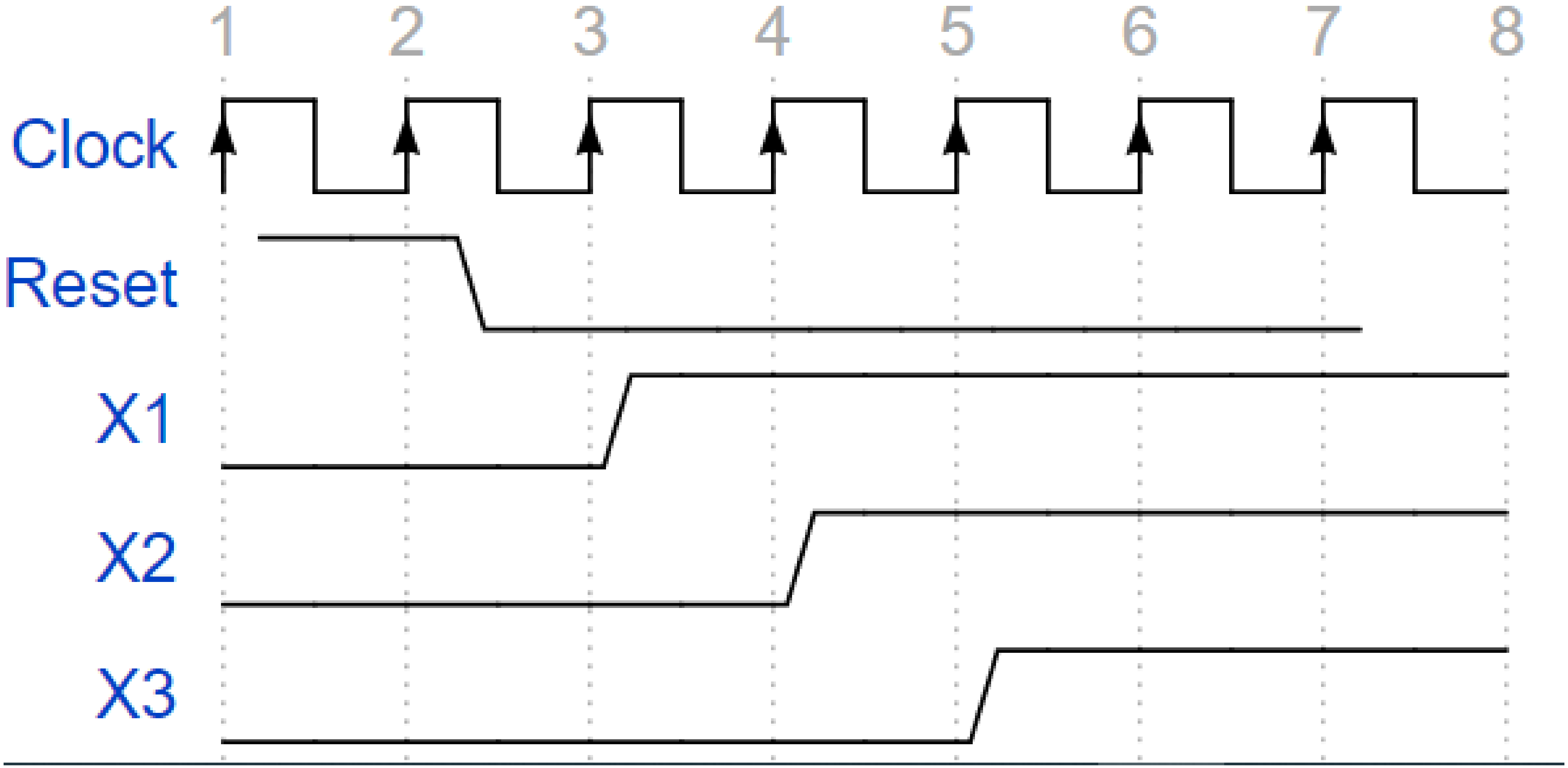
```
TestProcess : process(Reset, Clock)
begin
    if Reset = ‘1’ then
        X1 <= ‘0’;
        X2 <= ‘0’;
        X3 <= ‘0’;
    elsif rising_edge(Clock) then
         X1 <= ‘1’;
         X2 <= X1;
         X3 <= X2;
    end if;
end process;
```

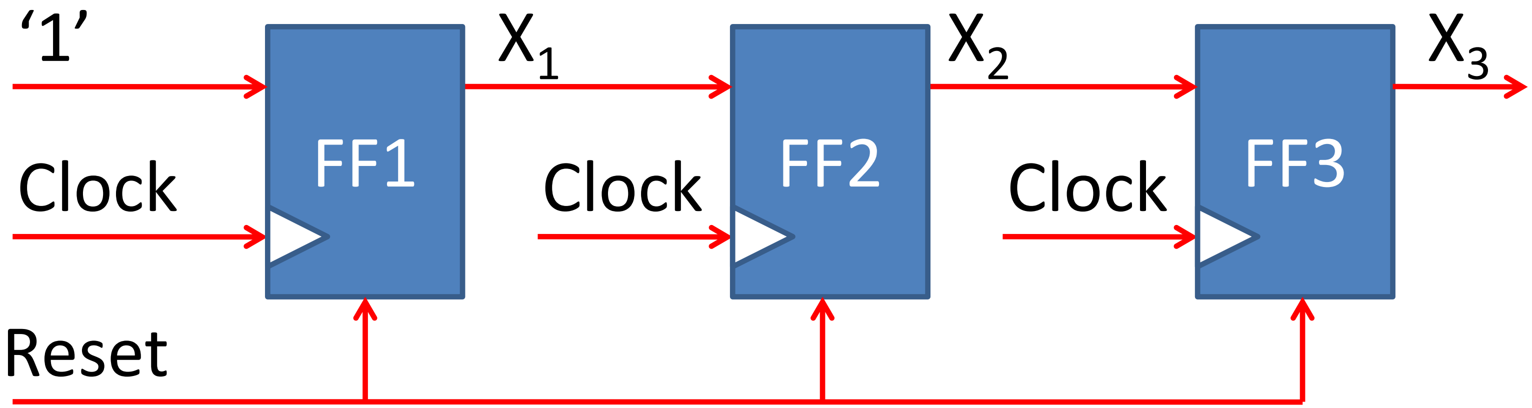
```
TestProcess : process(Reset, Clock)
begin
    if Reset = ‘1’ then
        X1 <= ‘0’;
        X2 <= ‘0’;
        X3 <= ‘0’;
    elsif rising_edge(Clock) then
         X1 <= ‘1’;
         X2 <= X1;
         X3 <= X2;
    end if;
end process;
```

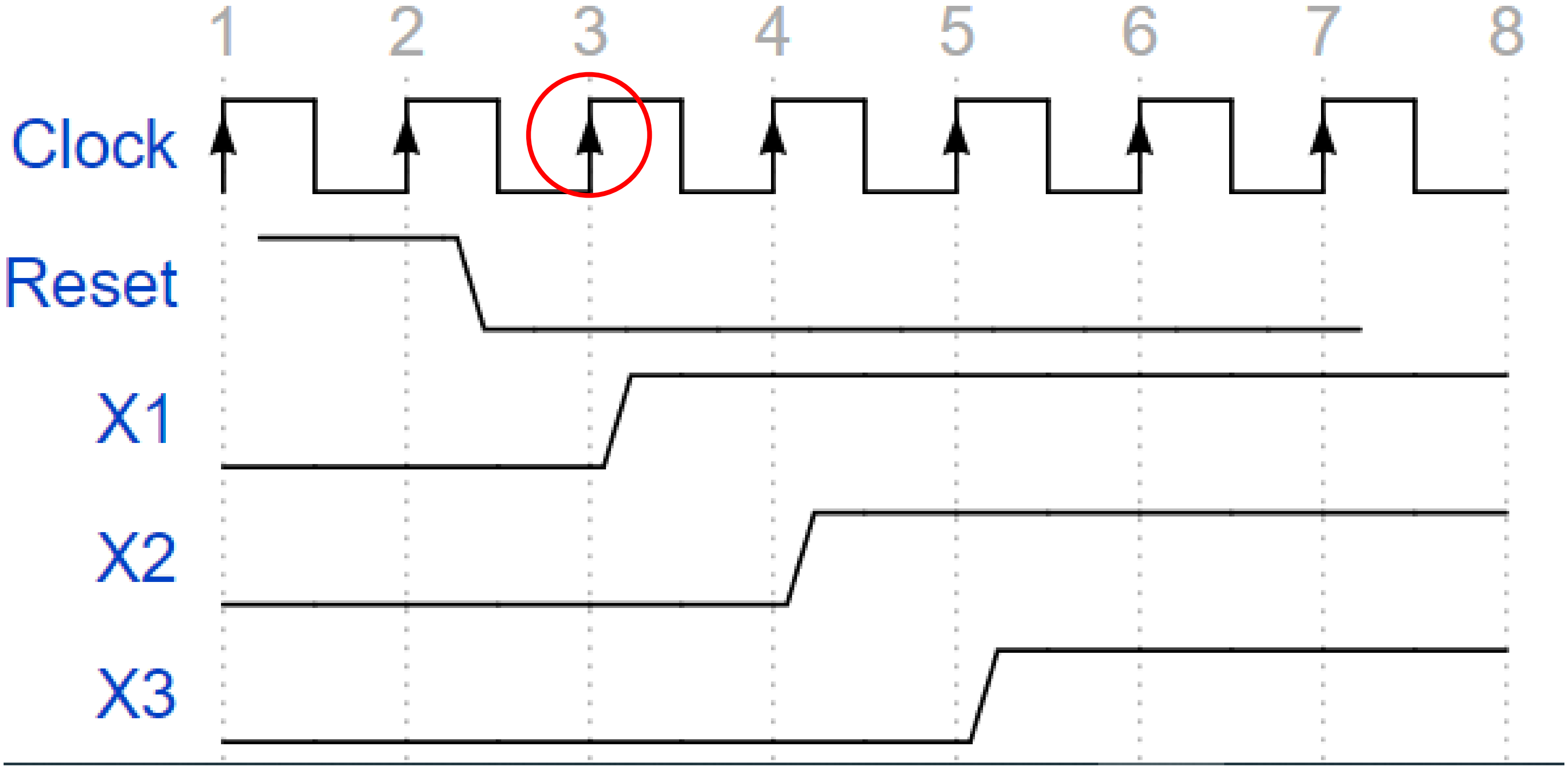

'1'
Clock
Reset
FF1
X1
Clock
FF2
X2
Clock
FF3
X3

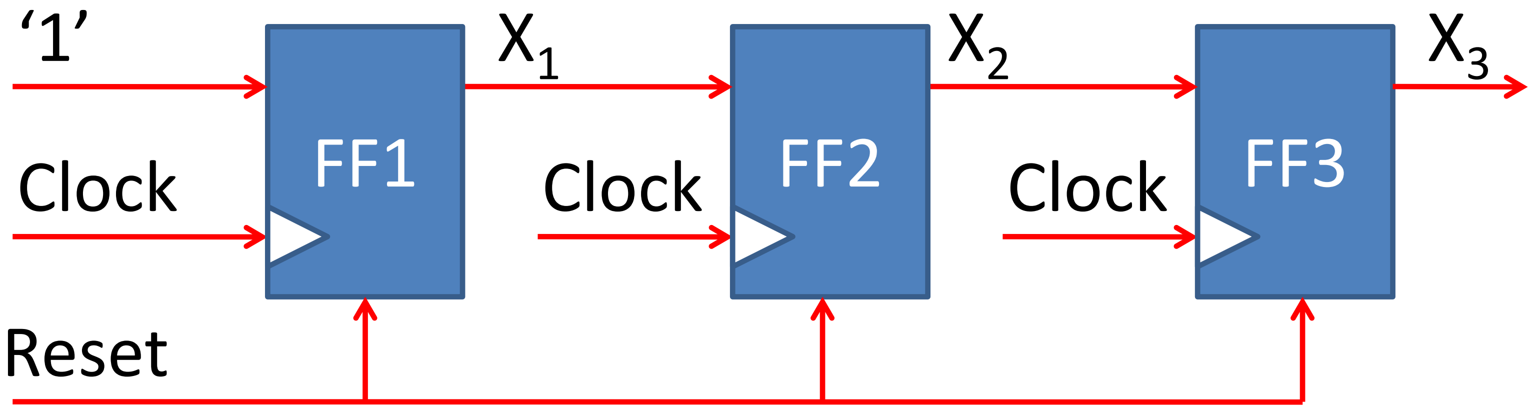

‘1’
Clock
Reset
FF1
$X_1$
Clock
FF2
$X_2$
Clock
FF3
$X_3$

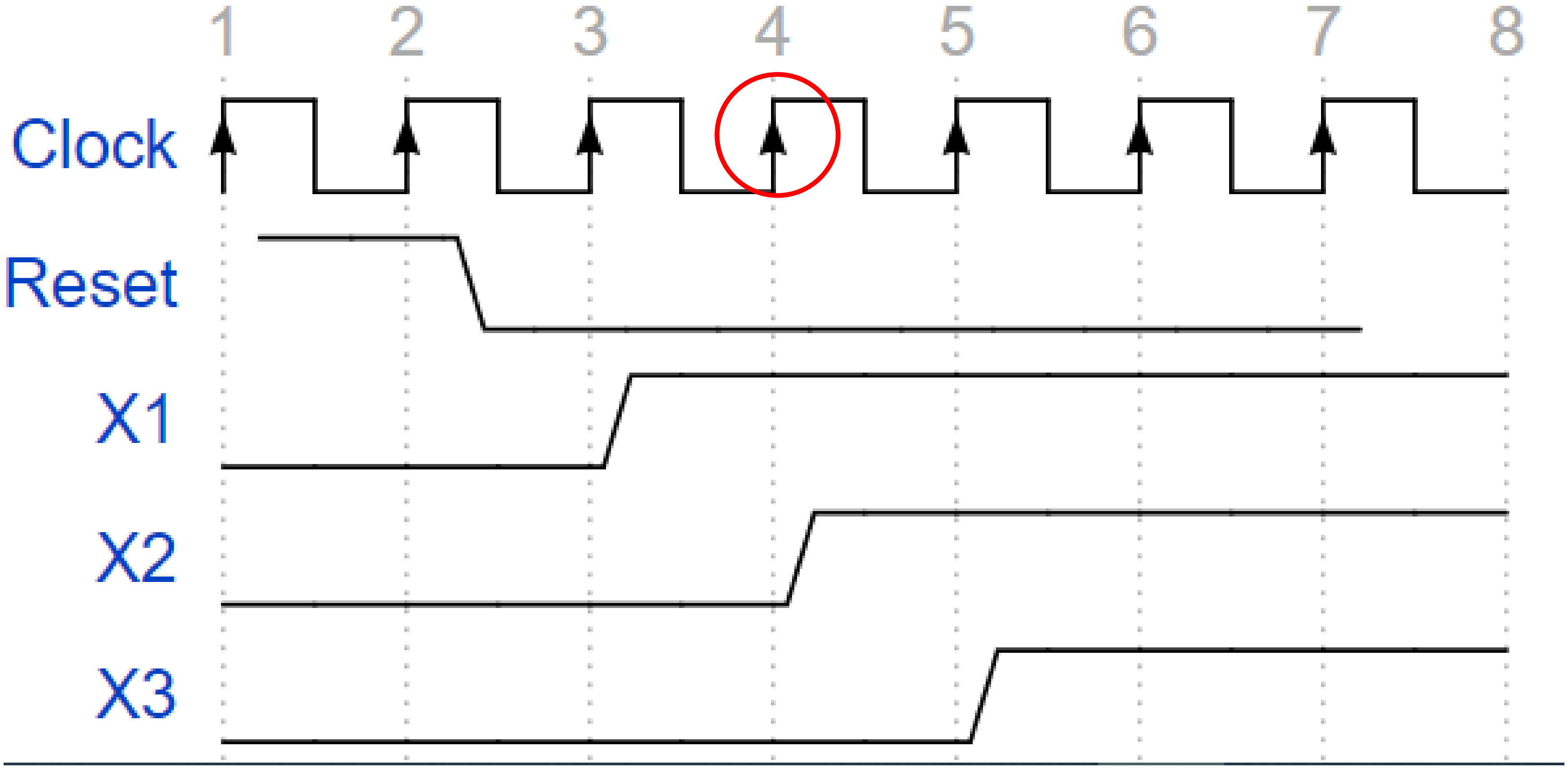

'1'
Clock
Reset
FF1
$X_1$
Clock
FF2
$X_2$
Clock
FF3
$X_3$

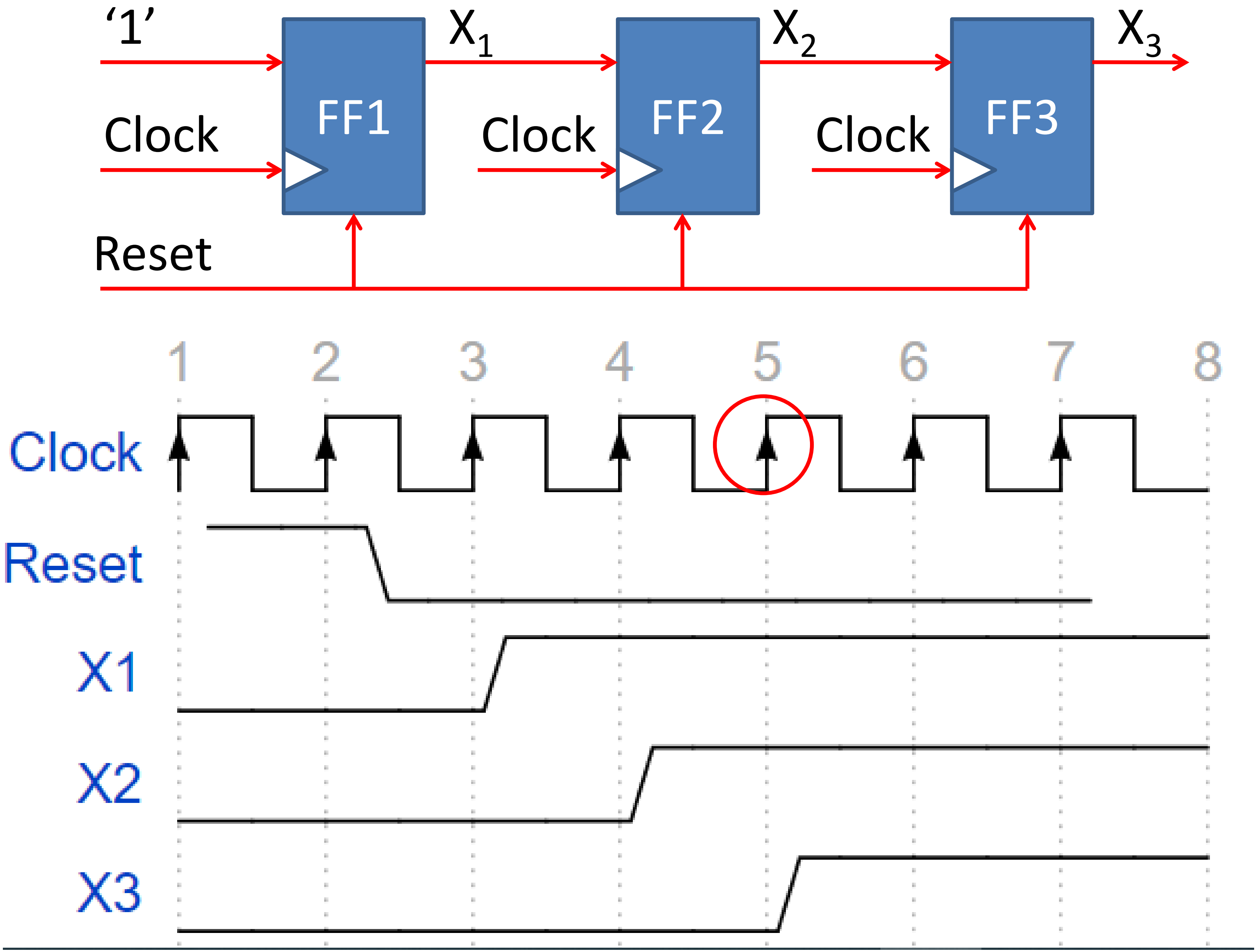

1
2
3
4
5
6
7
8
Clock
Reset
X1
X2
X3


'1'
FF1
$X_1$
FF2
$X_2$
FF3
$X_3$
Clock
Clock
Clock
Reset

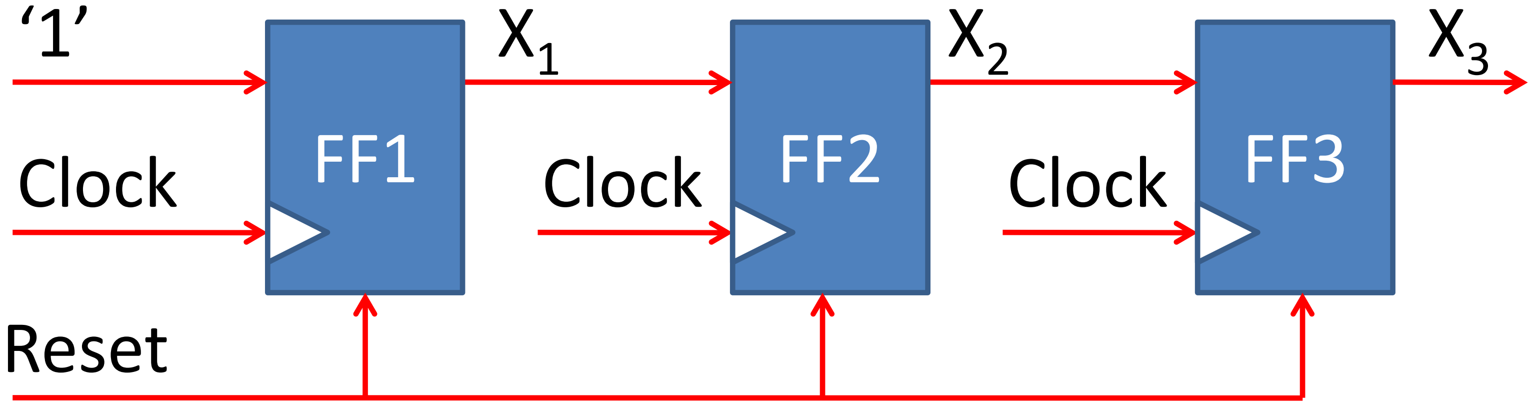

1
2
3
4
5
6
7
8
Clock
Reset
X1
X2
X3

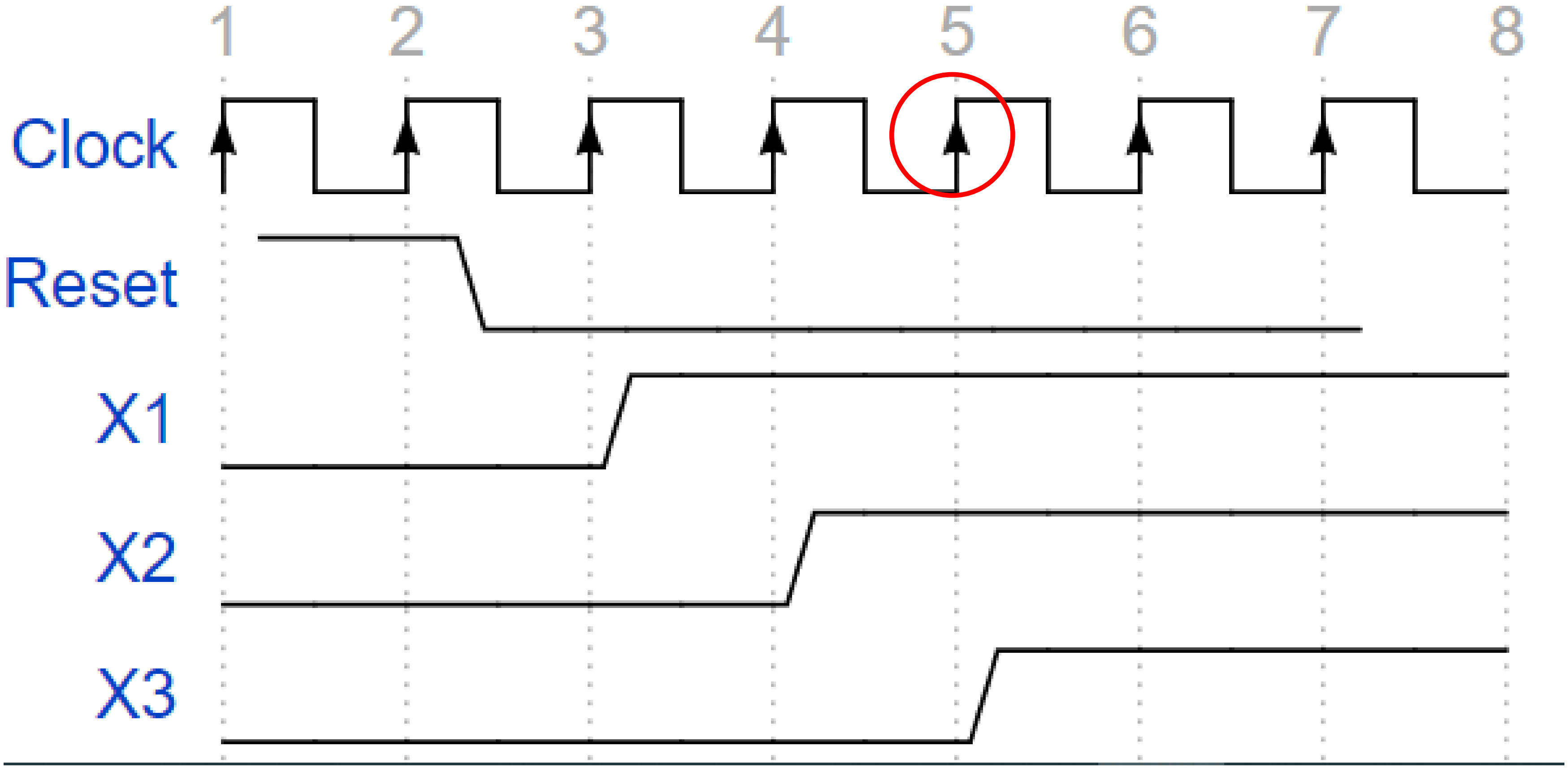

'1'
FF1
$X_1$
FF2
$X_2$
FF3
$X_3$
Clock
Clock
Clock
Reset


1
2
3
4
5
6
7
8
Clock
Reset
X1
X2
X3



'1'
X1
X2
X3
Clock
FF1
Clock
FF2
Clock
FF3
Reset
1
2
3
4
5
6
7
8
Clock
Reset
X1
X2
X3


'1'
X1
X2
X3
FF1
FF2
FF3
Clock
Clock
Clock
Reset


1
2
3
4
5
6
7
8
Clock
Reset
X1
X2
X3

```
TestProcess : process(Reset, Clock)
begin
    if Reset = ‘1’ then
        $X_1$ <= ‘0’;
        $X_2$ <= ‘0’;
    elsif rising_edge(Clock) then
         $X_1$ <= ‘1’;
         $X_2$ <= ‘1’;
    end if;
end process;
```

```
TestProcess : process(Reset, Clock)
begin
    if Reset = ‘1’ then
        X_1 <= ‘0’;
        X_2 <= ‘0’;
    elsif rising_edge(Clock) then
         X_1 <= ‘1’;
         X_2 <= ‘1’;
    end if;
end process;
```

```
TestProcess : process(Reset, Clock)
begin
    if Reset = ‘1’ then
        $X_1$ <= ‘0’;
elsif rising_edge(Clock) then
        $X_1$ <= $X_2$;
        $X_1$ <= $X_3$;
    end if;
end process;
```

```
TestProcess : process(Reset, Clock)
begin
    if Reset = ‘1’ then
        $X_1$ <= ‘0’;
elsif rising_edge(Clock) then
        $X_1$ <= $X_2$; -- VHDL will ignore this line!
        $X_1$ <= $X_3$;
    end if;
end process;
```

End Of Video

# Variables

# What Are Variables?

- Stores intermediate values within a **process**, **function** or **procedure**.

# What Are Variables?

- Stores intermediate values within a **process**, **function** or **procedure**.
- Cannot be used outside of process, function or procedure.

# What Are Variables?

- Stores intermediate values within a **process**, **function** or **procedure**.
- Cannot be used outside of process, function or procedure.
- Variable declaration is similar to signal declaration.

# What Are Variables?

- Stores intermediate values within a **process**, **function** or **procedure**.
- Cannot be used outside of process, function or procedure.
- Variable declaration is similar to signal declaration.
- Like signals, the initial value of a variable is ignored by synthesis!

```
TestProcess : process(Reset, Clock)
    variable X : integer; -- Declaration
begin









end process;
```

```vhdl
TestProcess : process(Reset, Clock)
      variable X : integer; -- Declaration
begin
    if Reset = ‘1’ then
        X := 0; -- Variable assignment
        Y <= 0;
    elsif rising_edge(Clock) then




    end if;
end process;
```

```
TestProcess : process(Reset, Clock)
      variable X : integer; -- Declaration
begin
    if Reset = ‘1’ then
       X := 0; -- Variable assignment
       Y <= 0;
    elsif rising_edge(Clock) then
       X := 7; -- X takes the value 7 immediately.




    end if;
end process;
```

```
TestProcess : process(Reset, Clock)
      variable X : integer; -- Declaration
begin
    if Reset = ‘1’ then
       X := 0; -- Variable assignment
       Y <= 0;
    elsif rising_edge(Clock) then
       X := 7; -- X takes the value 7 immediately.
       X := X + 1; -- X becomes 8 here.

    end if;
end process;
```

```
TestProcess : process(Reset, Clock)
      variable X : integer; -- Declaration
begin
    if Reset = ‘1’ then
      X := 0; -- Variable assignment
      Y <= 0;
    elsif rising_edge(Clock) then
      X := 7; -- X takes the value 7 immediately.
      X := X + 1; -- X becomes 8 here.

    end if;
end process;
```

|  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|
| A | 15 | 7 | 3 | 57 | 49 | 8 | 6 | 27 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **A** | 15 | 7 | 3 | 57 | 49 | 8 | 6 | 27 |

```
TestProcess : process(Reset, Clock)
      variable Temp : integer; -- Declaration
begin


end process;
```

|  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|
| **A** | 15 | 7 | 3 | 57 | 49 | 8 | 6 | 27 |

```
TestProcess : process(Reset, Clock)
      variable Temp : integer; -- Declaration
begin
    if Reset = ‘1’ then
       Temp := 0;
       Sum <= 0;
    elsif rising_edge(Clock) then




     end if;
end process;
```

|  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|
| **A** | 15 | 7 | 3 | 57 | 49 | 8 | 6 | 27 |

```
TestProcess : process(Reset, Clock)
        variable Temp: integer; -- Declaration
begin
     if Reset = ‘1’ then
         Temp := 0;
         Sum <= 0;
     elsif rising_edge(Clock) then

         for i in 0 to 7 loop
              Temp := Temp + A(i);
         end loop;

      end if;
end process;
```

|  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|
| **A** | 15 | 7 | 3 | 57 | 49 | 8 | 6 | 27 |

```
TestProcess : process(Reset, Clock)
      variable Temp: integer; -- Declaration
begin
    if Reset = ‘1’ then
       Temp := 0;
       Sum <= 0;
    elsif rising_edge(Clock) then
       Temp := 0;
       for i in 0 to 7 loop
           Temp := Temp + A(i);
       end loop;

     end if;
end process;
```

|  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|
| **A** | 15 | 7 | 3 | 57 | 49 | 8 | 6 | 27 |

```
TestProcess : process(Reset, Clock)
      variable Temp: integer; -- Declaration
begin
    if Reset = ‘1’ then
       Temp := 0;
       Sum <= 0;
    elsif rising_edge(Clock) then
       Temp := 0;
       for i in 0 to 7 loop
           Temp := Temp + A(i);
       end loop;
       Sum <= Temp;
     end if;
end process;
```

```
elsif rising_edge(Clock) then
    Temp:= 0;









end if;
```

```
elsif rising_edge(Clock) then
    Temp:= 0;
    Temp := Temp + A(0);









end if;
```

```
elsif rising_edge(Clock) then
    Temp:= 0;
    Temp := Temp + A(0);
    Temp := Temp + A(1);





end if;
```

```
elsif rising_edge(Clock) then
    Temp:= 0;
    Temp := Temp + A(0);
    Temp := Temp + A(1);
    Temp := Temp + A(2);
     .
     .
    Temp := Temp + A(6);
    Temp := Temp + A(7);

end if;
```

Comes from the FOR loop

```
elsif rising_edge(Clock) then
    Temp:= 0;
    Temp := Temp + A(0);
    Temp := Temp + A(1);
    Temp := Temp + A(2);
     .
     .
    Temp := Temp + A(6);
    Temp := Temp + A(7);
    Sum <= Temp;
end if;
```

Comes from the FOR loop

```
TestProcess : process(Reset, Clock)
begin
    if Reset = ‘1’ then
        Sum <= 0;
    elsif rising_edge(Clock) then
        Sum <= A(0) + A(1) + .....  A(7);
   end if;
end process;
```

```
TestProcess : process(Reset, Clock)
begin
    if Reset = ‘1’ then
             Sum <= 0;
    elsif rising_edge(Clock) then
         Sum <= 0;
         for i in 0 to 15 loop
              Sum <= Y + A(i);
         end loop;
   end if;
end process;
```

This code will not work!

```
elsif rising_edge(Clock) then
    Sum <= 0;
    Sum <= Sum + A(0);
    Sum <= Sum + A(1);
   .
   .
    Sum <= Sum + A(6);
    Sum <= Sum + A(7);
end if;
```

```vhdl
elsif rising_edge(Clock) then
    Sum <= 0;
    Sum <= Sum + A(0);
    Sum <= Sum + A(1);
   .
   .
    Sum <= Sum + A(6);
    Sum <= Sum + A(7); -- Only this line will be executed.
end if;
```

# Summary

- Variables capture their assignments immediately.
- Can have multiple assignments.
- Used to accumulate results.
- Used to store intermediate values in a calculation.

# Functions  And Procedures

# Functions & Procedures

- Similar to functions in software programming languages.

# Functions & Procedures

- Similar to functions in software programming languages.
- Used to perform commonly repeated operations or tasks.

# Functions & Procedures

- Similar to functions in software programming languages.
- Used to perform commonly repeated operations or tasks.
- Called from VHDL code to perform tasks.

# Functions & Procedures

- Similar to functions in software programming languages.

- Used to perform commonly repeated operations or tasks.

- Called from VHDL code to perform tasks.

- Cannot use registers. Only logic gates.

# Functions & Procedures

- Similar to functions in software programming languages.
- Used to perform commonly repeated operations or tasks.
- Called from VHDL code to perform tasks.
- Cannot use registers. Only logic gates.
- Can be declared in architecture, processes & packages.

# Functions

- Similar to functions in software programming languages.
- Used to perform regularly used algorithms.
- Can take zero or more inputs.
  - Functions cannot change input values.
- Must return an output value.
  - Cannot return void or omit a return value.
- Cannot contain wait statements.

# Basic Function Template

```
function FunctionName (input parameters)
return ReturnType is
   -- Constant or variable declarations
Begin
   -- Write the function code here.
   return Something;
end;
```

# Basic Function Template

```
function FunctionName (input parameters)
return ReturnType is
   -- Constant or variable declarations
Begin
   -- Write the function code here.
   return Something;
end;
```

# Function Example

```
function    Increment (InputNum    : integer)
```

# Function Example

```
function    Increment (InputNum   : integer)
return  integer  is
```

# Function Example

```
function     Increment (InputNum    : integer)
return  integer  is
        variable    temp    :  integer;
```

# Function Example

```
function    Increment (InputNum   : integer)
return  integer  is
      variable    temp    :  integer;
begin
      temp := InputNum + 1;
      return temp;
end;
```

# Function Example

```
function    Increment (InputNum   : integer)
return  integer  is
        variable    temp   :  integer;
begin
        temp := InputNum + 1;
        return temp;
end;


C <= Increment(4);   -- Function Usage
```

# Procedures

- Similar to Functions.
- Does NOT return a value.
- Parameters can be in, out or inout.
    - In – used to pass values into a procedure
    - Out – used to pass values out of a procedure. Can have more than one.
    - Inout – used to pass a value into a procedure where it can be modified and passed back out again.

# Procedures

- Similar to Functions.
- Does NOT return a value.
- Parameters can be in, out or inout.
    - In – used to pass values into a procedure
    - Out – used to pass values out of a procedure. Can have more than one.
    - Inout – used to pass a value into a procedure where it can be modified and passed back out again.

# Procedures

- Similar to Functions.
- Does NOT return a value.
- Parameters can be in, out or inout.
    - In – used to pass values into a procedure
    - Out – used to pass values out of a procedure. Can have more than one.
    - Inout – used to pass a value into a procedure where it can be modified and passed back out again.

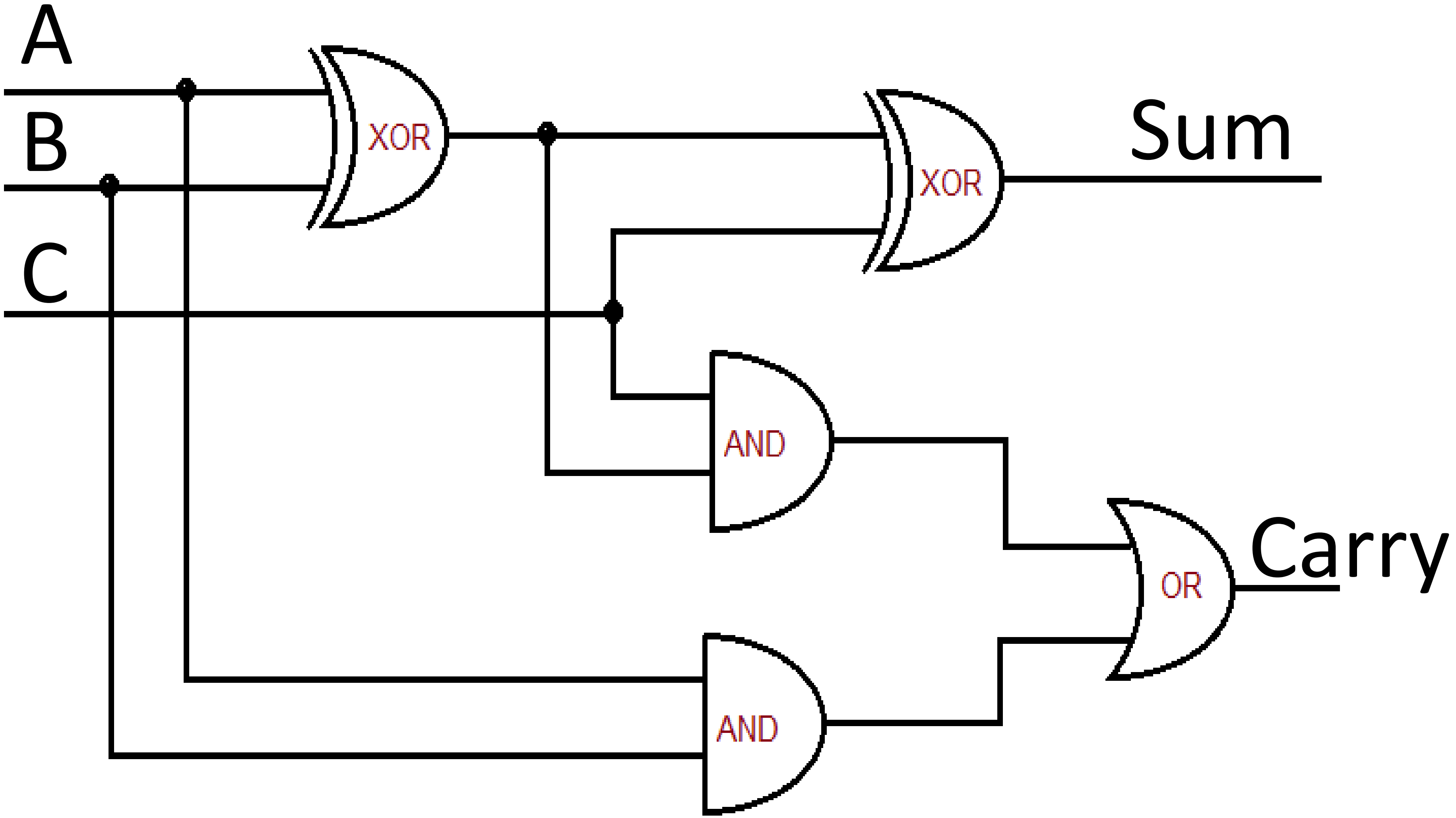
# Basic Procedure Template

```
procedure ProcedureName (parameters) is
   -- Constant or variable declarations
Begin
   -- Write the procedure code here.
end;
```

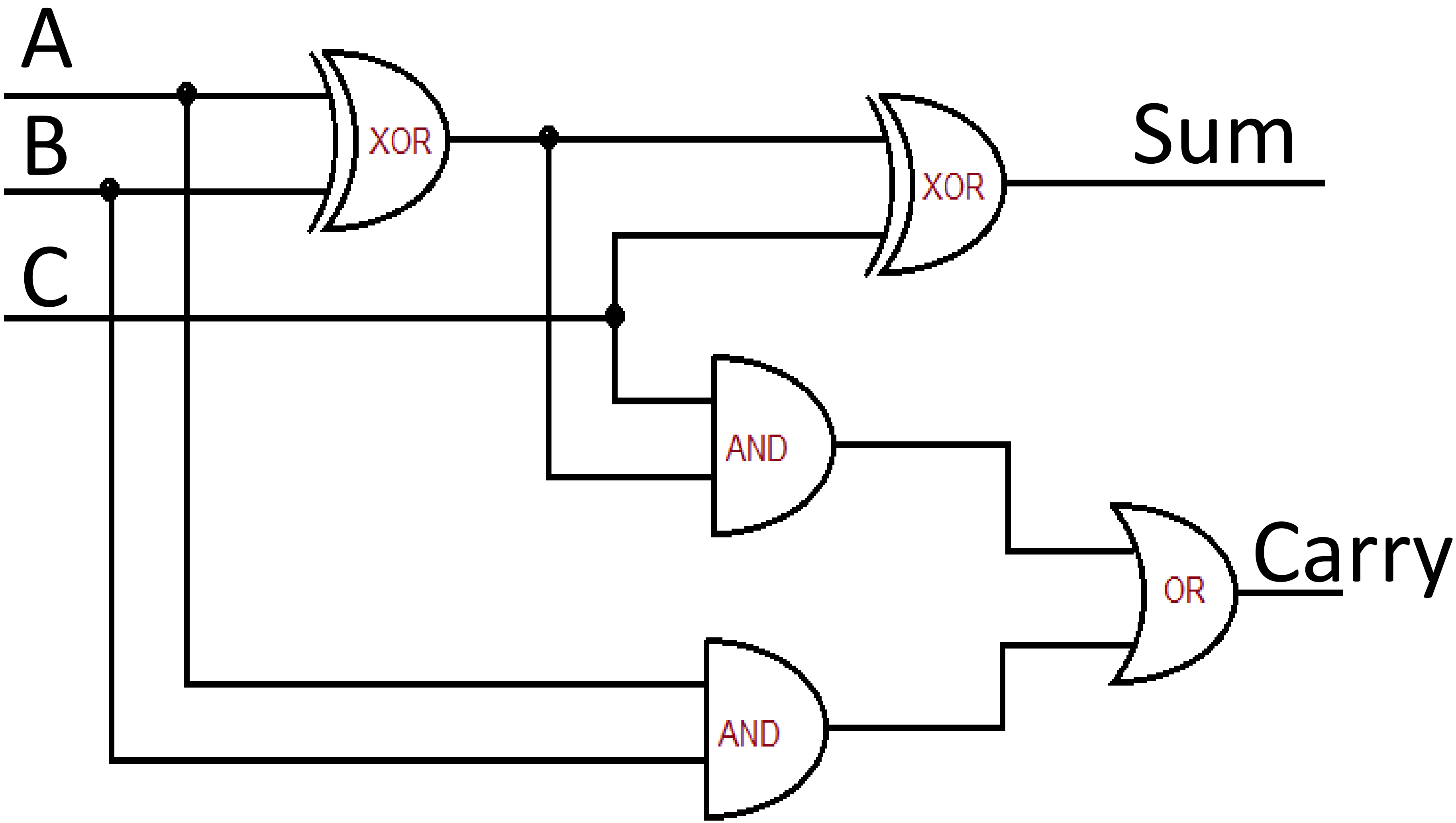
# Basic Procedure Template

```
procedure ProcedureName (parameters) is
   -- Constant or variable declarations
Begin
   -- Write the procedure code here.
end;
```

# Full Adder Circuit



# Full Adder Circuit



# Procedure Example

```
procedure FullAdder
```

# Procedure Example

```
procedure FullAdder
( signal A, B, C       : in  std_logic;
  signal Sum, Carry : out std_logic
) is
```

# Procedure Example

```
procedure FullAdder
( signal A, B, C       : in  std_logic;
  signal Sum, Carry : out std_logic
) is
begin
   Sum <= A xor B xor C;
   Carry <= (A and B) or (A and C) or (B and C);
end;
```

# Procedure Example

```
procedure FullAdder
( signal A, B, C         : in  std_logic;
  signal Sum, Carry : out std_logic
) is
begin
   Sum <= A xor B xor C;
   Carry <= (A and B) or (A and C) or (B and C);
end;

FullAdder (P, Q, R, SO, CO); -- Function Usage
```

End of Video

# Packages And Libraries

# Package

- Groups various declarations to be shared to be shared among several designs.

- A package can contain these items :
  - Signals & Constants
  - Data Types & Sub Types
  - Functions & Procedures
  - Components
  - Files
  - Attributes

# Packages We Have Seen Before

- Need to declare std_logic_1164 package to use :
  - Std_logic and std_logic_vector types

- Need to declare numeric_std package to use :
  - Unsigned and signed types

# Without Using Packages

```
Architecture Arch1 of Entity1 is

    ..............

    type TLightType is (RED, AMBER, GREEN);

    Signal TrafficLight : TLightType;

    ..............

Begin
```

# Without Using Packages

```
Architecture Arch1 of Entity1 is

    ..............

    type TLightType is (RED, AMBER, GREEN);

    Signal TrafficLight : TLightType;

    ..............

Begin
```

# Without Using Packages

VHDL File 1

```
Architecture Arch1 of Module1 is
    type TLightType is (RED, AMBER, GREEN);
    Signal TrafficLight : TLightType;
Begin
```

VHDL File 2

```
Architecture Arch1 of Module2 is
    type TLightType is (RED, AMBER, GREEN);
    Signal TrafficLight : TLightType;
Begin
```

# Without Using Packages

VHDL File 1

```
Architecture Arch1 of Module1 is
    type TLightType is (RED, AMBER, GREEN);
    Signal TrafficLight : TLightType;
Begin
```

VHDL File 2

```
Architecture Arch1 of Module2 is
    type TLightType is (RED, AMBER, GREEN);
    Signal TrafficLight : TLightType;
Begin
```

# Writing A Basic Package

MyPkg.vhd

```
package MyPackage is

    type TLightType is (RED, AMBER, GREEN);

end MyPackage;
```

# Writing A Basic Package

MyPkg.vhd

```
package MyPackage is

    type TLightType is (RED, AMBER, GREEN);

end MyPackage;
```

# Example Using Packages

VHDL File 1

```
use work.MyPackage.all;
Architecture Arch1 of Module1 is
    Signal TrafficLight : TLightType;
Begin
```

VHDL File 2

```
use work.MyPackage.all;
Architecture Arch1 of Module2 is
    Signal TrafficLight : TLightType;
Begin
```

# Example Using Packages

VHDL File 1

```
use work.MyPackage.all;
Architecture Arch1 of Module1 is
    Signal TrafficLight : TLightType;
Begin
```

VHDL File 2

```
use work.MyPackage.all;
Architecture Arch1 of Module2 is
    Signal TrafficLight : TLightType;
Begin
```

# Example Using Packages

VHDL File 1

```
use work.MyPackage.all;
Architecture Arch1 of Module1 is
    Signal TrafficLight : TLightType;
Begin
```

VHDL File 2

```
use work.MyPackage.all;
Architecture Arch1 of Module2 is
    Signal TrafficLight : TLightType;
Begin
```

# Libraries

- Is a **container** that contains **compiled design units**.

# Libraries

- Is a **container** that contains **compiled design units**.
- The compiler compiles a design modules into an **intermediate form** and stores this in a library.

# Libraries

- Is a **container** that contains **compiled design units**.
- The compiler compiles a design modules into an **intermediate form** and stores this in a library.
- By default, the intermediate forms are stored in the Library you are **currently working in**.

# Libraries

- Is a **container** that contains **compiled design units**.
- The compiler compiles a design modules into an **intermediate form** and stores this in a library.
- By default, the intermediate forms are stored in the Library you are **currently working in**.
- Most simulators and synthesis tools **automatically create** this default Library for you.

# Libraries

- Is a **container** that contains **compiled design units**.
- The compiler compiles a design modules into an **intermediate form** and stores this in a library.
- By default, the intermediate forms are stored in the Library you are **currently working in**.
- Most simulators and synthesis tools **automatically create** this default Library for you.
- The library you are working in will have a name of its own.

# Libraries

- Is a **container** that contains **compiled design units**.
- The compiler compiles a design modules into an **intermediate form** and stores this in a library.
- By default, the intermediate forms are stored in the Library you are **currently working in**.
- Most simulators and synthesis tools **automatically create** this default Library for you.
- The library you are working in will have a name of its own.
- You can use the keyword Work to refer to (or point to) the library you are currently working in.

# Libraries

- Every library has a name.
- To use  a library, you must declare it at the top of your file.
- This makes all the design units in the library available for use in the current design.

**Example:**

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

# Summary

- Creating & using simple VHDL packages.
- Packages are used to share declarations.
- Libraries store compiled design units.
- Current active library is where compiler stores compiled code by default.
- Work is used to refer to current active library.

# Parameterised Components

# Generics

- Generics are a special type of constant.
- Constant is declared in:
  - The declarative part of the architecture (for module scope).
  - VHDL packages (for global scope)
  - Value of a constant is the same across whole design.
- Generic is declared in:
  - The entity declaration.
  - Value of generic unique to each module instance.

# Generics

- Generics are a special type of constant.
- Constant is declared in:
  - The declarative part of the architecture (for module scope).
  - VHDL packages (for global scope)
  - Value of a constant is the same across whole design.
- Generic is declared in:
  - The entity declaration.
  - Value of generic unique to each module instance.

