Architecture des Ordinateurs I

Part I: VHDL and Logic Design
The Language VHDL

Paolo.Ienne@epfl.ch
EPFL – I&C – LAP

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Recommended Books:

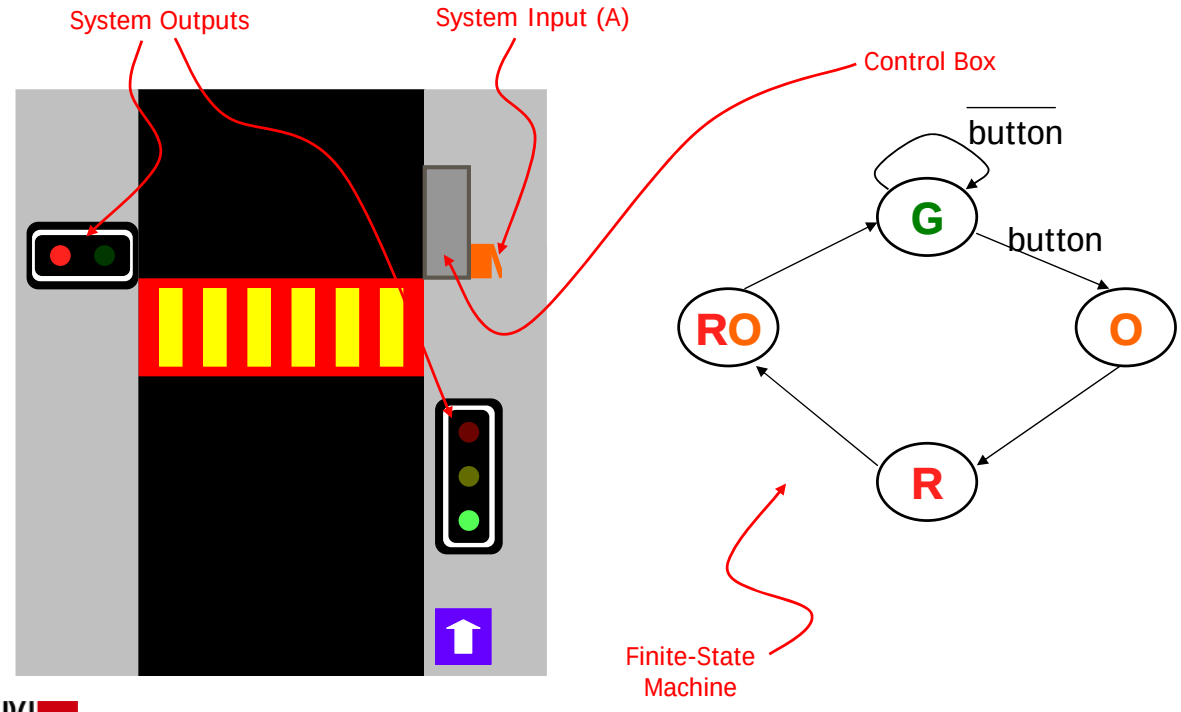
- John F. Wakerly
  Digital design (3rd edition)
  Prentice Hall, 2001
- Peter J. Ashenden
  The designer's guide to VHDL (2nd edition)
  Morgan Kaufmann, 2001
- Peter J. Ashenden
  The student's guide to VHDL
  Morgan Kaufmann, 1998
- James R. Armstrong and F. Gail Gray
  VHDL design: Representation and synthesis (2nd edition)
  Prentice Hall, 2000
- Jacques Weber et Maurice Meaudre
  VHDL: Du langage au circuit, du circuit au langage
  Masson, 1997
- Roland Airiau, Jean-Michel Bergé, Vincent Olive et Jacques Rouillard
  VHDL: Langage, modélisation, synthèse (2ème édition)
  PPUR, 1998