# Generics

- Generics are a special type of constant.
- Constant is declared in:
  - The declarative part of the architecture (for module scope).
  - VHDL packages (for global scope)
  - Value of a constant is the same across whole design.
- Generic is declared in:
  - The entity declaration.
  - Value of generic unique to each module instance.

```
entity Adder is
    port
    (
        A   : in   unsigned(7 downto 0);
        B   : in   unsigned(7 downto 0);
        C   : out unsigned(7 downto 0)
     );
end entity;

architecture rtl of Adder is
begin
        C <= A + B;
end architecture;
```

```
entity Adder is
    port
    (
        A   : in  unsigned(7 downto 0);
        B   : in  unsigned(7 downto 0);
        C   : out unsigned(7 downto 0)
     );
end entity;

architecture rtl of Adder is
begin
        C <= A + B;
end architecture;
```

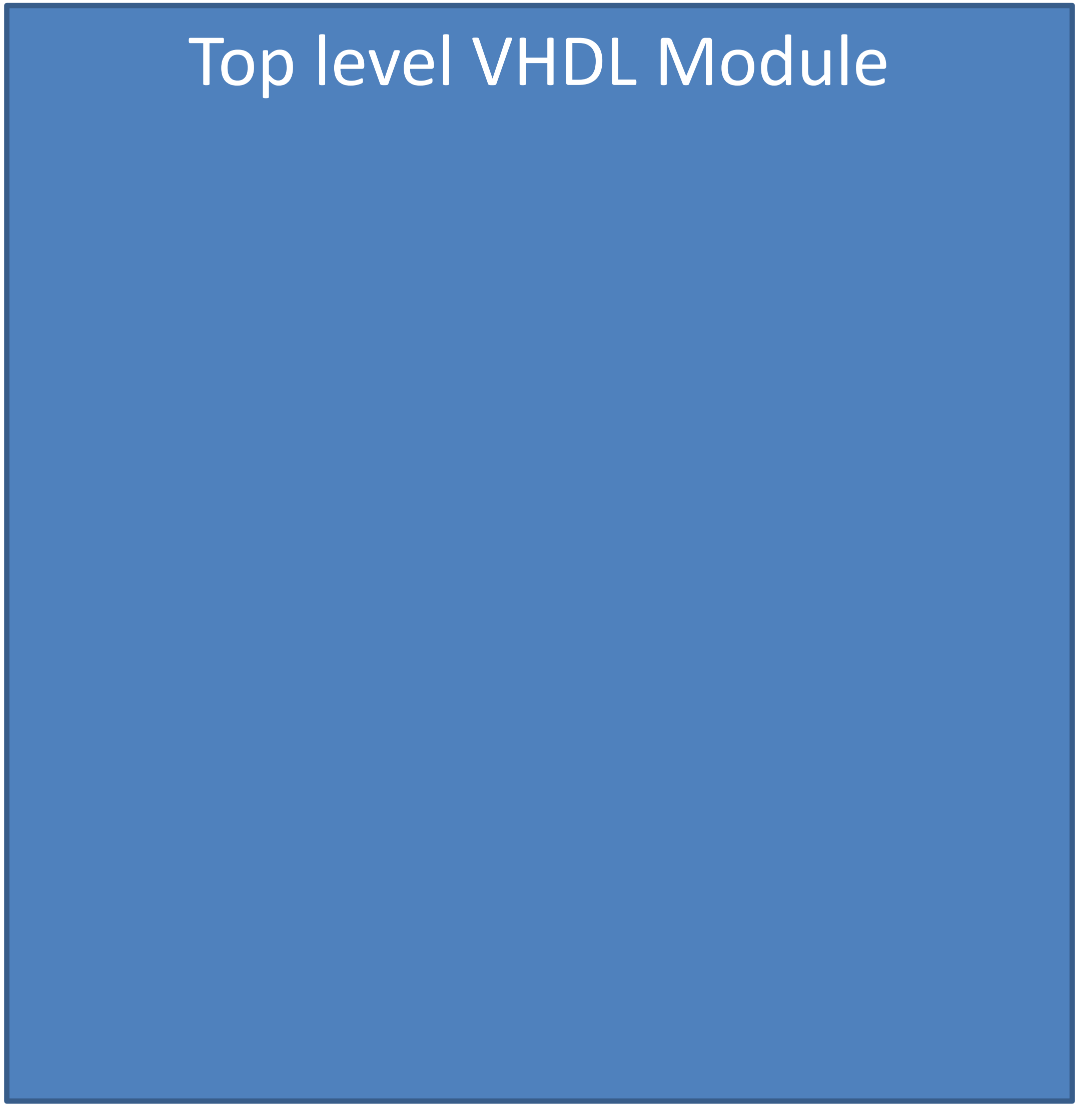
```
entity Adder is
```

We are going to declare our generic here

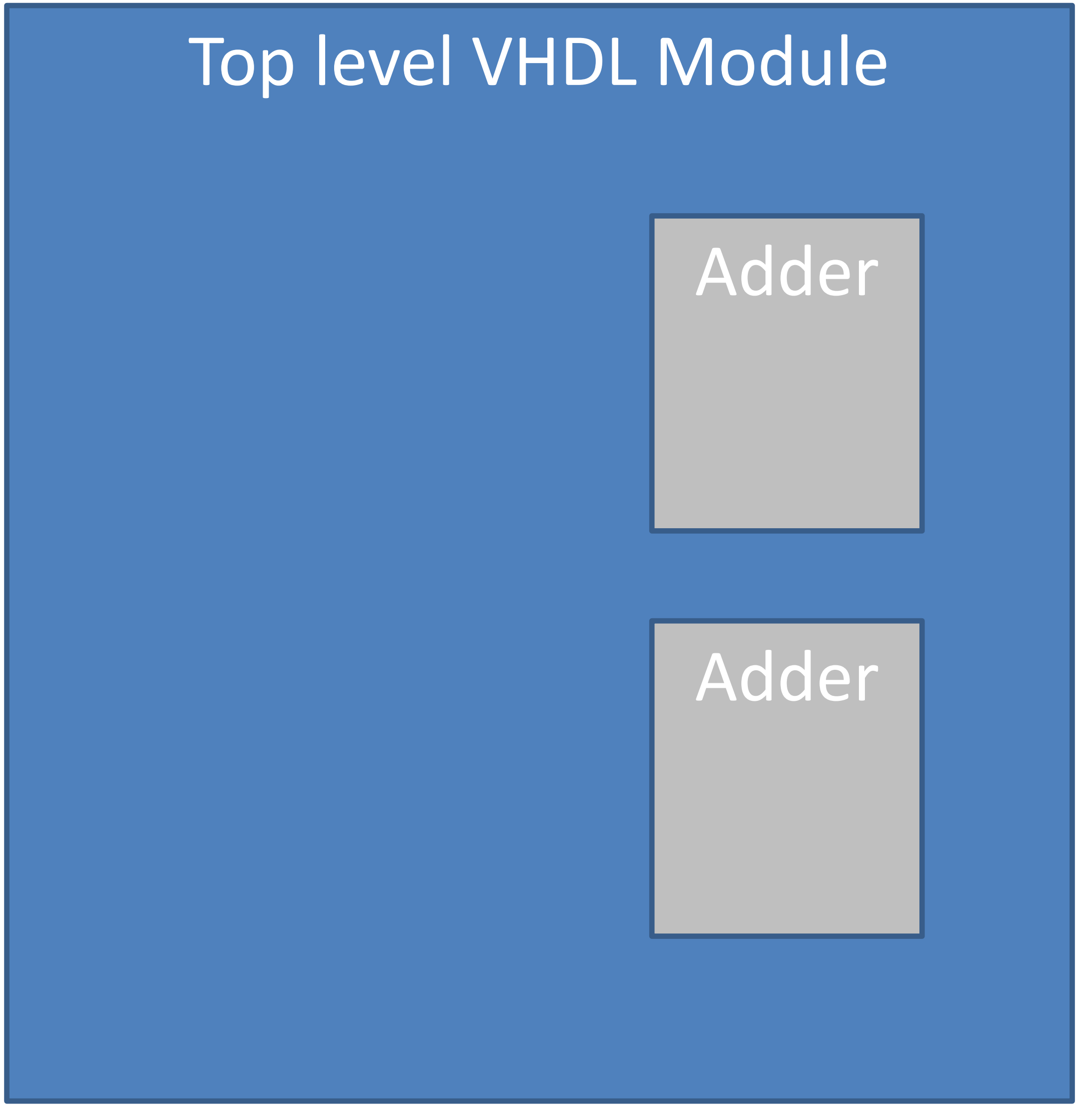
```
    port
    (
        A   : in   unsigned(7 downto 0);
        B   : in   unsigned(7 downto 0);
        C   : out unsigned(7 downto 0)
     );
end entity;

architecture rtl of Adder is
begin
        C <= A + B;
end architecture;
```

```
entity Adder is
   generic (
         N : integer := 8
   );
    port
    (
         A    : in    unsigned(7 downto 0);
         B    : in    unsigned(7 downto 0);
         C    : out unsigned(7 downto 0)
     );
end entity;

architecture rtl of Adder is
begin
         C <= A + B;
end architecture;
```

```
entity Adder is
  generic (
        N : integer := 8
  );
   port
   (
        A   : in   unsigned(N-1 downto 0);
        B   : in   unsigned(N-1 downto 0);
        C   : out unsigned(N-1 downto 0)
    );
end entity;

architecture rtl of Adder is
begin
        C <= A + B;
end architecture;
```



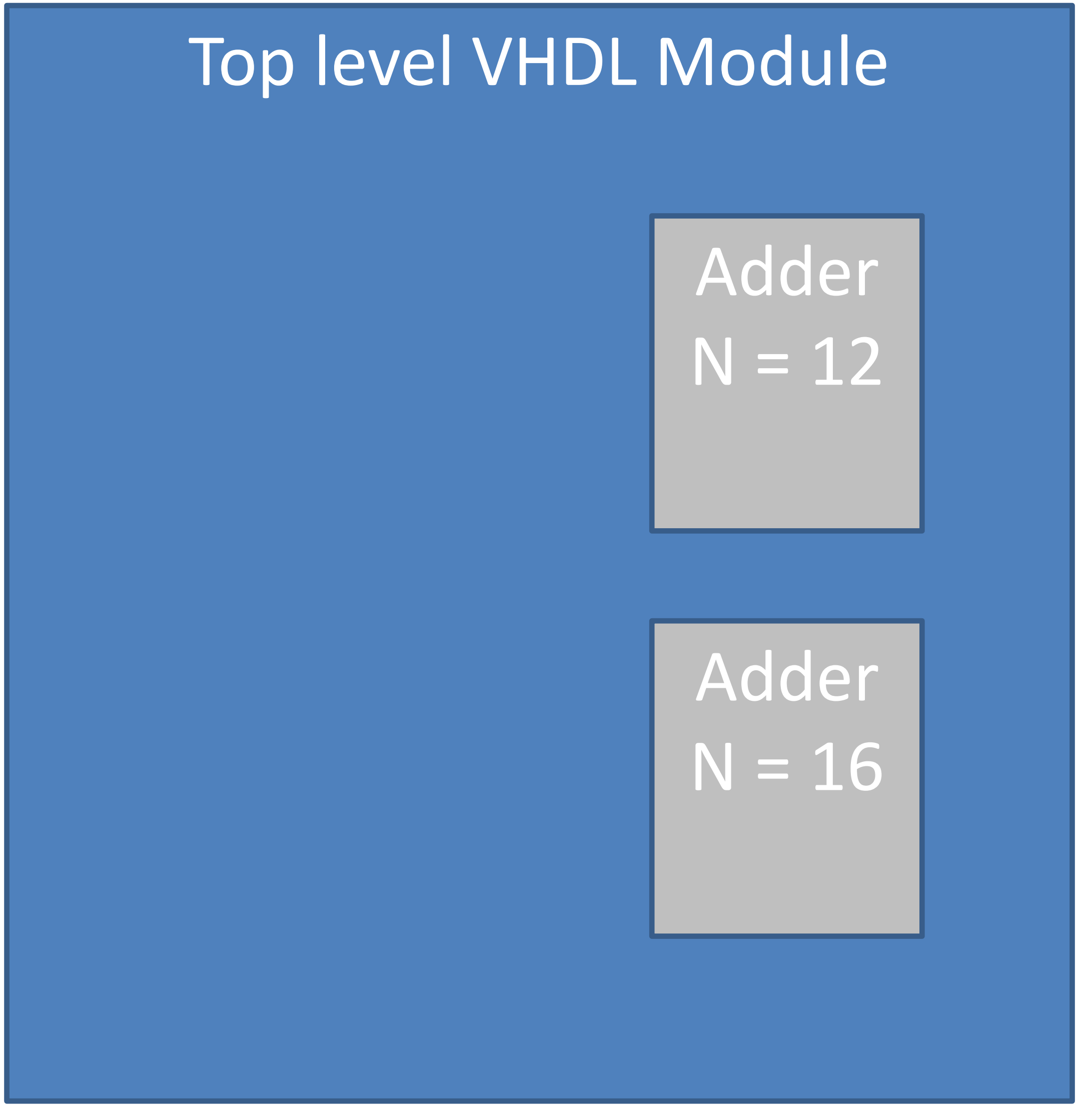
Top level VHDL Module



Top level VHDL Module

Adder

Adder



Top level VHDL Module

Adder

N = 12

Adder

N = 16

```
Adder12Bit : Adder
generic map (
     N => 12
)
port
(
     ...
);
```

We write this code in our Top Level VHDL module to instantiate the two Adder components

```
Adder12Bit : Adder
generic map (
    N => 12
)
port
(
    ...
);

Adder16Bit : Adder
generic map (
    N => 16
)
port
(
    ...
);
```

We write this code in
our Top Level VHDL
module to instantiate
the two Adder
components

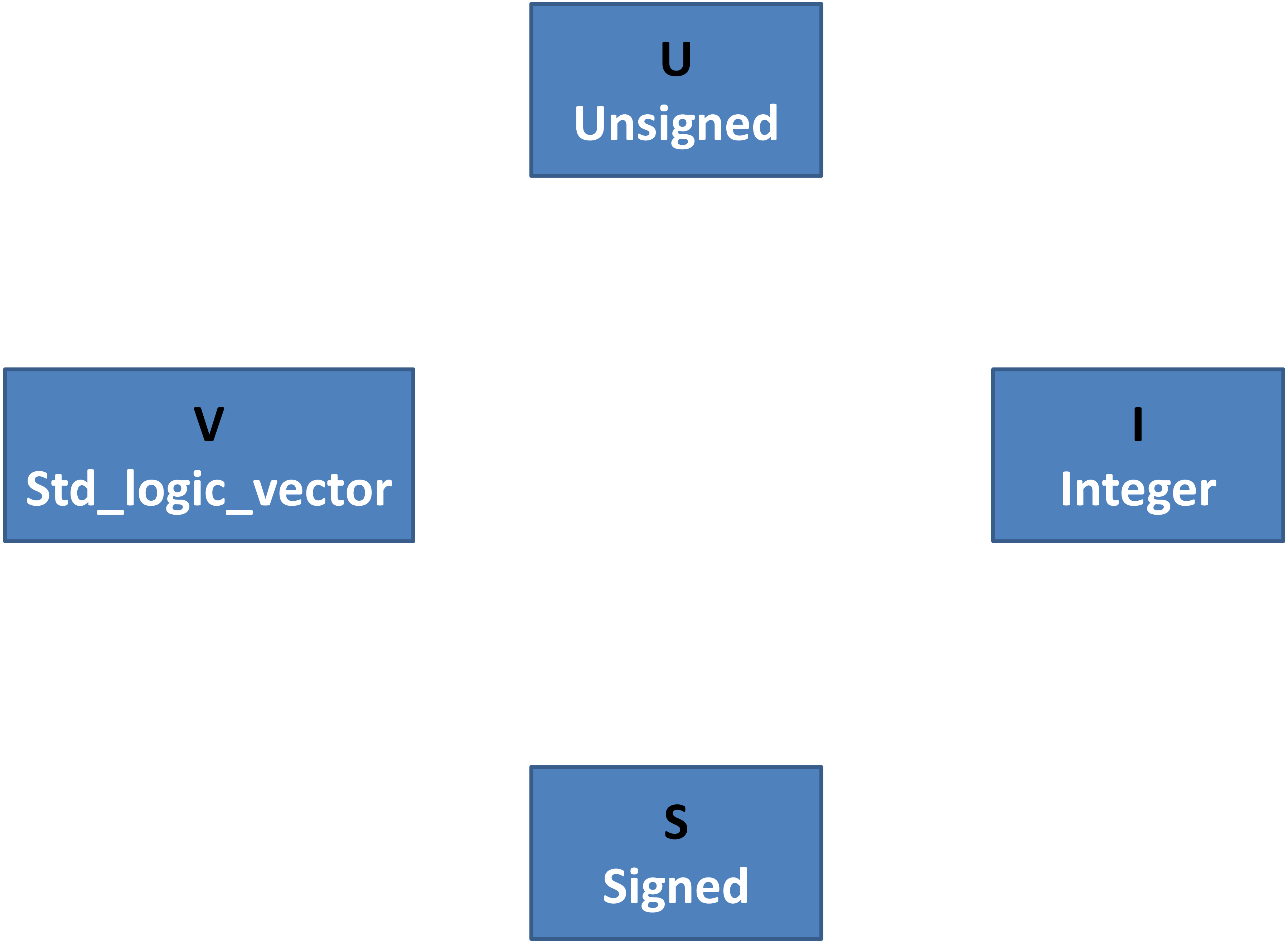
# Summary

- Generics are a special type of constant.
- Generics allow VHDL modules to be parameterised.
- Parameterised modules :
  - Allow their instances to be configurable.
  - Makes it easier to reuse in other projects.
  - Reduces code redundancy.

# Type Conversions

# Need For Type Conversion

- VHDL is very strict about data types.
    - VHDL is a strongly typed language.
- Cannot use different types in the same expression.
- When performing a signal assignment, the data type of LHS must match the RHS exactly.
- Type conversion functions & Type Casting.

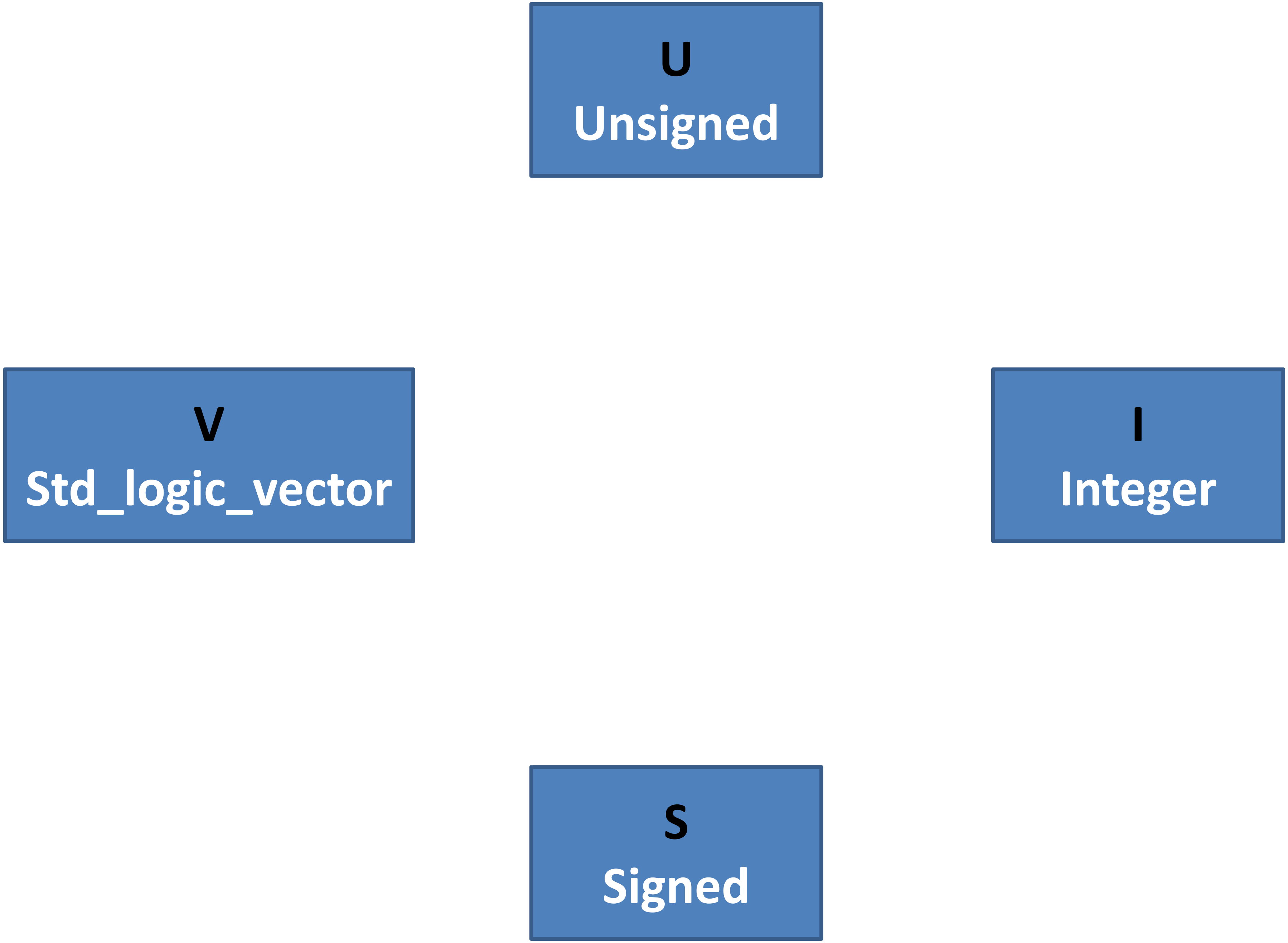


U
Unsigned

V
Std_logic_vector

I
Integer

S
Signed

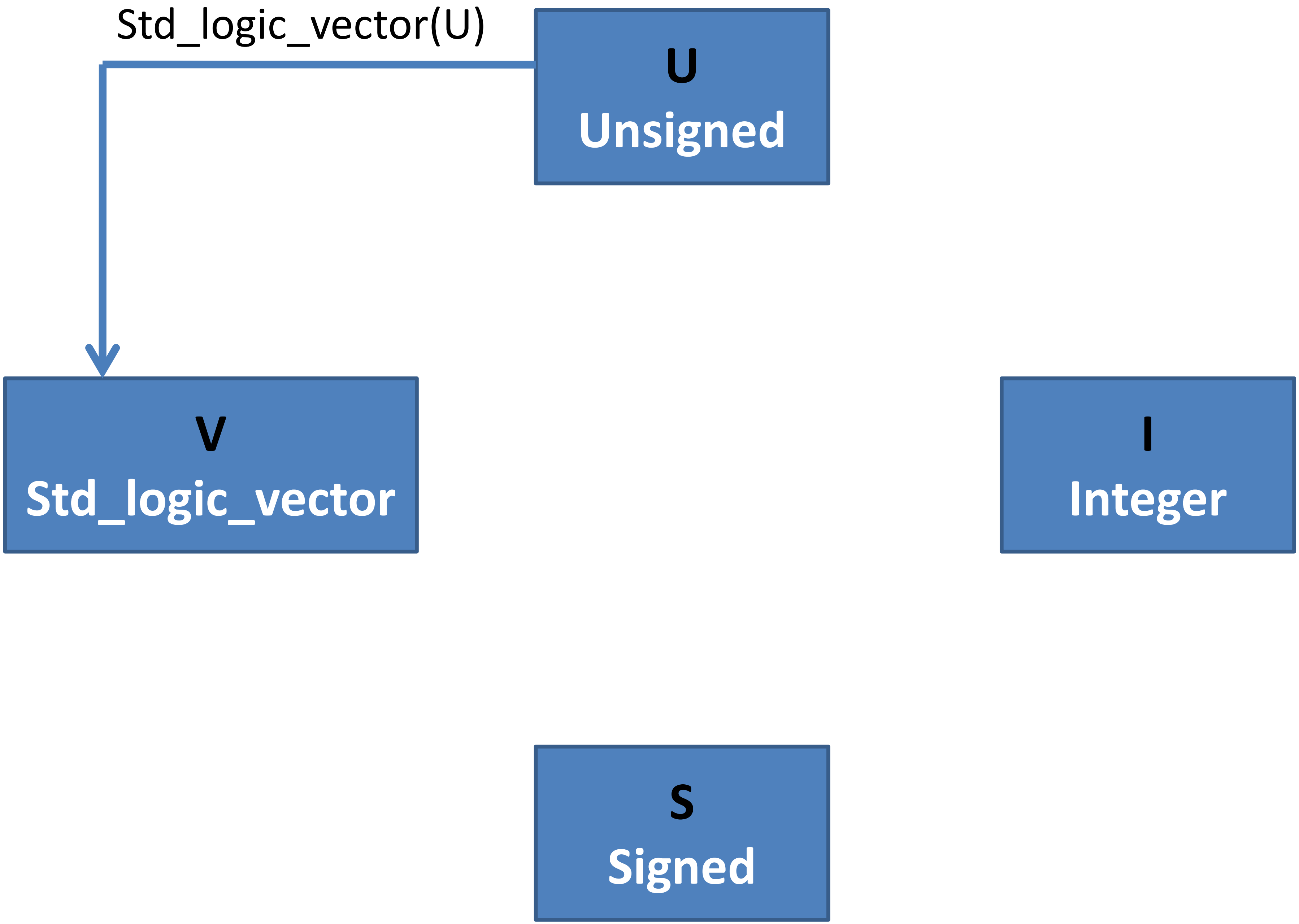


U
Unsigned

V
Std_logic_vector

I
Integer

S
Signed



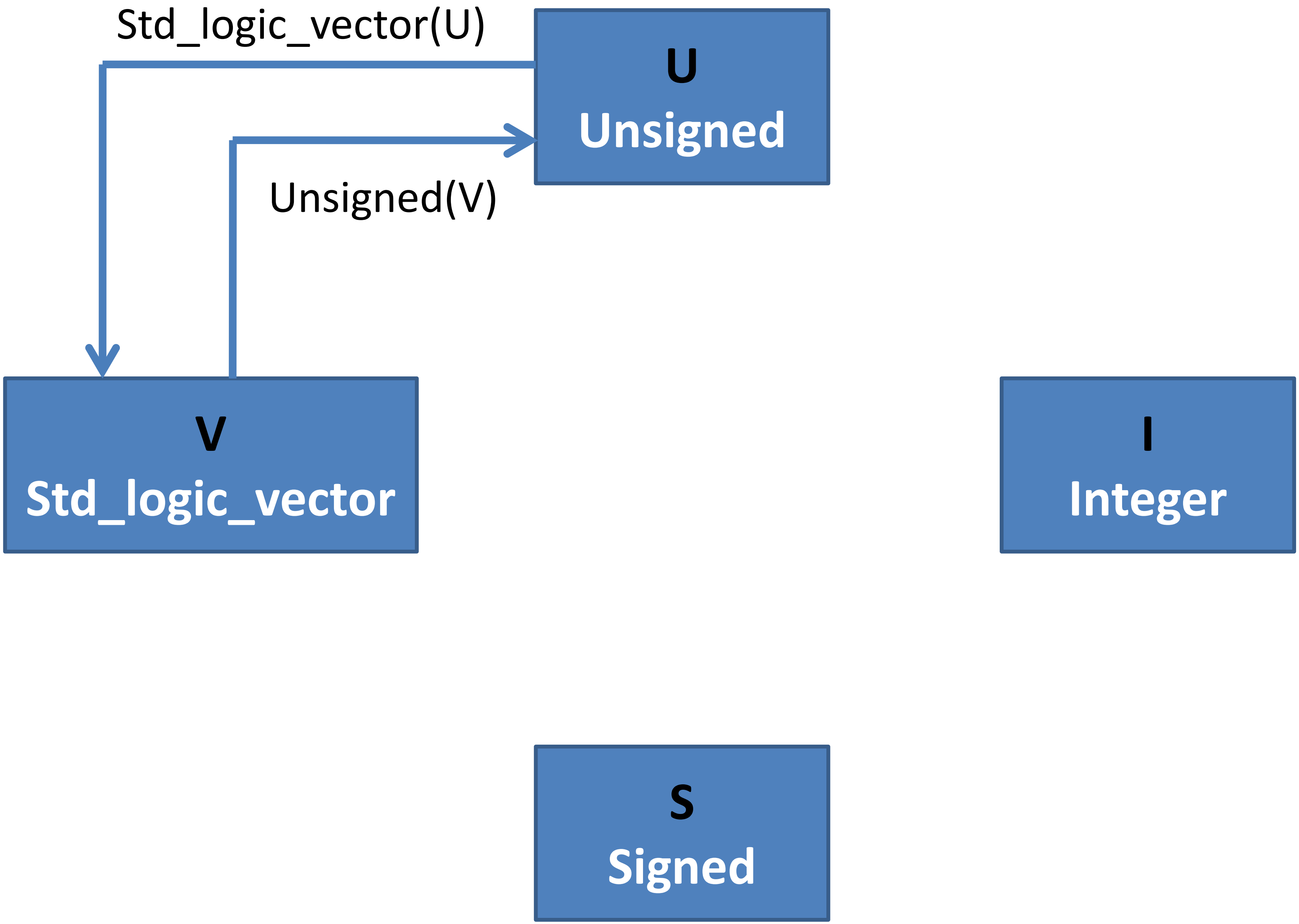
Std_logic_vector(U)

U
Unsigned

V
Std_logic_vector

I
Integer

S
Signed



Std_logic_vector(U)

U

Unsigned

Unsigned(V)

V

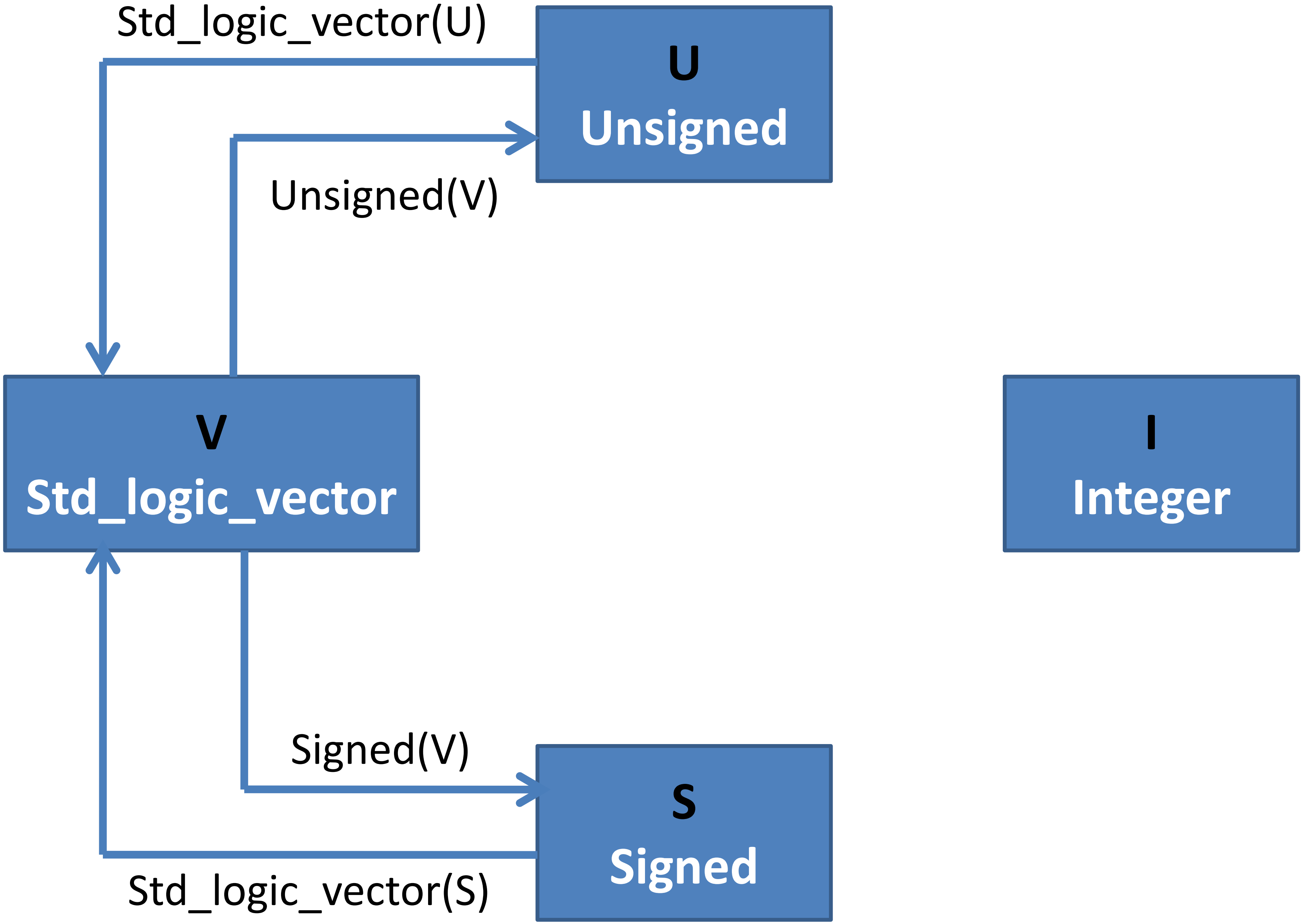
Std_logic_vector

I

Integer

S

Signed



Std_logic_vector(U)

U

Unsigned

Unsigned(V)

V

Std_logic_vector

I

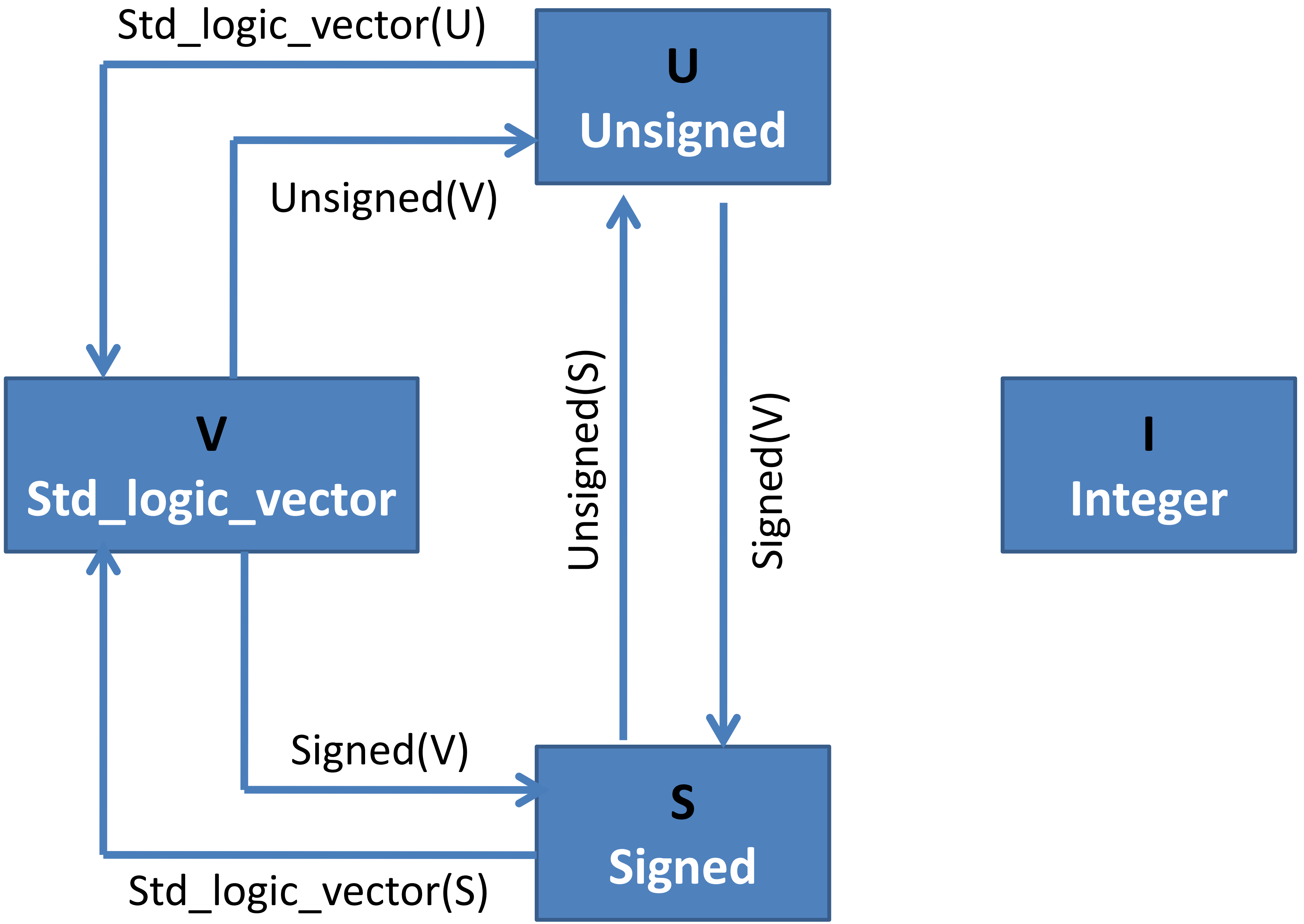
Integer

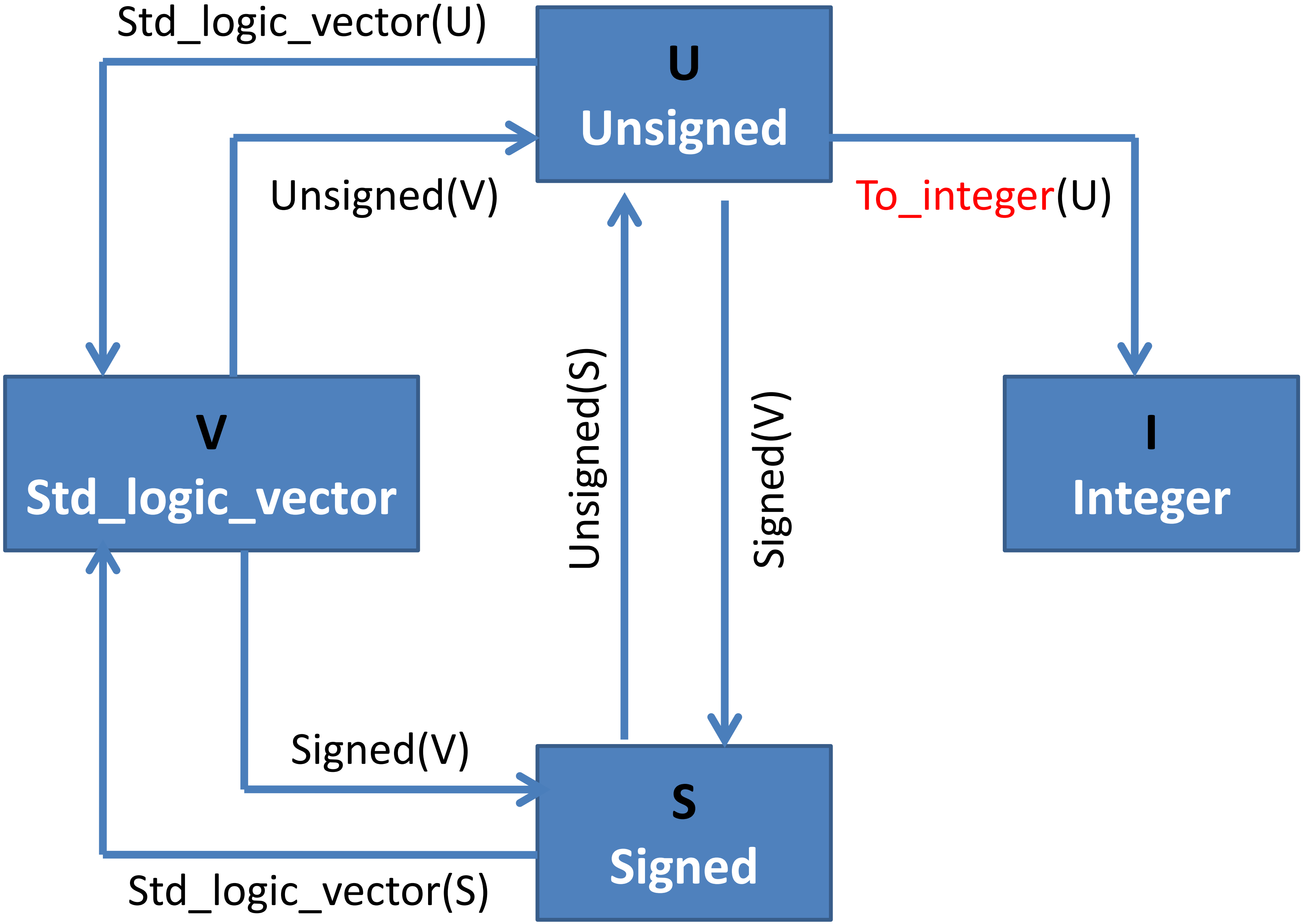
Signed(V)

S

Signed

Std_logic_vector(S)


Std_logic_vector(U)
U
Unsigned
Unsigned(V)
Unsigned(S)
Signed(V)
V
Std_logic_vector
I
Integer
Signed(V)
S
Signed
Std_logic_vector(S)



Std_logic_vector(U)

U

Unsigned

Unsigned(V)

To_integer(U)

V

Std_logic_vector

Unsigned(S)

Signed(V)

I

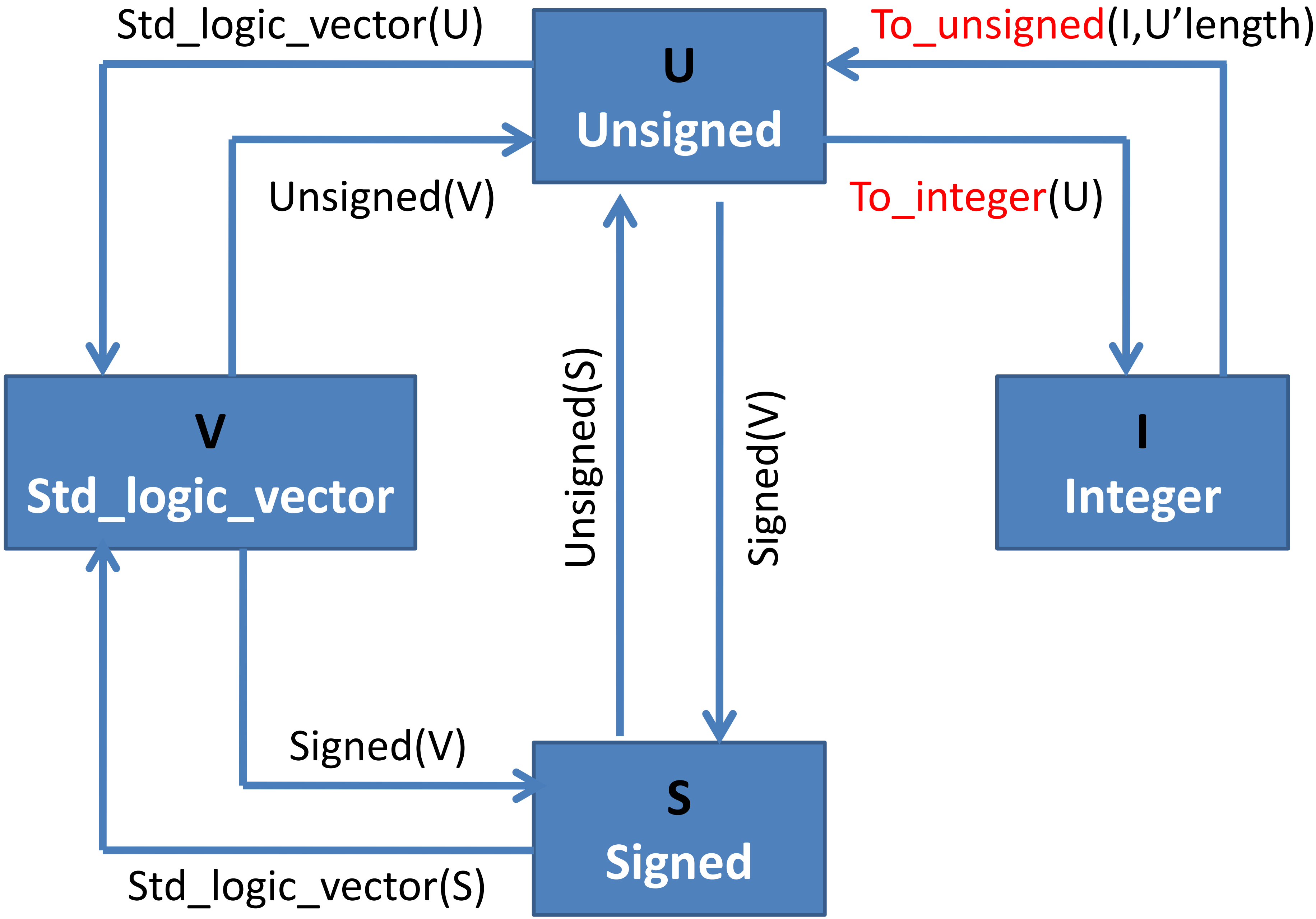
Integer

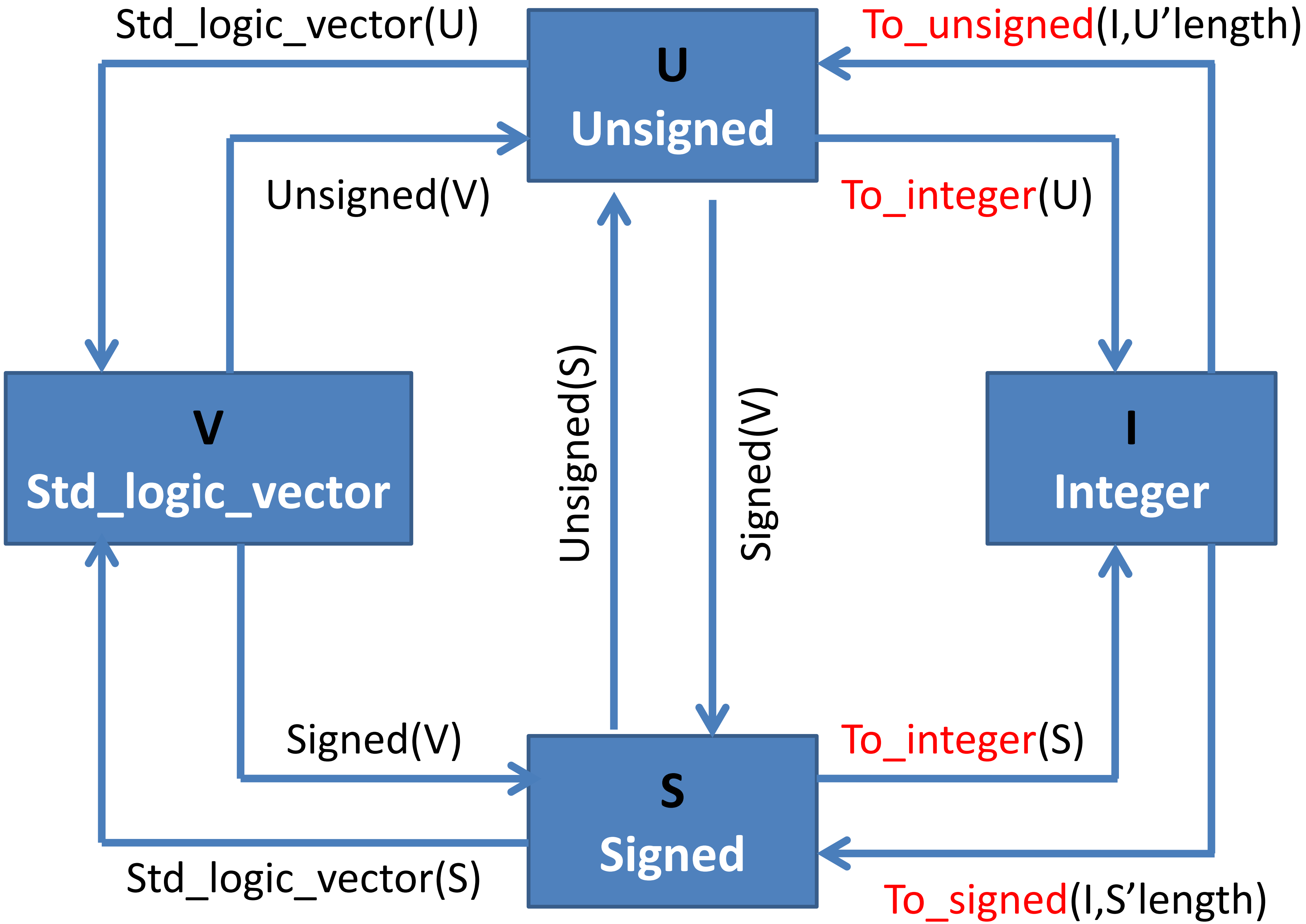
Signed(V)

S

Signed

Std_logic_vector(S)


Std_logic_vector(U)
U
Unsigned
To_unsigned(I,U'length)
Unsigned(V)
To_integer(U)
Unsigned(S)
Signed(V)
V
Std_logic_vector
I
Integer
Signed(V)
S
Signed
Std_logic_vector(S)



U
Unsigned

V
Std_logic_vector

I
Integer

S
Signed

Std_logic_vector(U)

Unsigned(V)

To_unsigned(I,U’length)

To_integer(U)

Unsigned(S)

Signed(V)

Signed(V)

To_integer(S)

Std_logic_vector(S)

To_signed(I,S’length)

# Writing Type Conversion Functions

```
function to_std_logic( A : in boolean)
return std_logic is
begin
    if  A  then
        return ‘1’;
    else
        return ‘0’;
    end if;
end;
```

# Summary

- The need for type conversion in VHDL.
- Type conversions provided by numeric standard package.
- We can write our own type conversion functions.
- Type conversion does not cost extra FPGA resources.

# Others Keyword

# Others Keyword

- To cover all choices that have not been explicitly covered in a CASE statement

# Others Keyword

- To cover all choices that have not been explicitly covered in a CASE statement

- Assign all elements of an array.

# Others Keyword

```
signal A : std_logic_vector (7 downto 0);
```

```
A <= “00000000”;
```

# Others Keyword

```
signal A : std_logic_vector (7 downto 0);
```

```
A <= “00000000”;

A <= x”00”;
```

# Others

```
signal A : std_logic_vector (7 downto 0);
```

```
A <= “00000000”;

A <= x”00”;

A <= (others => ‘0’);
```

# Tri-State Drivers

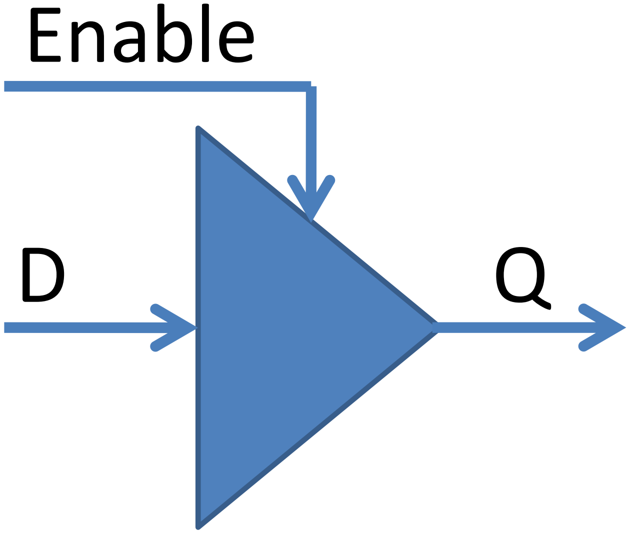
# Tri-State Drivers

- Are only available on IO pins of FPGA.

# Tri-State Drivers

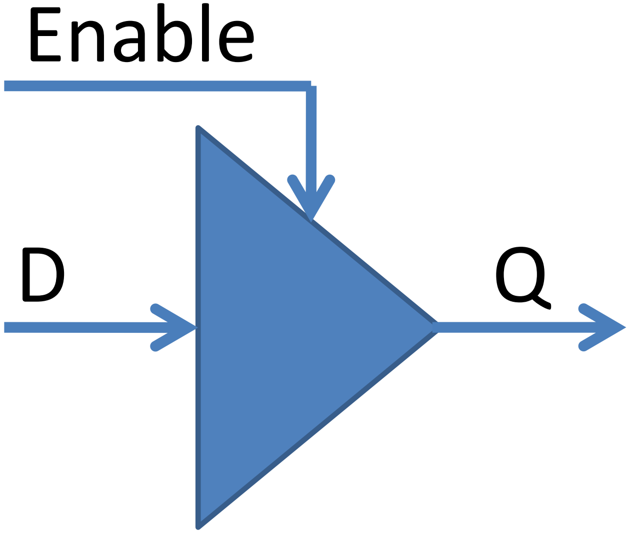
- Are only available on IO pins of FPGA.
- Cannot have Tri-state signals internal to the FPGA.

# Tri-State Drivers



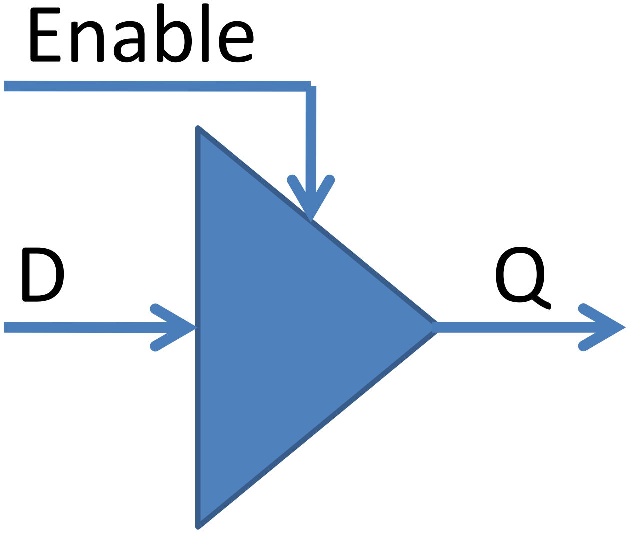
# Tri-State Drivers

```
TriState_Example : process (D , Enable)
begin



end process;
```



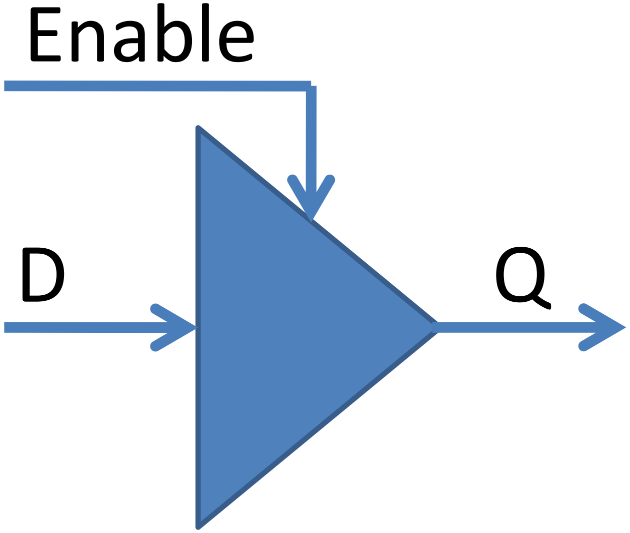
# Tri-State Drivers

```
TriState_Example : process (D , Enable)
begin
        if Enable = ‘1’ then
                Q  <= D;



end process;
```



# Tri-State Drivers

```
TriState_Example : process (D , Enable)
begin
        if Enable = ‘1’ then
                Q  <= D;
        else
                Q <= ‘Z’;
        end if;
end process;
```



# Comparators

# Comparators

# VHDL for Comparator

```
architecture rtl of SomeRandomVHDLModule is
        signal A            : integer;
        signal B            : integer;
        signal Out          : std_logic;
begin
```

# VHDL for Comparator

```
architecture rtl of SomeRandomVHDLModule is
    signal A        : integer;
    signal B        : integer;
    signal Out      : std_logic;
begin
```

# VHDL for Comparator

```
architecture rtl of SomeRandomVHDLModule is
    signal A        : integer;
    signal B        : integer;
    signal Out      : std_logic;
begin
    MyComparatorProcess : process (A, B)
    begin
```

# VHDL for Comparator

```
architecture rtl of SomeRandomVHDLModule is
    signal A        : integer;
    signal B        : integer;
    signal Out      : std_logic;
begin
    MyComparatorProcess : process (A, B)
    begin
        if A > B then
            Out <= ‘1’;
```

# VHDL for Comparator

```
architecture rtl of SomeRandomVHDLModule is
    signal A            : integer;
    signal B            : integer;
    signal Out          : std_logic;
begin
    MyComparatorProcess : process (A, B)
    begin
        if A > B then
            Out <= ‘1’;
        else
            Out <= ‘0’;
        end if;
```

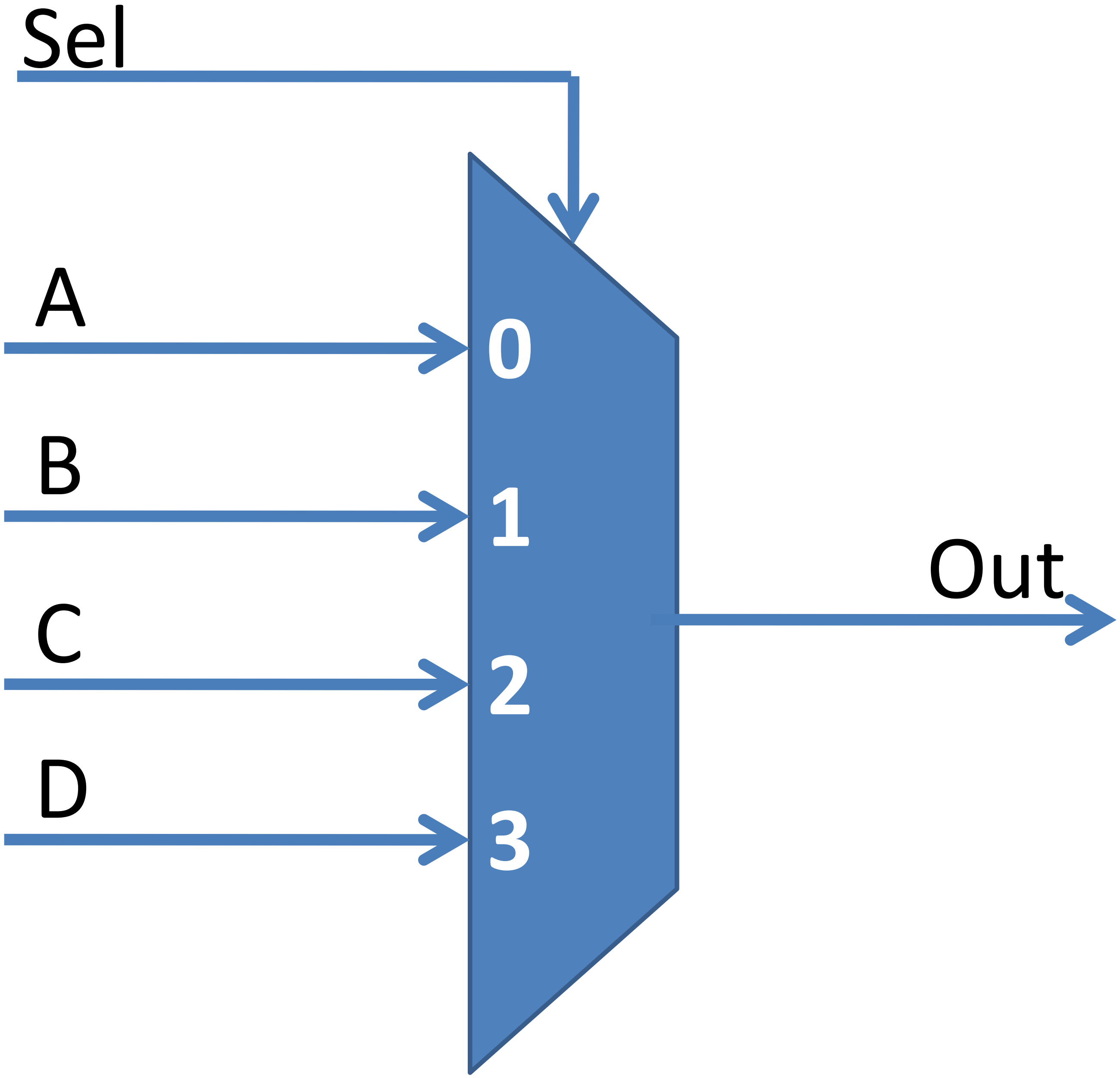
# VHDL for Comparator

```
architecture rtl of SomeRandomVHDLModule is
        signal A            : integer;
        signal B            : integer;
        signal Out          : std_logic;
begin
        MyComparatorProcess : process (A, B)
        begin
                if A > B then
                     Out <= ‘1’;
                else
                     Out <= ‘0’;
                end if;
        end process;

…
…
…
…

end rtl;
```

# VHDL for Comparator

```
architecture rtl of SomeRandomVHDLModule is
    signal A        : integer;
    signal B        : integer;
    signal Out      : std_logic;
begin
    MyComparatorProcess : process (A, B)
    begin
        if A > B then
            Out <= ‘1’;
        else
            Out <= ‘0’;
        end if;
    end process;

…
…
…
…

end rtl;
```

# Multiplexers


Sel
A
B
C
D
0
1
2
3
Out

# VHDL for Multiplexer

```
architecture rtl of SomeRandomVHDLModule is
        signal A        : integer;
        signal B        : integer;
        signal C        : integer;
        signal D        : integer;
        signal Out      : integer;
        signal Sel      : integer range 0 to 3;
begin
```

# VHDL for Multiplexer

```
architecture rtl of SomeRandomVHDLModule is
        signal A          : integer;
        signal B          : integer;
        signal C          : integer;
        signal D          : integer;
        signal Out        : integer;
        signal Sel        : integer range 0 to 3;
begin

        MyMuxProcess : process (A, B, C, D, Sel)
        begin
```

# VHDL for Multiplexer

```
architecture rtl of SomeRandomVHDLModule is
    signal A        : integer;
    signal B        : integer;
    signal C        : integer;
    signal D        : integer;
    signal Out      : integer;
    signal Sel      : integer range 0 to 3;
begin

    MyMuxProcess : process (A, B, C, D, Sel)
    begin
        case Sel is
```

# VHDL for Multiplexer

```
architecture rtl of SomeRandomVHDLModule is
    signal A        : integer;
    signal B        : integer;
    signal C        : integer;
    signal D        : integer;
    signal Out      : integer;
    signal Sel      : integer range 0 to 3;
begin

    MyMuxProcess : process (A, B, C, D, Sel)
    begin
        case Sel is
                when 0 => Out <= A;
```

# VHDL for Multiplexer

```
architecture rtl of SomeRandomVHDLModule is
    signal A            : integer;
    signal B            : integer;
    signal C            : integer;
    signal D            : integer;
    signal Out          : integer;
    signal Sel          : integer range 0 to 3;
begin

    MyMuxProcess : process (A, B, C, D, Sel)
    begin
        case Sel is
                when 0 => Out <= A;
                when 1 => Out <= B;
```

# VHDL for Multiplexer

```
architecture rtl of SomeRandomVHDLModule is
        signal A            : integer;
        signal B            : integer;
        signal C            : integer;
        signal D            : integer;
        signal Out          : integer;
        signal Sel          : integer range 0 to 3;
begin

        MyMuxProcess : process (A, B, C, D, Sel)
        begin
                case Sel is
                            when 0 => Out <= A;
                            when 1 => Out <= B;
                            when 2 => Out <= C;
```

# VHDL for Multiplexer

```
architecture rtl of SomeRandomVHDLModule is
    signal A        : integer;
    signal B        : integer;
    signal C        : integer;
    signal D        : integer;
    signal Out      : integer;
    signal Sel      : integer range 0 to 3;
begin

    MyMuxProcess : process (A, B, C, D, Sel)
    begin
        case Sel is
                when 0 => Out <= A;
                when 1 => Out <= B;
                when 2 => Out <= C;
                when 3 => Out <= D;
```

# VHDL for Multiplexer

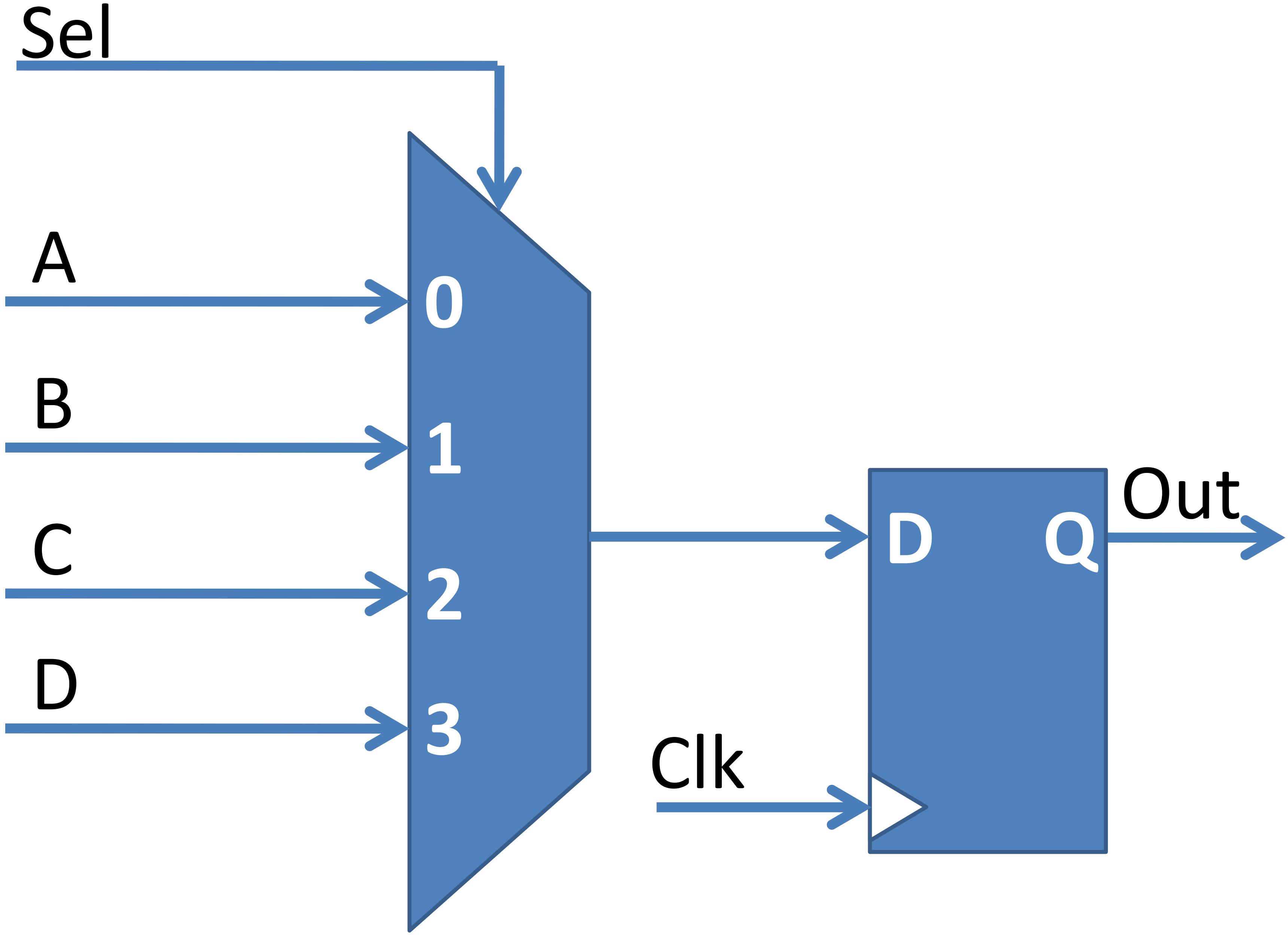
```
architecture rtl of SomeRandomVHDLModule is
    signal A        : integer;
    signal B        : integer;
    signal C        : integer;
    signal D        : integer;
    signal Out      : integer;
    signal Sel      : integer range 0 to 3;
begin

    MyMuxProcess : process (A, B, C, D, Sel)
    begin
        case Sel is
                when 0 => Out <= A;
                when 1 => Out <= B;
                when 2 => Out <= C;
                when 3 => Out <= D;
                when others =>
```

# VHDL for Multiplexer

```
architecture rtl of SomeRandomVHDLModule is
    signal A          : integer;
    signal B          : integer;
    signal C          : integer;
    signal D          : integer;
    signal Out        : integer;
    signal Sel        : integer range 0 to 3;
begin

    MyMuxProcess : process (A, B, C, D, Sel)
    begin
        case Sel is
                when 0 => Out <= A;
                when 1 => Out <= B;
                when 2 => Out <= C;
                when 3 => Out <= D;
                when others =>
        end case;
```

# VHDL for Multiplexer

```
architecture rtl of SomeRandomVHDLModule is
        signal A          : integer;
        signal B          : integer;
        signal C          : integer;
        signal D          : integer;
        signal Out        : integer;
        signal Sel        : integer range 0 to 3;
begin

        MyMuxProcess : process (A, B, C, D, Sel)
        begin
                case Sel is
                                when 0 => Out <= A;
                                when 1 => Out <= B;
                                when 2 => Out <= C;
                                when 3 => Out <= D;
                                when others =>
                end case;
        end process;

end rtl;
```


Sel
A
0
B
1
C
2
D
3
D
Q
Out
Clk

# VHDL for Multiplexer

```
MyMuxProcess : process (reset, clk)
begin
```

# VHDL for Multiplexer

```
MyMuxProcess : process (reset, clk)
begin
    if reset = ‘1’ then
            Out <= 0;
```

# VHDL for Multiplexer

```
MyMuxProcess : process (reset, clk)
begin
    if reset = ‘1’ then
            Out <= 0;
    elsif rising_edge(clk) then
```

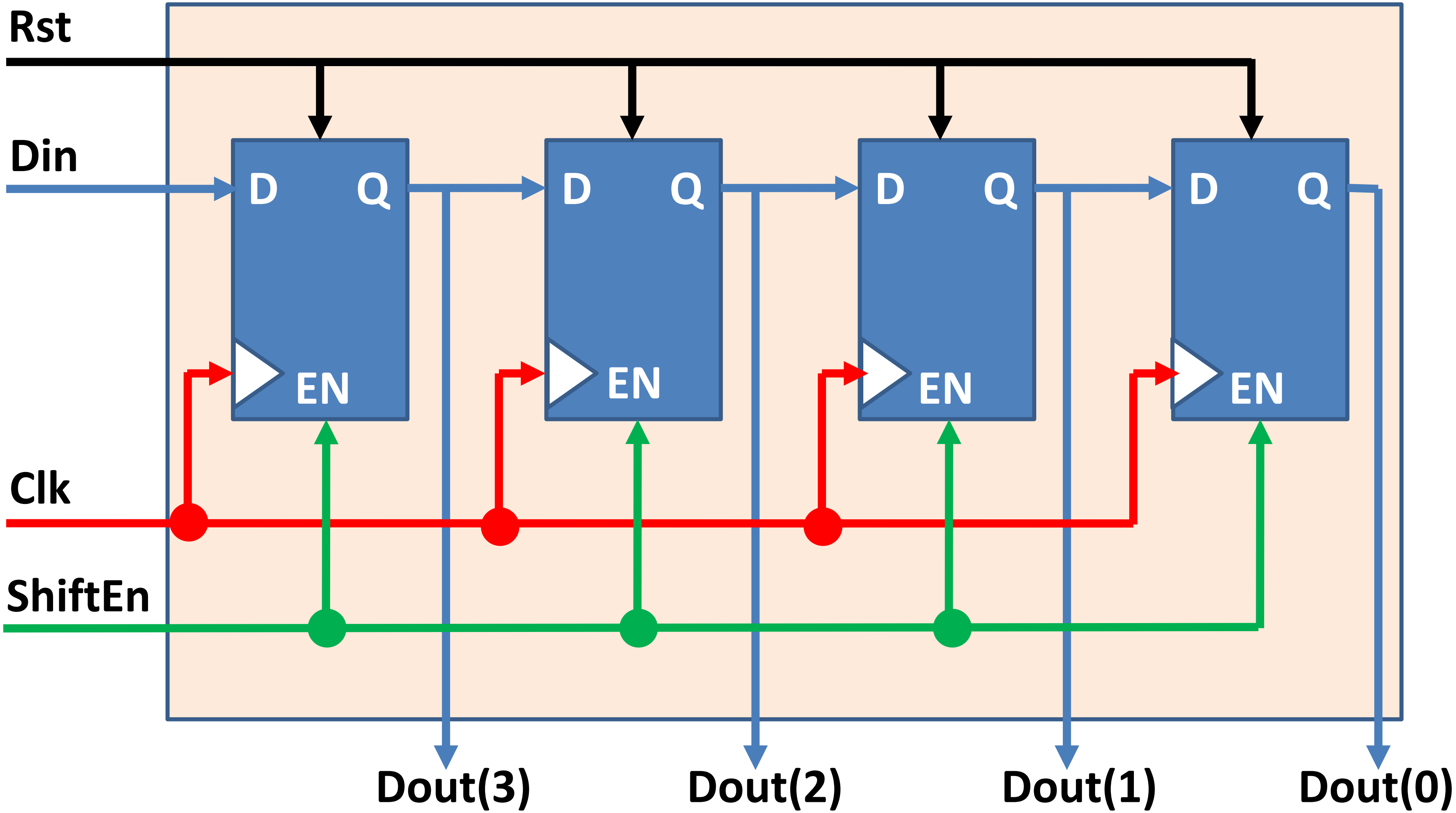
# VHDL for Multiplexer

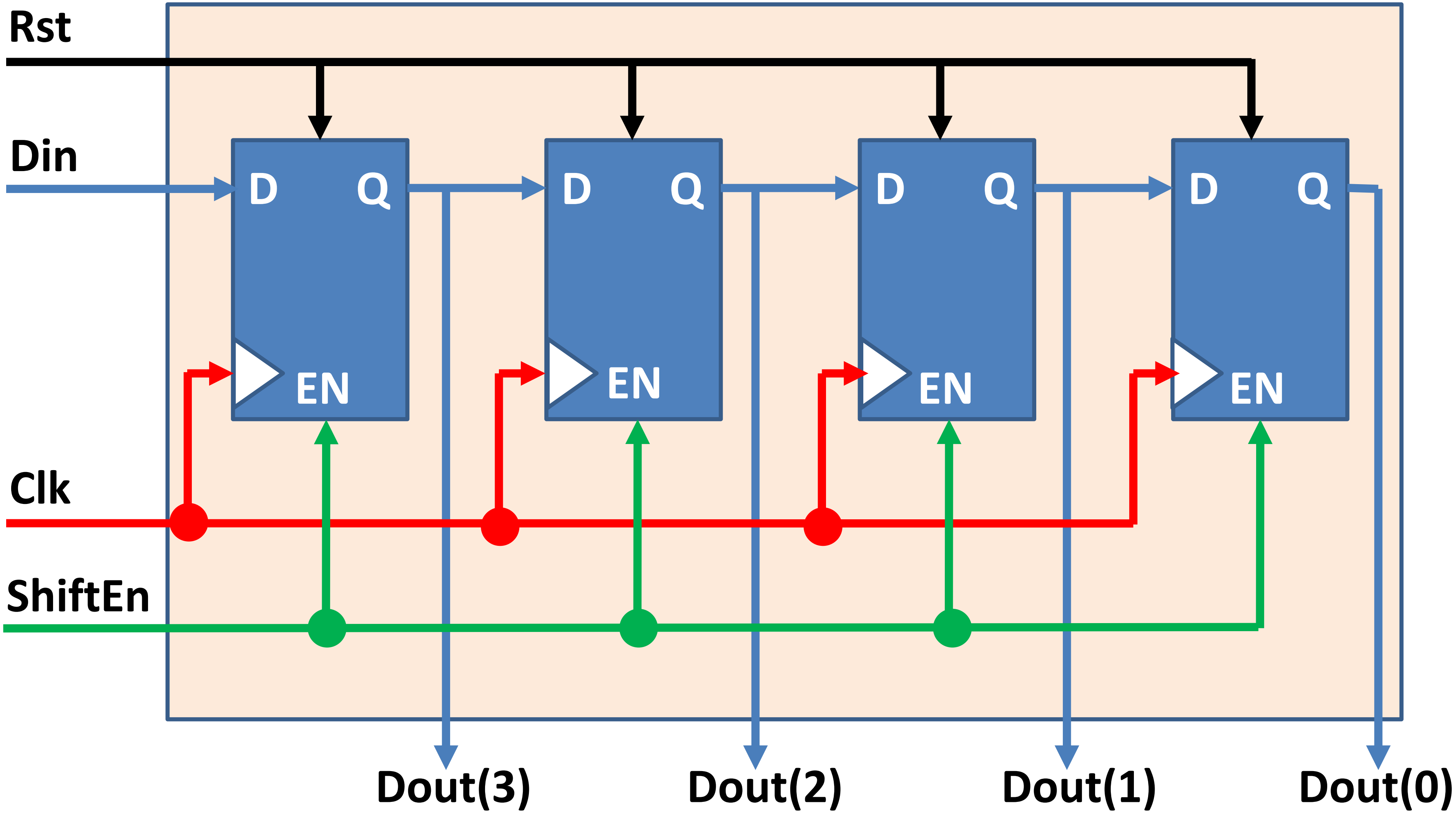
```
MyMuxProcess : process (reset, clk)
begin
    if reset = ‘1’ then
            Out <= 0;
    elsif rising_edge(clk) then
            case Sel is
                    when 0 => Out <= A;
                    when 1 => Out <= B;
                    when 2 => Out <= C;
                    when 3 => Out <= D;
                    when others =>
            end case;
```

# VHDL for Multiplexer

```
MyMuxProcess : process (reset, clk)
begin
    if reset = ‘1’ then
            Out <= 0;
    elsif rising_edge(clk) then
            case Sel is
                    when 0 => Out <= A;
                    when 1 => Out <= B;
                    when 2 => Out <= C;
                    when 3 => Out <= D;
                    when others =>
            end case;
    end if;
end process;
```

# Shift Registers


Rst
Din
D
Q
D
Q
D
Q
D
Q
EN
EN
EN
EN
Clk
ShiftEn
Dout(3)
Dout(2)
Dout(1)
Dout(0)


Rst
Din
D
Q
EN
D
Q
EN
D
Q
EN
D
Q
EN
Clk
ShiftEn
Dout(3)
Dout(2)
Dout(1)
Dout(0)

# VHDL for Shift Register

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

# VHDL for Shift Register

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

entity ShiftRegisterChain is
```

# VHDL for Shift Register

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

entity ShiftRegisterChain is
    generic (
        CHAIN_LENGTH : integer
    );
```

# VHDL for Shift Register

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

entity ShiftRegisterChain is
    generic (
        CHAIN_LENGTH : integer
    );
    port(
        Clk     : in std_logic;
        Rst     : in std_logic;
        ShiftEn : in std_logic;
        Din     : in std_logic;
        Dout    : out std_logic_vector(CHAIN_LENGTH-1 downto 0)
    );
```

# VHDL for Shift Register

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

entity ShiftRegisterChain is
    generic (
        CHAIN_LENGTH : integer
    );
    port(
        Clk     : in std_logic;
        Rst     : in std_logic;
        ShiftEn : in std_logic;
        Din     : in std_logic;
        Dout    : out std_logic_vector(CHAIN_LENGTH-1 downto 0)
    );
end entity;
```

# VHDL for Shift Register

```
architecture rtl of ShiftRegisterChain is
begin
```

# VHDL for Shift Register

```
architecture rtl of ShiftRegisterChain is
begin


    ShiftRegisterProcess : process(rst, clk)
```

# VHDL for Shift Register

```
architecture rtl of ShiftRegisterChain is
begin



    ShiftRegisterProcess : process(rst, clk)
    begin
```

# VHDL for Shift Register

```
architecture rtl of ShiftRegisterChain is
begin



    ShiftRegisterProcess : process(rst, clk)
    begin
        if rst = ‘1’ then
            Dout <= (others => ‘0’);
```

# VHDL for Shift Register

```
architecture rtl of ShiftRegisterChain is
begin



    ShiftRegisterProcess : process(rst, clk)
    begin
        if rst = ‘1’ then
            Dout <= (others => ‘0’);
        elsif rising_edge(clk) then
```

# VHDL for Shift Register

```
architecture rtl of ShiftRegisterChain is
begin



    ShiftRegisterProcess : process(rst, clk)
    begin
        if rst = ‘1’ then
            Dout <= (others => ‘0’);
        elsif rising_edge(clk) then
            Dout <= Din  & Dout (Dout’left downto 1) ;
```

# VHDL for Shift Register

```
Dout  :  out std_logic_vector(CHAIN_LENGTH-1 downto 0);

Dout <= Din & Dout (Dout ’left downto 1);
```

# VHDL for Shift Register

```
Dout  :  out std_logic_vector(CHAIN_LENGTH-1 downto 0);

Dout <= Din & Dout (Dout ’left downto 1);
```

# VHDL for Shift Register

```
Dout  :  out std_logic_vector(3 downto 0);
```

```
Dout <= Din & Dout (Dout ’left downto 1);
```

# VHDL for Shift Register

```
Dout  :  out std_logic_vector(3 downto 0);


Dout <= Din & Dout (Dout ’left downto 1);
```

# VHDL for Shift Register

```
Dout  :  out std_logic_vector(3 downto 0);
```

```
Dout <= Din & Dout (3 downto 1);
```

# VHDL for Shift Register

```
Dout  :  out std_logic_vector(3 downto 0);


Dout <= Din & Dout (3 downto 1);
```

| Index | 3 | 2 | 1 | 0 |
|---|---|---|---|---|
| Value | Dout(3) | Dout(2) | Dout(1) | Dout(0) |

# VHDL for Shift Register

```
Dout  :  out std_logic_vector(3 downto 0);
```

```
Dout <= Din & Dout (3 downto 1);
```

| Index | 3 | 2 | 1 | 0 |
|---|---|---|---|---|
| Value | Dout(3) | Dout(2) | Dout(1) | Dout(0) |

# VHDL for Shift Register

```
Dout  :  out std_logic_vector(3 downto 0);
```

```
Dout <= Din & Dout (3 downto 1);
```

| Index | 3 | 2 | 1 | 0 |
|---|---|---|---|---|
| Value | Dout(3) | Dout(2) | Dout(1) | Dout(0) |

# VHDL for Shift Register

```
Dout  :  out std_logic_vector(3 downto 0);


Dout <= Din & Dout (3 downto 1);
```

| Index | 3 | 2 | 1 | 0 |
|---|---|---|---|---|
| Value | Dout(3) | Dout(2) | Dout(1) | Dout(0) |

# VHDL for Shift Register

```
Dout  :  out std_logic_vector(3 downto 0);
```

```
Dout <= Din & Dout (3 downto 1);
```

| Index | 3 | 2 | 1 | 0 |
|---|---|---|---|---|
| Value | Dout(3) | Dout(2) | Dout(1) | Dout(0) |

<=

| Index | 3 | 2 | 1 | 0 |
|---|---|---|---|---|
| Value | Din | Dout(3) | Dout(2) | Dout(1) |

# VHDL for Shift Register

```
Dout  :  out std_logic_vector(3 downto 0);
```

```
Dout <= Din & Dout (3 downto 1);
```

| Index | 3 | 2 | 1 | 0 |
|---|---|---|---|---|
| Value | Dout(3) | Dout(2) | Dout(1) | Dout(0) |

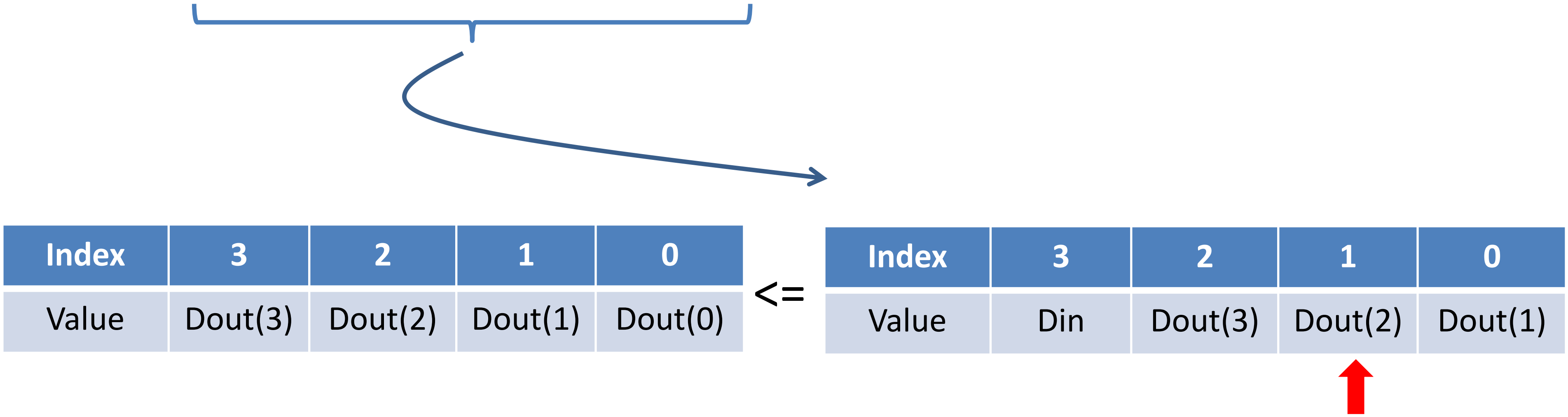
<=

| Index | 3 | 2 | 1 | 0 |
|---|---|---|---|---|
| Value | Din | Dout(3) | Dout(2) | Dout(1) |

# VHDL for Shift Register

```
Dout  :  out std_logic_vector(3 downto 0);
```

```
Dout <= Din & Dout (3 downto 1);
```



| Index | 3 | 2 | 1 | 0 |
|---|---|---|---|---|
| Value | Dout(3) | Dout(2) | Dout(1) | Dout(0) |

<=

| Index | 3 | 2 | 1 | 0 |
|---|---|---|---|---|
| Value | Din | Dout(3) | Dout(2) | Dout(1) |

# VHDL for Shift Register

```
Dout  :  out std_logic_vector(3 downto 0);
```

```
Dout <= Din & Dout (3 downto 1);
```

| Index | 3 | 2 | 1 | 0 |
|---|---|---|---|---|
| Value | Dout(3) | Dout(2) | Dout(1) | Dout(0) |

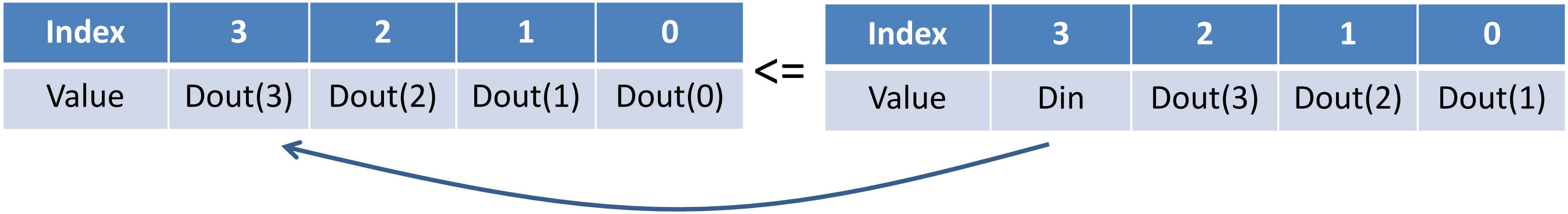
<=

| Index | 3 | 2 | 1 | 0 |
|---|---|---|---|---|
| Value | Din | Dout(3) | Dout(2) | Dout(1) |

# VHDL for Shift Register

```
Dout  :  out std_logic_vector(3 downto 0);
```

```
Dout <= Din & Dout (3 downto 1);
```



| Index | 3 | 2 | 1 | 0 |
|---|---|---|---|---|
| Value | Dout(3) | Dout(2) | Dout(1) | Dout(0) |

<=

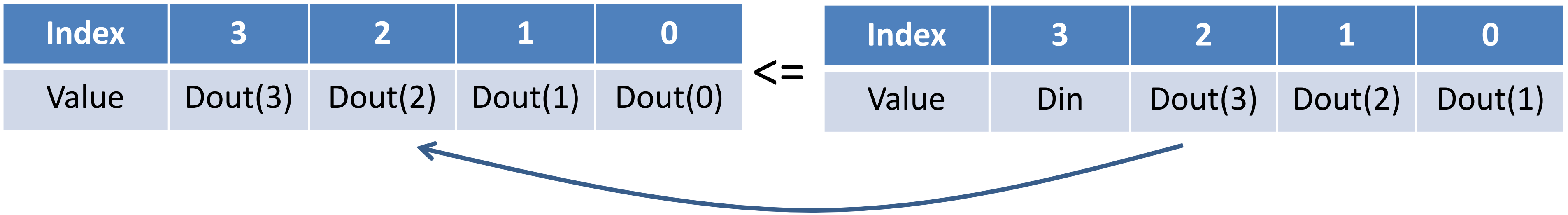
| Index | 3 | 2 | 1 | 0 |
|---|---|---|---|---|
| Value | Din | Dout(3) | Dout(2) | Dout(1) |

```
Dout(3) <= Din;
```

# VHDL for Shift Register

```
Dout  :  out std_logic_vector(3 downto 0);
```

```
Dout <= Din & Dout (3 downto 1);
```



| Index | 3 | 2 | 1 | 0 |
|---|---|---|---|---|
| Value | Dout(3) | Dout(2) | Dout(1) | Dout(0) |

<=

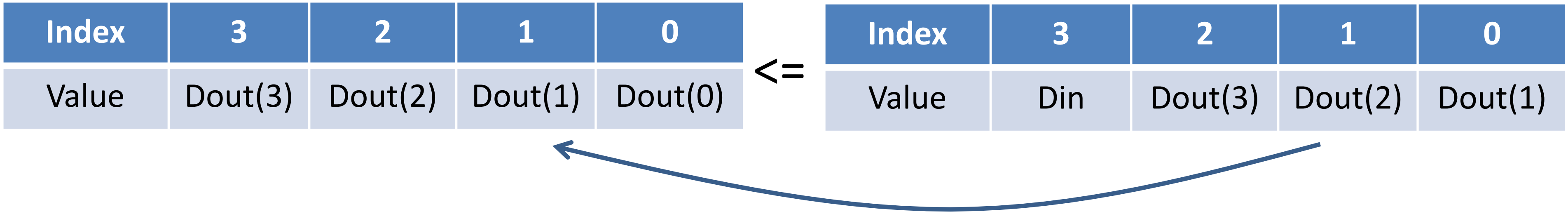
| Index | 3 | 2 | 1 | 0 |
|---|---|---|---|---|
| Value | Din | Dout(3) | Dout(2) | Dout(1) |

```
Dout(3) <= Din;
Dout(2) <= Dout(3);
```

# VHDL for Shift Register

```
Dout  :  out std_logic_vector(3 downto 0);
```

```
Dout <= Din & Dout (3 downto 1);
```



| Index | 3 | 2 | 1 | 0 |
|---|---|---|---|---|
| Value | Dout(3) | Dout(2) | Dout(1) | Dout(0) |

<=

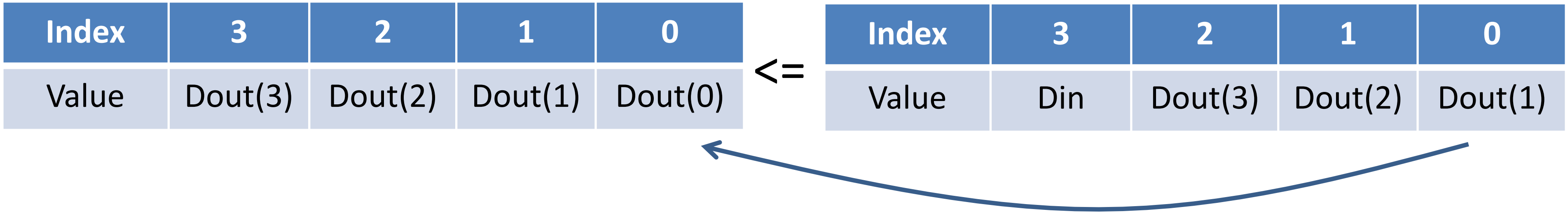
| Index | 3 | 2 | 1 | 0 |
|---|---|---|---|---|
| Value | Din | Dout(3) | Dout(2) | Dout(1) |

```
Dout(3) <= Din;
Dout(2) <= Dout(3);
Dout(1) <= Dout(2);
```

# VHDL for Shift Register

```
Dout  :  out std_logic_vector(3 downto 0);
```

```
Dout <= Din & Dout (3 downto 1);
```



| Index | 3 | 2 | 1 | 0 |
|---|---|---|---|---|
| Value | Dout(3) | Dout(2) | Dout(1) | Dout(0) |

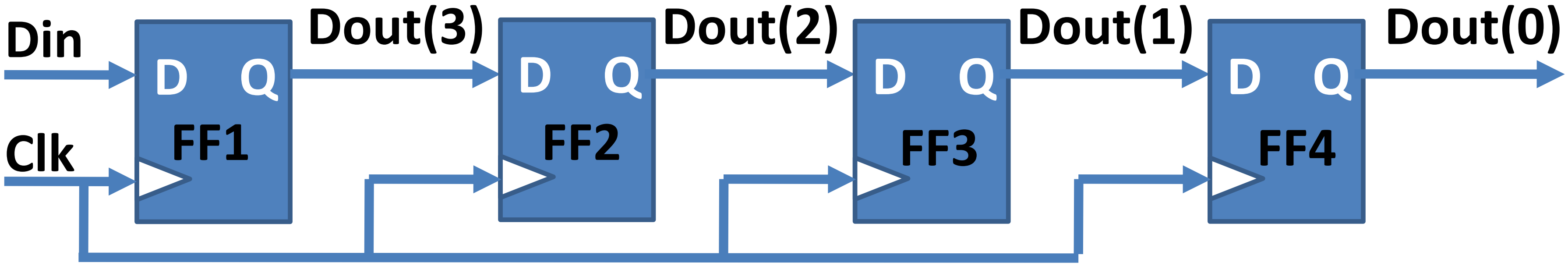
<=

| Index | 3 | 2 | 1 | 0 |
|---|---|---|---|---|
| Value | Din | Dout(3) | Dout(2) | Dout(1) |

```
Dout(3) <= Din;
Dout(2) <= Dout(3);
Dout(1) <= Dout(2);
Dout(0) <= Dout(1);
```

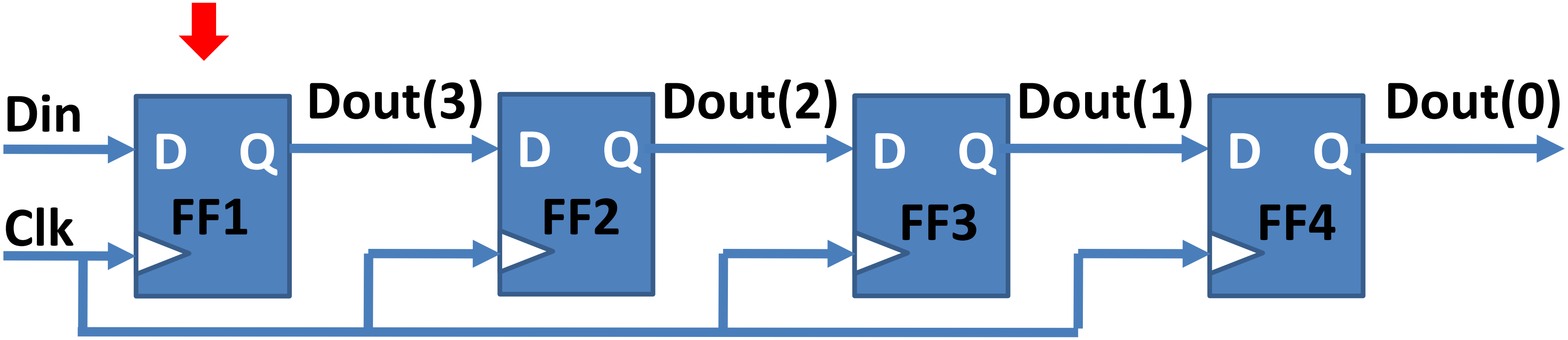
# VHDL for Shift Register

```
Dout(3) <= Din;
Dout(2) <= Dout(3);
Dout(1) <= Dout(2);
Dout(0) <= Dout(1);
```



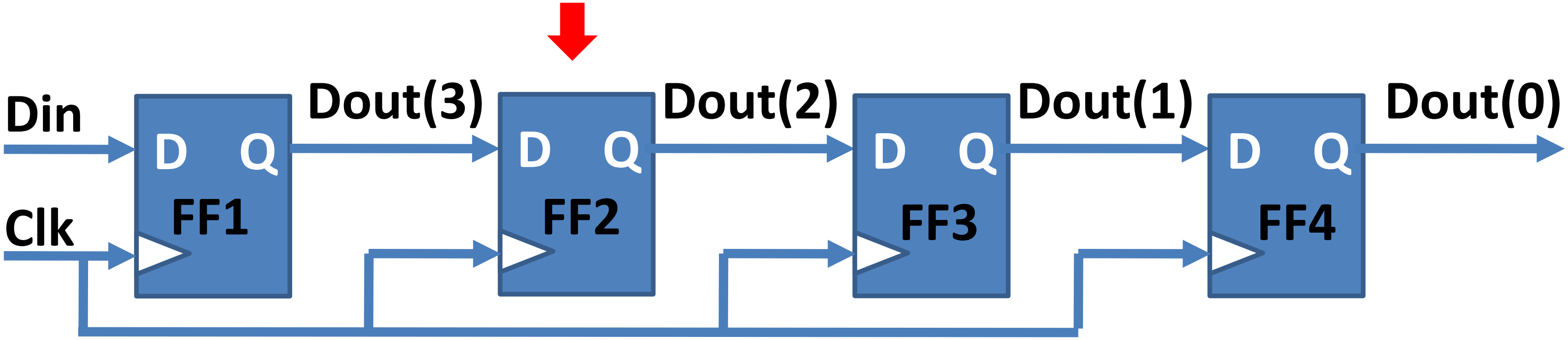
# VHDL for Shift Register

```
Dout(3) <= Din;
Dout(2) <= Dout(3);
Dout(1) <= Dout(2);
Dout(0) <= Dout(1);
```



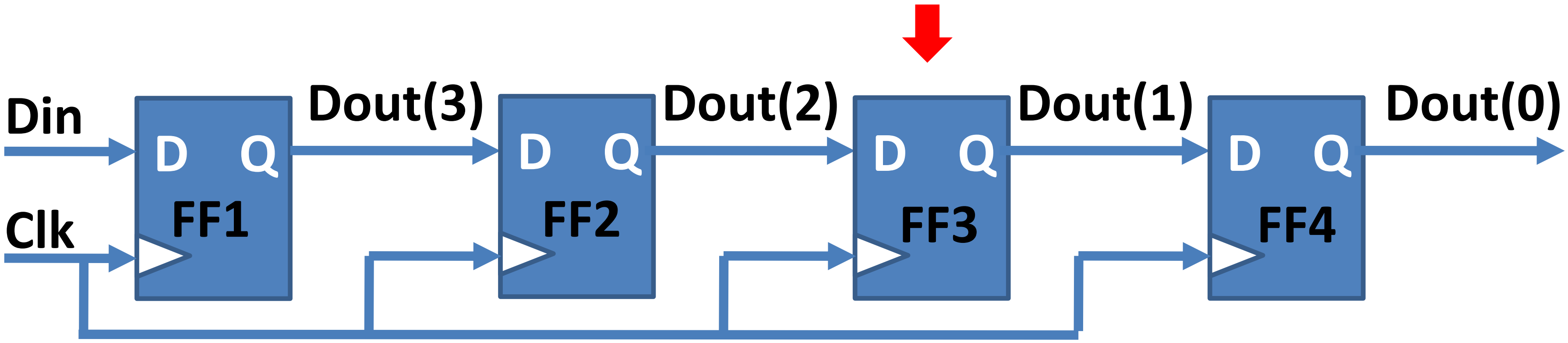
# VHDL for Shift Register

```
Dout(3) <= Din;
Dout(2) <= Dout(3);
Dout(1) <= Dout(2);
Dout(0) <= Dout(1);
```



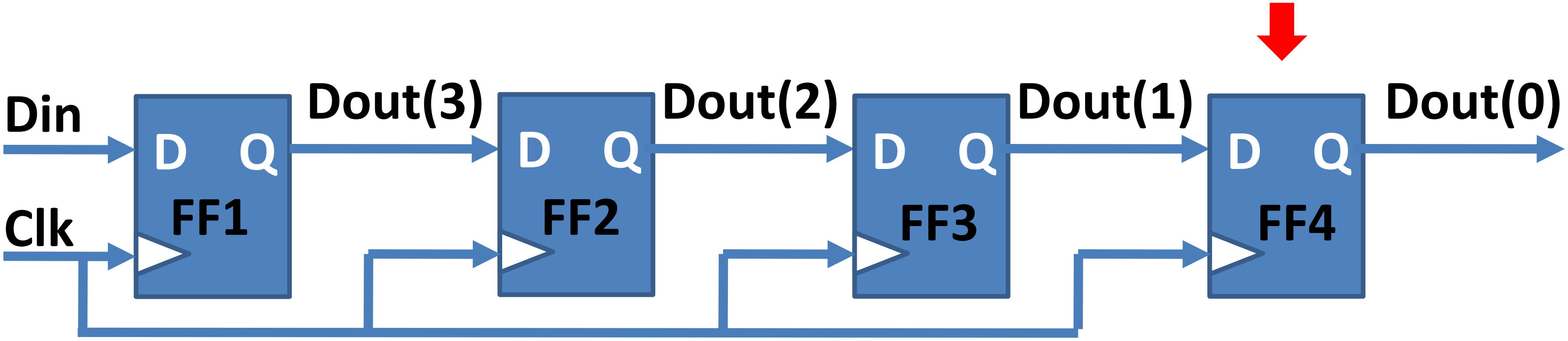
# VHDL for Shift Register

```
Dout(3) <= Din;
Dout(2) <= Dout(3);
Dout(1) <= Dout(2);
Dout(0) <= Dout(1);
```



# VHDL for Shift Register

```
Dout(3) <= Din;
Dout(2) <= Dout(3);
Dout(1) <= Dout(2);
Dout(0) <= Dout(1);
```



# VHDL for Shift Register

```
architecture rtl of ShiftRegisterChain is
begin


    ShiftRegisterProcess : process(rst, clk)
    begin
        if rst = ‘1’ then
            Dout <= (others => ‘0’);
        elsif rising_edge(clk) then
            Dout <= Din  & Dout (Dout’left downto 1) ;
        end if;
    end process;
```

# VHDL for Shift Register

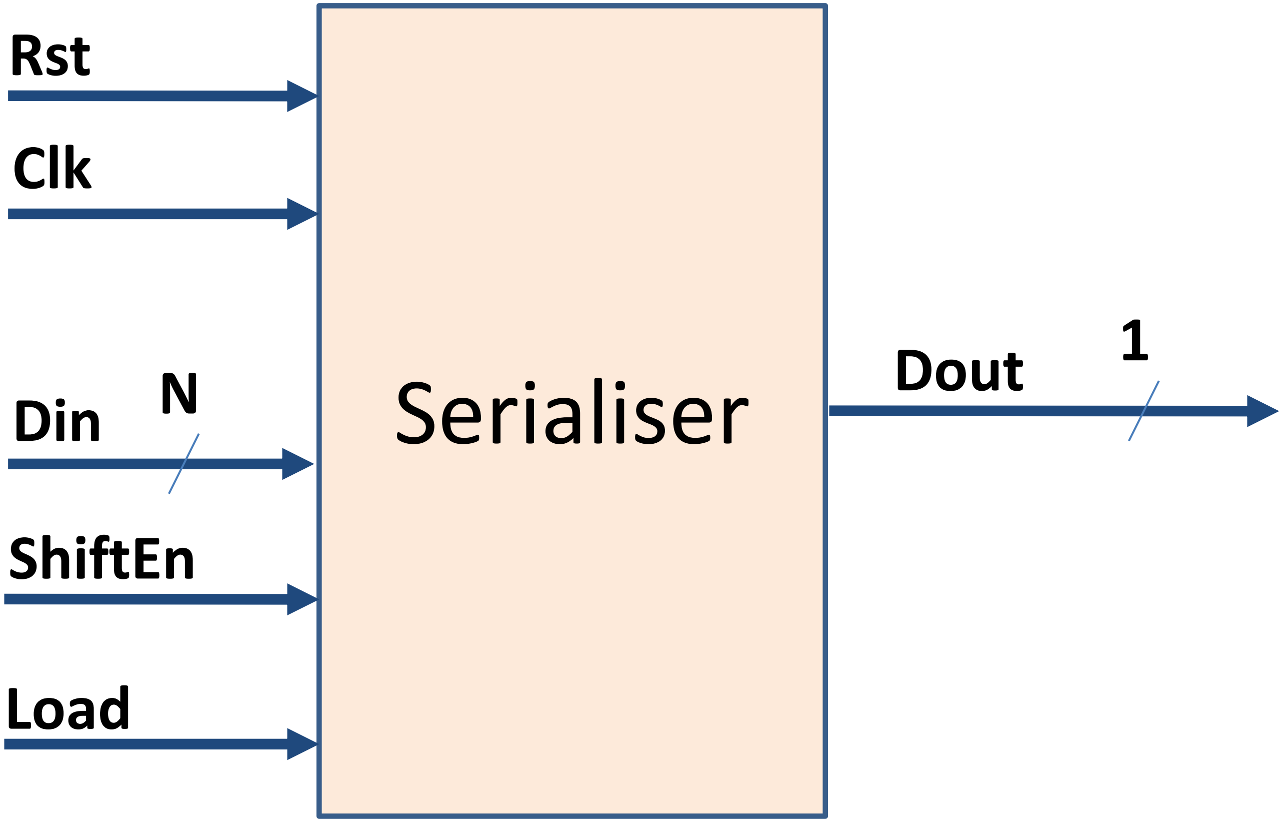
```
architecture rtl of ShiftRegisterChain is
begin



    ShiftRegisterProcess : process(rst, clk)
    begin
        if rst = ‘1’ then
             Dout <= (others => ‘0’);
        elsif rising_edge(clk) then
            if ShiftEn = ‘1’ then
                Dout <= Din  & Dout (Dout’left downto 1) ;
            end if;
        end if;
    end process;
```

# VHDL for Shift Register

```
architecture rtl of ShiftRegisterChain is
begin



    ShiftRegisterProcess : process(rst, clk)
    begin
        if rst = ‘1’ then
             Dout <= (others => ‘0’);
        elsif rising_edge(clk) then
            if ShiftEn = ‘1’ then
                Dout <= Din  & Dout (Dout’left downto 1) ;
            end if;
        end if;
    end process;
end rtl;
```

# Serialiser

# Serialiser



Rst

Clk

Din

N

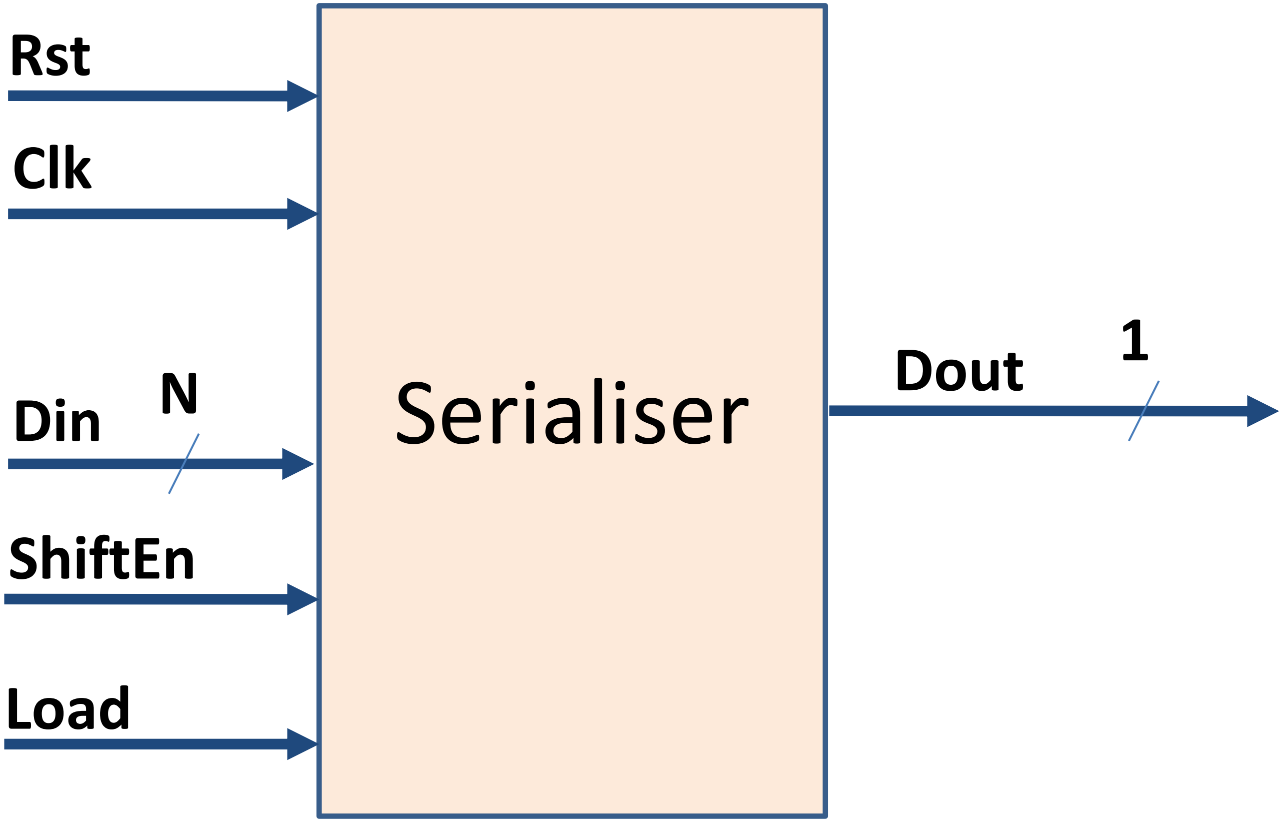
ShiftEn

Load

Serialiser

Dout

1


Rst
Clk
Din
N
ShiftEn
Load
Serialiser
Dout
1

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

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

entity Serialiser is
```

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

entity Serialiser is
	generic (
		DataWidth : integer
	);
```

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

entity Serialiser is
    generic (
        DataWidth : integer
    );
    port(
        Clk      : in std_logic;
        Rst      : in std_logic;
        ShiftEn  : in std_logic;
        Load     : in std_logic;
        Din      : in std_logic_vector(DataWidth -1 downto 0);
        Dout     : out std_logic
    );
end entity;
```

```vhdl
architecture rtl of Serialiser is
architecture rtl of Serialiser is
        Signal DataRegister : std_logic_vector(DataWidth-1 downto 0);
Begin
```

```vhdl
architecture rtl of Serialiser is
        Signal DataRegister : std_logic_vector(DataWidth-1 downto 0);
Begin
        SerialiserProcess : process(Rst,Clk)
        begin
```

```
architecture rtl of Serialiser is
    Signal DataRegister : std_logic_vector(DataWidth-1 downto 0);
Begin
    SerialiserProcess : process(Rst,Clk)
    begin
        if Rst = ‘1’ then
                DataRegister <= (others => ‘0’);
```

```
architecture rtl of Serialiser is
    Signal DataRegister : std_logic_vector(DataWidth-1 downto 0);
Begin
    SerialiserProcess : process(Rst,Clk)
    begin
        if Rst = ‘1’ then
                DataRegister <= (others => ‘0’);
        elsif rising_edge(Clk) then
```

```
architecture rtl of Serialiser is
    Signal DataRegister : std_logic_vector(DataWidth-1 downto 0);
Begin
    SerialiserProcess : process(Rst,Clk)
    begin
        if Rst = ‘1’ then
                DataRegister <= (others => ‘0’);
        elsif rising_edge(Clk) then
                if Load = ‘1’ then
                        DataRegister <= DIN;
```

```
architecture rtl of Serialiser is
    Signal DataRegister : std_logic_vector(DataWidth-1 downto 0);
Begin
    SerialiserProcess : process(Rst,Clk)
    begin
        if Rst = ‘1’ then
                DataRegister <= (others => ‘0’);
        elsif rising_edge(Clk) then
                if Load = ‘1’ then
                        DataRegister <= DIN;
                elsif ShiftEn = ‘1’ then
```

```
architecture rtl of Serialiser is
    Signal DataRegister : std_logic_vector(DataWidth-1 downto 0);
Begin
    SerialiserProcess : process(Rst,Clk)
    begin
        if Rst = ‘1’ then
                DataRegister <= (others => ‘0’);
        elsif rising_edge(Clk) then
                if Load = ‘1’ then
                        DataRegister <= DIN;
                elsif ShiftEn = ‘1’ then
                         DataRegister <= ‘0’ & DataRegister(DataWidth-1 downto 1);
```

```
architecture rtl of Serialiser is
    Signal DataRegister : std_logic_vector(DataWidth-1 downto 0);
Begin
    SerialiserProcess : process(rst,clk)
    begin
        if rst = ‘1’ then
                DataRegister <= (others => ‘0’);
        elsif rising_edge(clk) then
                if Load = ‘1’ then
                        DataRegister <= DIN;
                elsif ShiftEn = ‘1’ then
                         DataRegister <= ‘0’ & DataRegister(DataWidth-1 downto 1);
                end if;
        end if;
    end process;
```

```
architecture rtl of Serialiser is
    Signal DataRegister : std_logic_vector(DataWidth-1 downto 0);
Begin
    SerialiserProcess : process(rst,clk)
    begin
        if rst = ‘1’ then
                DataRegister <= (others => ‘0’);
        elsif rising_edge(clk) then
                if Load = ‘1’ then
                        DataRegister <= DIN;
                elsif ShiftEn = ‘1’ then
                         DataRegister <= ‘0’ & DataRegister(DataWidth-1 downto 1);
                end if;
        end if;
    end process;
    Dout <= DataRegister(0);
```