1

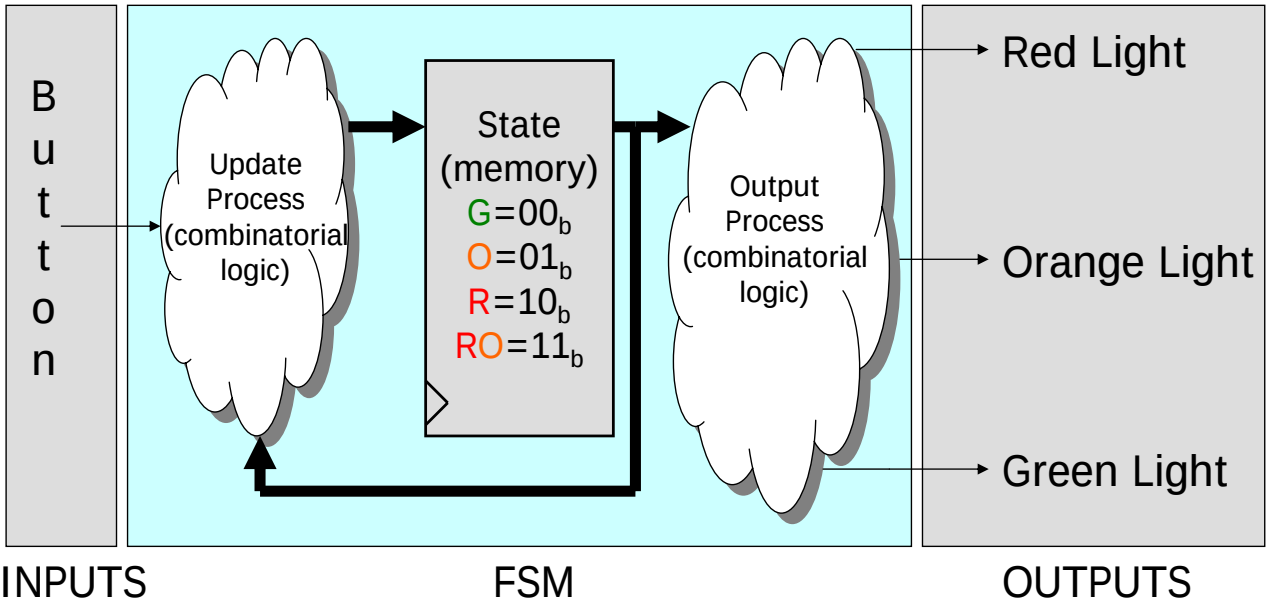
Introduction

A simple traffic-light controller

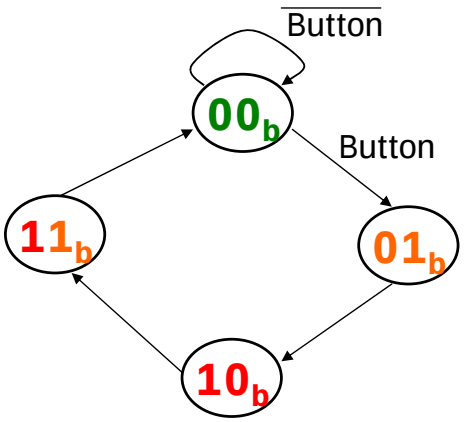
Example: A Traffic Light



Control Box



Classic Approach

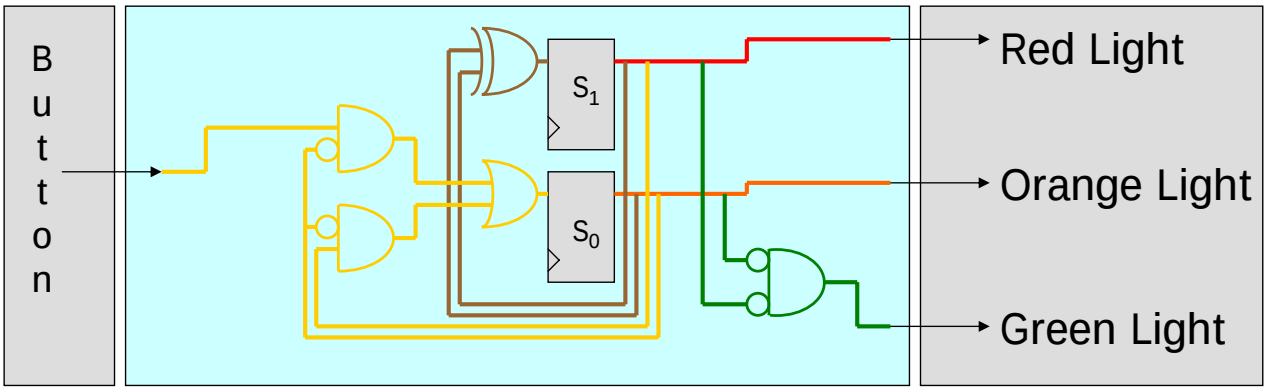


State	B	Next State	Green light	Orange light	Red light
00_b	0_b	00_b	1_b	0_b	0_b
00_b	1_b	01_b	1_b	0_b	0_b
01_b	X	10_b	0_b	1_b	0_b
10_b	X	11_b	0_b	0_b	1_b
11_b	X	00_b	0_b	1_b	1_b