```
architecture rtl of Serialiser is
    Signal DataRegister : std_logic_vector(DataWidth-1 downto 0);
Begin
    SerialiserProcess : process(rst,clk)
    begin
        if rst = ‘1’ then
                DataRegister <= (others => ‘0’);
        elsif rising_edge(clk) then
                if Load = ‘1’ then
                        DataRegister <= DIN;
                elsif ShiftEn = ‘1’ then
                         DataRegister <= ‘0’ & DataRegister(DataWidth-1 downto 1);
                end if;
        end if;
    end process;
    Dout <= DataRegister(0);
end rtl;
```

# RAMs

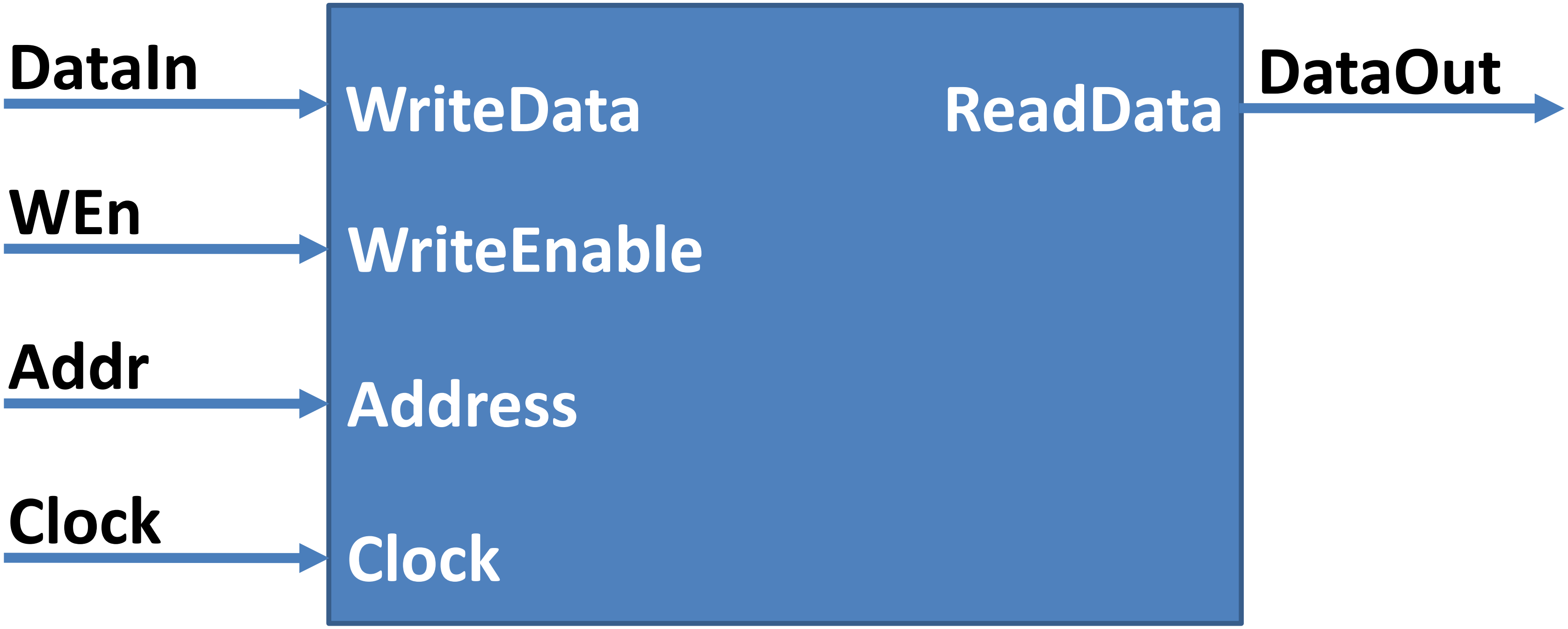
# RAM Implementation

- FPGAs implement RAM using :
  1. Register Banks (for smaller memories)
  2. Block RAM (for larger memories)
- Synthesiser infers how to implement RAM by looking at your VHDL code.
- Synthesiser by default goes for Register bank implementation.
- VHDL code must follow a particular template to get the synthesiser to use Block RAM.

# RAM Implementation

- FPGAs implement RAM using :
  1. Register Banks (for smaller memories)
  2. Block RAM (for larger memories)
- Synthesiser infers how to implement RAM by looking at your VHDL code.
- Synthesiser by default goes for Register bank implementation.
- VHDL code must follow a particular template to get the synthesiser to use Block RAM.

# Single Port RAM



DataIn

WEn

Addr

Clock

WriteData

WriteEnable

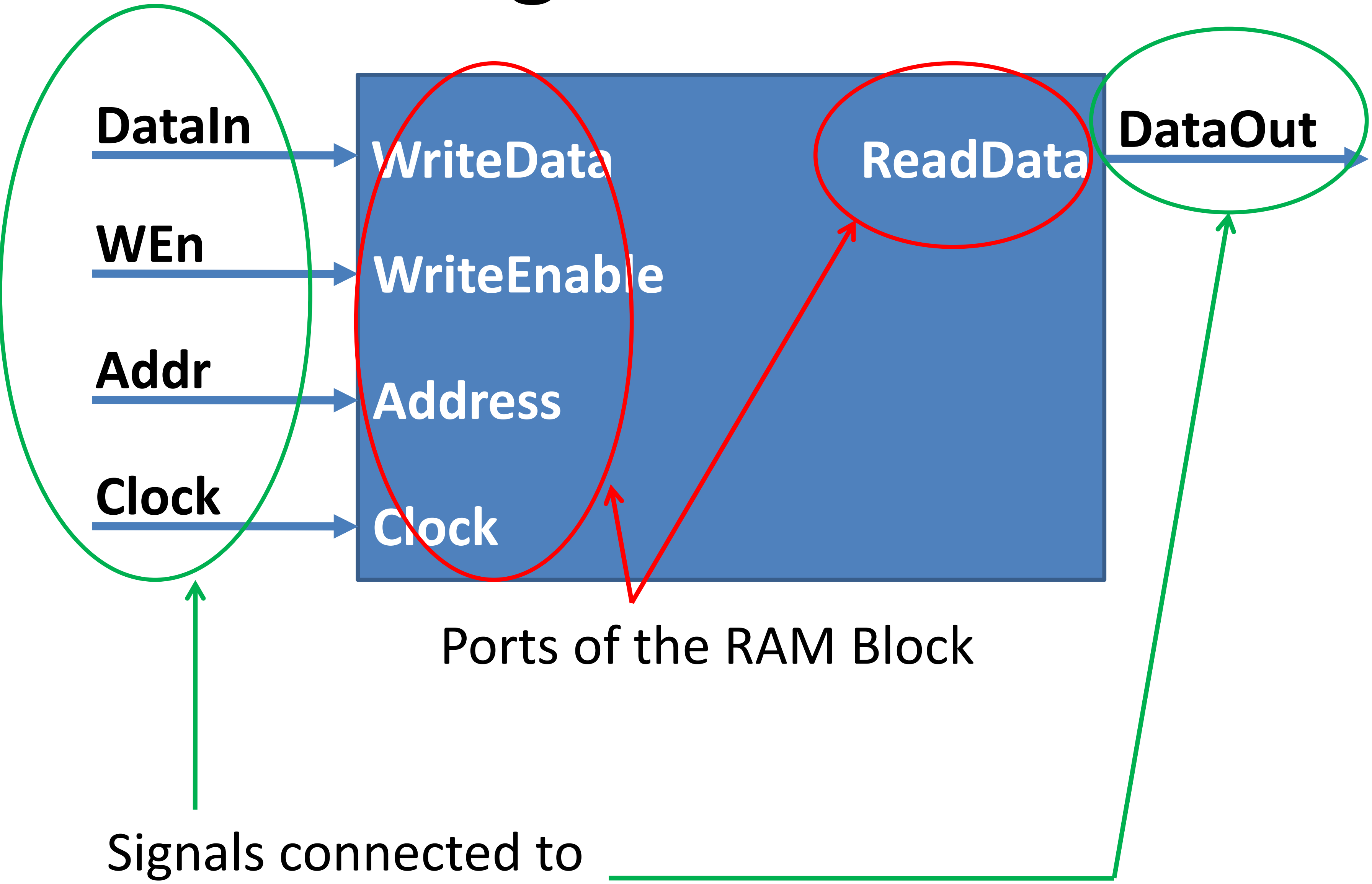
Address

Clock

ReadData

DataOut

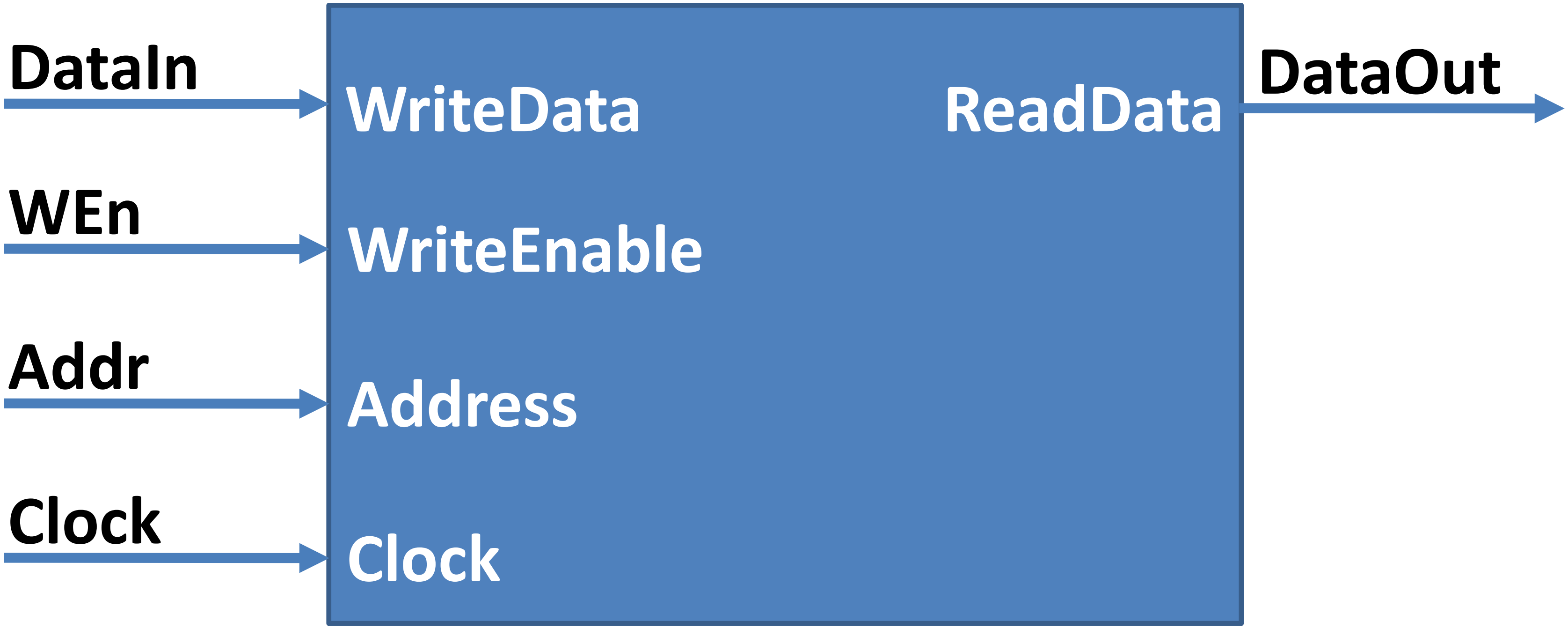
# Single Port RAM



Ports of the RAM Block

Signals connected to the ports

# Single Port RAM



DataIn

WEn

Addr

Clock

WriteData

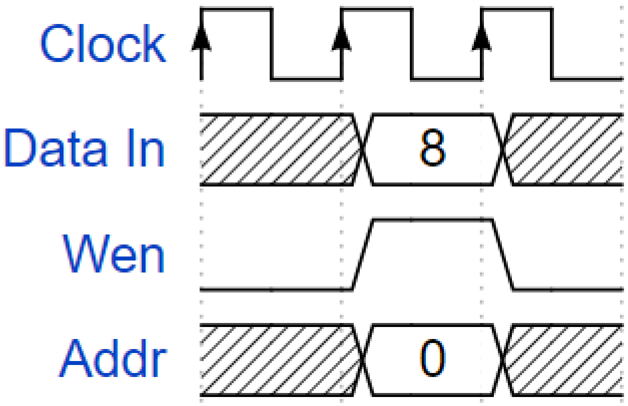
WriteEnable

Address

Clock

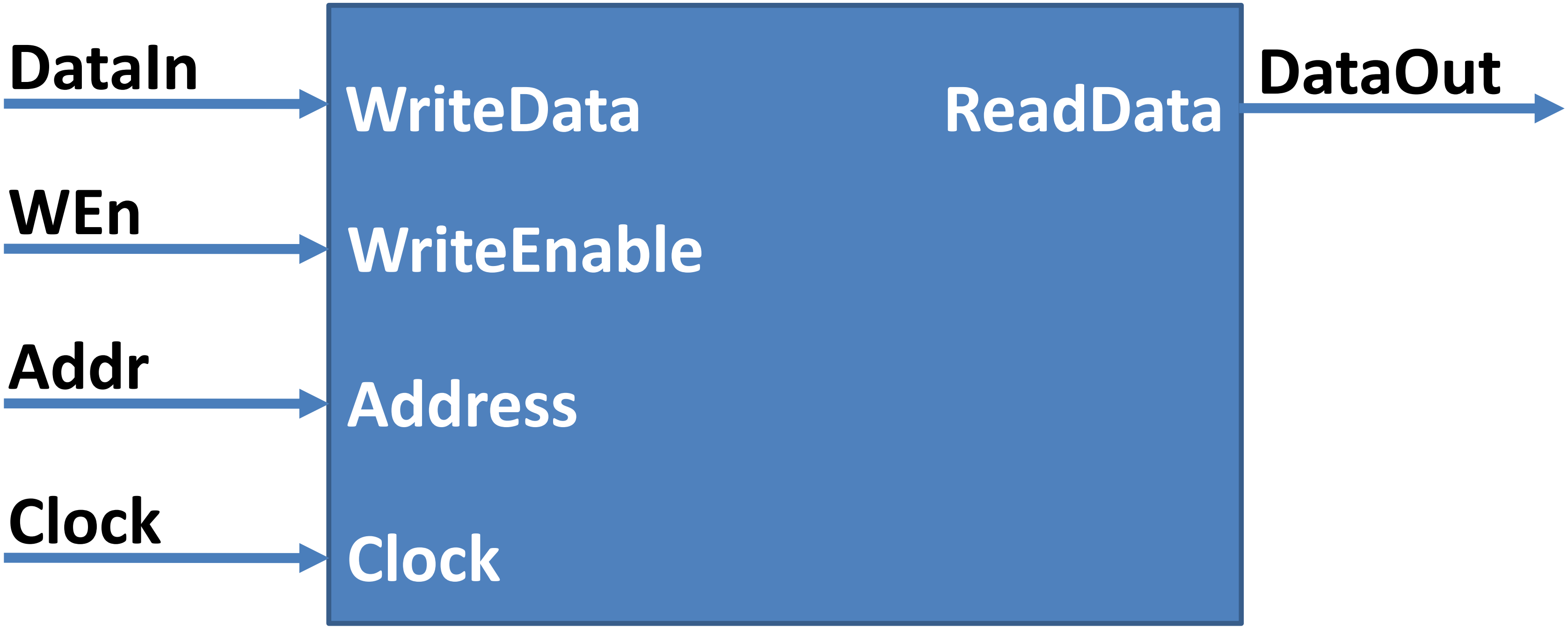
ReadData

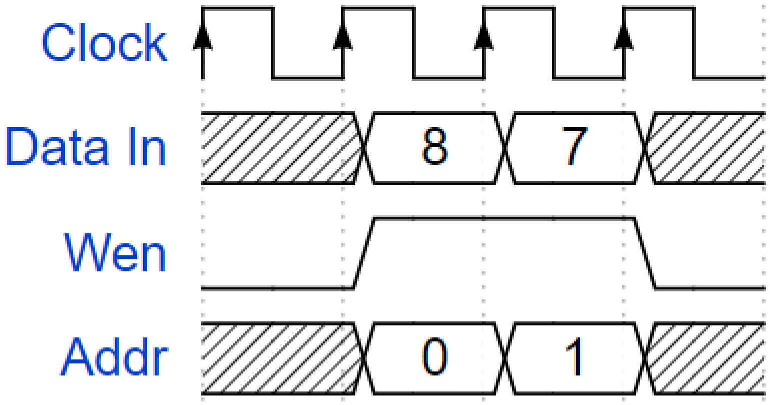
DataOut



Write Single Word

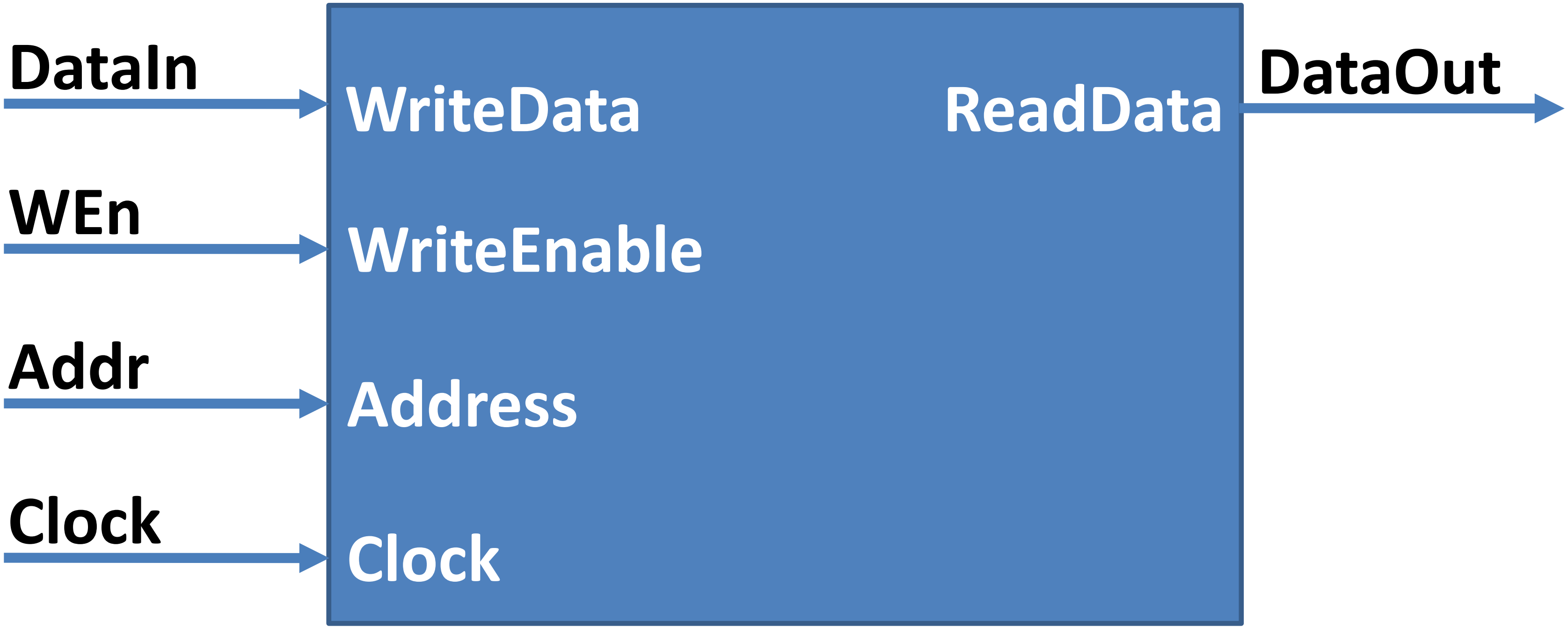
# Single Port RAM





Write Two Words

# Single Port RAM


DataIn

WEn

Addr

Clock

WriteData

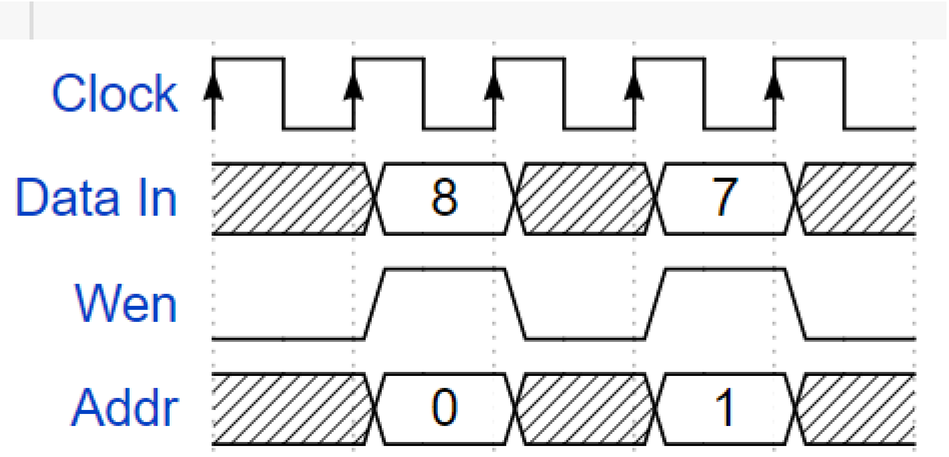
WriteEnable

Address

Clock

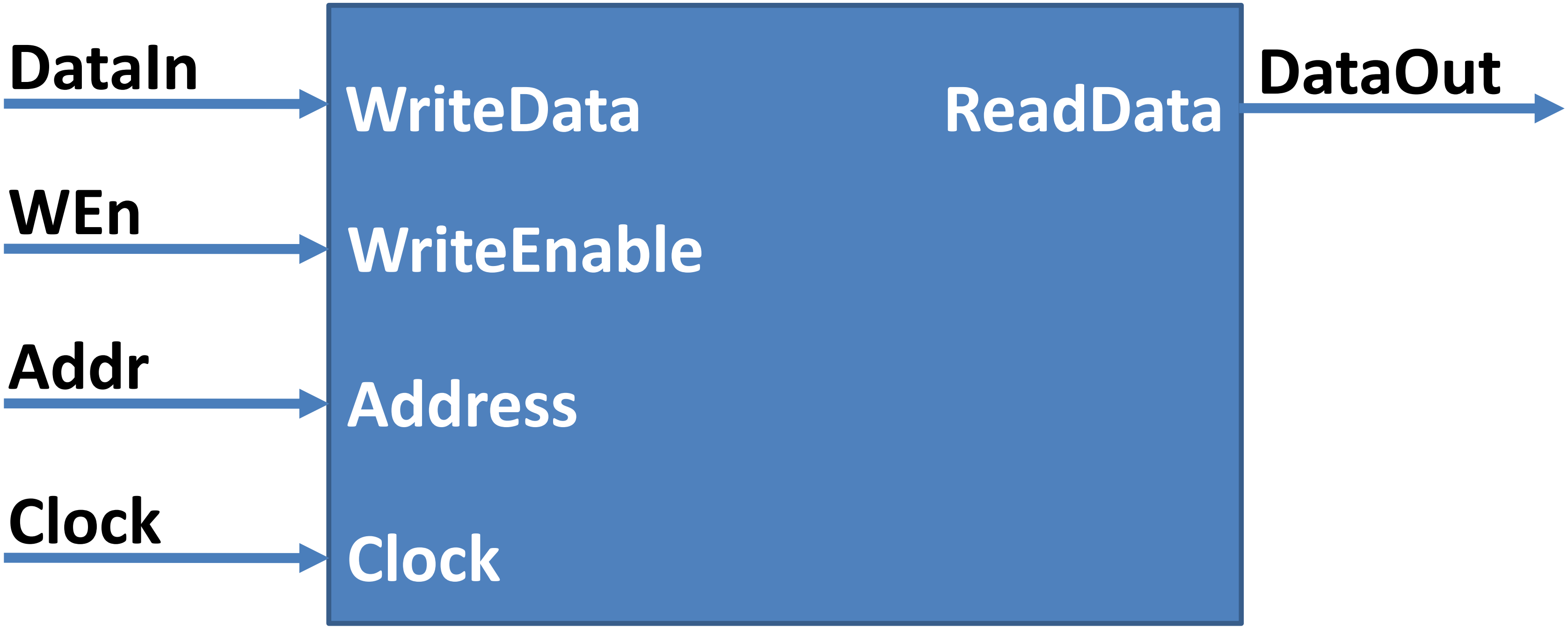
ReadData

DataOut



Write Two Words

# Single Port RAM



DataIn

WEn

Addr

Clock

WriteData

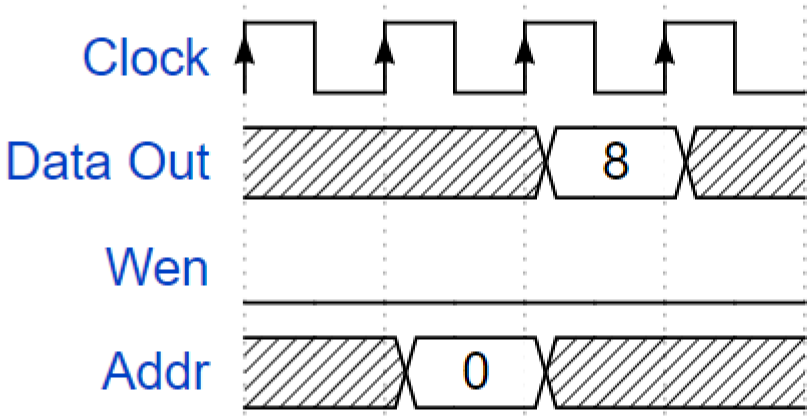
WriteEnable

Address

Clock

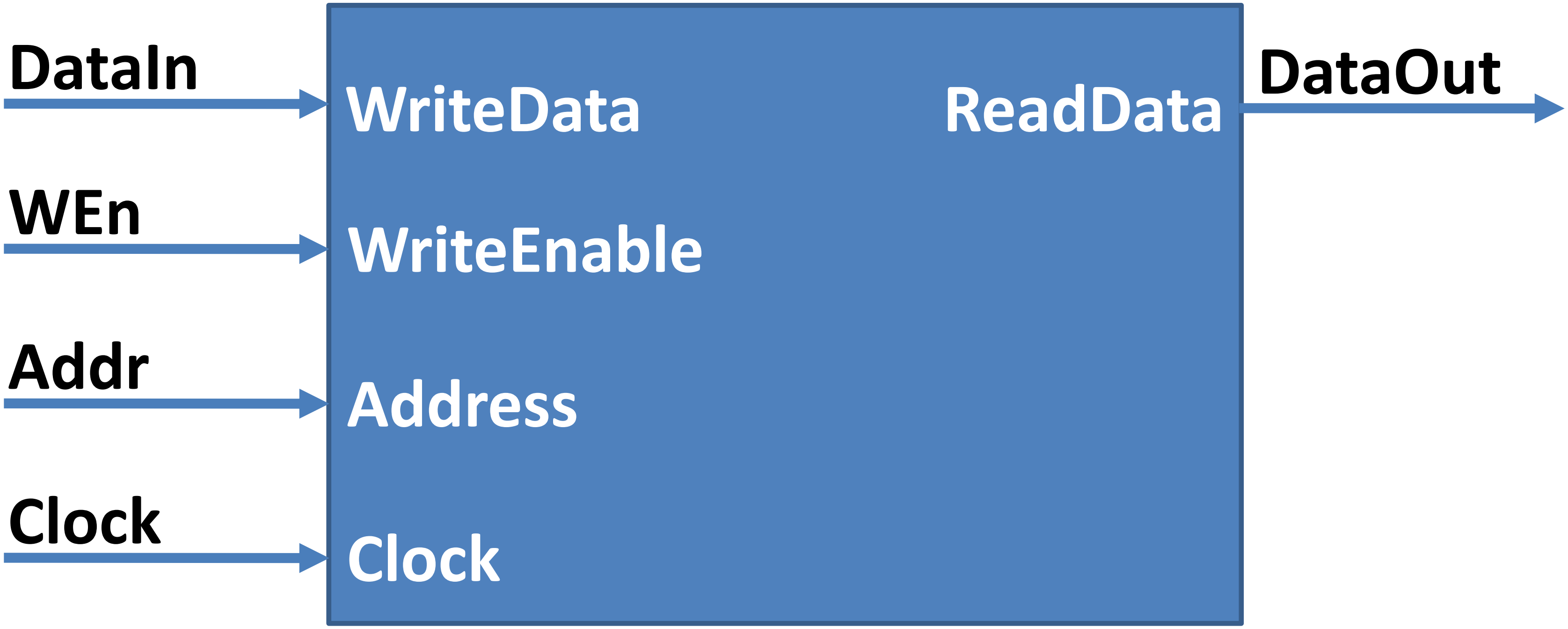
ReadData

DataOut



Read Single Word

# Single Port RAM



DataIn

WEn

Addr

Clock

WriteData

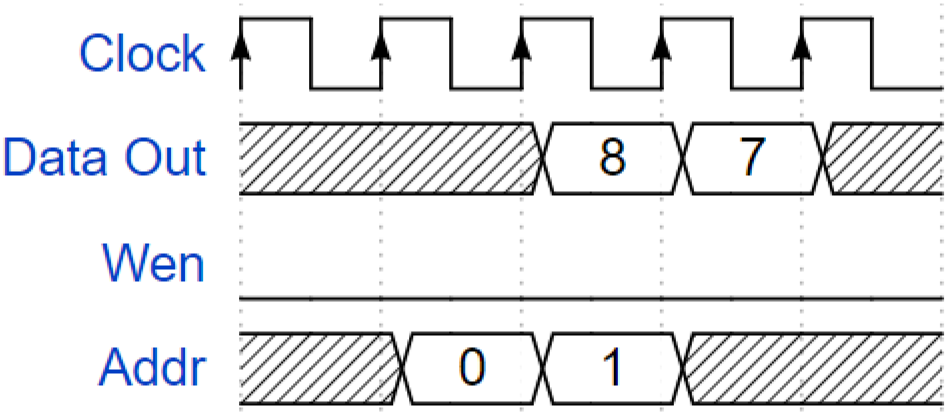
WriteEnable

Address

Clock

ReadData

DataOut



Read Two Words

# VHDL for Single Port RAM

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.numeric_std.all;
```

# VHDL for Single Port RAM

```
entity SinglePortRAM is
    generic(
        Dwidth : integer;
        Awidth : integer
    );
    port(
        Clock       : in   std_logic;
        WriteData   : in   std_logic_vector(Dwidth - 1 downto 0);
        ReadData    : out  std_logic_vector(Dwidth - 1 downto 0);
        Address     : in   std_logic_vector(Awidth - 1 downto 0);
        WriteEnable : in   std_logic
    );
end SinglePortRAM ;
```

# VHDL for Single Port RAM

```
entity SinglePortRAM is
    generic(
        Dwidth : integer;
        Awidth : integer
    );
    port(
        Clock           : in   std_logic;
        WriteData       : in   std_logic_vector(Dwidth - 1 downto 0);
        ReadData        : out  std_logic_vector(Dwidth - 1 downto 0);
        Address         : in   std_logic_vector(Awidth - 1 downto 0);
        WriteEnable     : in   std_logic
    );
end SinglePortRAM ;
```

# VHDL for Single Port RAM

```
entity SinglePortRAM is
    generic(
        Dwidth : integer;
        Awidth : integer
    );
    port(
        Clock           : in   std_logic;
        WriteData       : in   std_logic_vector(Dwidth - 1 downto 0);
        ReadData        : out  std_logic_vector(Dwidth - 1 downto 0);
        Address         : in   std_logic_vector(Awidth - 1 downto 0);
        WriteEnable     : in   std_logic
    );
end SinglePortRAM ;
```

# VHDL for Single Port RAM

```
architecture rtl of SinglePortRAM is
```

# VHDL for Single Port RAM

```
architecture rtl of SinglePortRAM is
    -- Build a 2D array type for the RAM
    type memory_t is array(0 to 2**Awidth -1 ) of
    std_logic_vector((Dwidth-1) downto 0);
```

# VHDL for Single Port RAM

```
architecture rtl of SinglePortRAM is
    -- Build a 2D array type for the RAM
    type memory_t is array(0 to 2**Awidth -1 ) of
    std_logic_vector((Dwidth-1) downto 0);
    -- Declare the RAM signal
     signal MyRam : memory_t ;
begin
```

# VHDL for Single Port RAM

```
architecture rtl of SinglePortRAM is
    -- Build a 2D array type for the RAM
    type memory_t is array(0 to 2**Awidth -1 ) of
    std_logic_vector((Dwidth-1) downto 0);
    -- Declare the RAM signal
     signal MyRam : memory_t ;
begin
    RAMProcess : process (Clock) begin
```

# VHDL for Single Port RAM

```
architecture rtl of SinglePortRAM is
    -- Build a 2D array type for the RAM
    type memory_t is array(0 to 2**Awidth -1 ) of
    std_logic_vector((Dwidth-1) downto 0);
    -- Declare the RAM signal
     signal MyRam : memory_t ;
begin
    RAMProcess : process (Clock) begin
        if rising_edge(Clock) then
```

# VHDL for Single Port RAM

```
architecture rtl of SinglePortRAM is
    -- Build a 2D array type for the RAM
    type memory_t is array(0 to 2**Awidth -1 ) of
    std_logic_vector((Dwidth-1) downto 0);
    -- Declare the RAM signal
     signal MyRam : memory_t ;
begin
    RAMProcess : process (Clock) begin
        if rising_edge(Clock) then
            if WriteEnable = '1'  then
                MyRam (to_integer(unsigned(Address))) <= WriteData;
            end if;
```

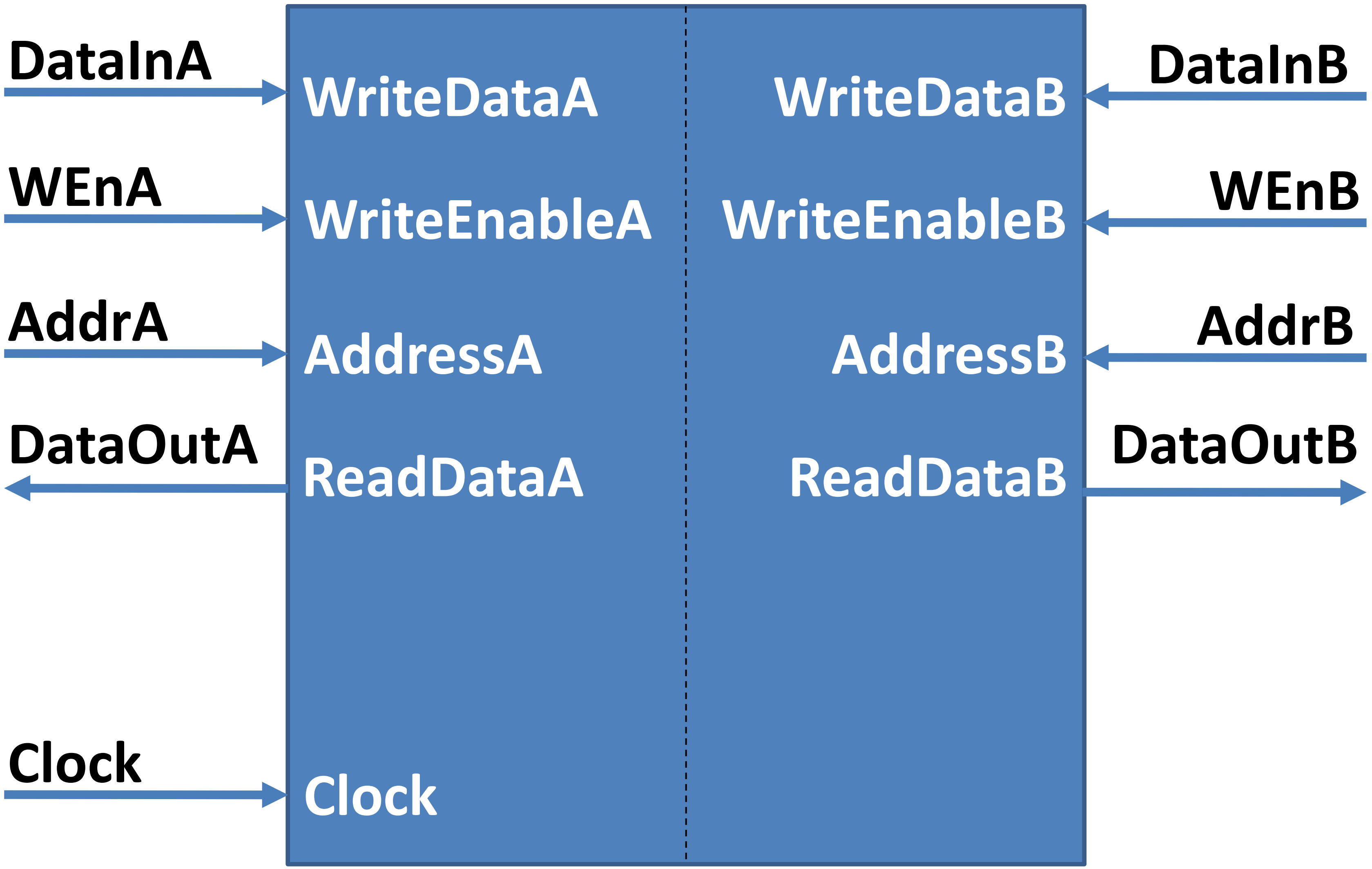
# VHDL for Single Port RAM

```
architecture rtl of SinglePortRAM is
    -- Build a 2D array type for the RAM
    type memory_t is array(0 to 2**Awidth -1 ) of
    std_logic_vector((Dwidth-1) downto 0);
    -- Declare the RAM signal
     signal MyRam : memory_t ;
begin
    RAMProcess : process (Clock) begin
        if rising_edge(Clock) then
            if WriteEnable = '1'  then
                MyRam (to_integer(unsigned(Address))) <= WriteData;
            end if;
            ReadData <= MyRam(to_integer(unsigned(Address)));
```

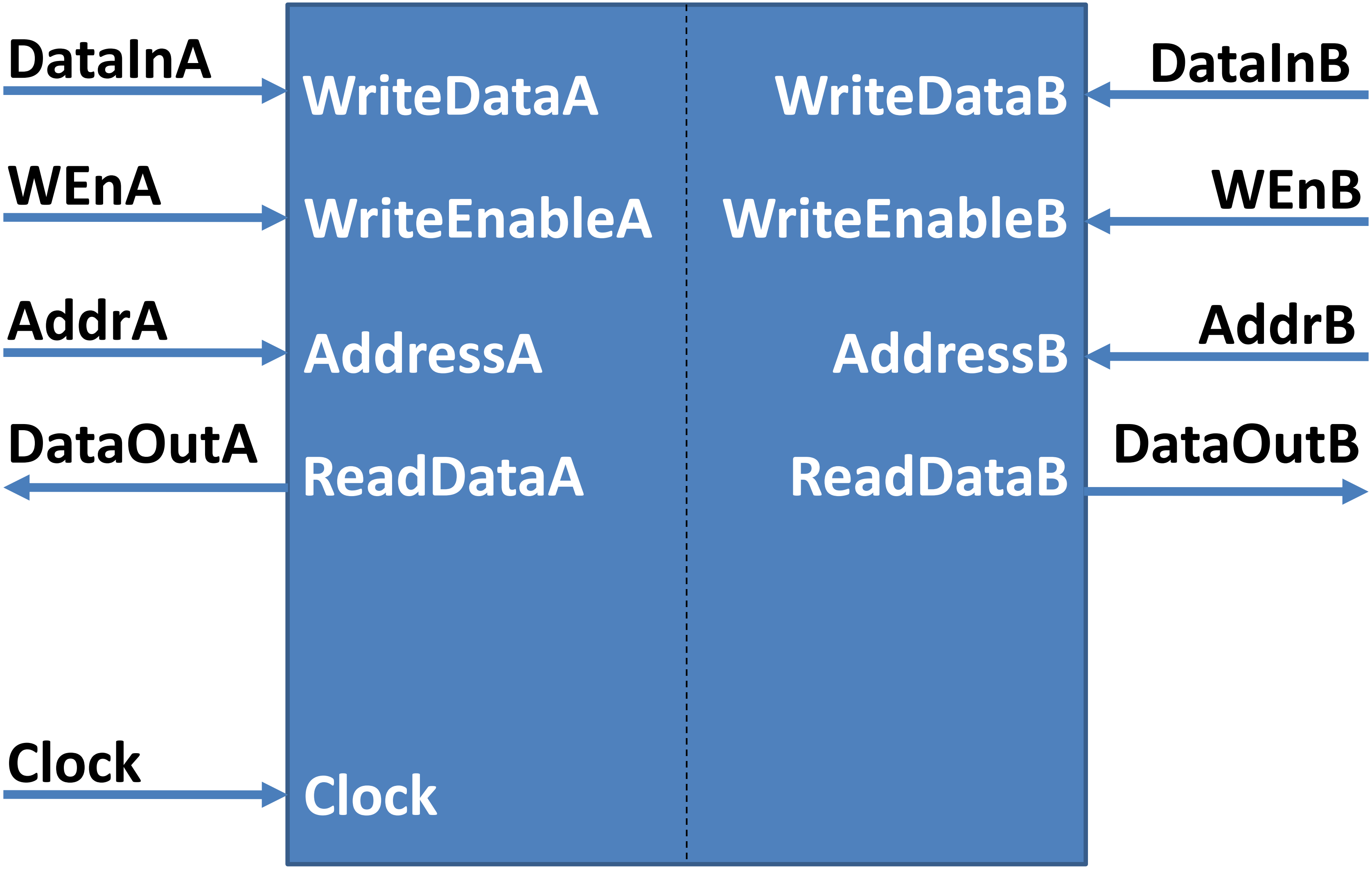
# VHDL for Single Port RAM

```
architecture rtl of SinglePortRAM is
    -- Build a 2D array type for the RAM
    type memory_t is array(0 to 2**Awidth -1 ) of
    std_logic_vector((Dwidth-1) downto 0);
    -- Declare the RAM signal
     signal MyRam : memory_t ;
begin
    RAMProcess : process (Clock) begin
        if rising_edge(Clock) then
            if WriteEnable = '1'  then
                MyRam (to_integer(unsigned(Address))) <= WriteData;
            end if;
            ReadData <= MyRam(to_integer(unsigned(Address)));
        end if;
     end process;
end rtl;
```

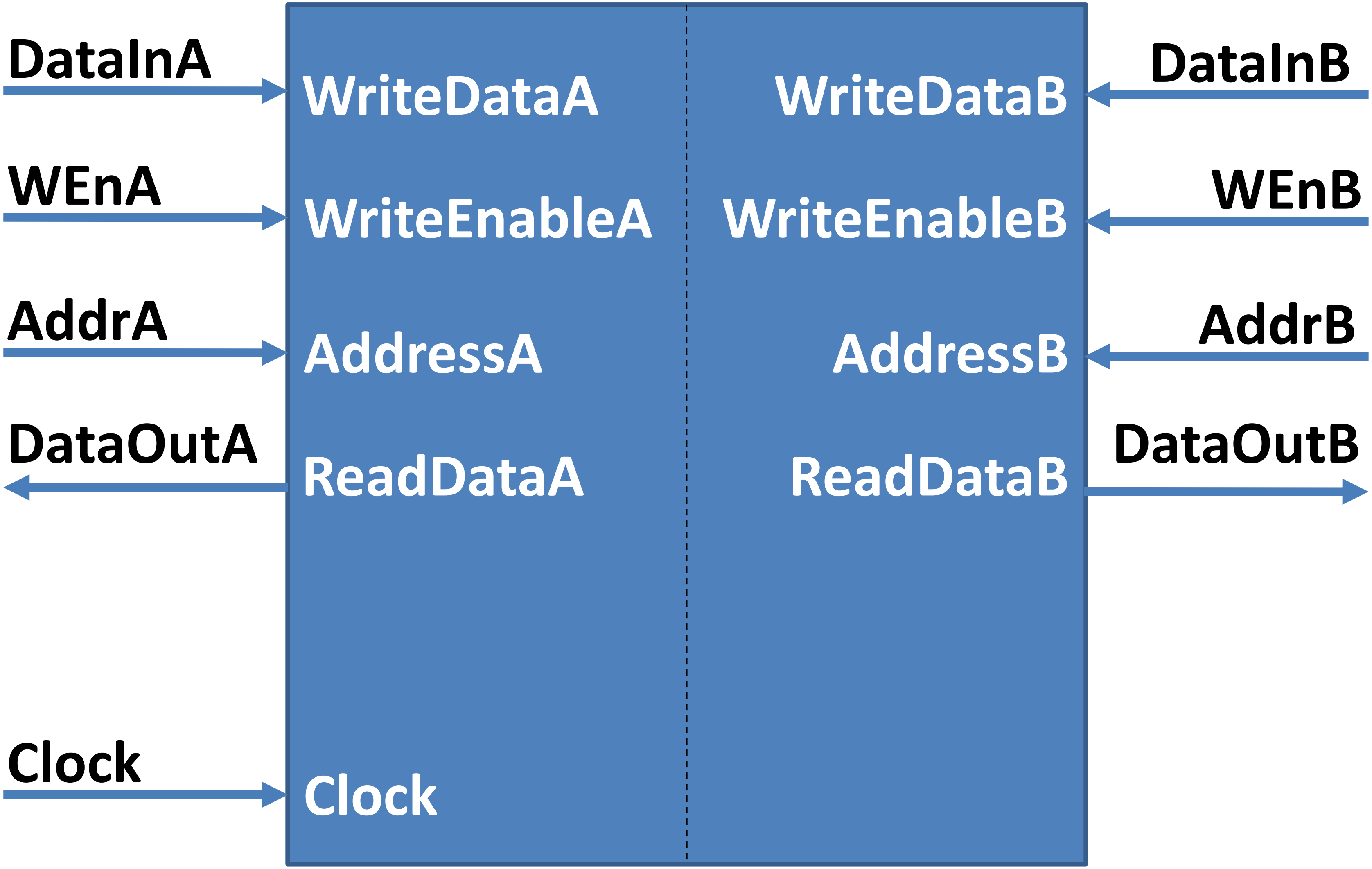
# Dual Port RAM



# Dual Port RAM



# Dual Port RAM



# VHDL for Dual Port RAM

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

# VHDL for Dual Port RAM

```
entity DualPortRAM is
	generic(
		Dwidth : integer;
		Awidth : integer
	);
	port(
		Clock			: in   std_logic;

		WriteDataA		: in   std_logic_vector(Dwidth - 1 downto 0);
		ReadDataA		: out  std_logic_vector(Dwidth - 1 downto 0);
		AddressA		: in   std_logic_vector(Awidth - 1 downto 0);
		WriteEnableA	: in   std_logic;

		WriteDataB		: in   std_logic_vector(Dwidth - 1 downto 0);
		ReadDataB		: out  std_logic_vector(Dwidth - 1 downto 0);
		AddressB		: in   std_logic_vector(Awidth - 1 downto 0);
		WriteEnableB	: in   std_logic
	);
end DualPortRAM ;
```

These ports implement the 1st “port” which I have called A.

These ports implement the 2nd “port” which I have called B.

# VHDL for Single Port RAM

```
architecture rtl of DualPortRAM is
        -- Build a 2D array type for the RAM
        type memory_t is array(0 to 2**Awidth -1 )   of   std_logic_vector((Dwidth-1) downto 0);
        -- Declare the RAM signal
         signal MyRam : memory_t ;
begin
```

# VHDL for Single Port RAM

```
architecture rtl of DualPortRAM is
        -- Build a 2D array type for the RAM
        type memory_t is array(0 to 2**Awidth -1 )   of   std_logic_vector((Dwidth-1) downto 0);
        -- Declare the RAM signal
         signal MyRam : memory_t ;
begin
        RAMProcess : process (Clock) begin
           if rising_edge(Clock) then
```

# VHDL for Single Port RAM

```
architecture rtl of DualPortRAM is
        -- Build a 2D array type for the RAM
        type memory_t is array(0 to 2**Awidth -1 )   of   std_logic_vector((Dwidth-1) downto 0);
        -- Declare the RAM signal
         signal MyRam : memory_t ;
begin
        RAMProcess : process (Clock) begin
           if rising_edge(Clock) then

                if WriteEnableA = '1'  then
                  MyRam (to_integer(unsigned(AddressA))) <= WriteDataA;
                end if;
                ReadDataA <= MyRam(to_integer(unsigned(AddressA)));
```

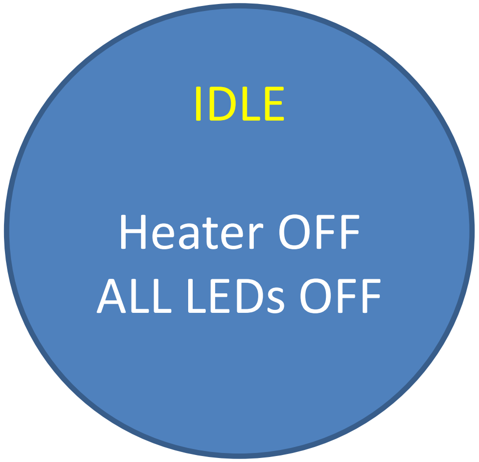
# VHDL for Single Port RAM

```
architecture rtl of DualPortRAM is
        -- Build a 2D array type for the RAM
        type memory_t is array(0 to 2**Awidth -1 )   of   std_logic_vector((Dwidth-1) downto 0);
        -- Declare the RAM signal
         signal MyRam : memory_t ;
begin
        RAMProcess : process (Clock) begin
           if rising_edge(Clock) then
                if WriteEnableA = '1'  then
                  MyRam (to_integer(unsigned(AddressA))) <= WriteDataA;
                end if;
                ReadDataA <= MyRam(to_integer(unsigned(AddressA)));

                if WriteEnableB = '1'  then
                  MyRam (to_integer(unsigned(AddressB))) <= WriteDataB;
                end if;
                ReadDataB <= MyRam(to_integer(unsigned(AddressB)));
```

# VHDL for Single Port RAM

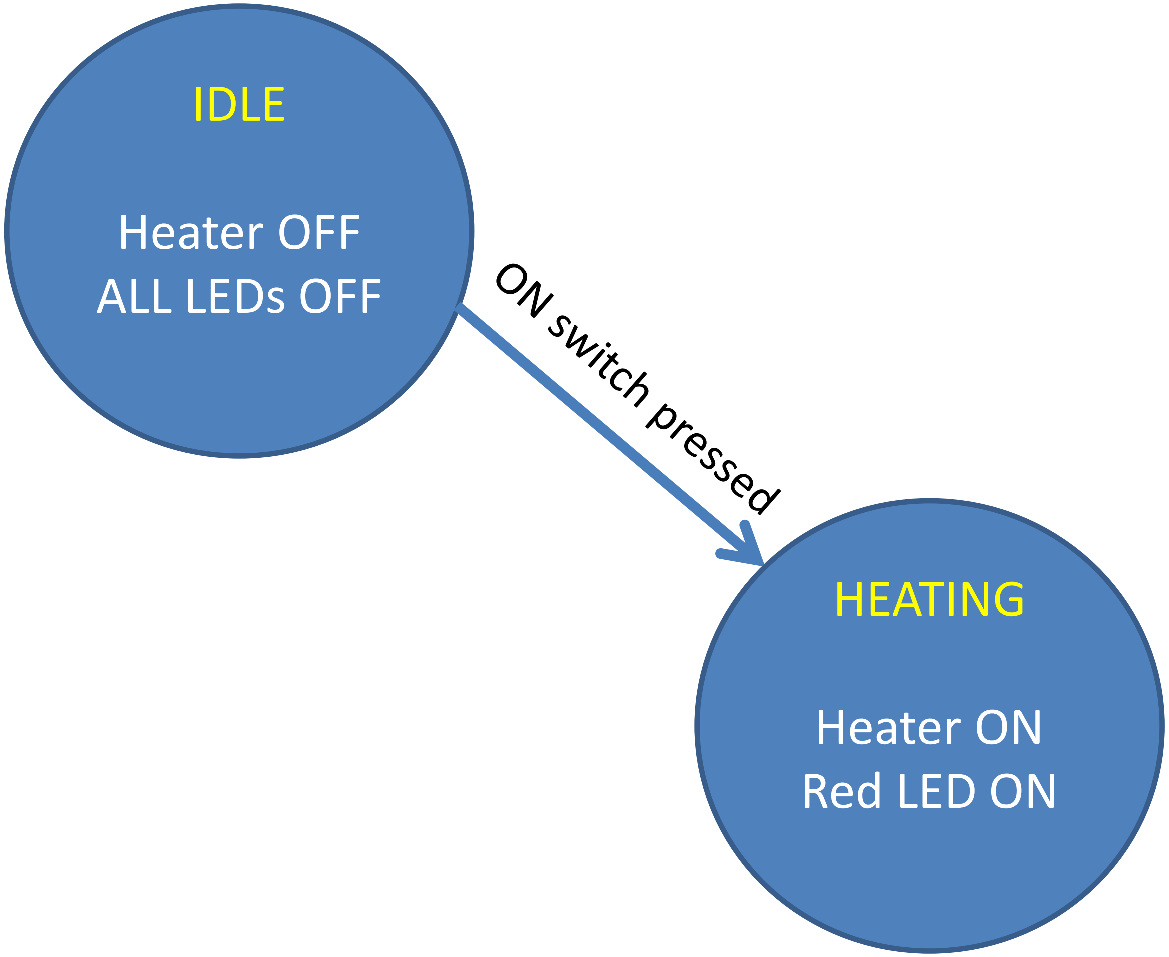
```
architecture rtl of DualPortRAM is
        -- Build a 2D array type for the RAM
        type memory_t is array(0 to 2**Awidth -1 )   of   std_logic_vector((Dwidth-1) downto 0);
        -- Declare the RAM signal
         signal MyRam : memory_t ;
begin
        RAMProcess : process (Clock) begin
           if rising_edge(Clock) then

                if WriteEnableA = '1'  then
                  MyRam (to_integer(unsigned(AddressA))) <= WriteDataA;
                end if;
                ReadDataA <= MyRam(to_integer(unsigned(AddressA)));

                if WriteEnableB = '1'  then
                  MyRam (to_integer(unsigned(AddressB))) <= WriteDataB;
                end if;
                ReadDataB <= MyRam(to_integer(unsigned(AddressB)));


           end if;
         end process;
end rtl;
```