Update process

Output process

Gate Level Implementation

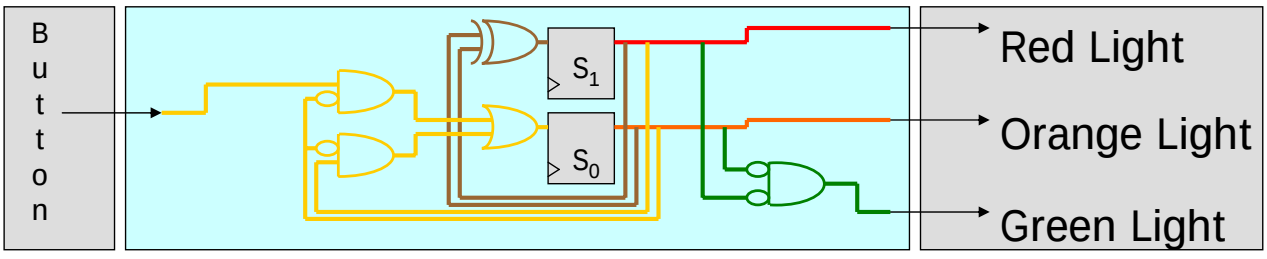


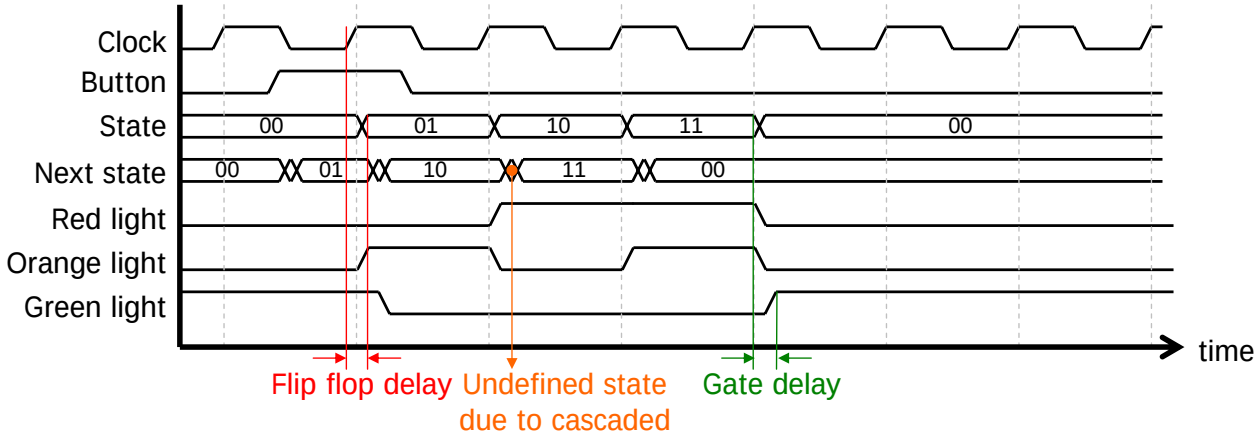
State	B	Next state
00	0	00
00	1	01
01	X	10
10	X	11
11	X	00

	Button			$\overline{\text{Button}}$
	S_1		$\overline{S_1}$	S_1
S_0	0	1	1	0
$\overline{S_0}$	1	0	0	1

	Button			$\overline{\text{Button}}$
	S_1		$\overline{S_1}$	S_1
S_0	0	0	0	0
$\overline{S_0}$	1	1	0	1