# Finite State Machines



IDLE

Heater OFF
ALL LEDs OFF



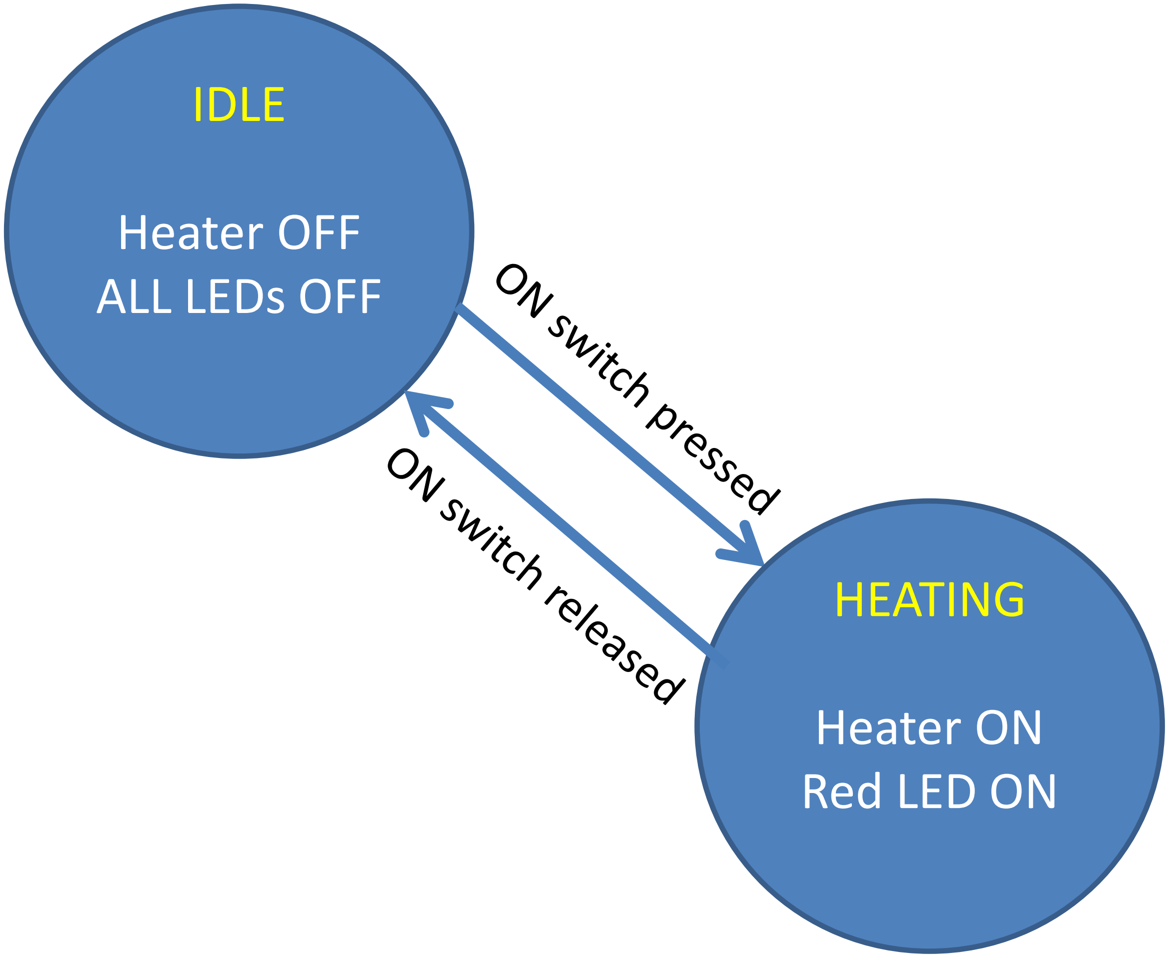
IDLE

Heater OFF
ALL LEDs OFF

ON switch pressed

HEATING

Heater ON
Red LED ON



IDLE

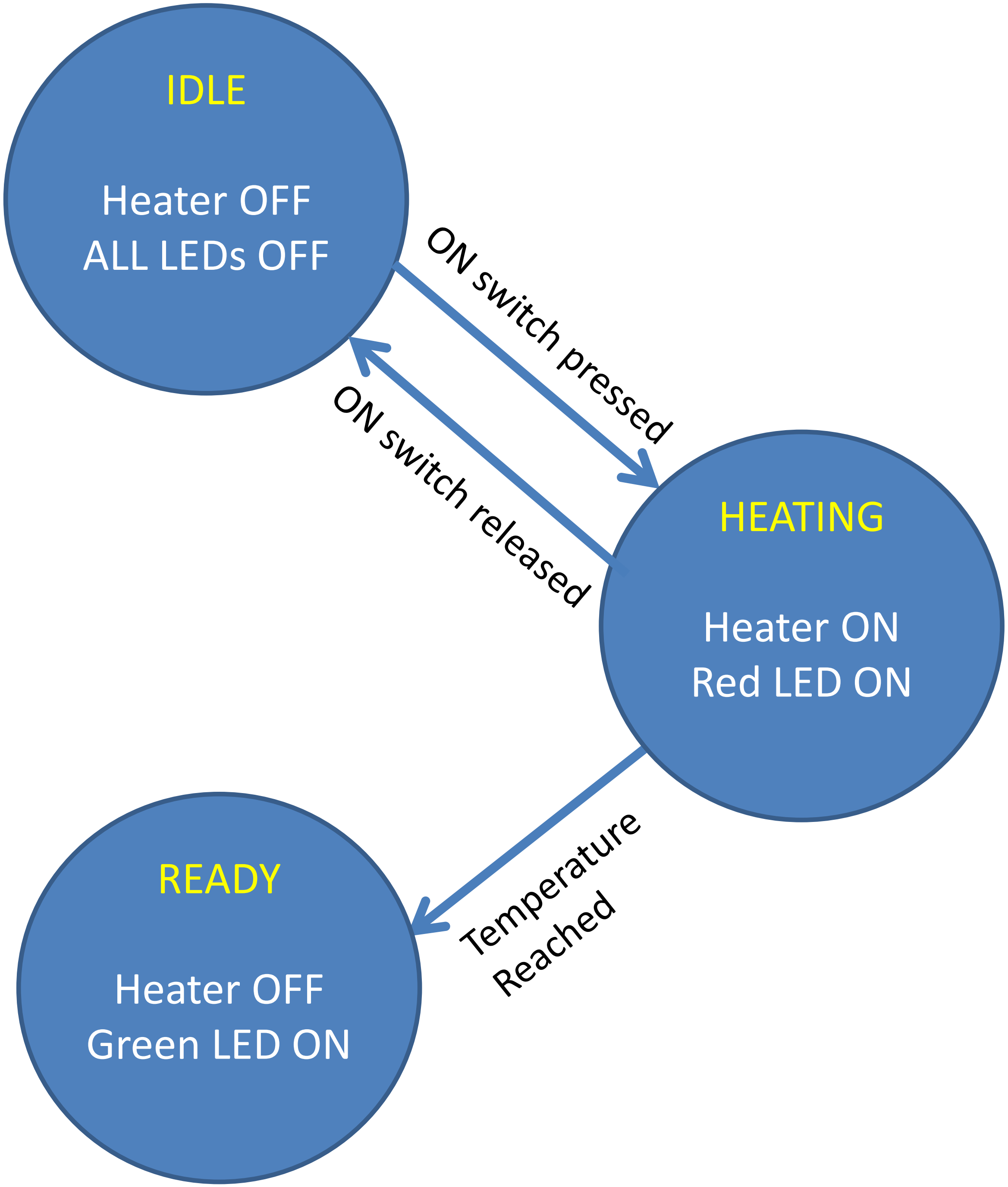
Heater OFF
ALL LEDs OFF

ON switch pressed

ON switch released

HEATING

Heater ON
Red LED ON



IDLE

Heater OFF
ALL LEDs OFF

ON switch pressed

ON switch released

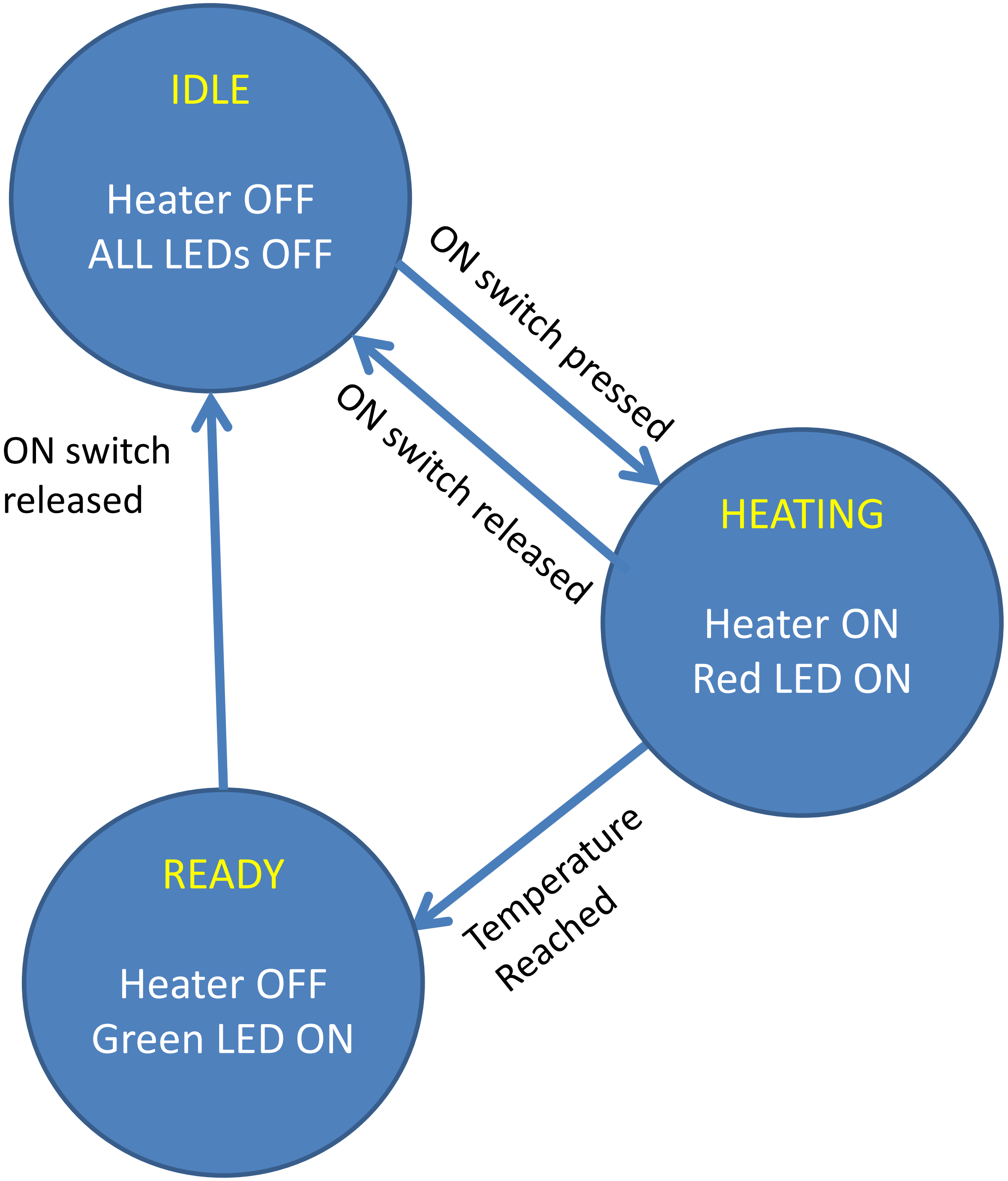
HEATING

Heater ON
Red LED ON

Temperature
Reached

READY

Heater OFF
Green LED ON



IDLE

Heater OFF
ALL LEDs OFF

ON switch pressed

ON switch released

HEATING

Heater ON
Red LED ON

Temperature
Reached

READY

Heater OFF
Green LED ON

ON switch
released

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

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

entity  HeaterFSM   is
      port(
            Clk             : in std_logic;
            Rst             : in std_logic;
            Sw              : in std_logic;
            Temp            : in std_logic;
            Heater          : out std_logic;
            RedLED         : out  std_logic;
            GreenLED    : out  std_logic
      );
end entity;
```

```
architecture rtl of HeaterFSM  is
architecture rtl of HeaterFSM  is
        type FSMStateType is (IDLE, HEATING, READY);
```

```
architecture rtl of HeaterFSM  is
    type FSMStateType is (IDLE, HEATING, READY);
    signal State : FSMStateType;
begin
```

```
architecture rtl of HeaterFSM  is
    type FSMStateType is (IDLE, HEATING, READY);
    signal State : FSMStateType;
begin
```

```
FSMProcess: process(rst , clk)
begin
        if rst = ‘1’ then
                    GreenLED <= ‘0’;
                    RedLED <= ‘0’;
                    Heater <= ‘0’;
                    State <= IDLE;
        elsif rising_edge(clk) then
```

```
FSMProcess: process(rst , clk)
begin
        if rst = ‘1’ then
                        GreenLED <= ‘0’;
                        RedLED <= ‘0’;
                        Heater <= ‘0’;
                        State <= IDLE;
        elsif rising_edge(clk) then
                        case State is
```

```
FSMProcess: process(rst , clk)
begin
        if rst = ‘1’ then
                     GreenLED <= ‘0’;
                     RedLED <= ‘0’;
                     Heater <= ‘0’;
                     State <= IDLE;
        elsif rising_edge(clk) then
                     case State is
                                   when IDLE =>
                                       GreenLED <= ‘0’;
                                       RedLED <= ‘0’;
                                       Heater <= ‘0’;
                                       if Sw = ‘1’ then
                                           State <= HEATING;
                                       end if;
```

```
FSMProcess: process(rst , clk)
begin
        if rst = ‘1’ then
                    GreenLED <= ‘0’;
                    RedLED <= ‘0’;
                    Heater <= ‘0’;
                    State <= IDLE;
        elsif rising_edge(clk) then
                    case State is
                                when IDLE =>
                                    GreenLED <= ‘0’;
                                    RedLED <= ‘0’;
                                    Heater <= ‘0’;
                                    if Sw = ‘1’ then
                                        State <= HEATING;
                                    end if;
                                when HEATING =>
                                    RedLED <= ‘1’;
                                    Heater <= ‘1’;

                                    if Sw = ‘0’ then
                                        State <= IDLE ;
                                    elsif Temp = ‘1’ then
                                        State <= READY ;
                                    end if;
```

```
FSMProcess: process(rst , clk)
begin
        if rst = ‘1’ then
                    GreenLED <= ‘0’;
                    RedLED <= ‘0’;
                    Heater <= ‘0’;
                    State <= IDLE;
        elsif rising_edge(clk) then
                    case State is
                            when IDLE =>
                                GreenLED <= ‘0’;
                                RedLED <= ‘0’;
                                Heater <= ‘0’;
                                if Sw = ‘1’ then
                                    State <= HEATING;
                                end if;
                            when HEATING =>
                                RedLED <= ‘1’;
                                Heater <= ‘1’;

                                if Sw = ‘0’ then
                                    State <= IDLE ;
                                elsif Temp = ‘1’ then
                                    State <= READY ;
                                end if;

                            when READY =>
                                 RedLED <= ‘0’;
                                 GreenLED <= ‘1’;
                                 Heater <= ‘0’;

                                 if Sw = ‘0’ then
                                    State <= IDLE ;
                                 end if;
```

```
FSMProcess: process(rst , clk)
begin
        if rst = ‘1’ then
                      GreenLED <= ‘0’;
                      RedLED <= ‘0’;
                      Heater <= ‘0’;
                      State <= IDLE;
        elsif rising_edge(clk) then
                      case State is
                                    when IDLE =>
                                         GreenLED <= ‘0’;
                                         RedLED <= ‘0’;
                                         Heater <= ‘0’;
                                         if Sw = ‘1’ then
                                             State <= HEATING;
                                         end if;
                                    when HEATING =>
                                         RedLED <= ‘1’;
                                         Heater <= ‘1’;

                                         if Sw = ‘0’ then
                                             State <= IDLE ;
                                         elsif Temp = ‘1’ then
                                             State <= READY ;
                                         end if;

                                    when READY =>
                                          RedLED <= ‘0’;
                                          GreenLED <= ‘1’;
                                          Heater <= ‘0’;

                                          if Sw = ‘0’ then
                                             State <= IDLE ;
                                          end if;

                                    when others =>  State <= IDLE ;
                      end case;
```

```
    FSMProcess: process(rst , clk)
    begin
        if rst = ‘1’ then
                GreenLED <= ‘0’;
                RedLED <= ‘0’;
                Heater <= ‘0’;
                State <= IDLE;
        elsif rising_edge(clk) then
                case State is
                        when IDLE =>
                            GreenLED <= ‘0’;
                            RedLED <= ‘0’;
                            Heater <= ‘0’;
                            if Sw = ‘1’ then
                                State <= HEATING;
                            end if;
                        when HEATING =>
                            RedLED <= ‘1’;
                            Heater <= ‘1’;

                            if Sw = ‘0’ then
                                State <= IDLE ;
                            elsif Temp = ‘1’ then
                                State <= READY ;
                            end if;

                        when READY =>
                             RedLED <= ‘0’;
                             GreenLED <= ‘1’;
                             Heater <= ‘0’;

                             if Sw = ‘0’ then
                                State <= IDLE ;
                             end if;

                        when others =>  State <= IDLE ;
                end case;
        end if;
     End process;
End rtl;
```

# Recommendations :

# Recommendations :

- Use a Registered Process with a Reset.

## Recommendations :

- Use a Synchronous Process with a Reset.
- Create an enumerated data type to define states.

# Recommendations :

- Use a Synchronous Process with a Reset.
- Create an enumerated data type to define states.
  - Use meaningful names for the states.

**Recommendations :**

- Use a Synchronous Process with a Reset.
- Create an enumerated data type to define states.
  - Use meaningful names for the states.
- Use a CASE statement to :
  - Implement the next state encoder.
  - Implement the output decoder encoder.

# Recommendations :

- Use a Synchronous Process with a Reset.

- Create an enumerated data type to define states.
  - Use meaningful names for the states.

- Use a CASE statement to :
  - Implement the next state encoder.
  - Implement the output decoder encoder.

- Best to have registered outputs.

## Recommendations :

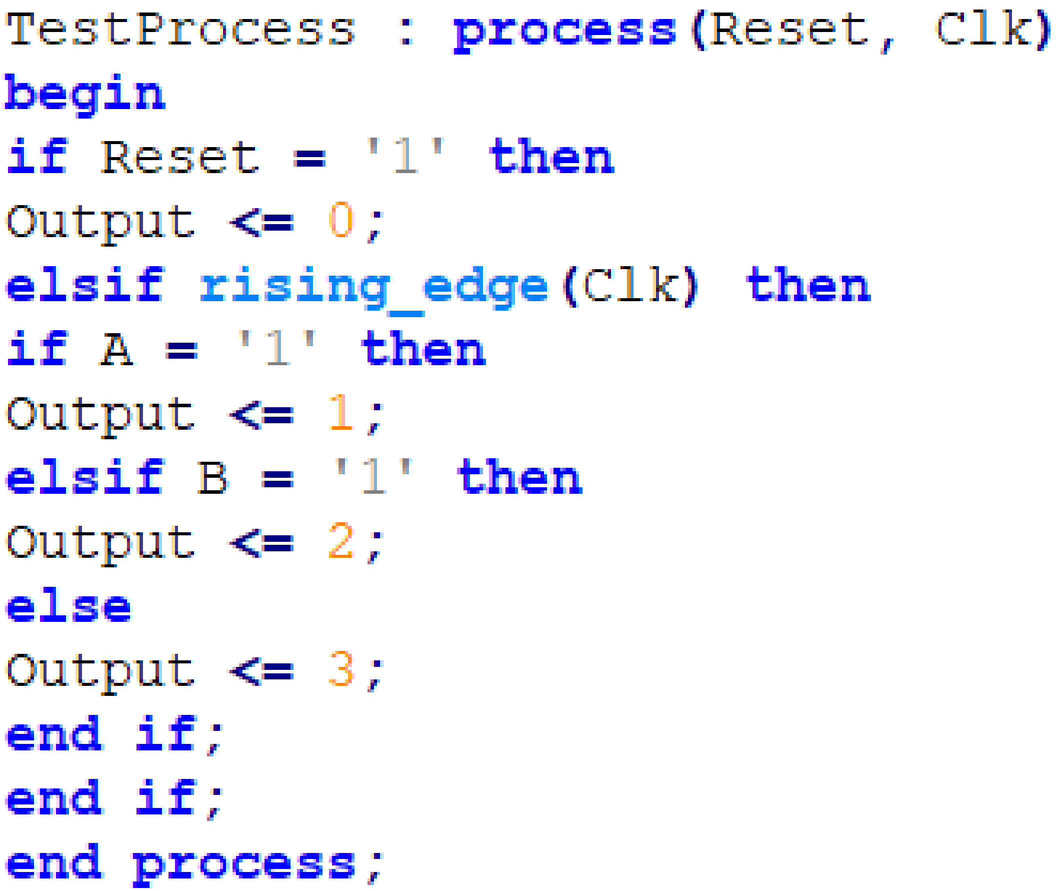
- Use a Synchronous Process with a Reset.
- Create an enumerated data type to define states.
  - Use meaningful names for the states.
- Use a CASE statement to :
  - Implement the next state encoder.
  - Implement the output decoder encoder.
- Best to have registered outputs.
- Use “When Others” to return to a default state (in the event of entering an illegal state)

# Good Design Practice 1

# Code Readability

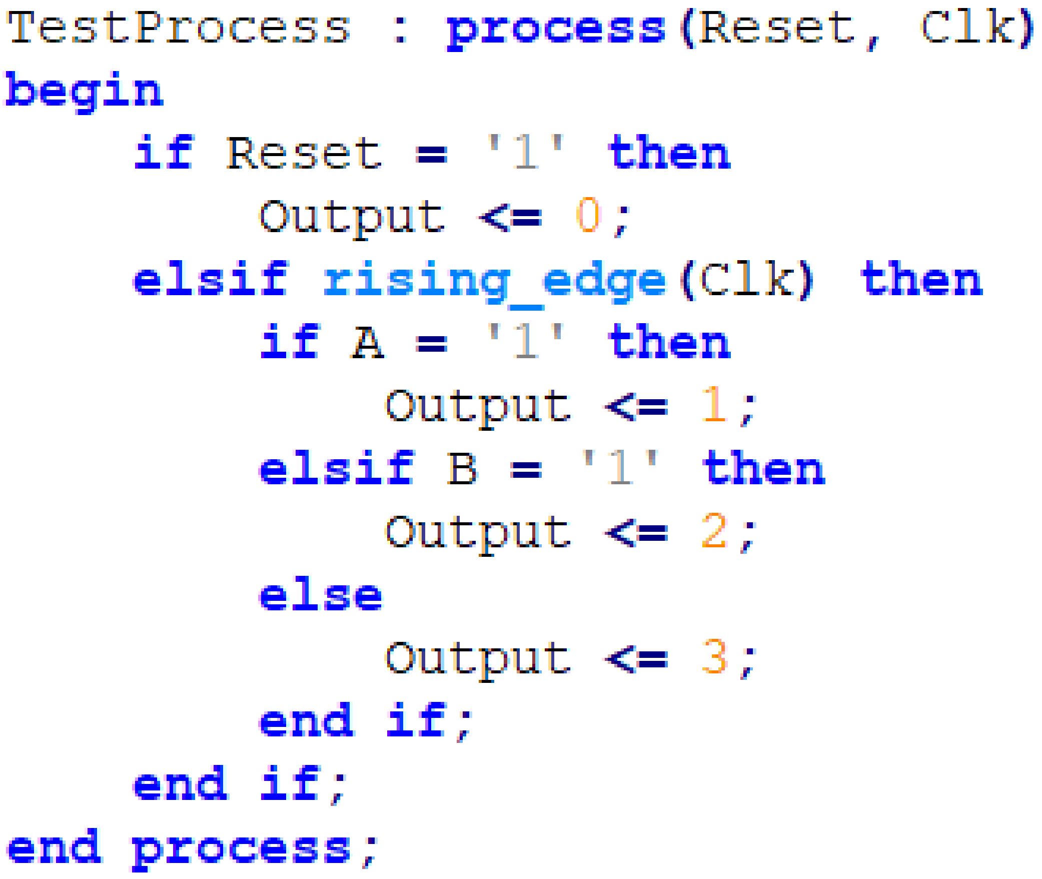
- Use meaningful names for signals, processes, generics, constants and entities.
- Make use of indentation to improve readability.
- Indent you code using :
  - Using the tab characters.
  - Or using spaces characters (typically 4 spaces)
- Text editors (such as notepad++ and Intel Quartus) can be configured to use either tabs or spaces.
- Disadvantage with using tabs :
  - Indentation may look wrong when files are opened with other editors.

# No Indentation



```
TestProcess : process(Reset, Clk)
begin
if Reset = '1' then
Output <= 0;
elsif rising_edge(Clk) then
if A = '1' then
Output <= 1;
elsif B = '1' then
Output <= 2;
else
Output <= 3;
end if;
end if;
end process;
```

# With Indentation



```
TestProcess : process(Reset, Clk)
begin
    if Reset = '1' then
        Output <= 0;
    elsif rising_edge(Clk) then
        if A = '1' then
            Output <= 1;
        elsif B = '1' then
            Output <= 2;
        else
            Output <= 3;
        end if;
    end if;
end process;
```

# Avoid Creating Latches

- Watch out for accidental latch creation inside combinational processes.
- Only a problem when there are conditional statements (if then / case).
- Make sure the outputs are defined in all possible cases.

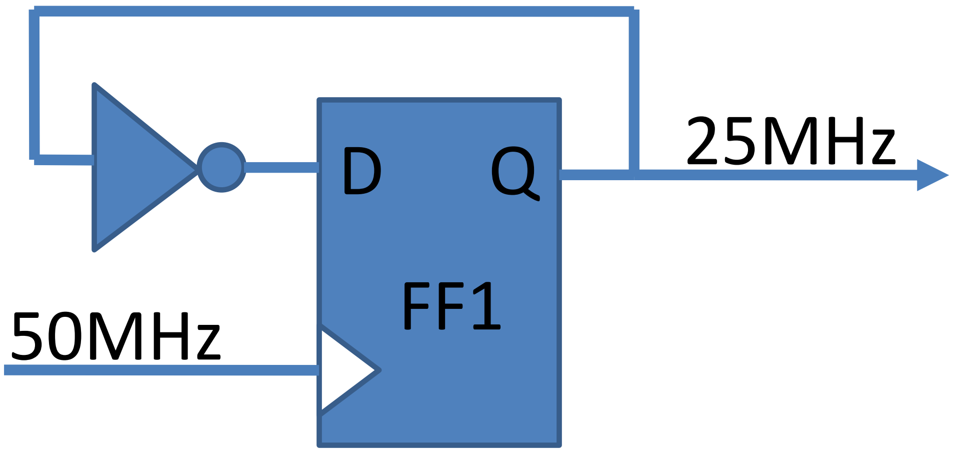
# Do Not Generate Clocks Using Logic

# Do Not Generate Clocks Using Logic

- Suppose our FPGA has a 50MHz clock input and we want to generate a 25MHz internal clock.

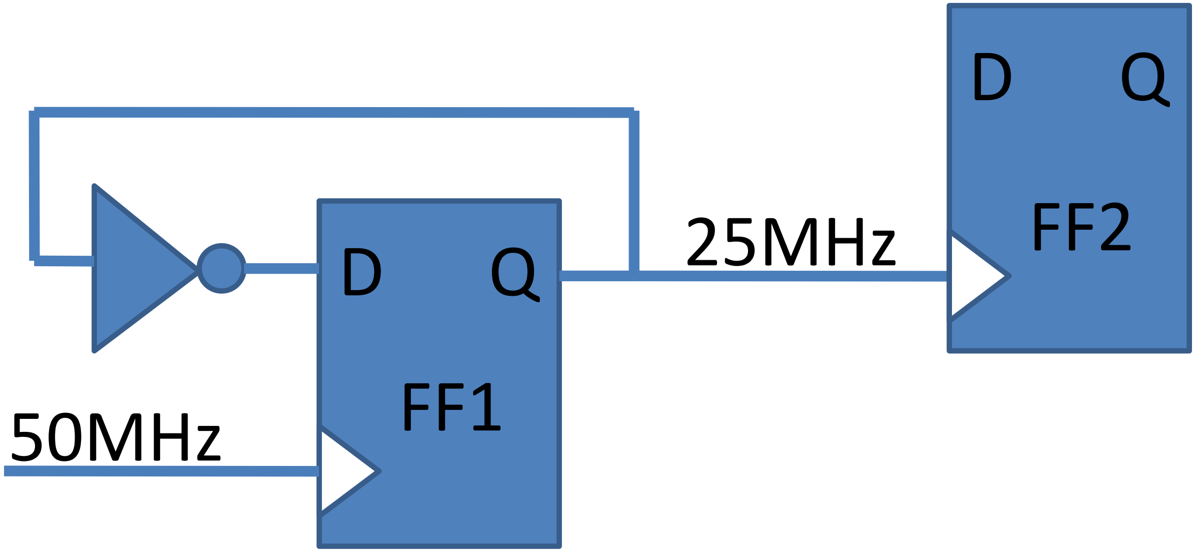
# Do Not Generate Clocks Using Logic

- Suppose our FPGA has a 50MHz clock input and we want to generate a 25MHz internal clock.



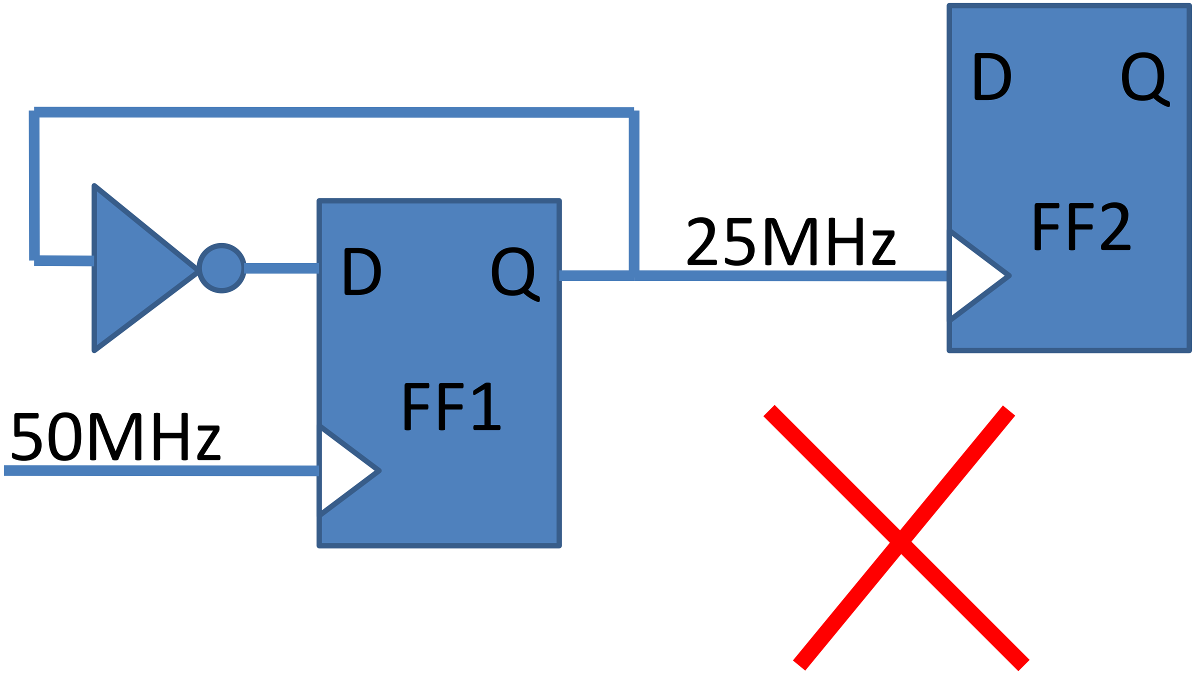
# Do Not Generate Clocks Using Logic

- Suppose our FPGA has a 50MHz clock input and we want to generate a 25MHz internal clock.



# Do Not Generate Clocks Using Logic

- Suppose our FPGA has a 50MHz clock input and we want to generate a 25MHz internal clock.

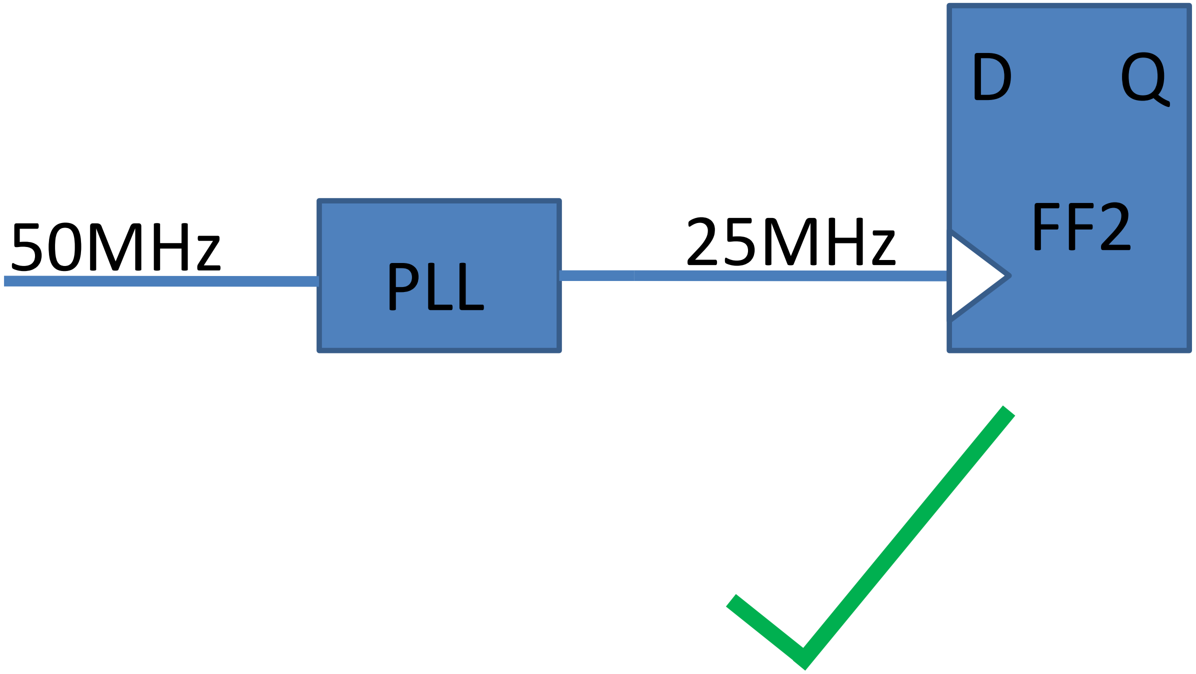


# Do Not Generate Clocks Using Logic

- Using a clocks generated by FPGA logic results in slower designs.
- Because FPGA will be forced to use non-global routing resources to some degree.
- These paths have much higher skew resulting in sub-optimal timing performance.

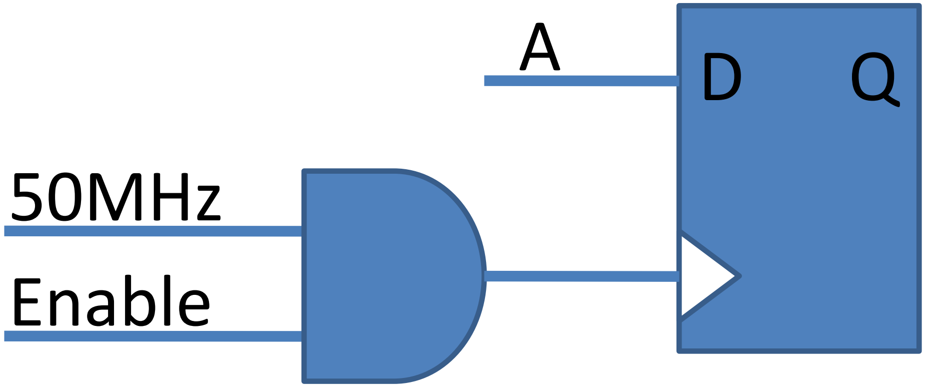
# Better Way To Generate Clocks

- Suppose our FPGA has a 50MHz clock input and we want to generate a 25MHz internal clock.



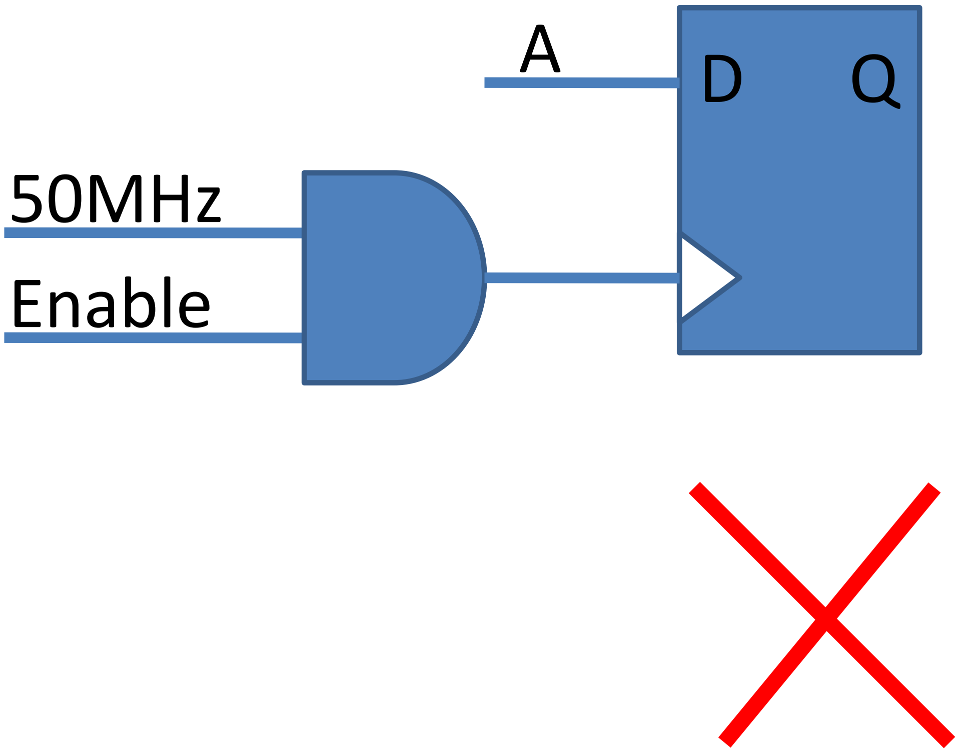
# Do Not Use Clock Gating

- For the same reasons, do not use clock gating.



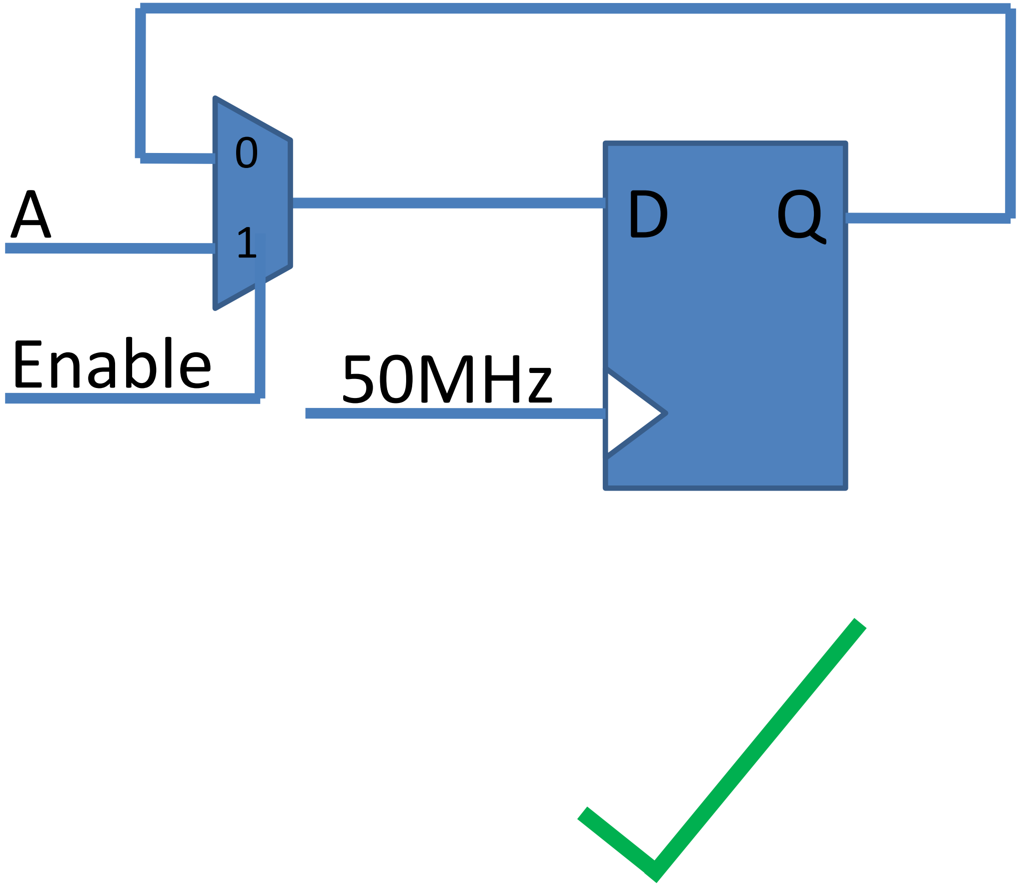
# Do Not Use Clock Gating

- For the same reasons, do not use clock gating.



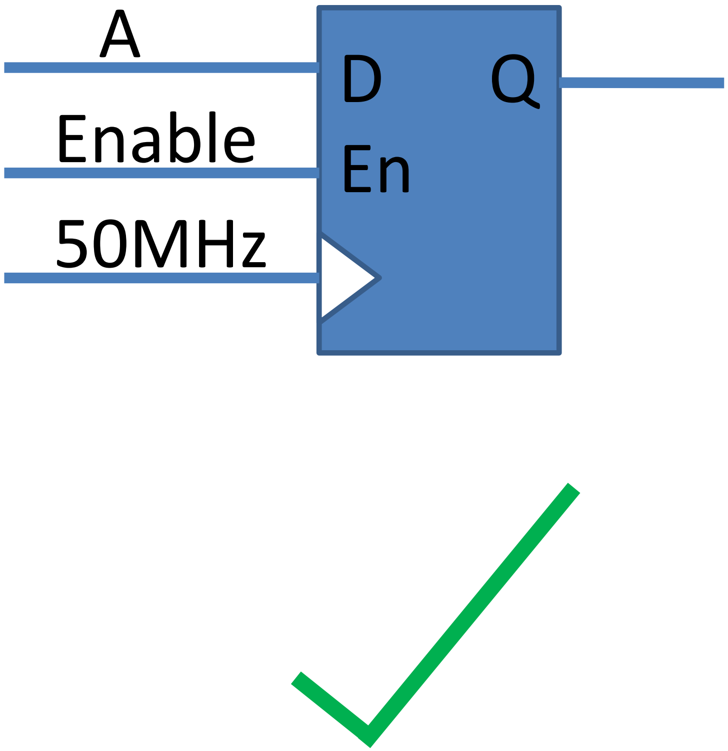
# Use Data Gating Instead

- For the same reasons, do not use clock gating.

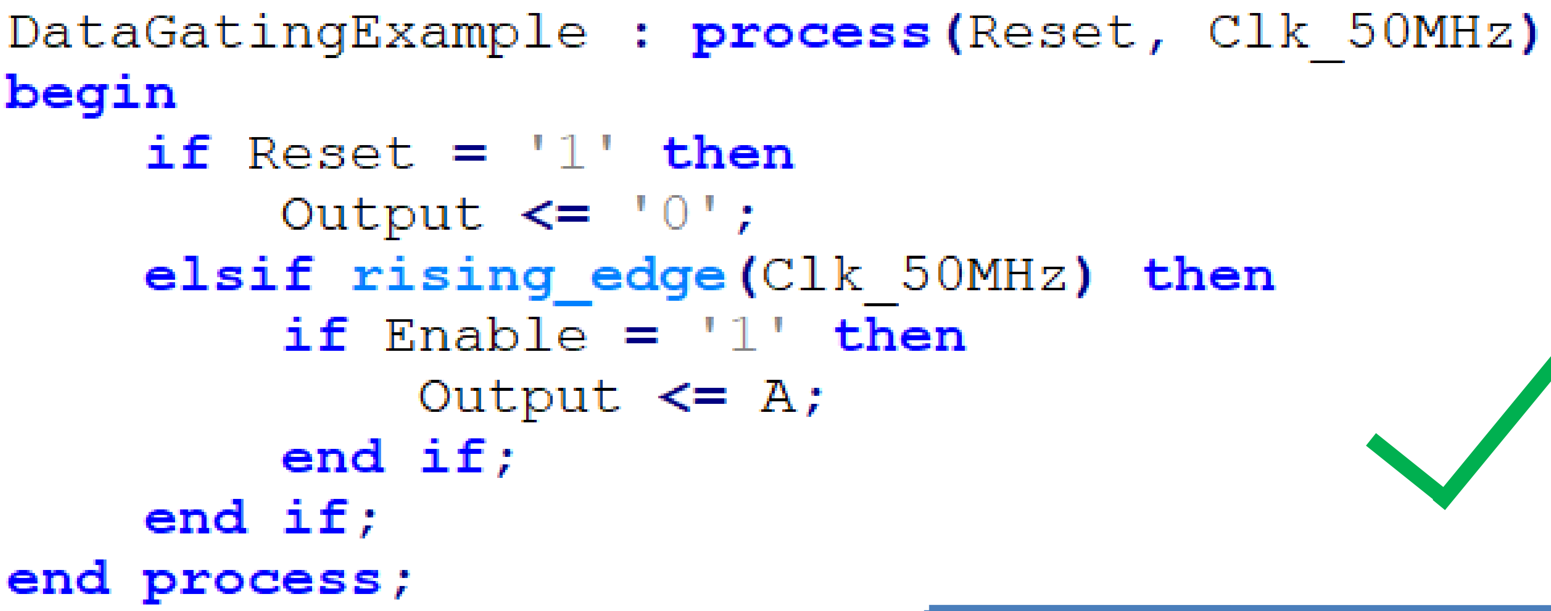


# Use Data Gating Instead

- For the same reasons, do not use clock gating.

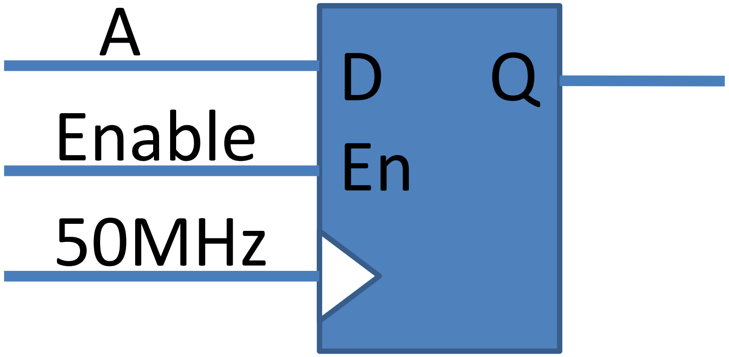


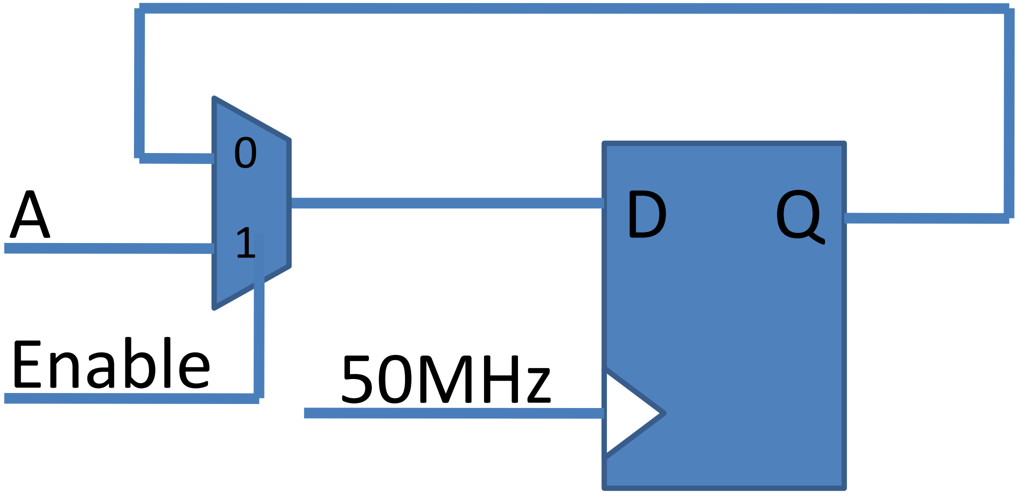
# Use Data Gating Instead



```
DataGatingExample : process(Reset, Clk_50MHz)
begin
    if Reset = '1' then
        Output <= '0';
    elsif rising_edge(Clk_50MHz) then
        if Enable = '1' then
            Output <= A;
        end if;
    end if;
end process;
```

✓

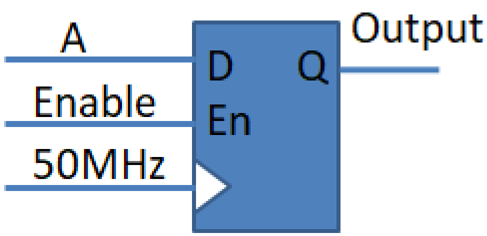


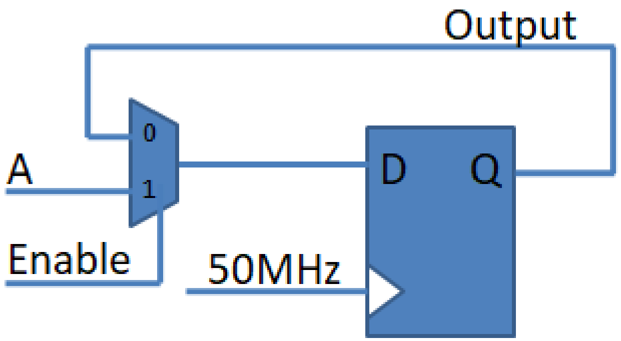


# Use Data Gating Instead

```vhdl
DataGatingExample : process(Reset, Clk_50MHz)
begin
    if Reset = '1' then
        Output <= '0';
    elsif rising_edge(Clk_50MHz) then
        if Enable = '1' then
            Output <= A;
        end if;
    end if;
end process;
```

✓



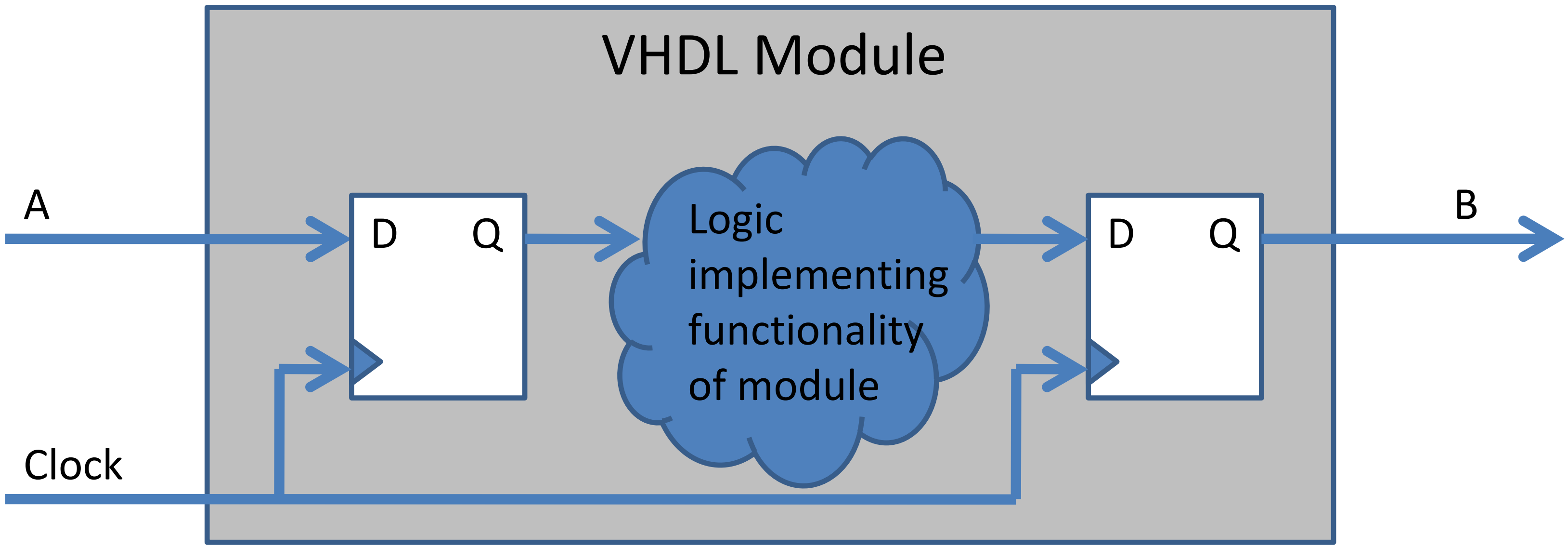


# Make Generous Use Of Registers

- FPGAs have plenty of registers.
- Using registers generously improves circuit timing performance.
- Register all inputs and outputs from ports.

# Make Generous Use Of Registers

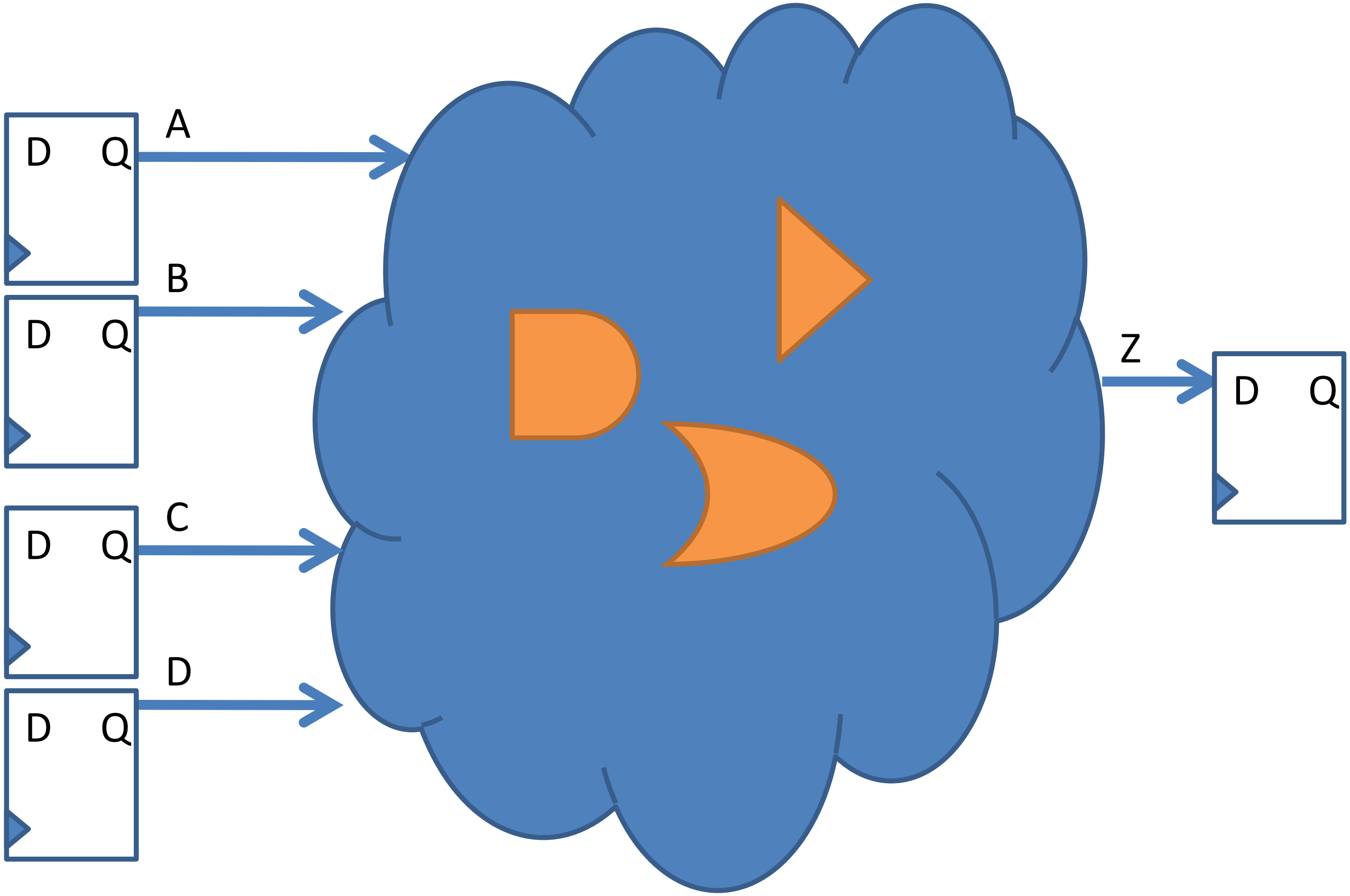
- Register **all** inputs and outputs ports of a VHDL module.



# Pipelining

- Large combinational functions have big delays leading to a timing bottleneck.
- This leads to a slower FPGA design.
- Pipelining can be very effective in mitigating these delays and boost timing performance.
- Pipelining uses extra registers to break up a large combinational function.

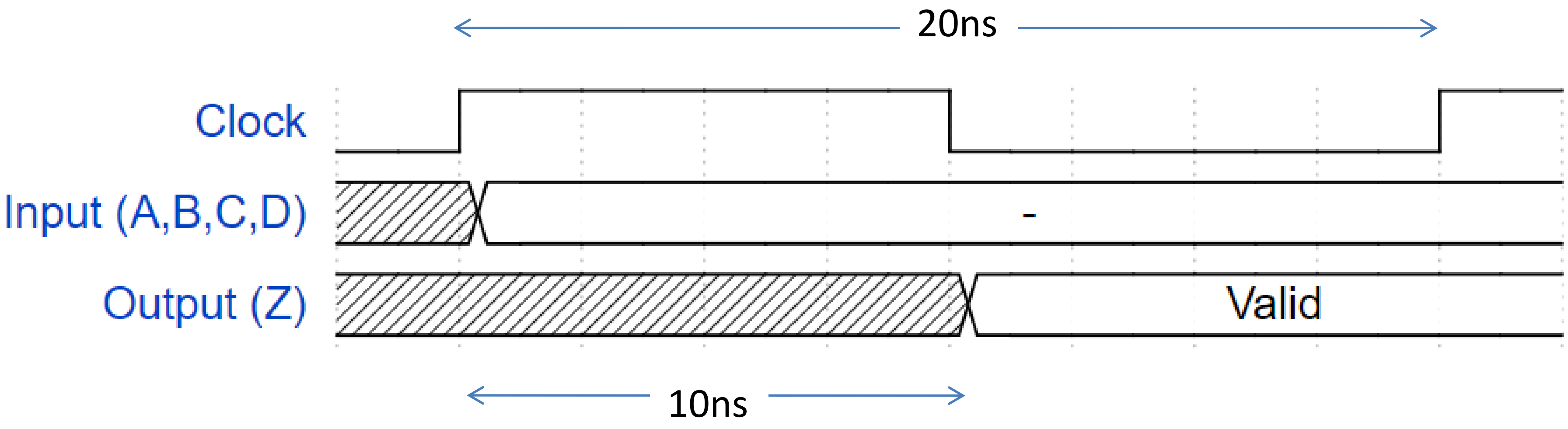
# Pipelining - Application



Logic delay = 10 ns

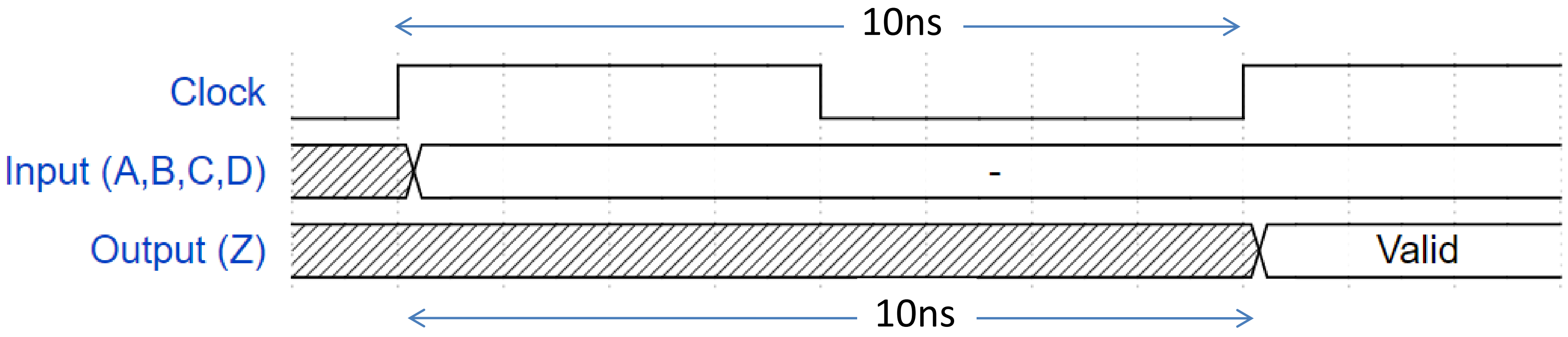
# Pipelining - Application

Clock = 50 MHz



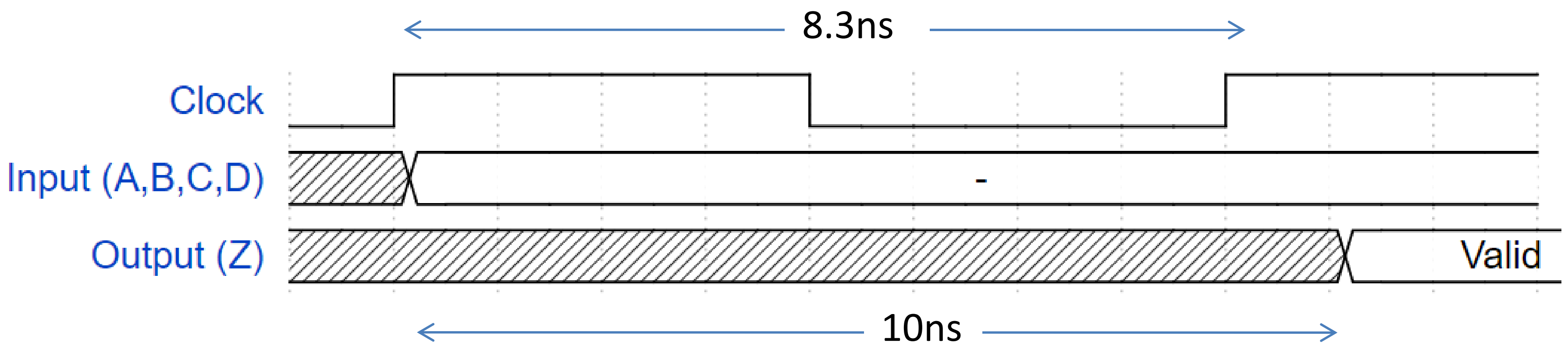
# Pipelining - Application

Clock = 100 MHz

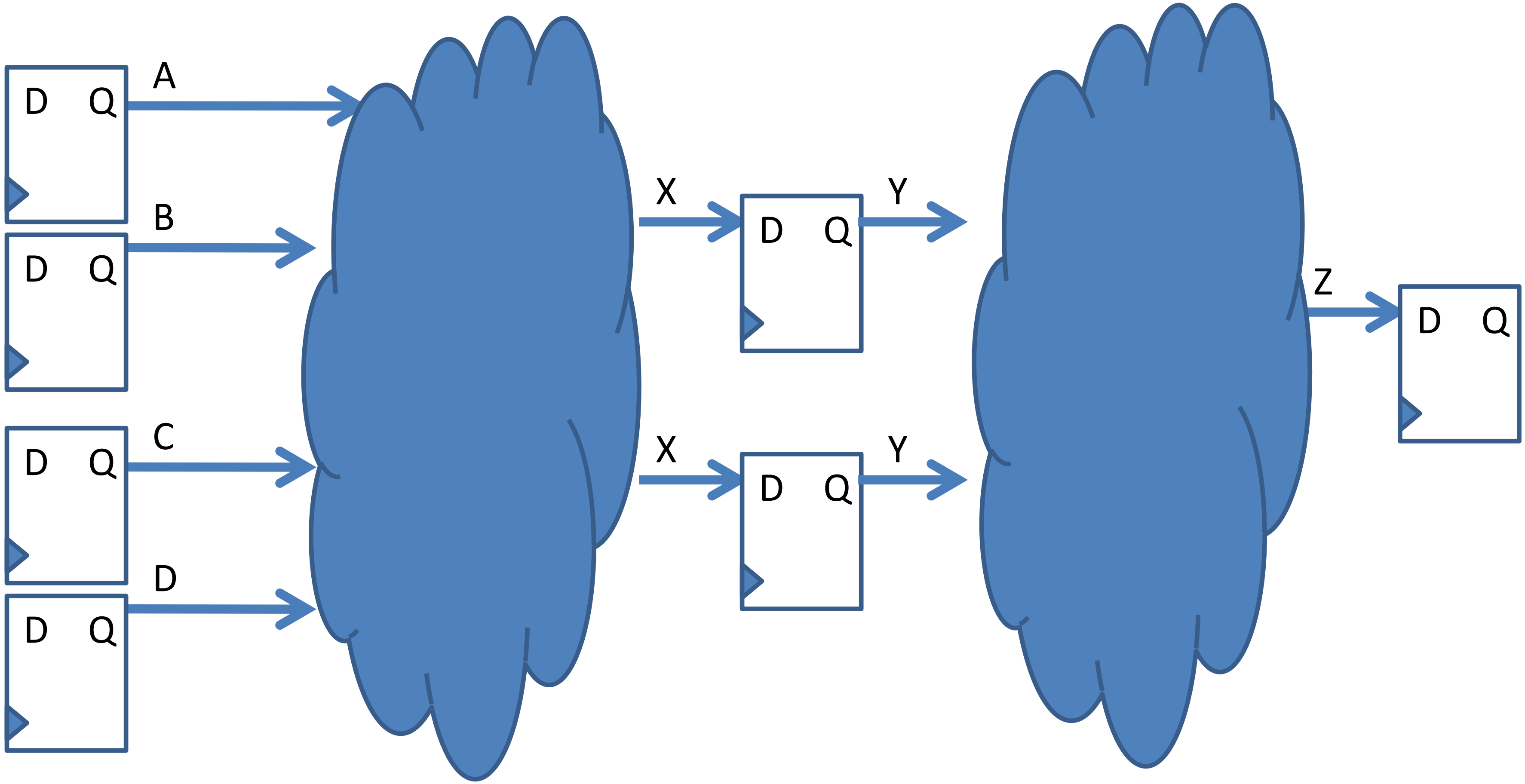


# Pipelining - Application

Clock = 120 MHz



# Pipelining - Application



Logic delay = 6 ns

Logic delay = 4ns

# Synchronous Systems

- Always design synchronous systems:
    - Having **registered** inputs and outputs.

# Synchronous Systems

- Always design synchronous systems:
    - Having **registered** inputs and outputs.
- Why?
    - To avoid all sorts of timing glitches that are possible on a purely combinational system.

# Topics

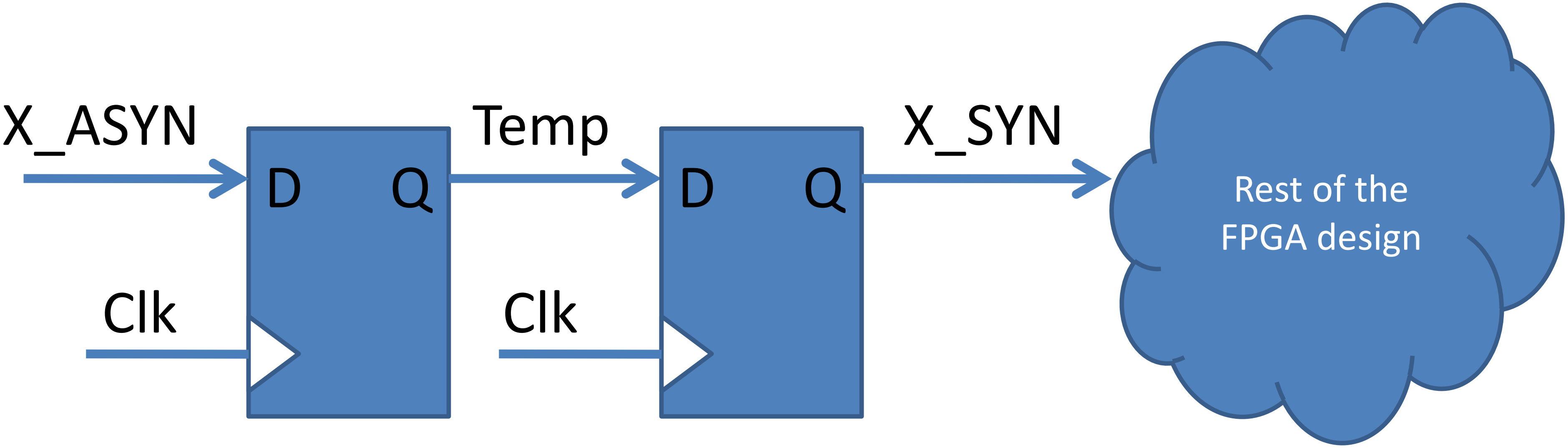
- Use meaningful names.
- Importance of indentation.
- Avoid creating latches.
- Avoid generate clocks using logic.
- Clock Gating vs Data Gating.
- Benefits of a register rich design.
- Benefits of a synchronous design.

# Good Design Practice 2

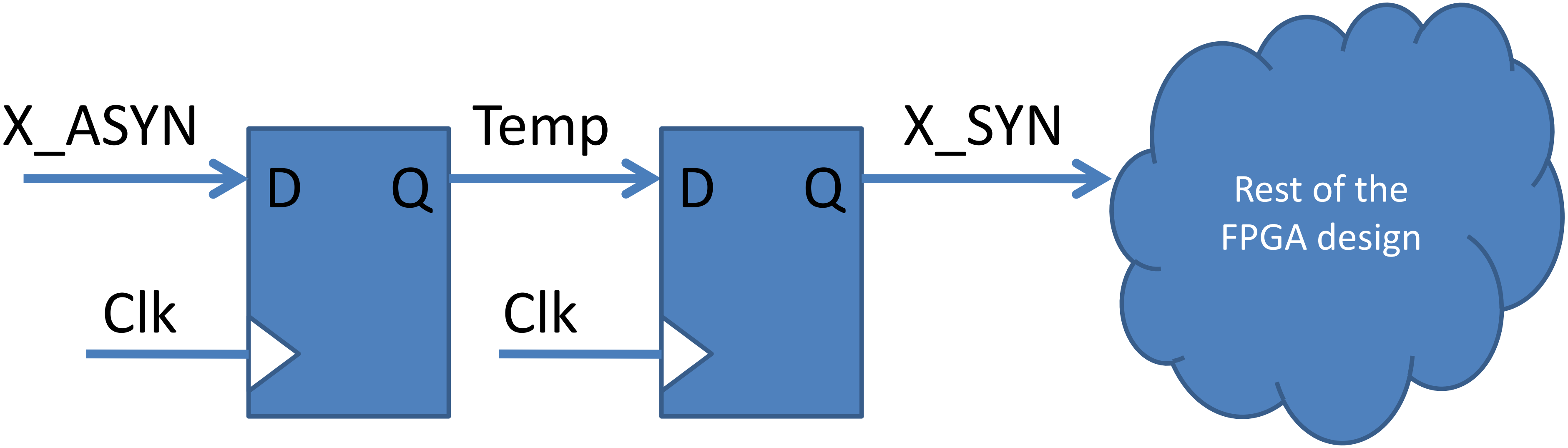
# Asynchronous Inputs

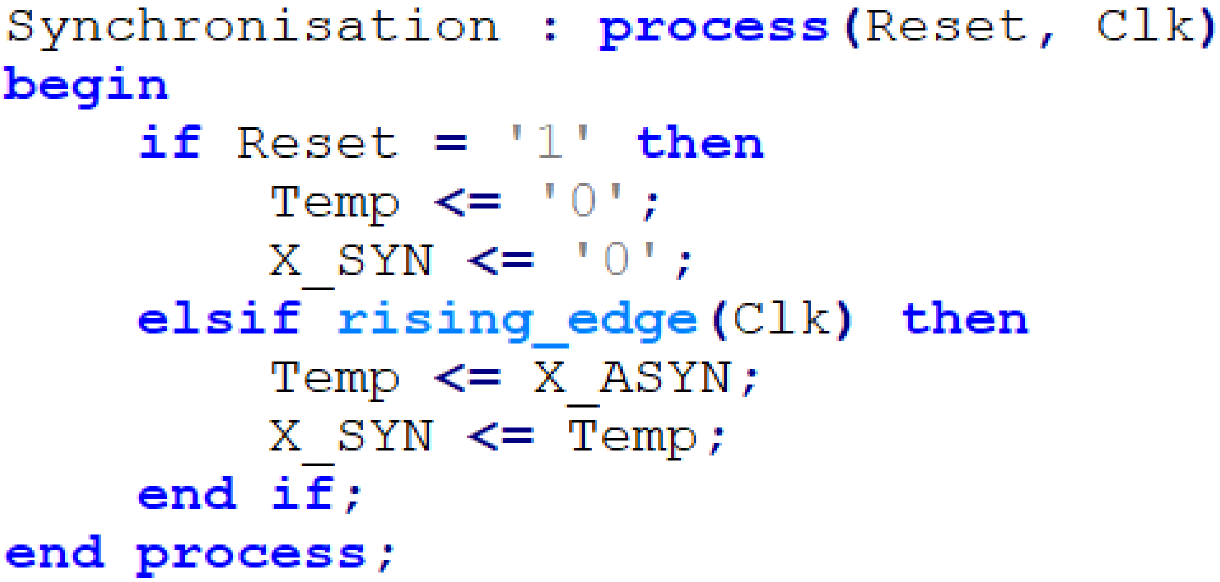
- Asynchronous inputs can change state at any point in time:
  - Push button input
  - RS232 Data
- These signals are not driven from a clock.
- We must make sure to synchronise **all** asynchronous inputs before using them.

# 2 Stage Synchronisation



# 2 Stage Synchronisation





```
Synchronisation : process(Reset, Clk)
begin
    if Reset = '1' then
        Temp <= '0';
        X_SYN <= '0';
    elsif rising_edge(Clk) then
        Temp <= X_ASYN;
        X_SYN <= Temp;
    end if;
end process;
```

# Crossing Clock Domains

- Similar to asynchronous inputs (danger of meta-stability)
- Usually, a signal or a bus going from one clock domain to another.
- The clock domain generating the signal can be running faster or slower than the receiving clock domain.

# Crossing Clock Domains

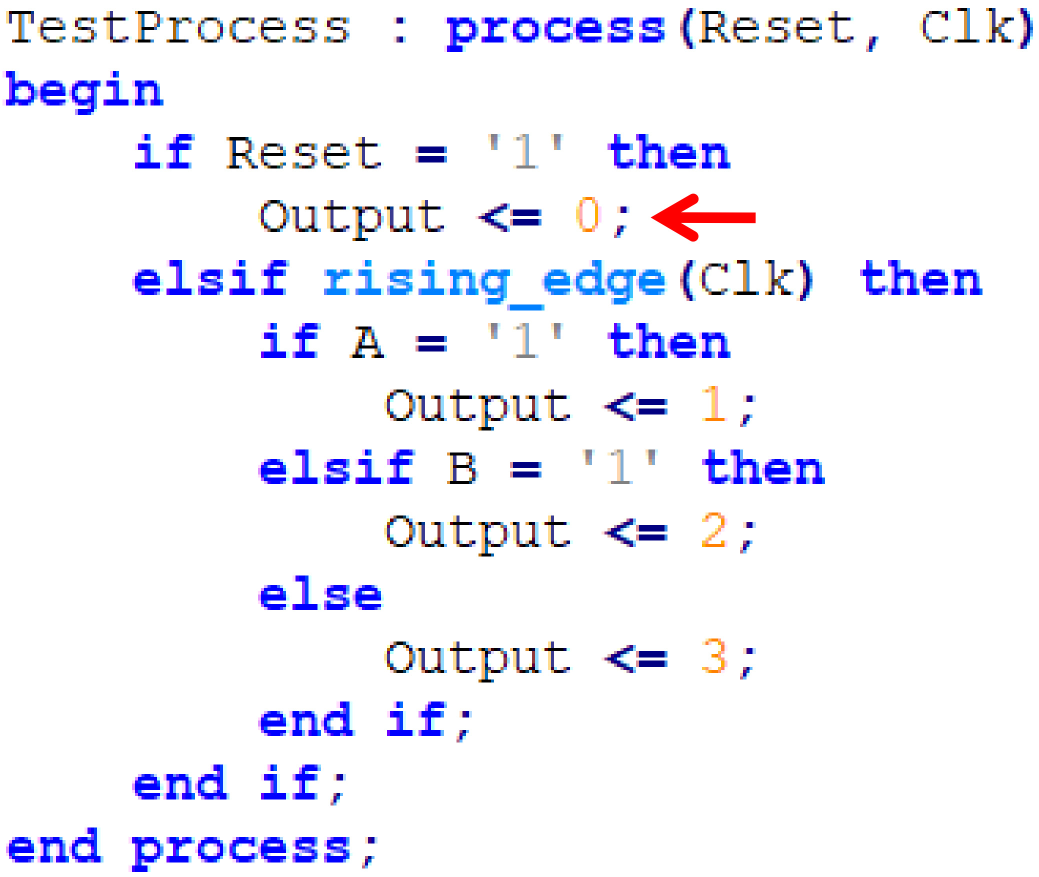
- Ways of dealing with clock domain crossing:
  - 2 – stage synchronisation (identical to the asynchronous input case)
  - Using additional signals for hand-shaking.
  - Using FIFOs

# Use Generics

- Makes modules reconfigurable.
- Allows for re-usable VHDL modules.
- Reduces code redundancy.
- Saves time in writing and testing code.

# Initial Values

- Reset **all** outputs of a synchronous process using a reset :

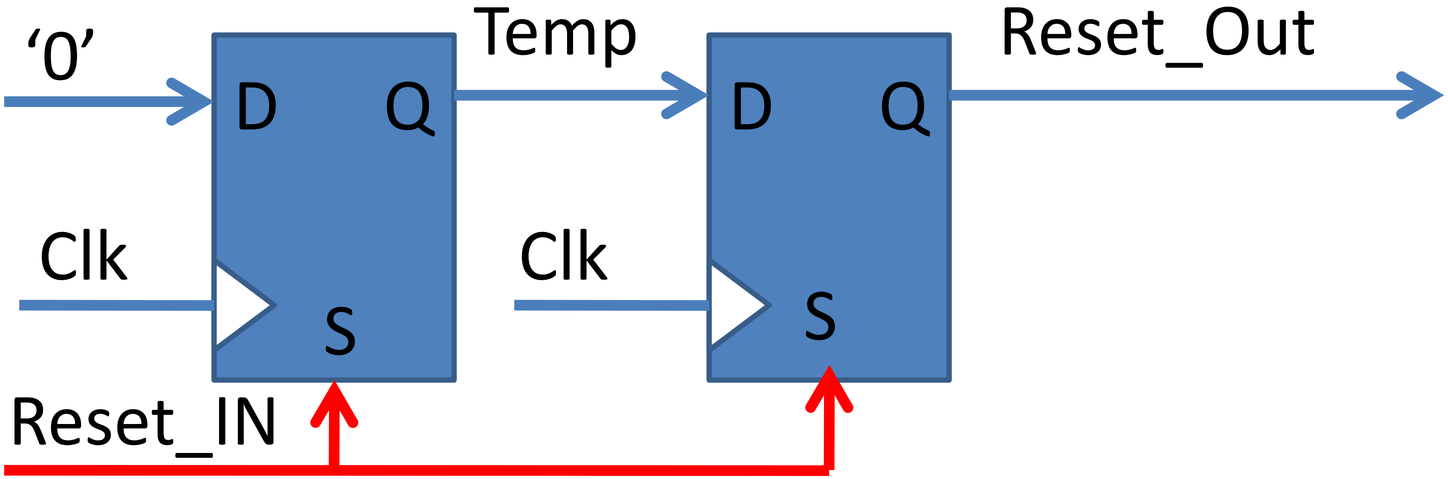


```
TestProcess : process(Reset, Clk)
begin
    if Reset = '1' then
        Output <= 0;
    elsif rising_edge(Clk) then
        if A = '1' then
            Output <= 1;
        elsif B = '1' then
            Output <= 2;
        else
            Output <= 3;
        end if;
    end if;
end process;
```

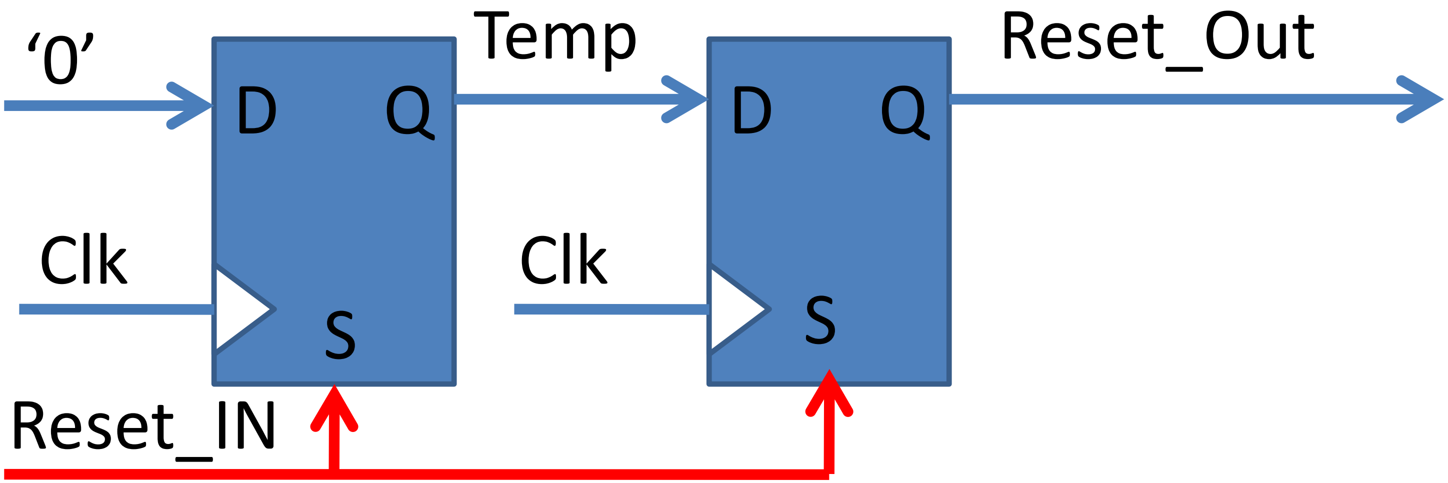
# Synchronised Reset De-Assertion

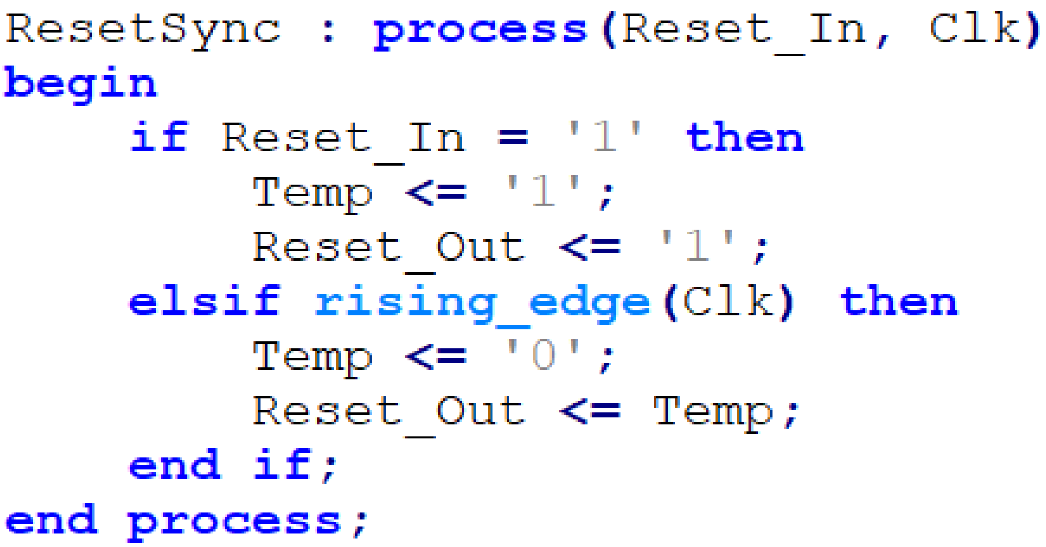
- Using a purely asynchronous reset can lead to problems.
- Reset signal takes different times to reach different parts within the FPGA (due to propagation delay).
- Therefore some registers of the FPGA will come out of reset slightly earlier than other parts.
- This has the potential to cause a glitch / bug.
- We must synchronise the de-assertion of the reset signal to avoid this type of fault.

# Synchronised Reset De-Assertion



# Synchronised Reset De-Assertion





```
ResetSync : process(Reset_In, Clk)
begin
    if Reset_In = '1' then
        Temp <= '1';
        Reset_Out <= '1';
    elsif rising_edge(Clk) then
        Temp <= '0';
        Reset_Out <= Temp;
    end if;
end process;
```

# More On Clocks And Resets

- Route clocks and resets on global routing.
- Global routing has very low skew.
- Global routing allows signals to reach all registers in the FPGA almost at the same time.
- Use PLL to clean up the input clock.
- PLLs can reduce the effect of noise, ringing and reflections that can appear on the input clock signal.

# Topics

- Dealing with Asynchronous inputs.
- Crossing clock domains.
- Reusable VHDL modules.
- Initialising signals.
- Synchronising the de-assertion of system reset.
- Route clocks and resets on global routing.
- Using PLL to clean clock

# Test Benches

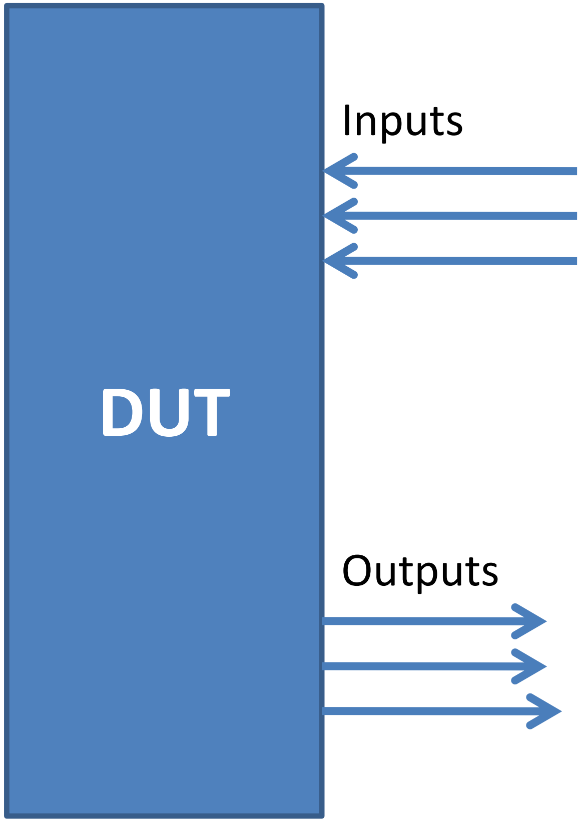
# What is a Test Bench?

- VHDL modules are tested using Test benches.
- A Test Bench is also a VHDL module that we must develop.
- In a basic test bench we :
    - Generate input vectors.
    - Define checks on output vectors.
    - What kinds of tests we want to run.
    - How long we run the tests for.

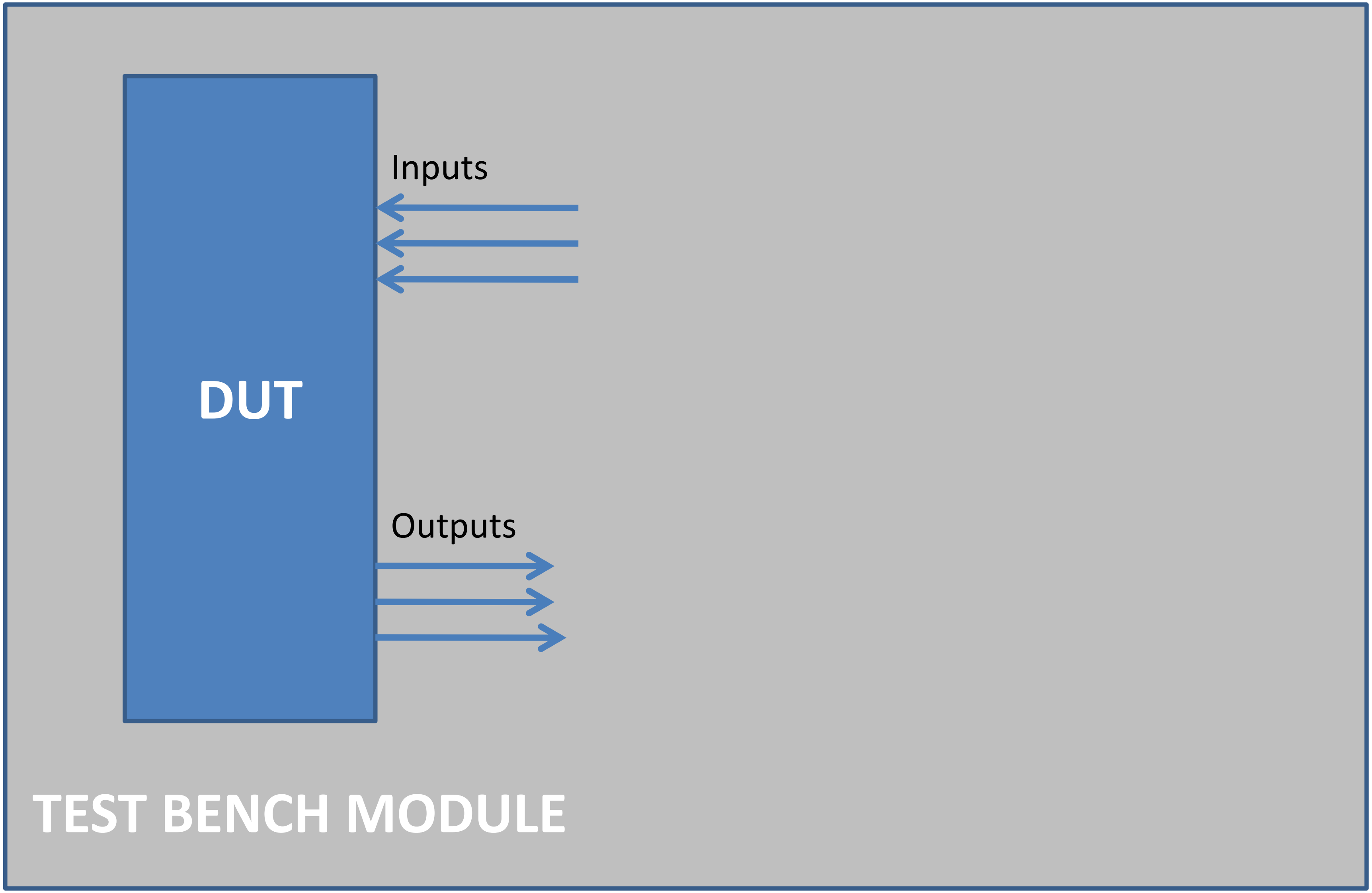
# Test Bench Model

DUT

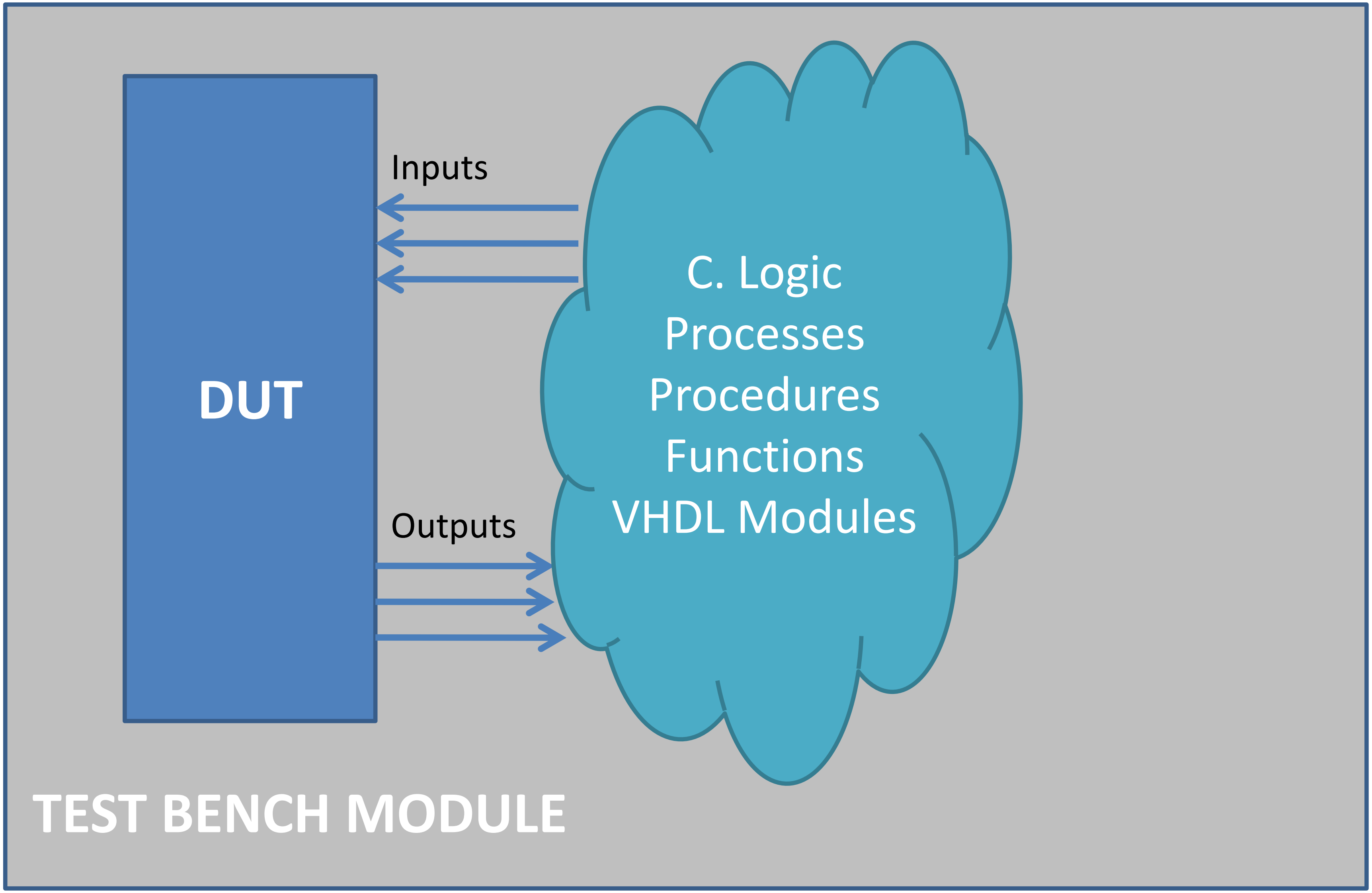
# Test Bench Model



# Test Bench Model



# Test Bench Model



DUT

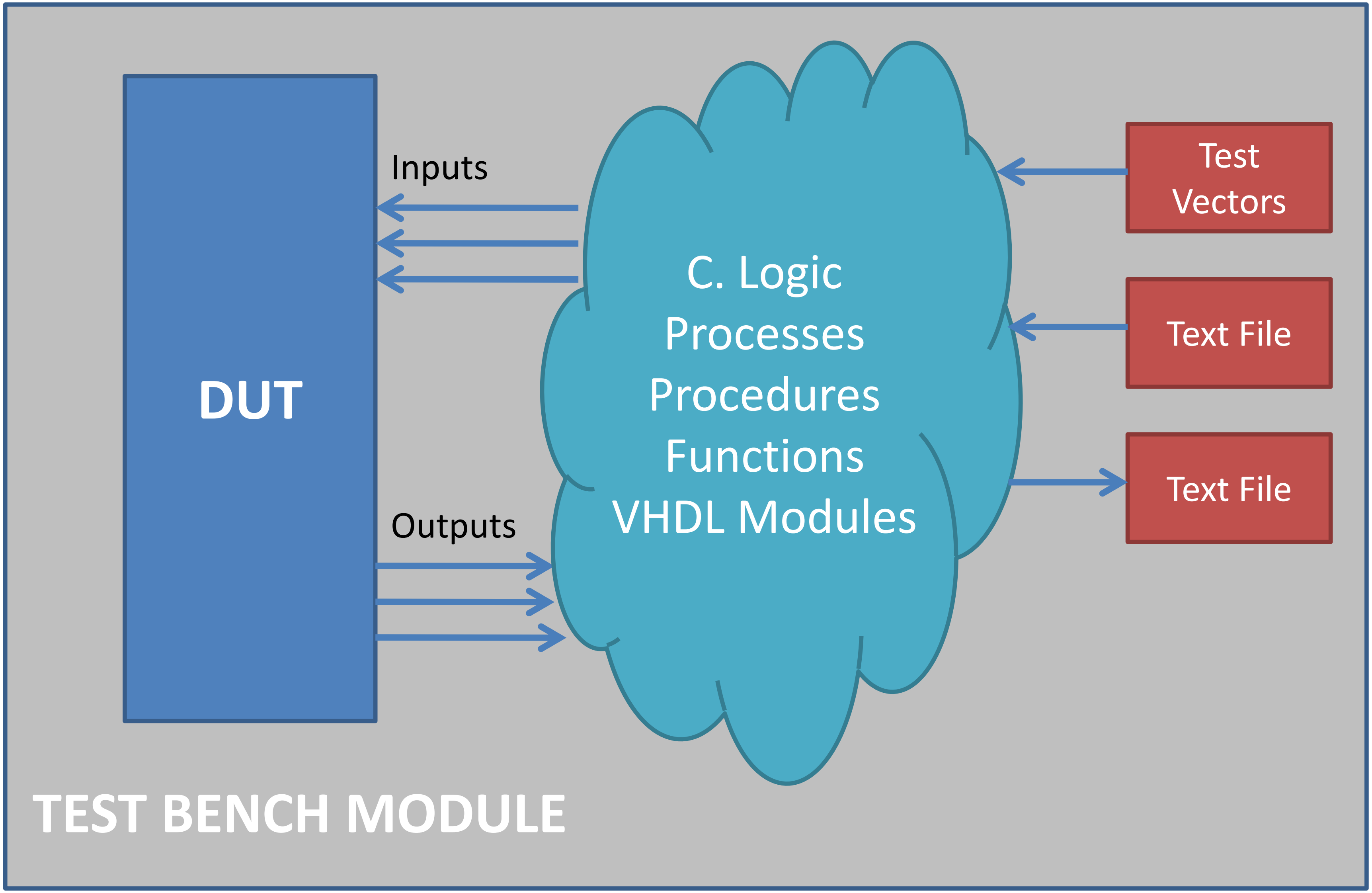
Inputs

Outputs

C. Logic
Processes
Procedures
Functions
VHDL Modules

TEST BENCH MODULE

# Test Bench Model



# Generating Messages

- The Test Bench can print messages on the simulator console window.

# Generating Messages

- The Test Bench can print messages on the simulator console window.

Syntax:

```
report   message_string    severity   severity_level;
```

# Generating Messages

- The Test Bench can print messages on the simulator console window.

Syntax:

```
report   message_string    severity   severity_level;
```

severity_level = note, warning, error, failure

# Generating Messages

- The Test Bench can print messages on the simulator console window.

Syntax:

```
report   message_string    severity   severity_level;
```

severity_level : note, warning, error, failure

e.g.:

```
report “This is a message”; -- severity = note.
```

# Generating Messages

- The Test Bench can print messages on the simulator console window.

Syntax:

```
report   message_string    severity   severity_level;
```

severity_level : note, warning, error, failure

e.g.:

```
report “This is a message”; -- severity = note.
report “This is a message” severity warning;
```

# Assert Statement

- The assert statement tests a Boolean condition.
- If false, it outputs a message string to the simulator output console.

# Assert Statement

- The assert statement tests a Boolean condition.
- If false, it outputs a message string to the simulator output console.

Syntax:

```
assert condition report string severity severity_level;
```

# Assert Statement

- The assert statement tests a Boolean condition.
- If false, it outputs a message string to the simulator output console.

Syntax:

```
assert condition report string severity severity_level;
```

e.g.:

```
assert Overflow = ‘0’
report ”An overflow has occurred!” severity warning;
```

# Using Text Files

- Must declare **Textio** package ("use std.textio.all;")
- Allows for more complex testing.
- Input vectors can be read from text files.
- Output vectors can be written to text files.
- Data types stored in files can be anything (real numbers, strings, std_logic_vectors, integers etc).
- Basic operations with text files :
  - Declaration of a text file.
  - Opening and closing a file of a text file.
  - Reading and or writing to a file.

# Declaring a Text File

Syntax :

```
file FileHandle : text;
```

# Declaring a Text File

Syntax :

```
file FileHandle : text;
```

e.g.:

```
file File1 : text;
```

# Declaring a Text File

Syntax :

```
file FileHandle : text;
```

e.g.:

```
file File1 : text;
```

Where to declare :

- Declarative region of the architecture
- Declarative region of a process

# Opening a Text File

Syntax :

file_open(FileHandle, “FileName.txt”,  OpenMode);

# Opening A Text File

Syntax :

```
file_open(FileHandle, “FileName.txt”,  OpenMode);
```

Open  Mode :

1. read_Mode
2. write_Mode
3. append

# Opening A Text File

Syntax :

file_open(FileHandle, “FileName.txt”, OpenMode);

Open Mode :

1. read_Mode
2. write_Mode
3. append

E.g.:

file_open(File1, “InputVectors.txt”, read_mode);

file_open(File2, “OutputVectors.txt”, write_mode);

# Closing A File

Syntax :

```
file_close(FileHandle);
```

# Closing A File

Syntax :

```
file_close(FileHandle);
```

E.g.:

```
file_close(File1);
```

# Test Benches

## Writing Data to Text Files

# Writing To a Text File

1. Open a text file in write mode.
2. Write to a line buffer:

```
write(LineBuffer, DataToWrite);
```

# Writing To a Text File

1. Open a text file in write mode.
2. Write to a line buffer:

    ```
    write(LineBuffer, DataToWrite);
    ```

3. Write line buffer to file:

    ```
    writeline(FileHandle, LineBuffer);
    ```

# Writing To a Text File

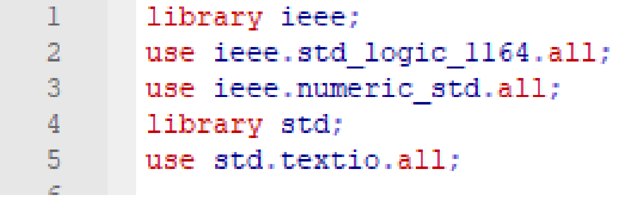
1. Open a text file in write mode.
2. Write to a line buffer:

   ```
   write(LineBuffer, DataToWrite);
   ```

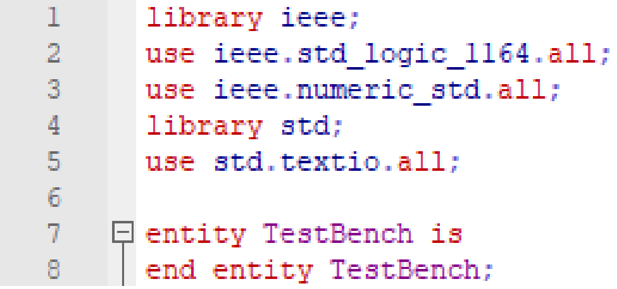
3. Write line buffer to file:

   ```
   writeline(FileHandle, LineBuffer);
   ```

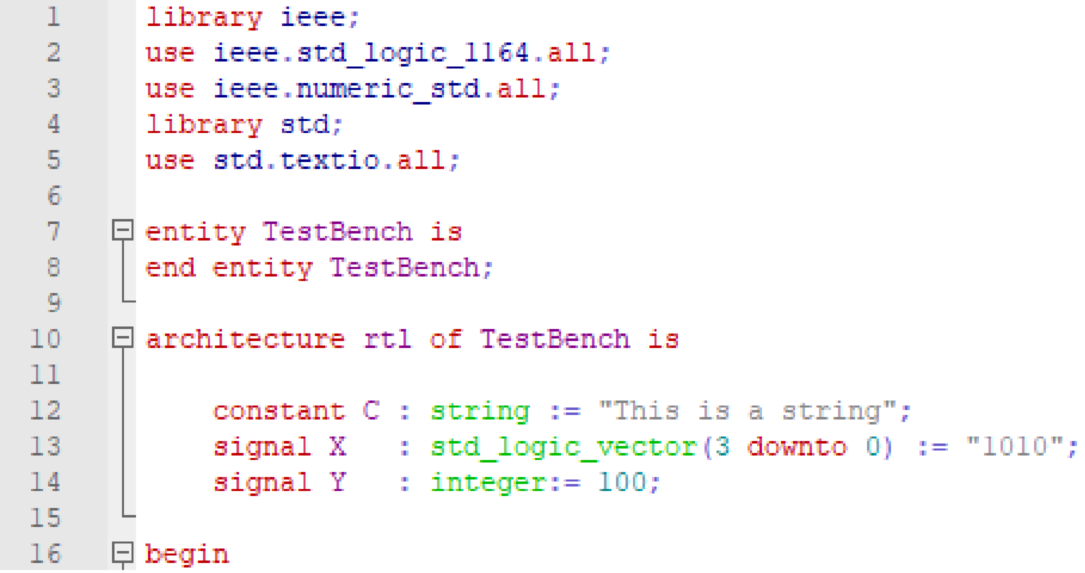
4. Close file.



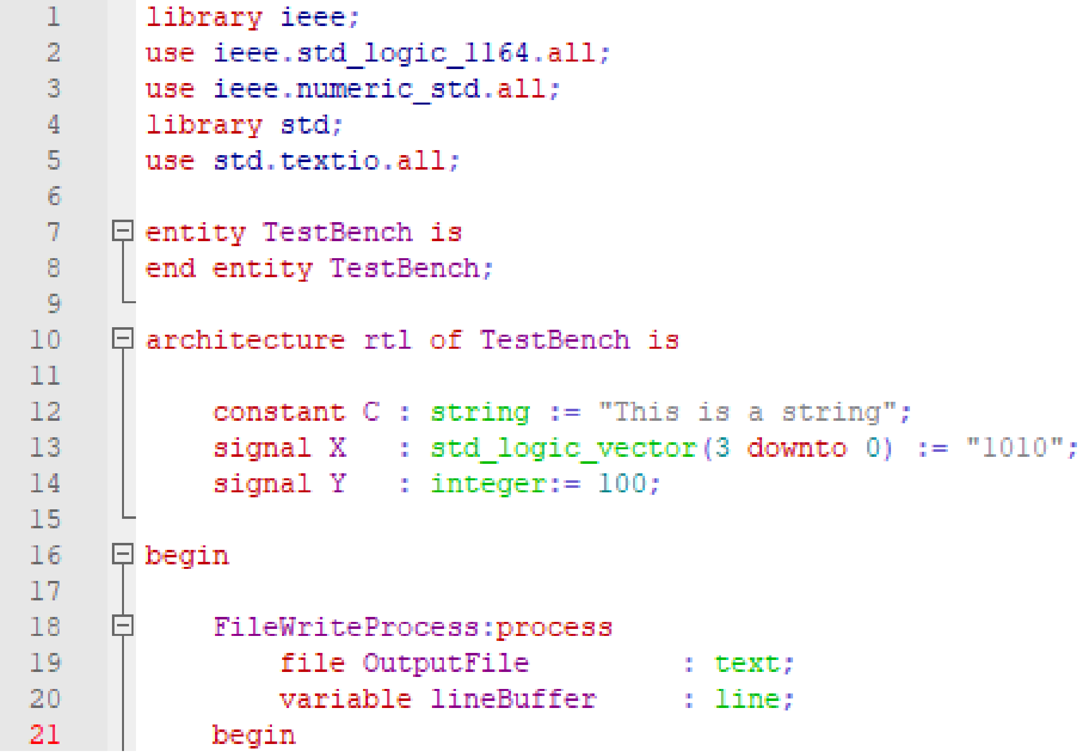
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library std;
use std.textio.all;
```



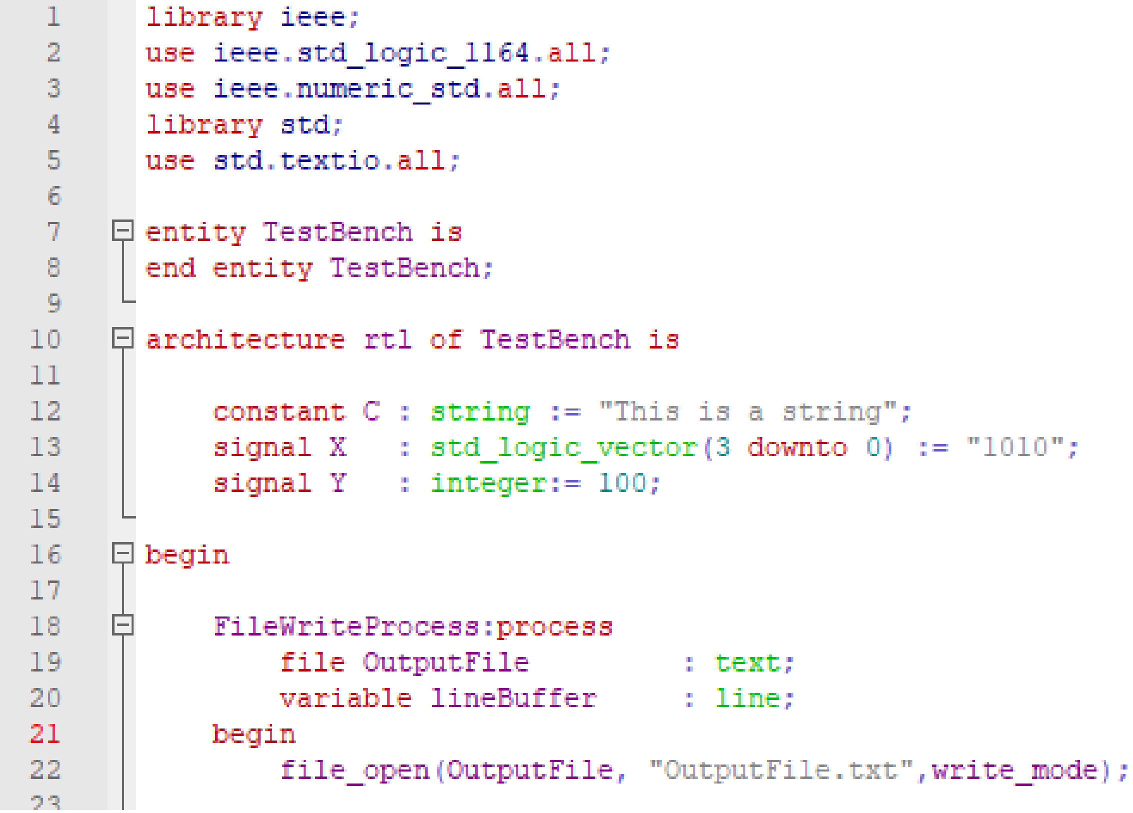
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library std;
use std.textio.all;

entity TestBench is
end entity TestBench;
```



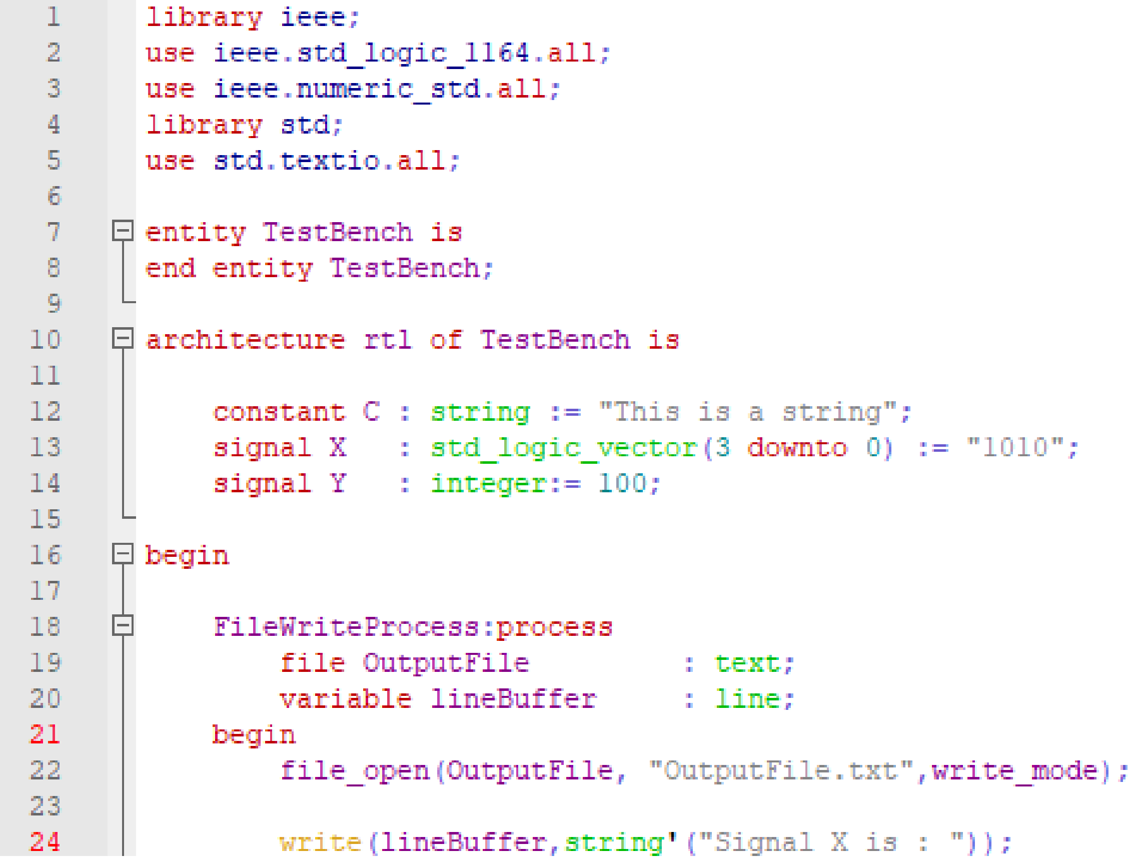
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library std;
use std.textio.all;

entity TestBench is
end entity TestBench;

architecture rtl of TestBench is

    constant C : string := "This is a string";
    signal X   : std_logic_vector(3 downto 0) := "1010";
    signal Y   : integer:= 100;

begin
```



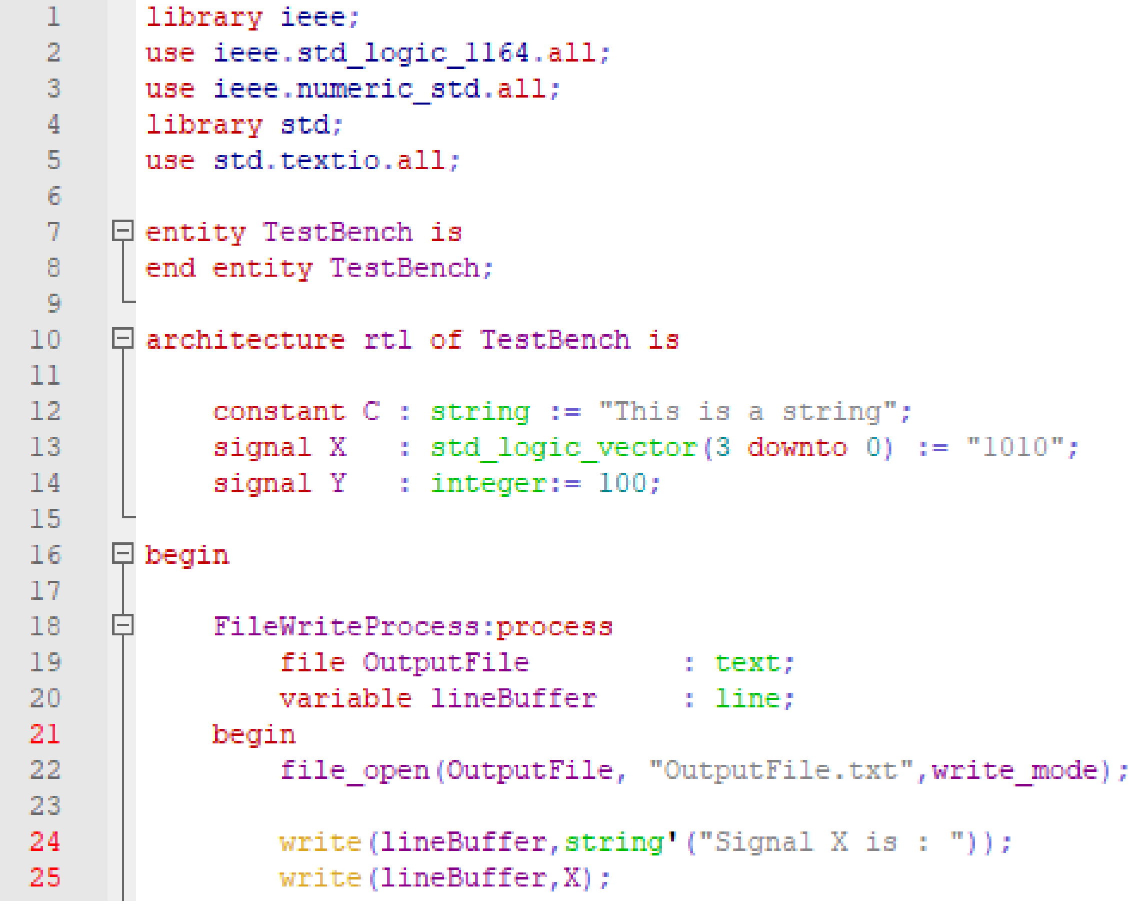
```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library std;
use std.textio.all;

entity TestBench is
end entity TestBench;

architecture rtl of TestBench is

    constant C : string := "This is a string";
    signal X   : std_logic_vector(3 downto 0) := "1010";
    signal Y   : integer:= 100;

begin

    FileWriteProcess:process
        file OutputFile         : text;
        variable lineBuffer     : line;
    begin
```



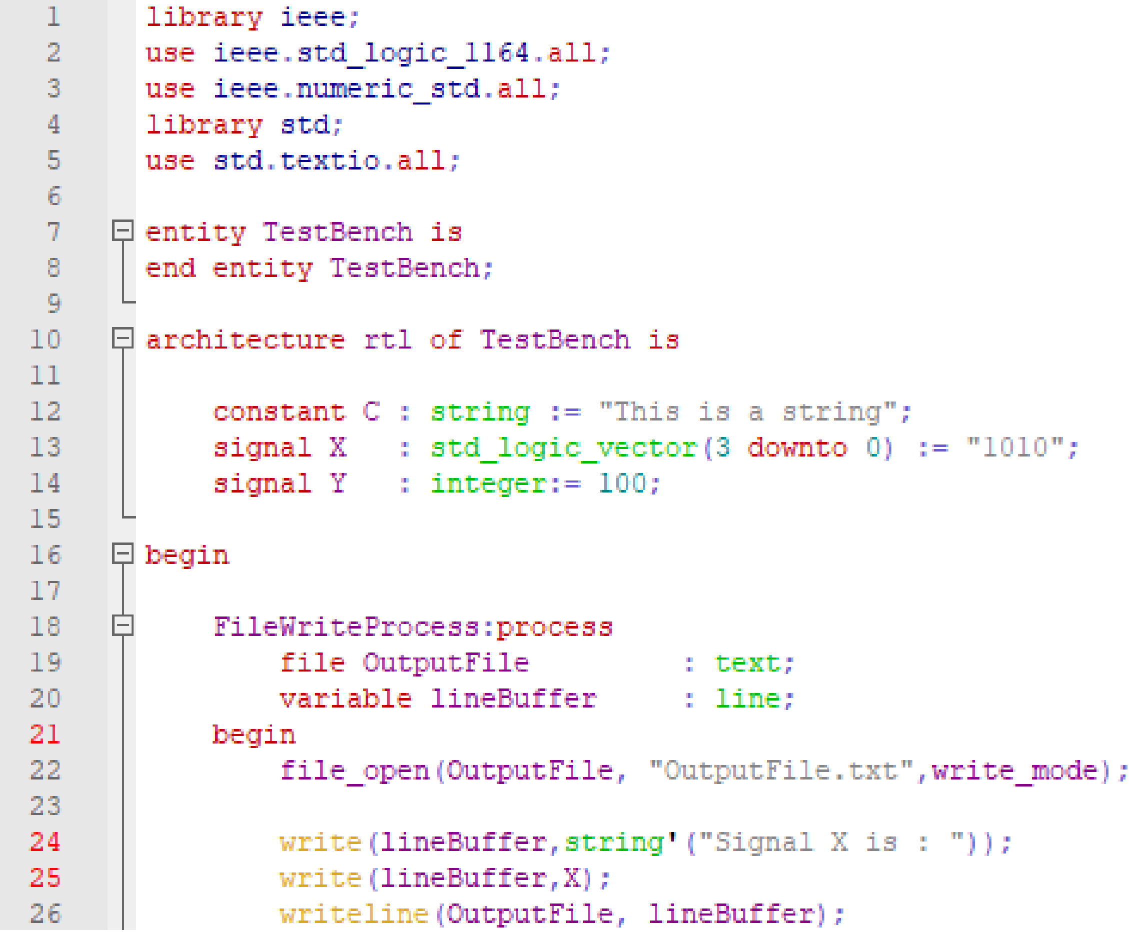
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library std;
use std.textio.all;

entity TestBench is
end entity TestBench;

architecture rtl of TestBench is

    constant C : string := "This is a string";
    signal X   : std_logic_vector(3 downto 0) := "1010";
    signal Y   : integer:= 100;

begin

    FileWriteProcess:process
        file OutputFile         : text;
        variable lineBuffer     : line;
    begin
        file_open(OutputFile, "OutputFile.txt",write_mode);
```



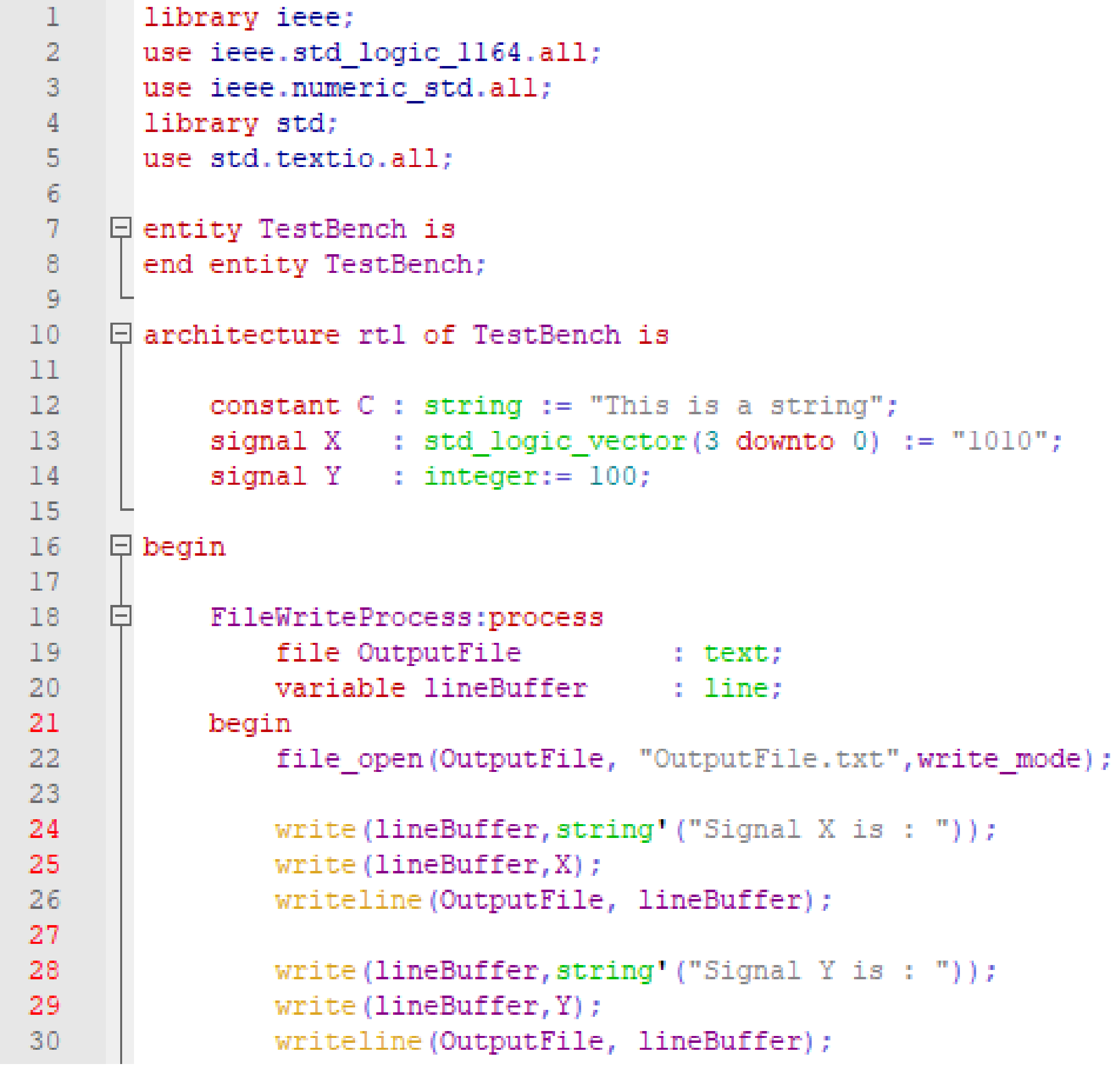
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library std;
use std.textio.all;

entity TestBench is
end entity TestBench;

architecture rtl of TestBench is

    constant C : string := "This is a string";
    signal X   : std_logic_vector(3 downto 0) := "1010";
    signal Y   : integer:= 100;

begin

    FileWriteProcess:process
        file OutputFile         : text;
        variable lineBuffer     : line;
    begin
        file_open(OutputFile, "OutputFile.txt",write_mode);

        write(lineBuffer,string'("Signal X is : "));
```



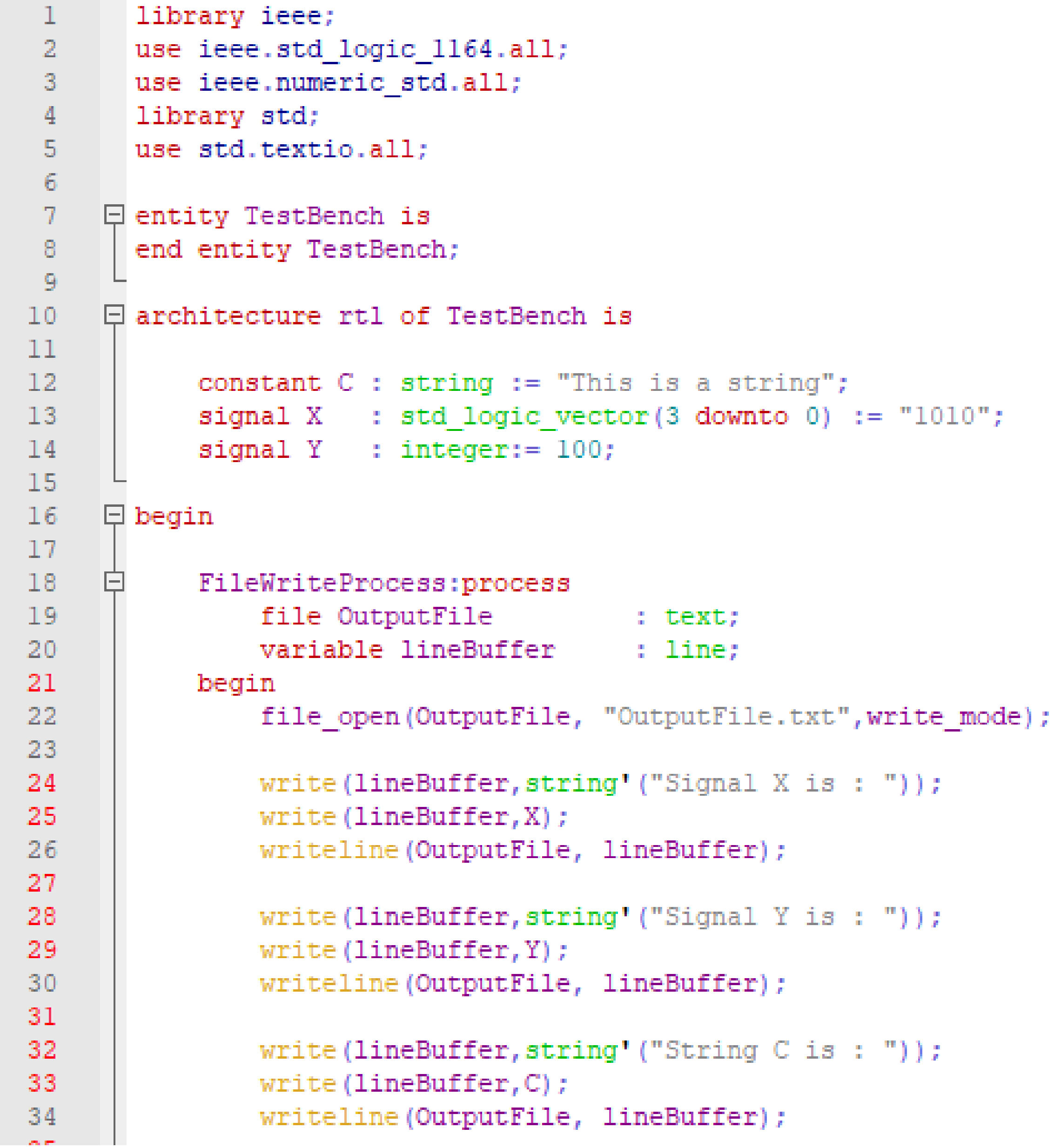
```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library std;
use std.textio.all;

entity TestBench is
end entity TestBench;

architecture rtl of TestBench is

    constant C : string := "This is a string";
    signal X   : std_logic_vector(3 downto 0) := "1010";
    signal Y   : integer:= 100;

begin

    FileWriteProcess:process
        file OutputFile         : text;
        variable lineBuffer     : line;
    begin
        file_open(OutputFile, "OutputFile.txt",write_mode);

        write(lineBuffer,string'("Signal X is : "));
        write(lineBuffer,X);
```



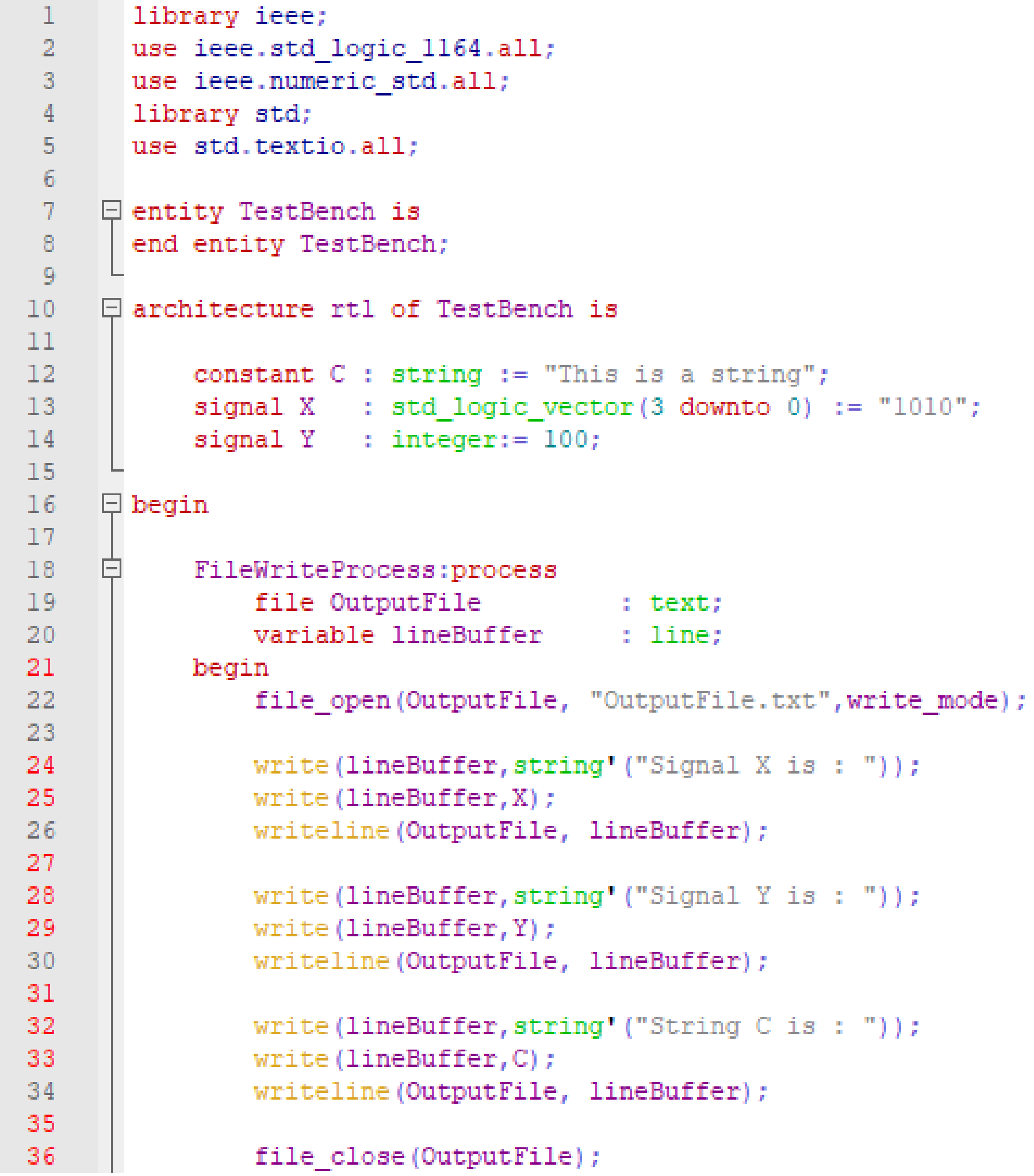
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library std;
use std.textio.all;

entity TestBench is
end entity TestBench;

architecture rtl of TestBench is

    constant C : string := "This is a string";
    signal X   : std_logic_vector(3 downto 0) := "1010";
    signal Y   : integer:= 100;

begin

    FileWriteProcess:process
        file OutputFile         : text;
        variable lineBuffer     : line;
    begin
        file_open(OutputFile, "OutputFile.txt",write_mode);

        write(lineBuffer,string'("Signal X is : "));
        write(lineBuffer,X);
        writeline(OutputFile, lineBuffer);
```



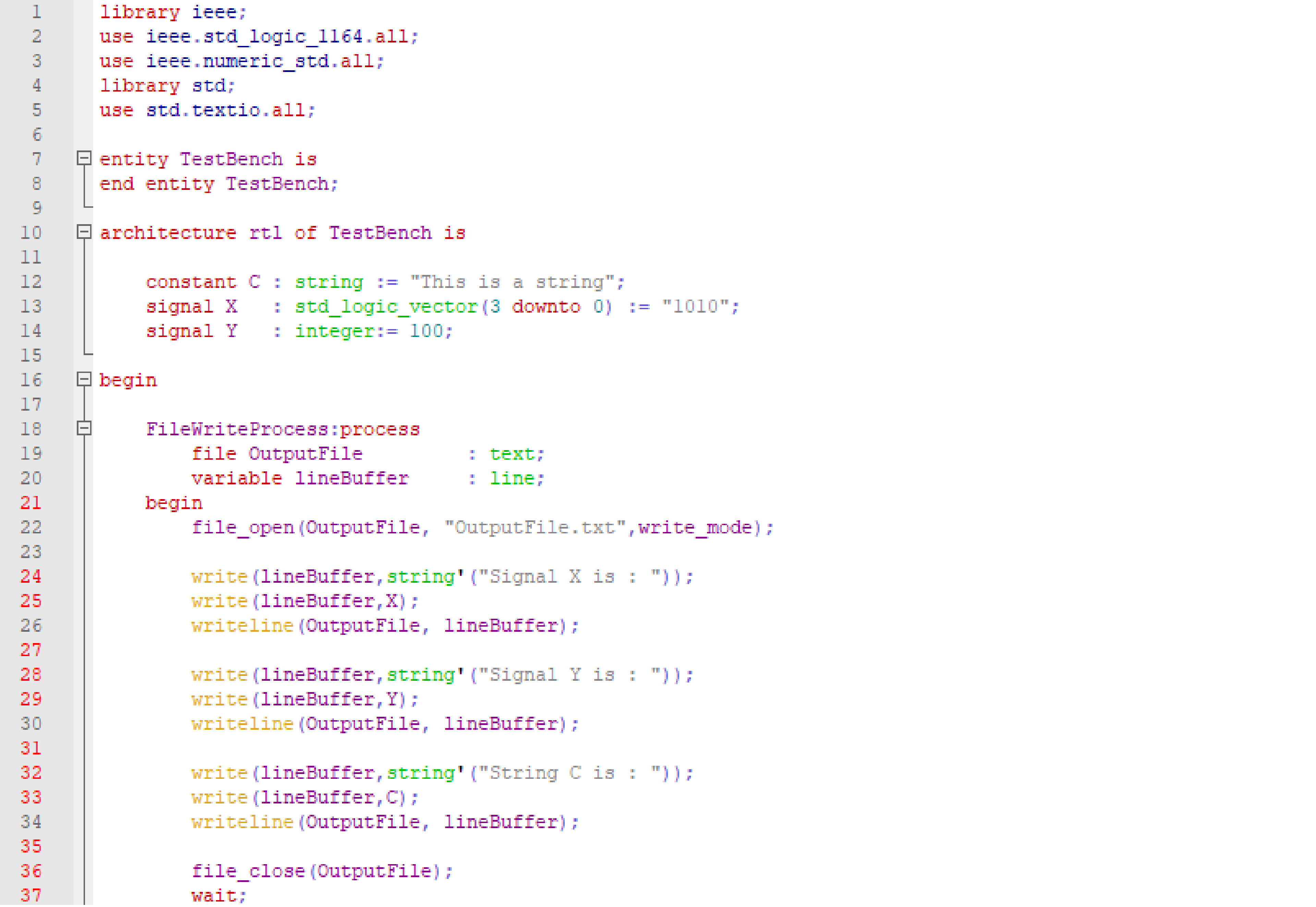
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library std;
use std.textio.all;

entity TestBench is
end entity TestBench;

architecture rtl of TestBench is

    constant C : string := "This is a string";
    signal X   : std_logic_vector(3 downto 0) := "1010";
    signal Y   : integer:= 100;

begin

    FileWriteProcess:process
        file OutputFile         : text;
        variable lineBuffer     : line;
    begin
        file_open(OutputFile, "OutputFile.txt",write_mode);

        write(lineBuffer,string'("Signal X is : "));
        write(lineBuffer,X);
        writeline(OutputFile, lineBuffer);

        write(lineBuffer,string'("Signal Y is : "));
        write(lineBuffer,Y);
        writeline(OutputFile, lineBuffer);
```



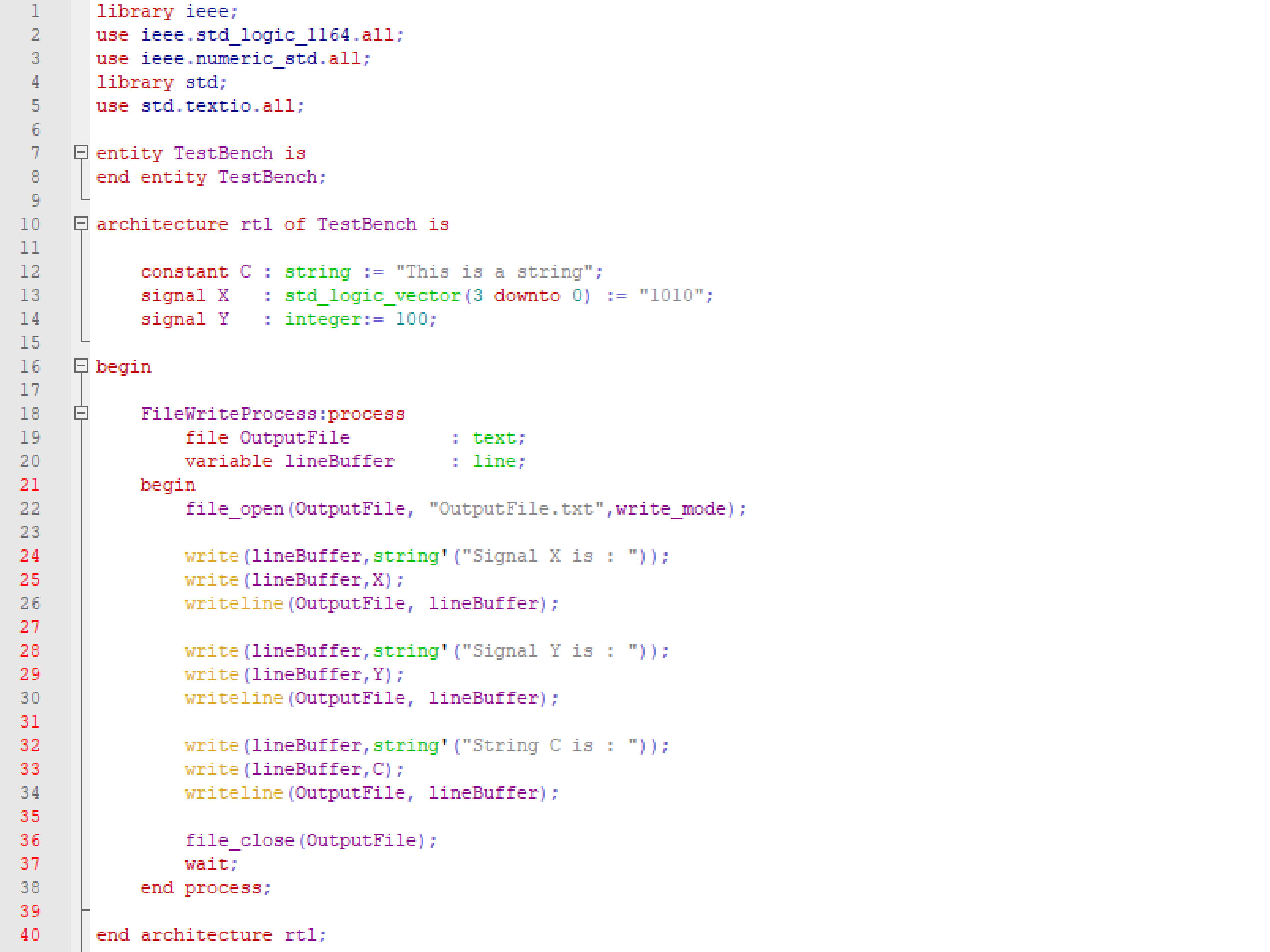
```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library std;
use std.textio.all;

entity TestBench is
end entity TestBench;

architecture rtl of TestBench is

    constant C : string := "This is a string";
    signal X   : std_logic_vector(3 downto 0) := "1010";
    signal Y   : integer:= 100;

begin

    FileWriteProcess:process
        file OutputFile         : text;
        variable lineBuffer     : line;
    begin
        file_open(OutputFile, "OutputFile.txt",write_mode);

        write(lineBuffer,string'("Signal X is : "));
        write(lineBuffer,X);
        writeline(OutputFile, lineBuffer);

        write(lineBuffer,string'("Signal Y is : "));
        write(lineBuffer,Y);
        writeline(OutputFile, lineBuffer);

        write(lineBuffer,string'("String C is : "));
        write(lineBuffer,C);
        writeline(OutputFile, lineBuffer);
```



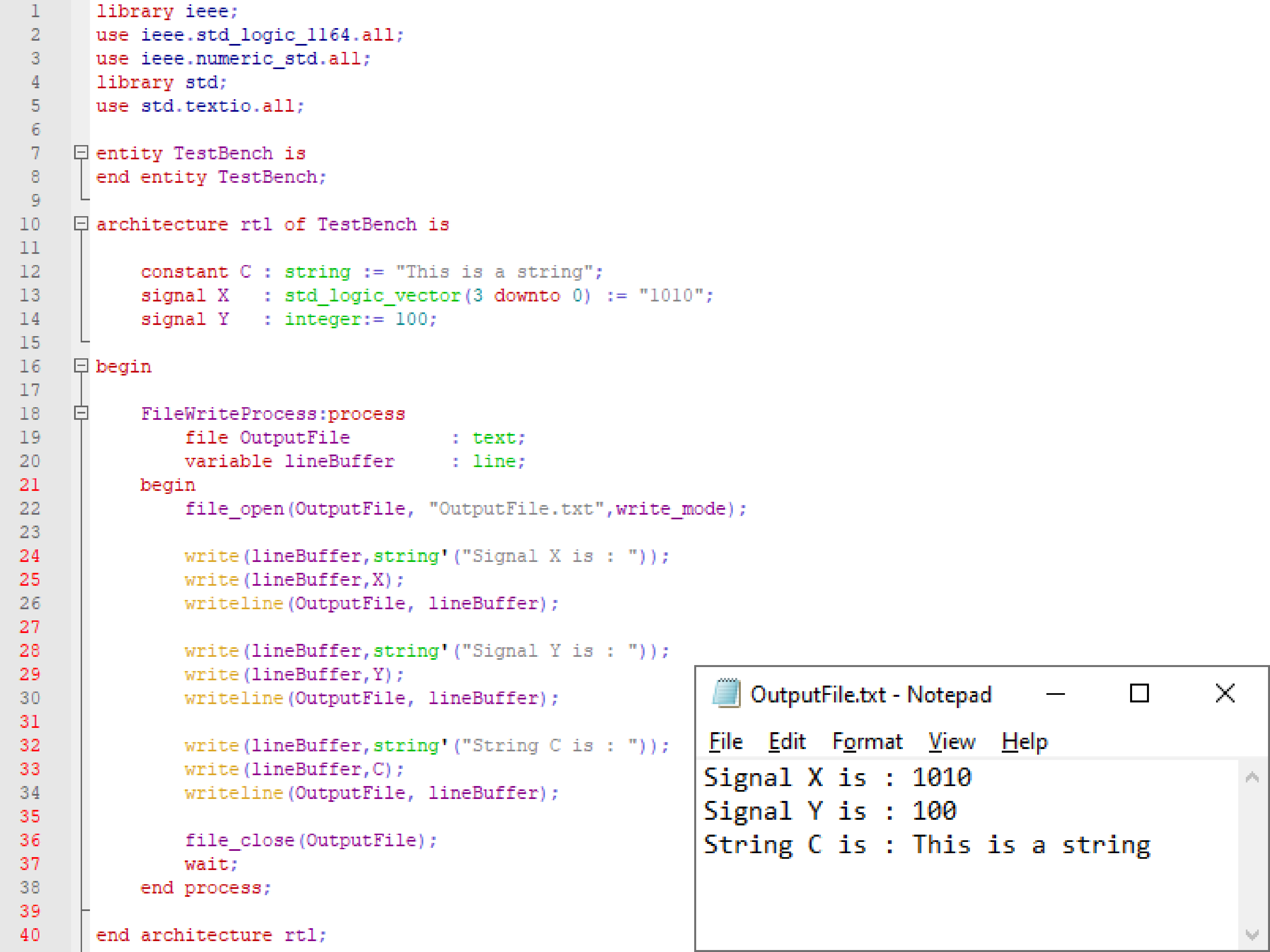
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library std;
use std.textio.all;

entity TestBench is
end entity TestBench;

architecture rtl of TestBench is

    constant C : string := "This is a string";
    signal X   : std_logic_vector(3 downto 0) := "1010";
    signal Y   : integer:= 100;

begin

    FileWriteProcess:process
        file OutputFile         : text;
        variable lineBuffer     : line;
    begin
        file_open(OutputFile, "OutputFile.txt",write_mode);

        write(lineBuffer,string'("Signal X is : "));
        write(lineBuffer,X);
        writeline(OutputFile, lineBuffer);

        write(lineBuffer,string'("Signal Y is : "));
        write(lineBuffer,Y);
        writeline(OutputFile, lineBuffer);

        write(lineBuffer,string'("String C is : "));
        write(lineBuffer,C);
        writeline(OutputFile, lineBuffer);

        file_close(OutputFile);
```



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

entity TestBench is
end entity TestBench;

architecture rtl of TestBench is

    constant C : string := "This is a string";
    signal X   : std_logic_vector(3 downto 0) := "1010";
    signal Y   : integer:= 100;

begin

    FileWriteProcess:process
        file OutputFile         : text;
        variable lineBuffer     : line;
    begin
        file_open(OutputFile, "OutputFile.txt",write_mode);

        write(lineBuffer,string'("Signal X is : "));
        write(lineBuffer,X);
        writeline(OutputFile, lineBuffer);

        write(lineBuffer,string'("Signal Y is : "));
        write(lineBuffer,Y);
        writeline(OutputFile, lineBuffer);

        write(lineBuffer,string'("String C is : "));
        write(lineBuffer,C);
        writeline(OutputFile, lineBuffer);

        file_close(OutputFile);
        wait;
```



```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library std;
use std.textio.all;

entity TestBench is
end entity TestBench;

architecture rtl of TestBench is

    constant C : string := "This is a string";
    signal X   : std_logic_vector(3 downto 0) := "1010";
    signal Y   : integer:= 100;

begin

    FileWriteProcess:process
        file OutputFile         : text;
        variable lineBuffer     : line;
    begin
        file_open(OutputFile, "OutputFile.txt",write_mode);

        write(lineBuffer,string'("Signal X is : "));
        write(lineBuffer,X);
        writeline(OutputFile, lineBuffer);

        write(lineBuffer,string'("Signal Y is : "));
        write(lineBuffer,Y);
        writeline(OutputFile, lineBuffer);

        write(lineBuffer,string'("String C is : "));
        write(lineBuffer,C);
        writeline(OutputFile, lineBuffer);

        file_close(OutputFile);
        wait;
    end process;

end architecture rtl;
```



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

entity TestBench is
end entity TestBench;

architecture rtl of TestBench is

    constant C : string := "This is a string";
    signal X   : std_logic_vector(3 downto 0) := "1010";
    signal Y   : integer:= 100;

begin

    FileWriteProcess:process
        file OutputFile         : text;
        variable lineBuffer     : line;
    begin
        file_open(OutputFile, "OutputFile.txt",write_mode);

        write(lineBuffer,string'("Signal X is : "));
        write(lineBuffer,X);
        writeline(OutputFile, lineBuffer);

        write(lineBuffer,string'("Signal Y is : "));
        write(lineBuffer,Y);
        writeline(OutputFile, lineBuffer);

        write(lineBuffer,string'("String C is : "));
        write(lineBuffer,C);
        writeline(OutputFile, lineBuffer);

        file_close(OutputFile);
        wait;
    end process;

end architecture rtl;
```

```
Signal X is : 1010
Signal Y is : 100
String C is : This is a string
```

# Test Benches

## Reading Data From Text Files

# Reading From a Text File

1. Open a text file in read mode.

# Reading From a Text File

1. Open a text file in read mode.
2. Read from a line from the text file:

```
readline(FileHandle, LineBuffer);
```

# Reading From a Text File

1. Open a text file in read mode.
2. Read from a line from the text file:

   ```
   readline(FileHandle, LineBuffer);
   ```

3. Read a data value from the line buffer to a signal:

   ```
   read(LineBuffer, SignalName);
   ```

# Reading From a Text File

1. Open a text file in read mode.
2. Read from a line from the text file:

   ```
   readline(FileHandle, LineBuffer);
   ```

3. Read a data value from the line buffer to a signal:

   ```
   read(LineBuffer, SignalName);
   ```

4. Repeat steps 2 and 3 until all lines of text file have been read.

# Reading From a Text File

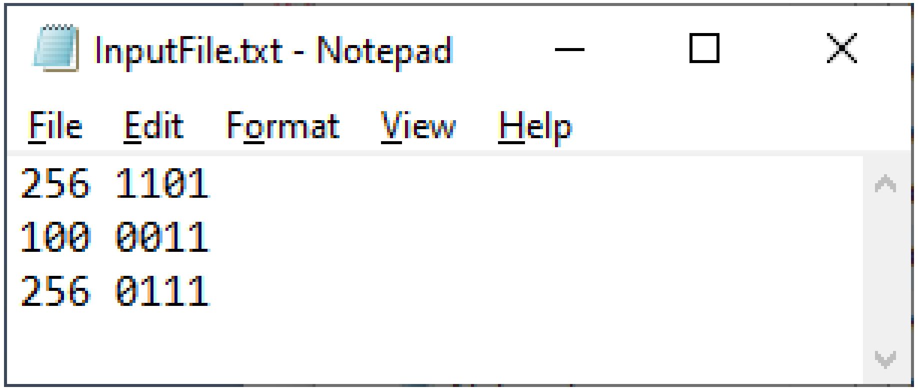
1. Open a text file in read mode.
2. Read from a line from the text file:

   ```
   readline(FileHandle, LineBuffer);
   ```

3. Read a data value from the line buffer to a signal:

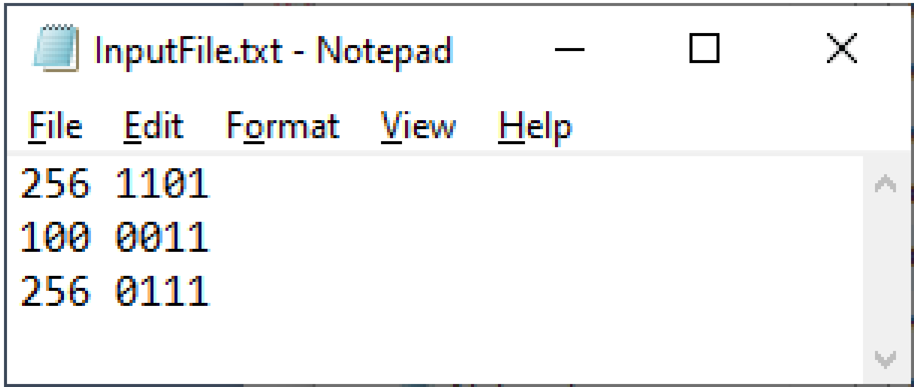
   ```
   read(LineBuffer, SignalName);
   ```

4. Repeat steps 2 and 3 until all lines of text file have been read.
5. Close file.

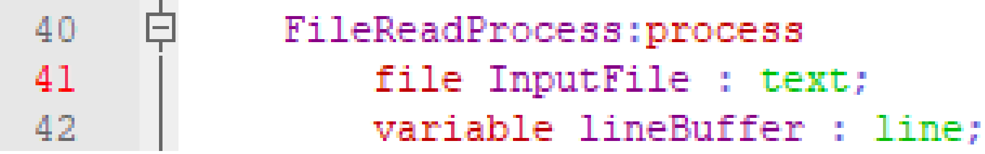


```
256 1101
100 0011
256 0111
```

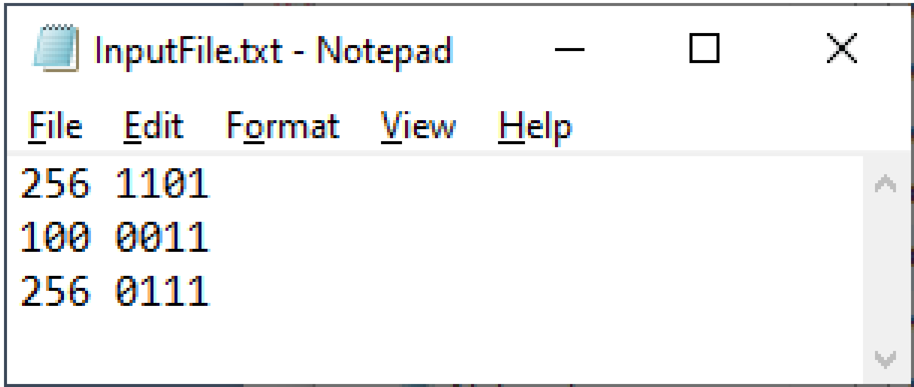
```
    FileReadProcess:process
```



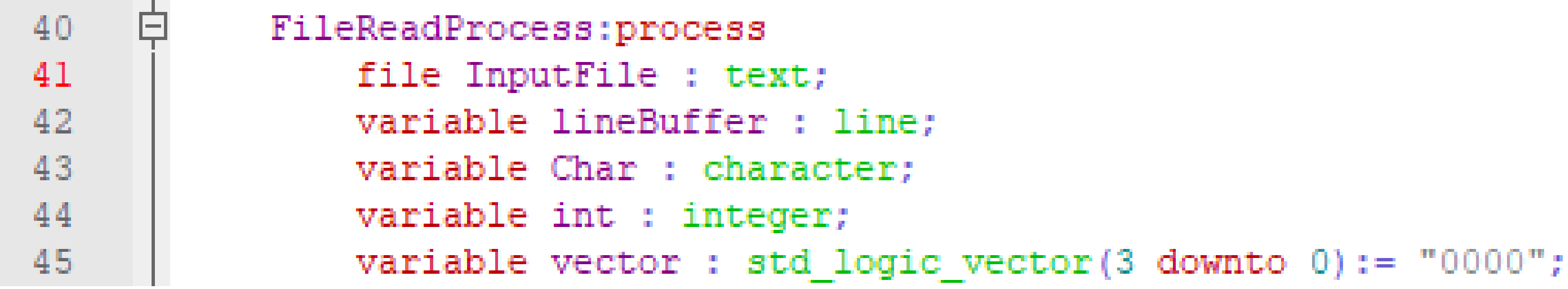
```
256 1101
100 0011
256 0111
```



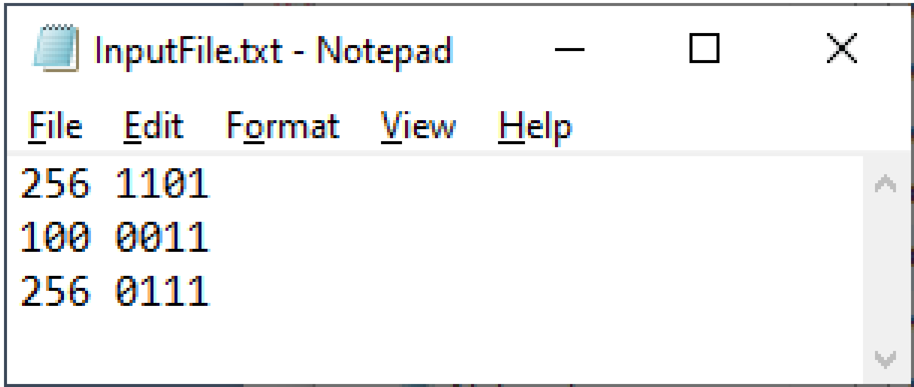
```
    FileReadProcess:process
        file InputFile : text;
        variable lineBuffer : line;
```



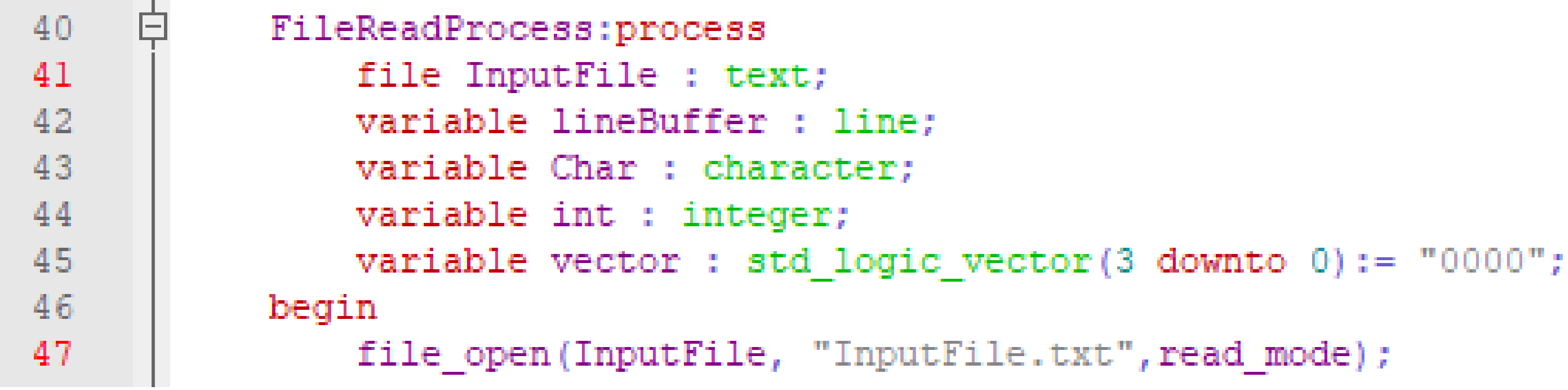
```
256 1101
100 0011
256 0111
```



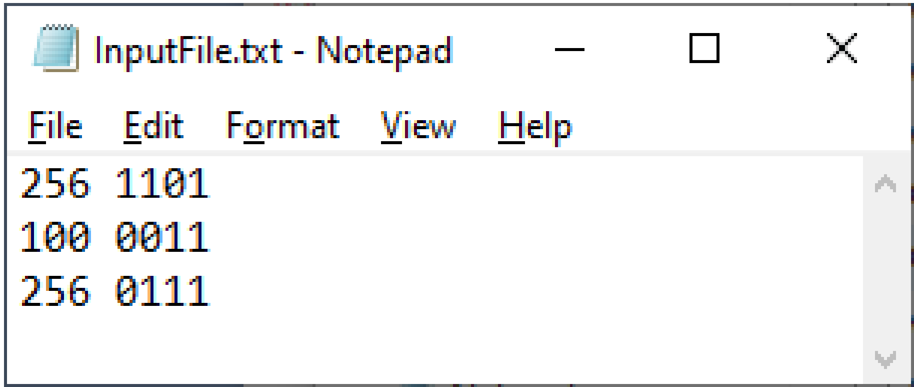
```
FileReadProcess:process
    file InputFile : text;
    variable lineBuffer : line;
    variable Char : character;
    variable int : integer;
    variable vector : std_logic_vector(3 downto 0):= "0000";
```



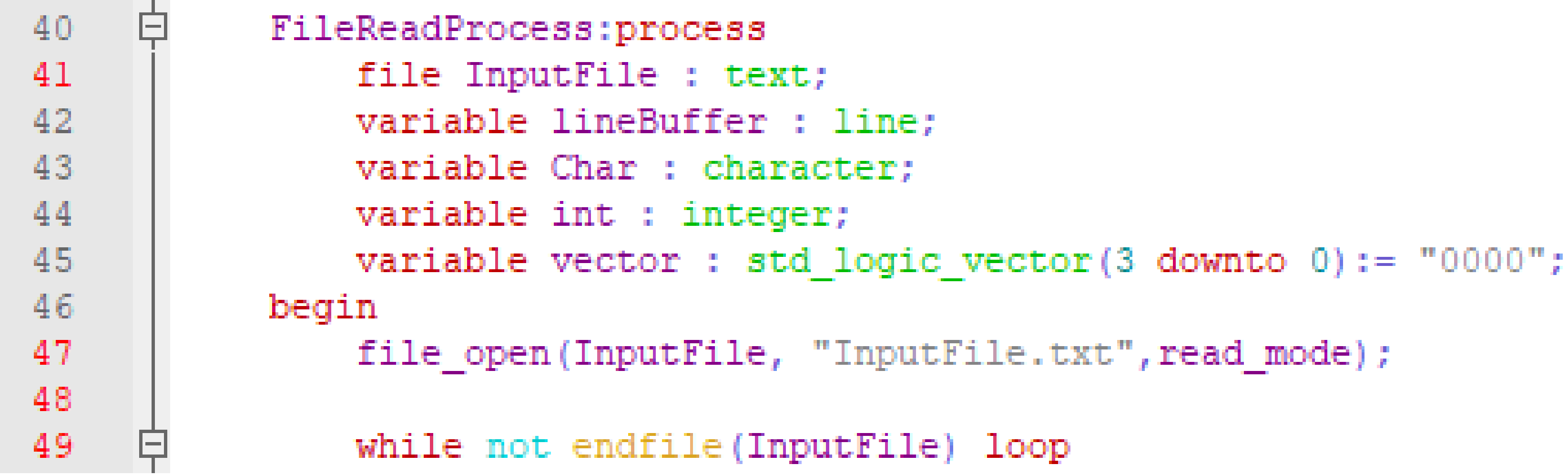
```
256 1101
100 0011
256 0111
```



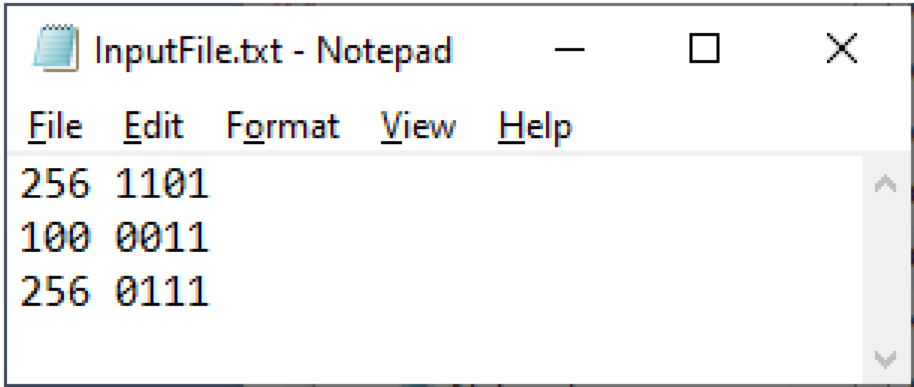
```
    FileReadProcess:process
        file InputFile : text;
        variable lineBuffer : line;
        variable Char : character;
        variable int : integer;
        variable vector : std_logic_vector(3 downto 0):= "0000";
    begin
        file_open(InputFile, "InputFile.txt",read_mode);
```



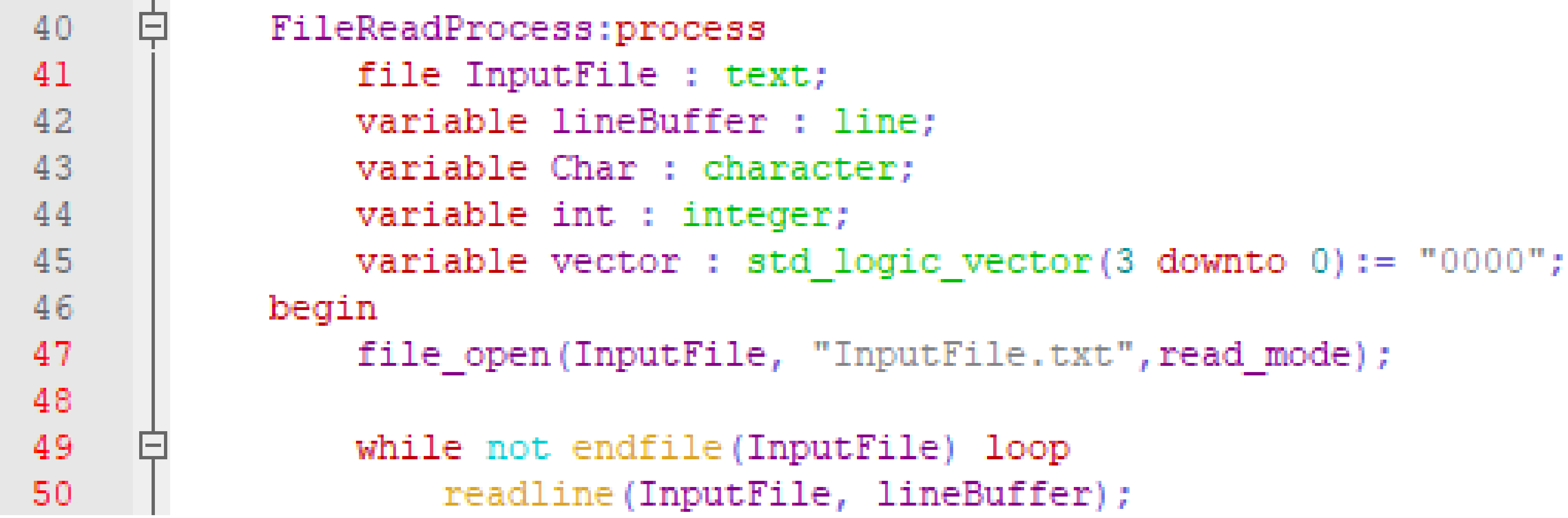
```
256 1101
100 0011
256 0111
```



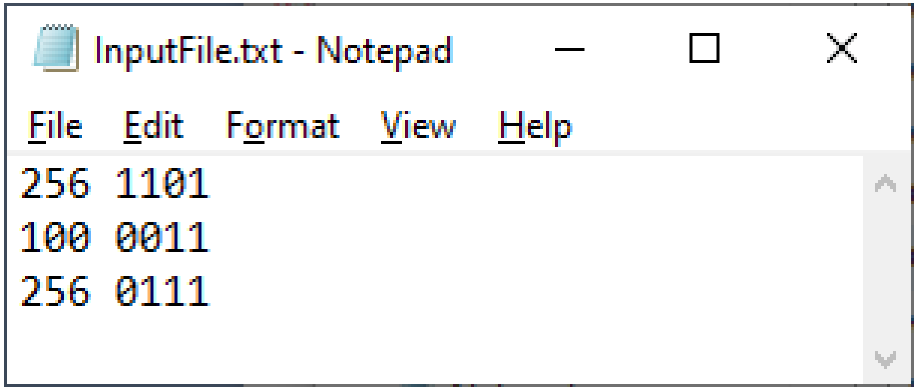
```
FileReadProcess:process
    file InputFile : text;
    variable lineBuffer : line;
    variable Char : character;
    variable int : integer;
    variable vector : std_logic_vector(3 downto 0):= "0000";
begin
    file_open(InputFile, "InputFile.txt",read_mode);

    while not endfile(InputFile) loop
```



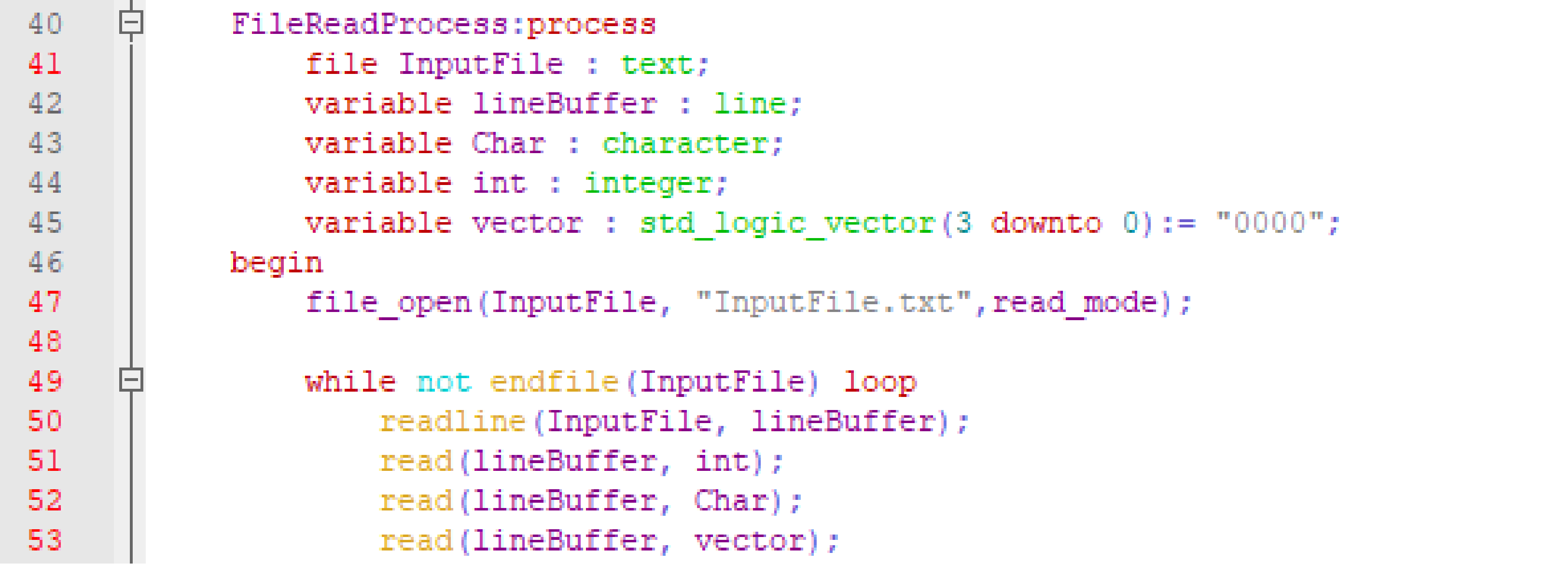
```
256 1101
100 0011
256 0111
```



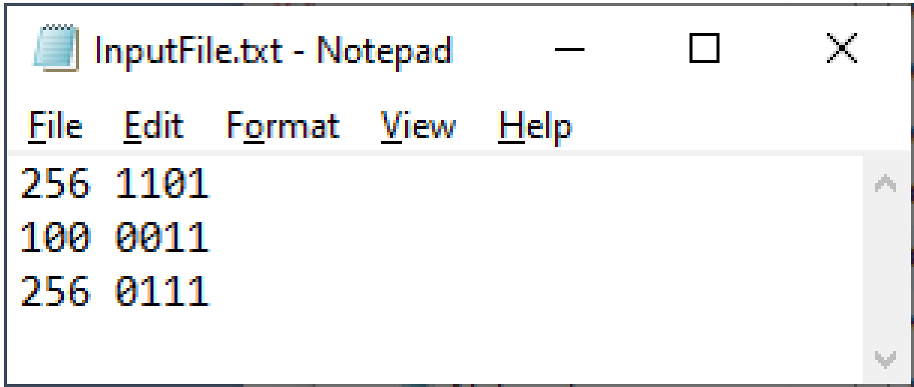
```
FileReadProcess:process
    file InputFile : text;
    variable lineBuffer : line;
    variable Char : character;
    variable int : integer;
    variable vector : std_logic_vector(3 downto 0):= "0000";
begin
    file_open(InputFile, "InputFile.txt",read_mode);

    while not endfile(InputFile) loop
        readline(InputFile, lineBuffer);
```



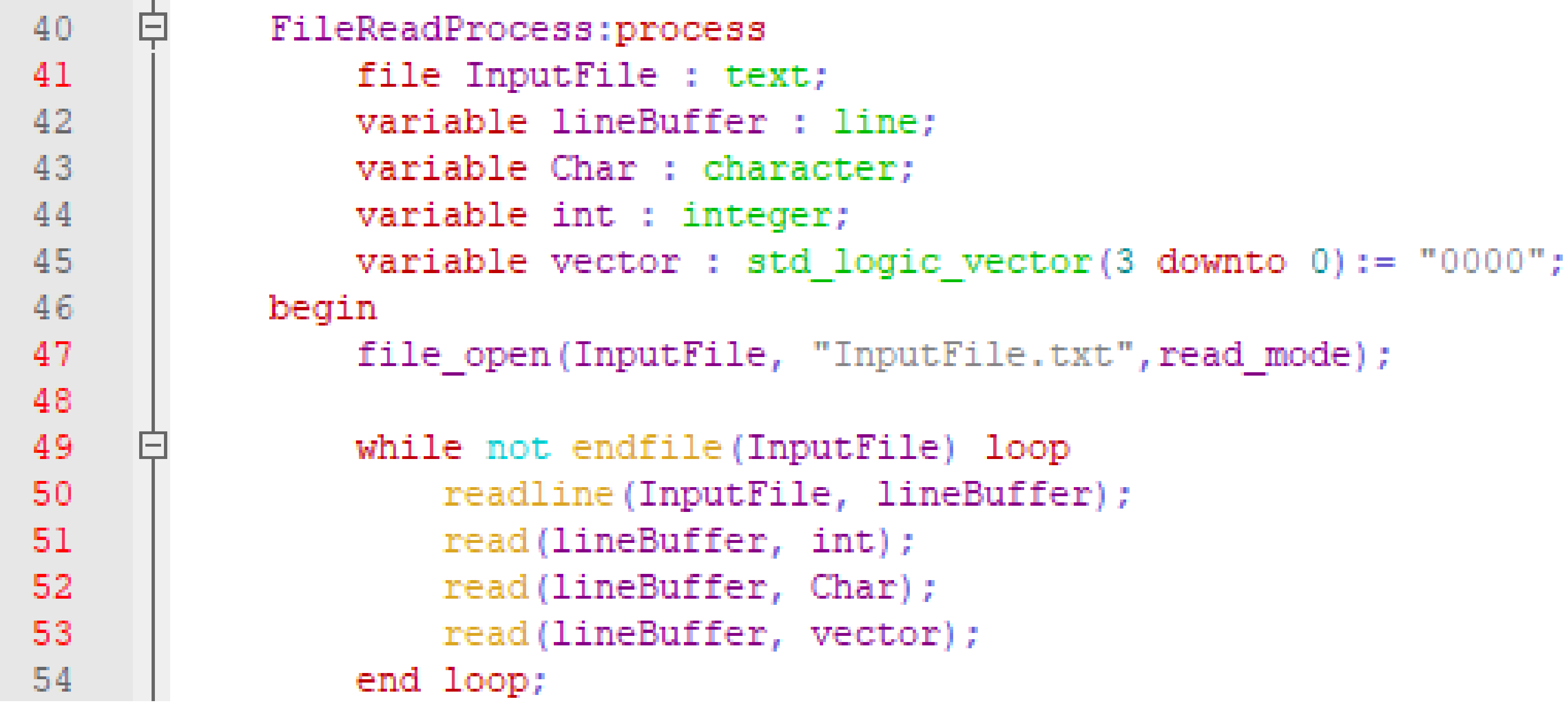
```
256 1101
100 0011
256 0111
```



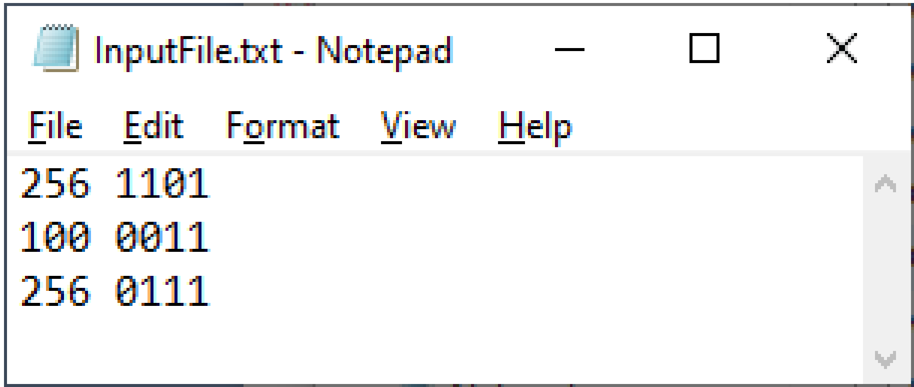
```
FileReadProcess:process
    file InputFile : text;
    variable lineBuffer : line;
    variable Char : character;
    variable int : integer;
    variable vector : std_logic_vector(3 downto 0):= "0000";
begin
    file_open(InputFile, "InputFile.txt",read_mode);

    while not endfile(InputFile) loop
        readline(InputFile, lineBuffer);
        read(lineBuffer, int);
        read(lineBuffer, Char);
        read(lineBuffer, vector);
```



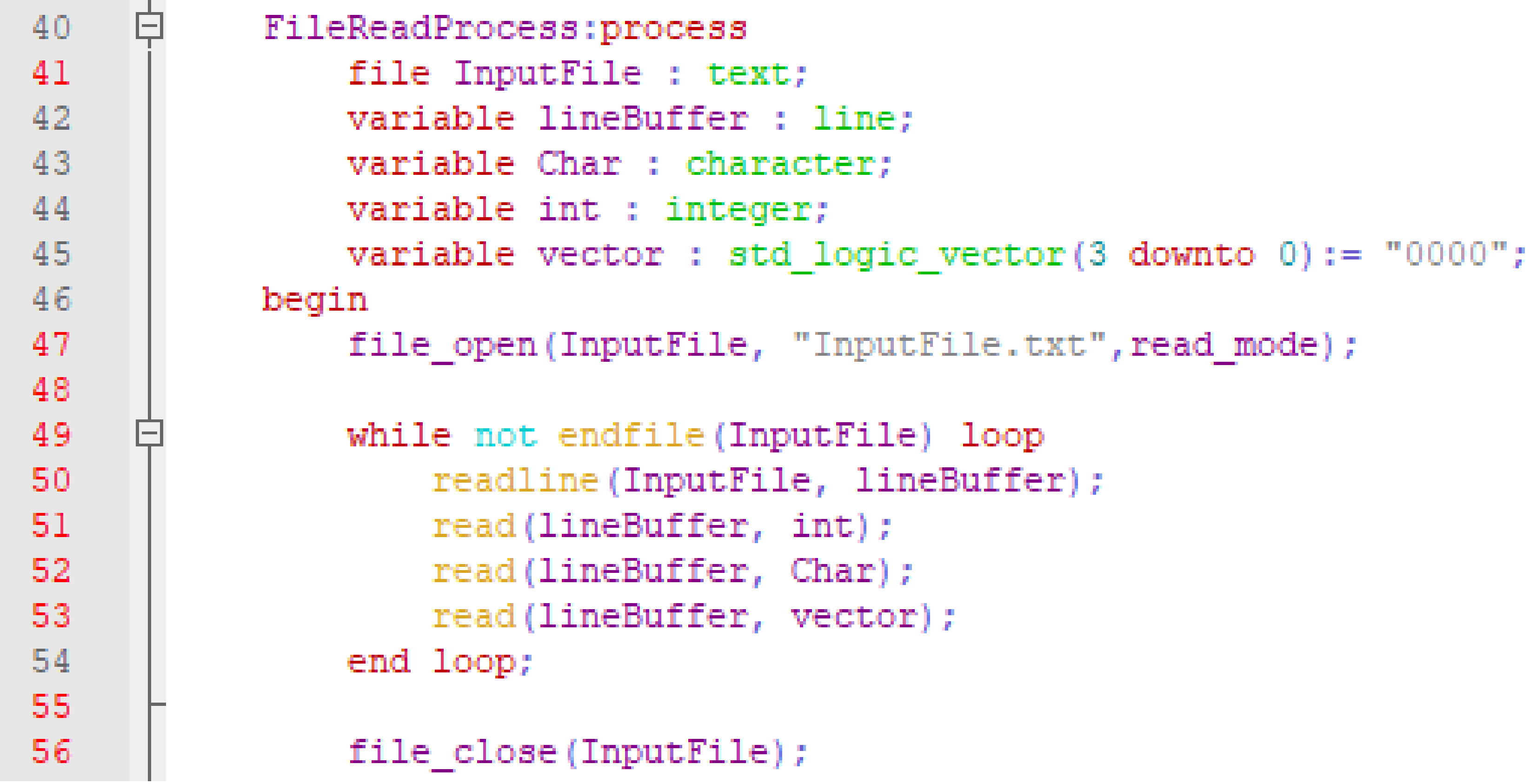
```
256 1101
100 0011
256 0111
```



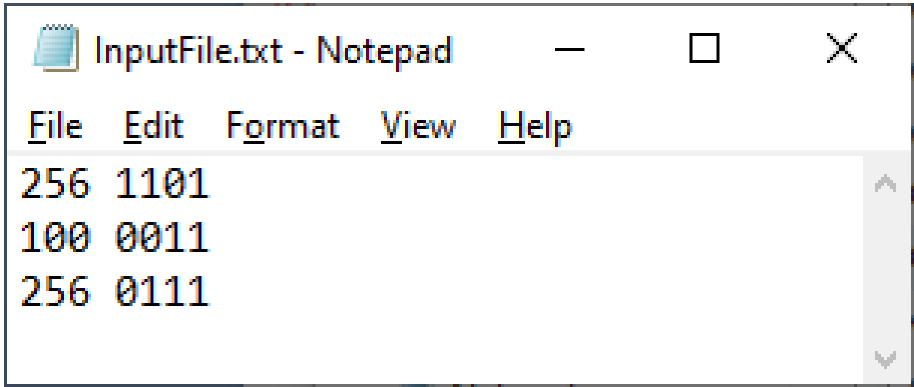
```
    FileReadProcess:process
        file InputFile : text;
        variable lineBuffer : line;
        variable Char : character;
        variable int : integer;
        variable vector : std_logic_vector(3 downto 0):= "0000";
    begin
        file_open(InputFile, "InputFile.txt",read_mode);

        while not endfile(InputFile) loop
            readline(InputFile, lineBuffer);
            read(lineBuffer, int);
            read(lineBuffer, Char);
            read(lineBuffer, vector);
        end loop;
```



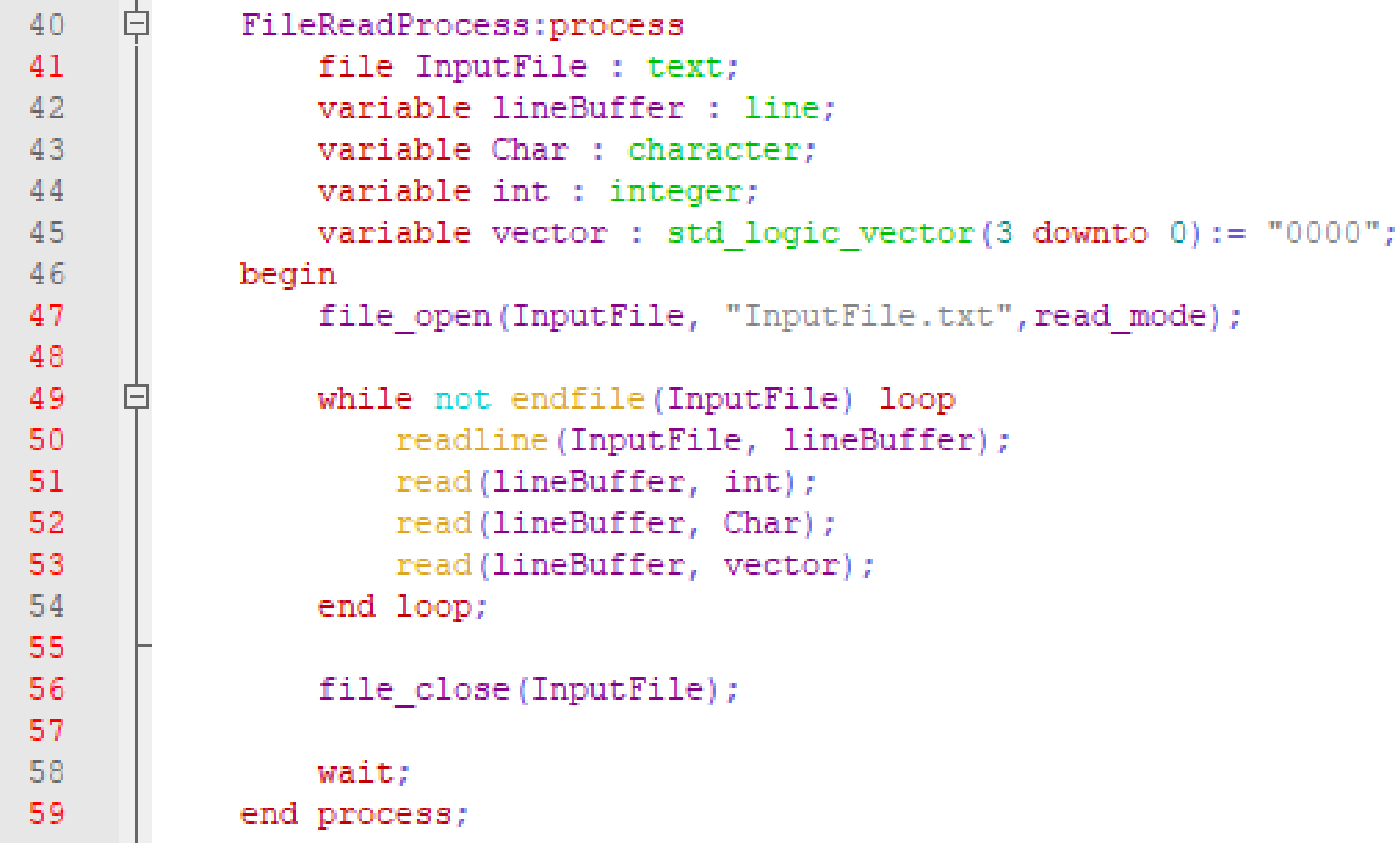
```
256 1101
100 0011
256 0111
```



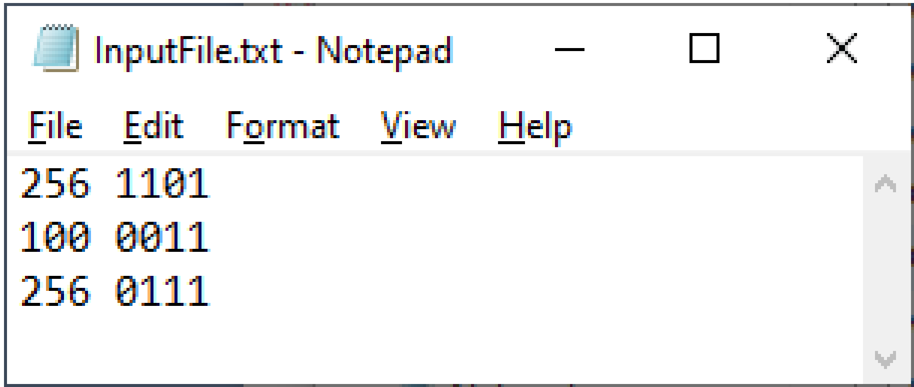
```
    FileReadProcess:process
        file InputFile : text;
        variable lineBuffer : line;
        variable Char : character;
        variable int : integer;
        variable vector : std_logic_vector(3 downto 0):= "0000";
    begin
        file_open(InputFile, "InputFile.txt",read_mode);

        while not endfile(InputFile) loop
            readline(InputFile, lineBuffer);
            read(lineBuffer, int);
            read(lineBuffer, Char);
            read(lineBuffer, vector);
        end loop;

        file_close(InputFile);
```



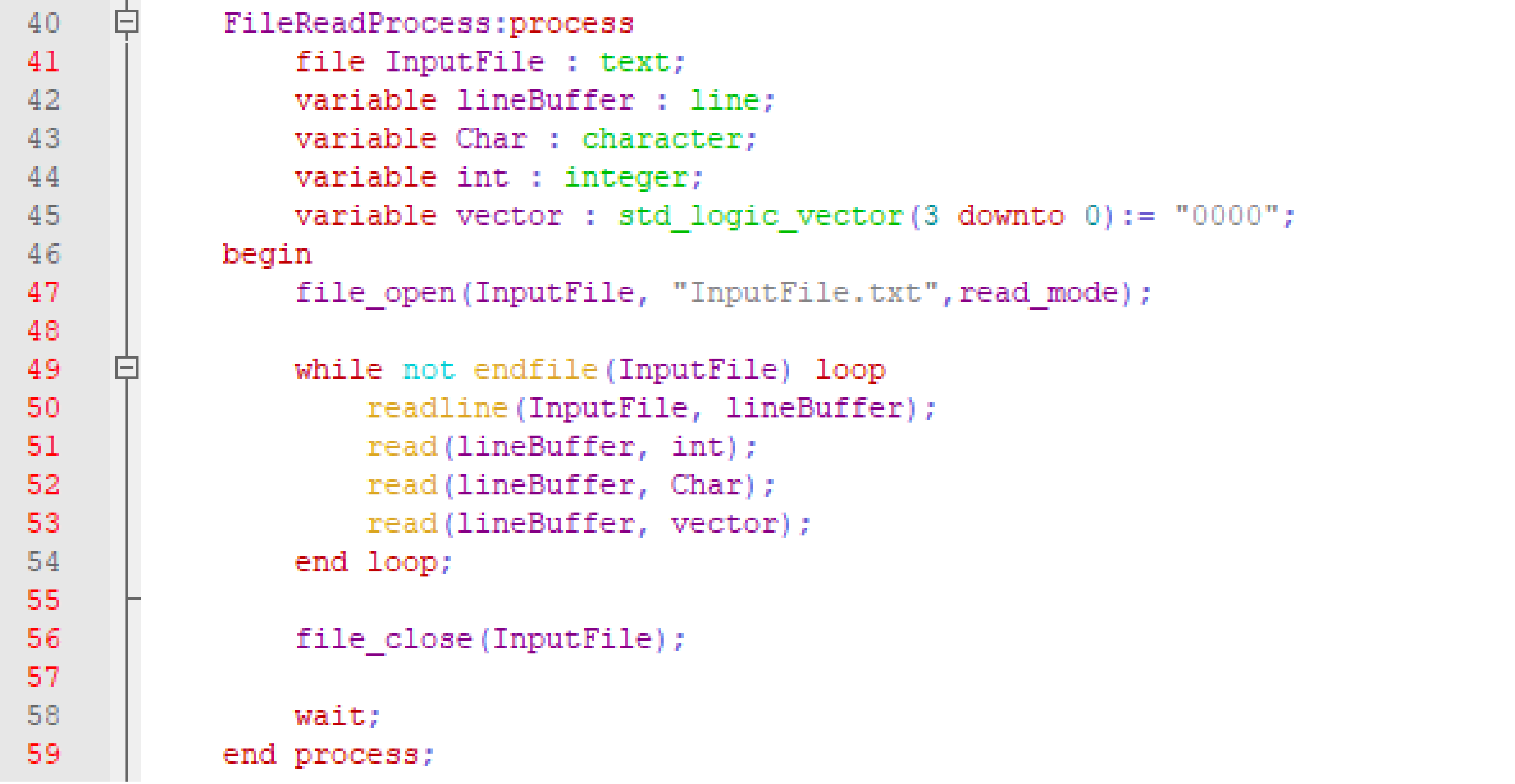
```
256 1101
100 0011
256 0111
```



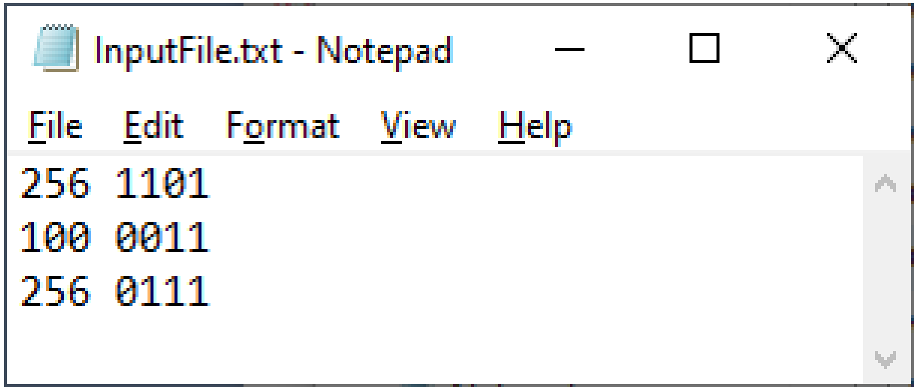
```
    FileReadProcess:process
        file InputFile : text;
        variable lineBuffer : line;
        variable Char : character;
        variable int : integer;
        variable vector : std_logic_vector(3 downto 0):= "0000";
    begin
        file_open(InputFile, "InputFile.txt",read_mode);

        while not endfile(InputFile) loop
            readline(InputFile, lineBuffer);
            read(lineBuffer, int);
            read(lineBuffer, Char);
            read(lineBuffer, vector);
        end loop;

        file_close(InputFile);

        wait;
    end process;
```



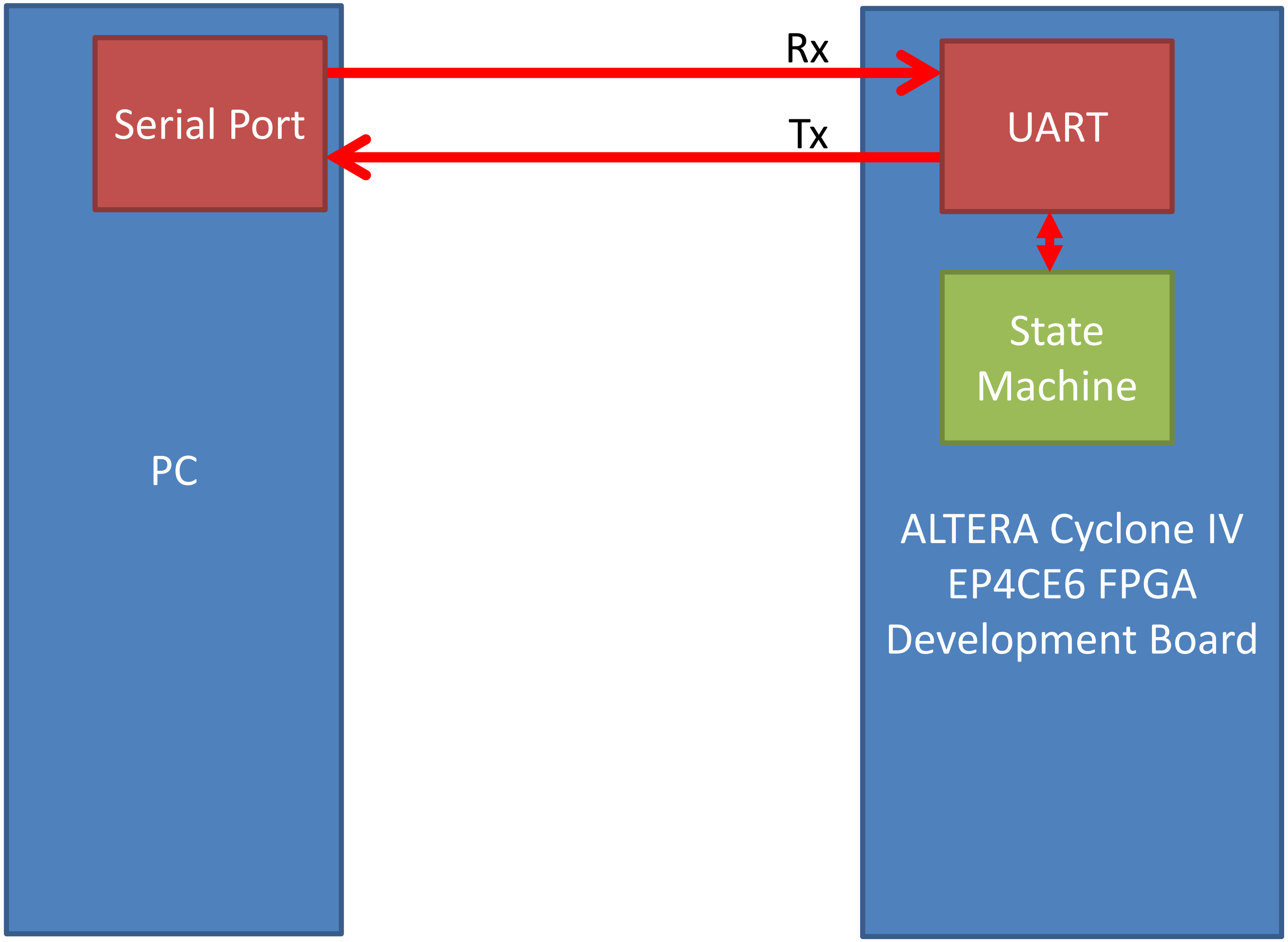
```
256 1101
100 0011
256 0111
```

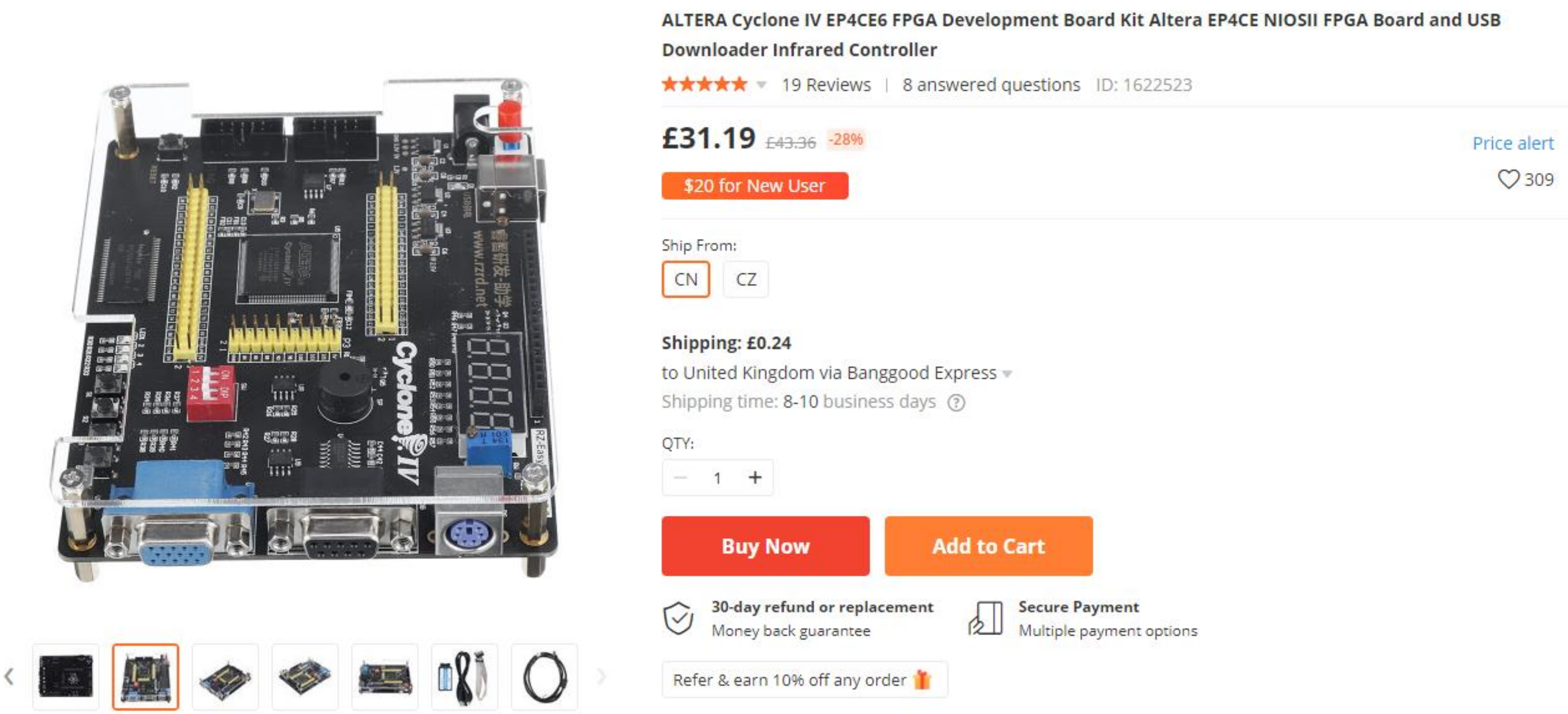


```
    FileReadProcess:process
        file InputFile : text;
        variable lineBuffer : line;
        variable Char : character;
        variable int : integer;
        variable vector : std_logic_vector(3 downto 0):= "0000";
    begin
        file_open(InputFile, "InputFile.txt",read_mode);

        while not endfile(InputFile) loop
            readline(InputFile, lineBuffer);
            read(lineBuffer, int);
            read(lineBuffer, Char);
            read(lineBuffer, vector);
        end loop;

        file_close(InputFile);

        wait;
    end process;
```



```
256 1101
100 0011
256 0111
```


Serial Port
PC
Rx
Tx
UART
State Machine
ALTERA Cyclone IV EP4CE6 FPGA Development Board



**ALTERA Cyclone IV EP4CE6 FPGA Development Board Kit Altera EP4CE NIOSII FPGA Board and USB Downloader Infrared Controller**

★★★★★ 19 Reviews | 8 answered questions ID: 1622523

**£31.19** ~~£43.36~~ -28%

$20 for New User

Ship From: CN CZ

**Shipping: £0.24**

to United Kingdom via Banggood Express

Shipping time: 8-10 business days

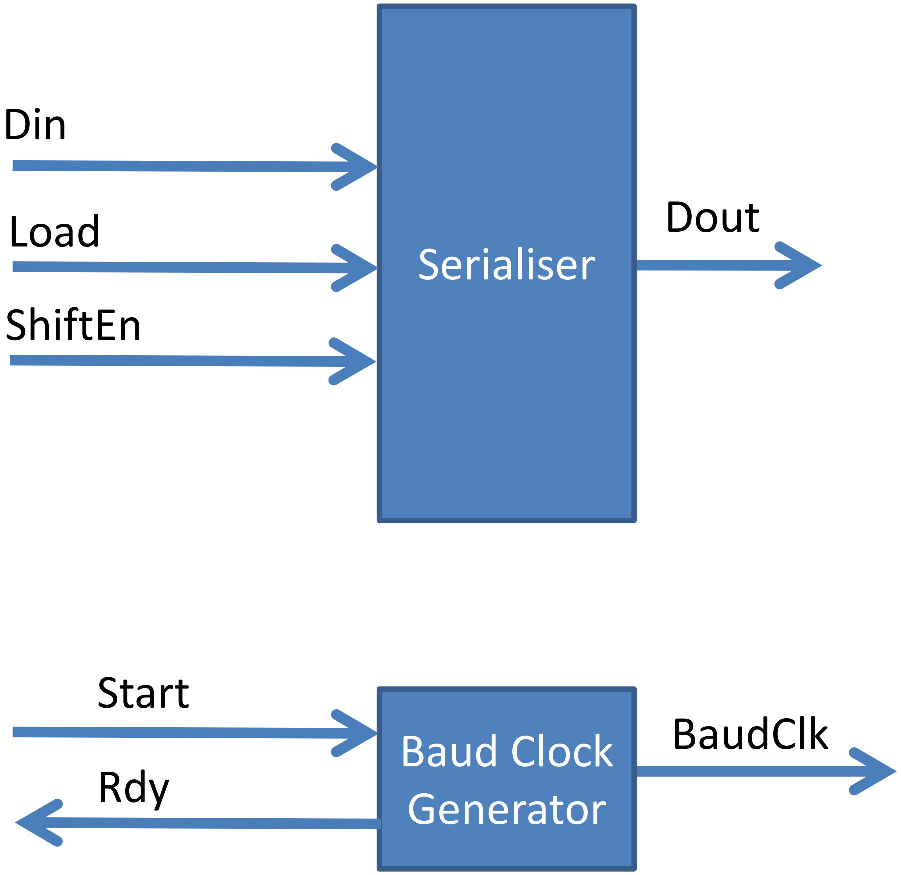
QTY: 1

**30-day refund or replacement**
Money back guarantee

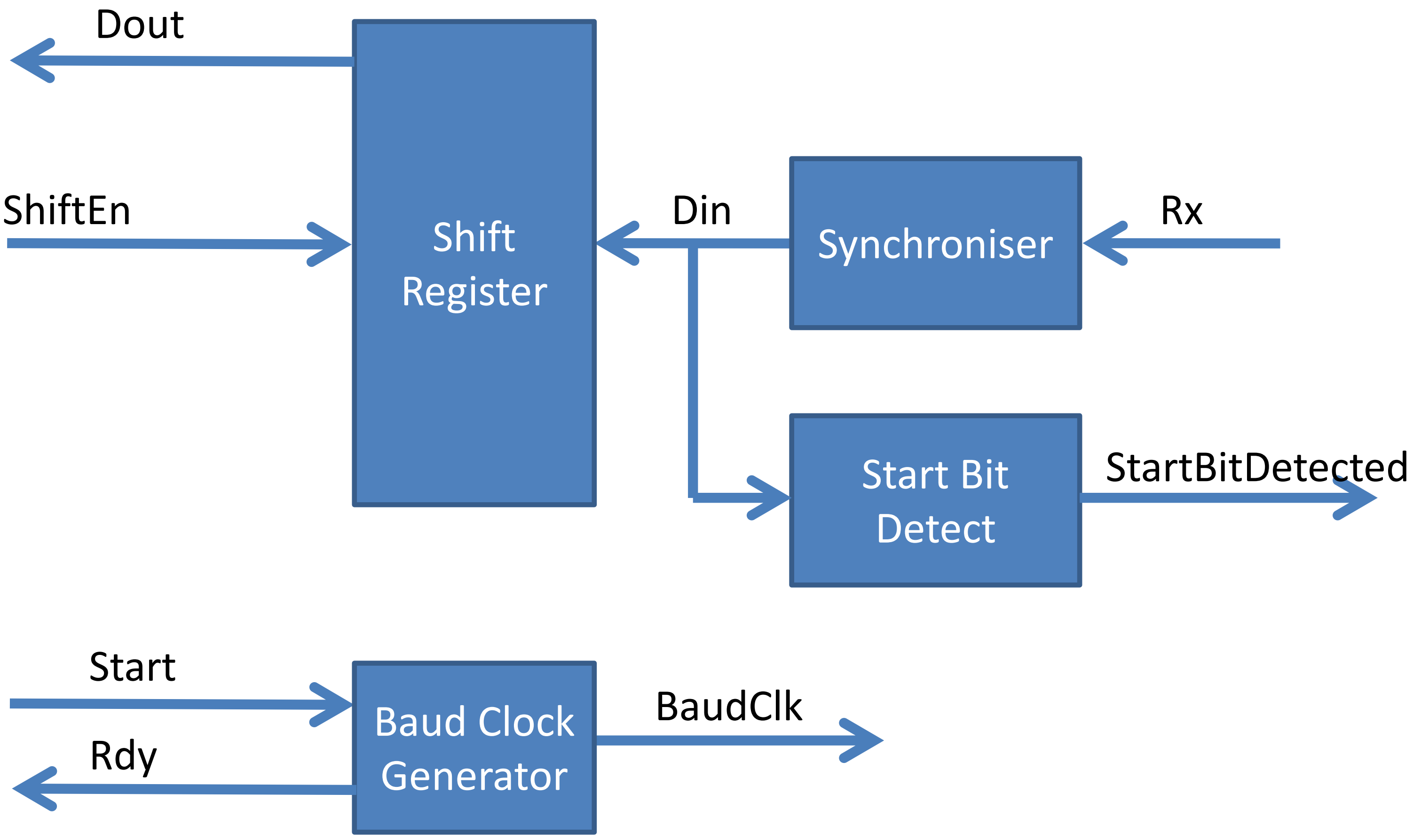
**Secure Payment**
Multiple payment options

Refer & earn 10% off any order

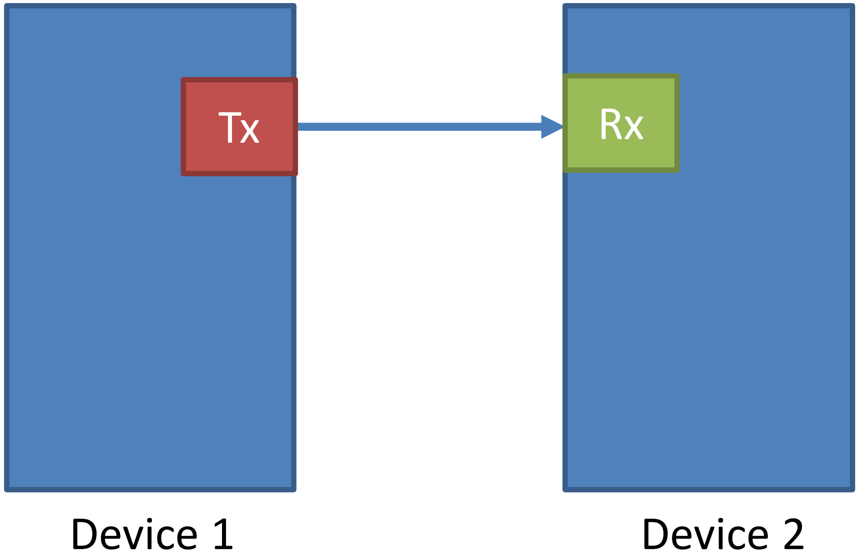
# UART Transmitter



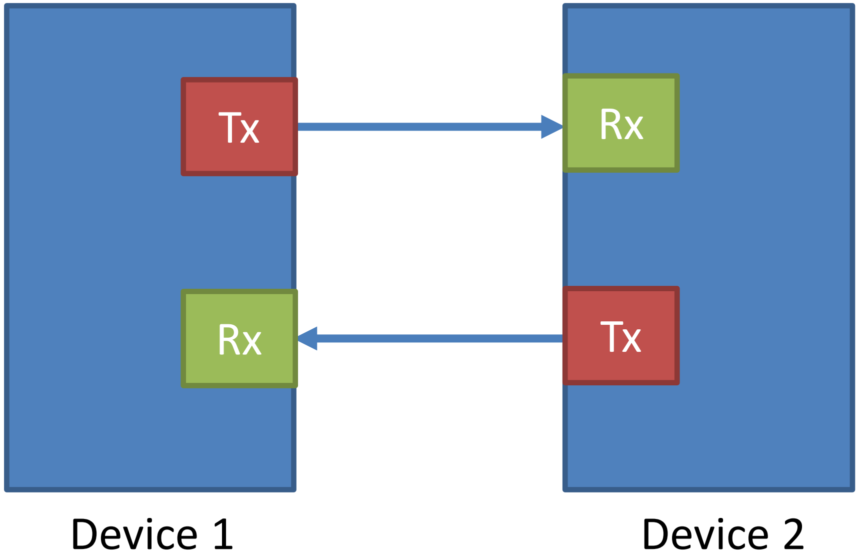
# UART Receiver



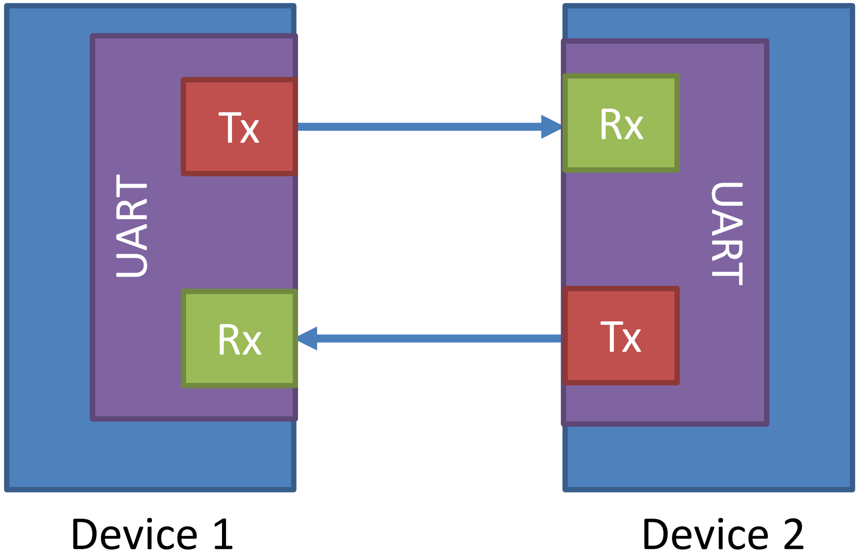
# RS232 Protocol



# RS232 Protocol



# RS232 Protocol



# RS232 Protocol

| Start Bit | Data Bits | Parity Bit | Stop Bit(s) |
|---|---|---|---|

Start Bit = Supports 1 start bit. Logic 0.
Data Bits = Supports 5 to 9 data bits
Parity Bit = Optional (Even or Odd Parity)
Stop Bits = Supports 1, 1.5 or 2 stop bits. Logic 1.

# RS232 Protocol

| Start Bit | Data Bits | Parity Bit | Stop Bit(s) |
|---|---|---|---|

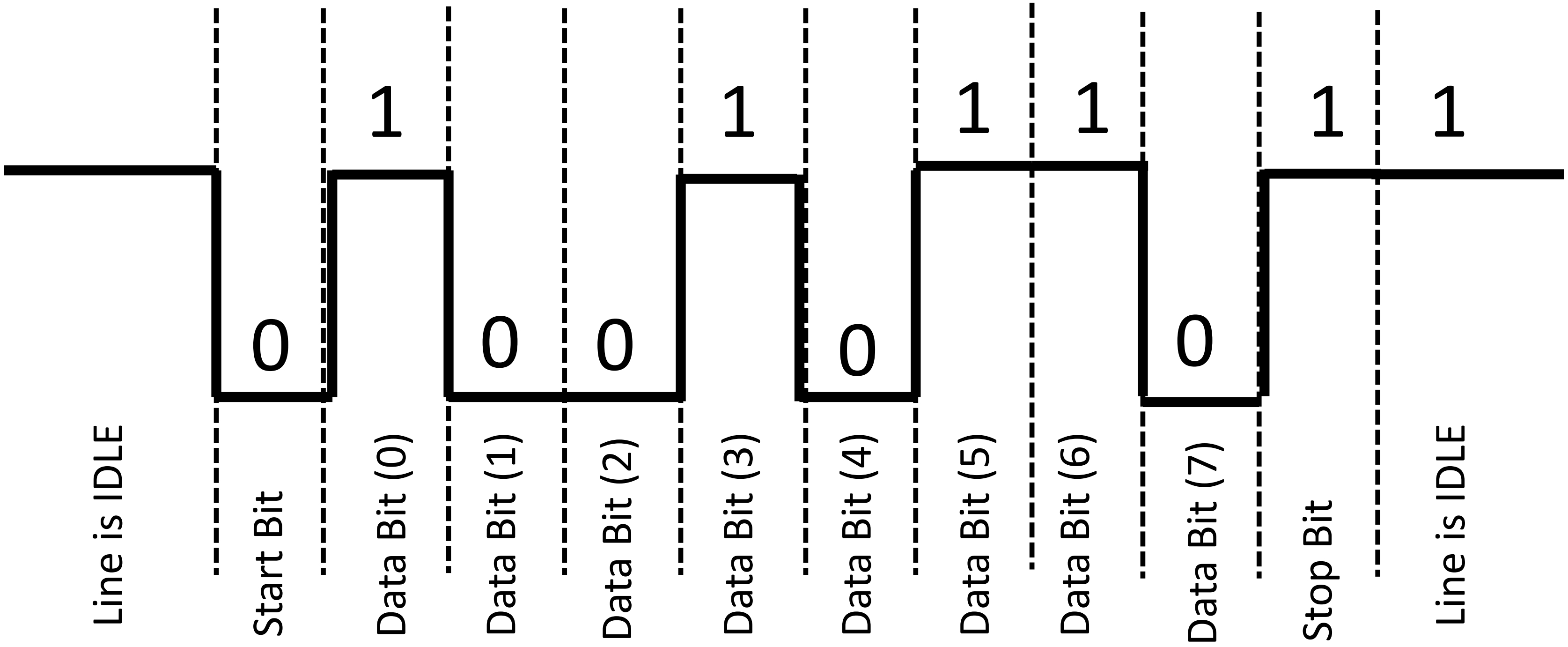
Start Bit = Supports 1 start bit. Logic 0.
Data Bits = Supports 5 to 9 data bits
Parity Bit = Optional (Even or Odd Parity)
Stop Bits = Supports 1, 1.5 or 2 stop bits. Logic 1.

# RS232 Protocol

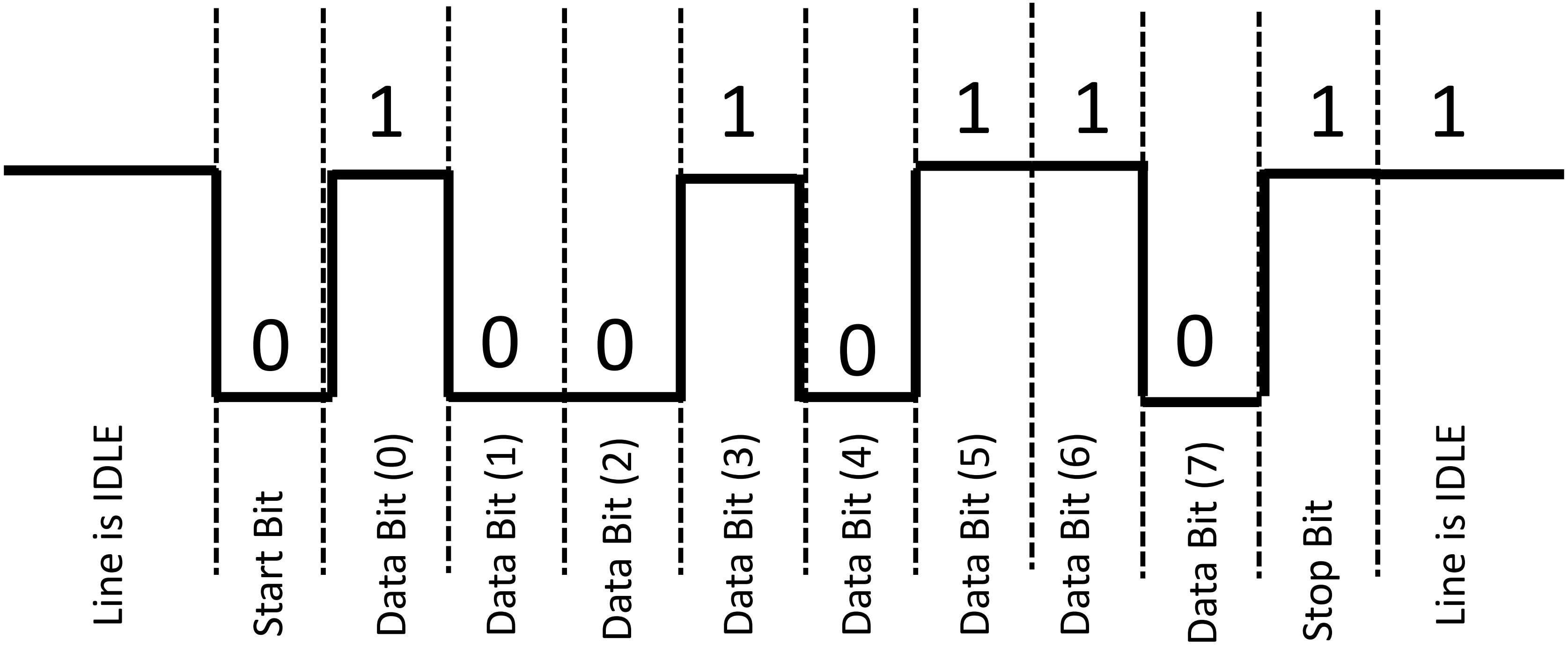
| Start Bit | Data Bits | Parity Bit | Stop Bit(s) |
|---|---|---|---|

Start Bit = Supports 1 start bit. Logic 0.
Data Bits = Supports 5 to 9 data bits
Parity Bit = Optional (Even or Odd Parity)
Stop Bits = Supports 1, 1.5 or 2 stop bits. Logic 1.

# Transmit Data Byte 0x69



# Transmit Data Byte 0x69



# Buad Rates

- 115200
- 57600
- 38400
- 19200
- 14400
- 9600

# Buad Rates

- 115200 (Bit period = 1/115200 = 8.7us)
- 57600
- 38400
- 19200
- 14400
- 9600