Signal Flow

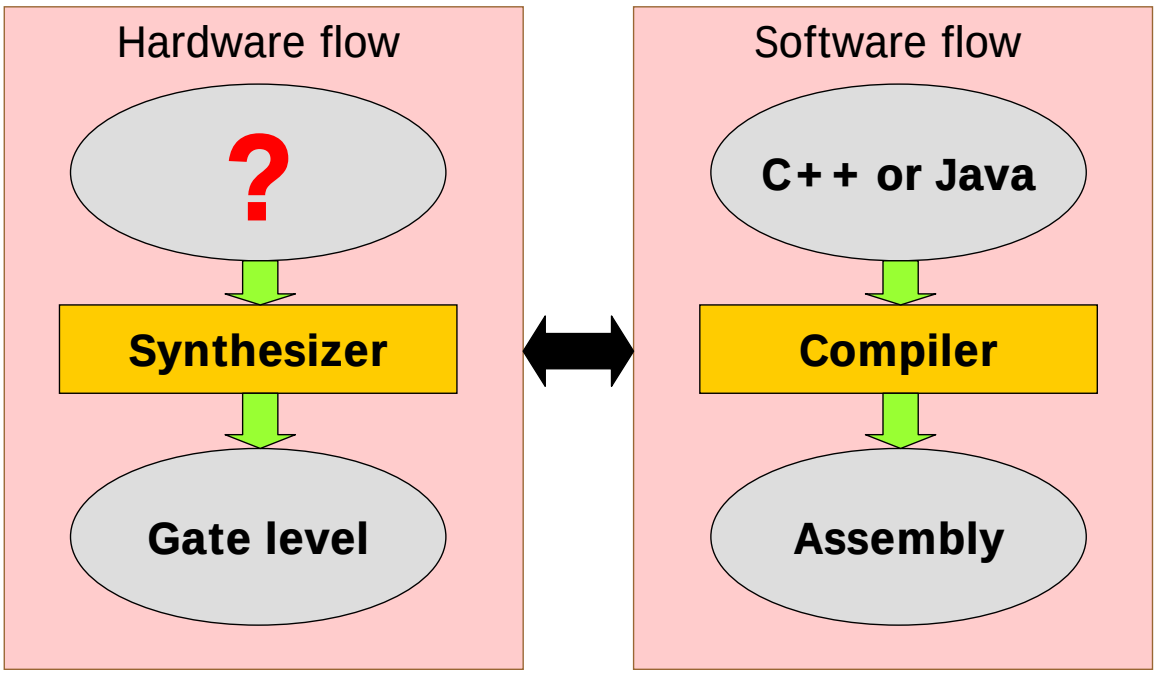




Is This the Way to Go?

- Design engineers used this method until about 1985
- For simple designs this method works, but as complexity increases new methodologies are required
- Compare gate-level designing with writing programs in assembly language
- We are not going to use gate-level design methods for 1 billion transistors!

Design Flow



?=VHDL

Very High-Speed Integrated Circuits (VHSIC)

Hardware **D**escription **L**anguage

- Formal language for specifying digital systems, equally good at structural as behavioral level
- Usage:
  - System description
  - Simulation
  - Conceptual modeling
  - Documentation
- Main characteristics:
  - Hierarchical
  - Event-driven simulation
  - Modular
  - Extensible
  - General language, strongly typed, similar to Ada

History

- 1980:
  Beginning of the project, financed by DoD (400M $US)
- 1982:
  Contracts for Intermetrics, IBM et Texas
- 1985:
  Version 7.2 released public domain
- 1987:
  Standard IEEE 1076 (VHDL-87)
- 1993:
  New version of the standard (VHDL-93)
- 2001:
  New version of the standard (VHDL-2001)

In This Course...

- VHDL is a very rich language and provides syntactical elements for describing:
  - Synthesizable digital systems
  - Functional description of complex components (processors, DSPs, etc.)
  - Description of libraries of elementary gates with internal delays rise and fall times, etc.
  - ...
- We will only use a subset, namely the description of *synthesizable zero-delay digital systems*

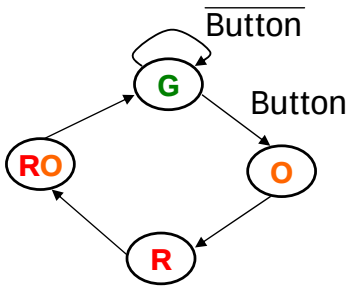
Simulation vs. Synthesis

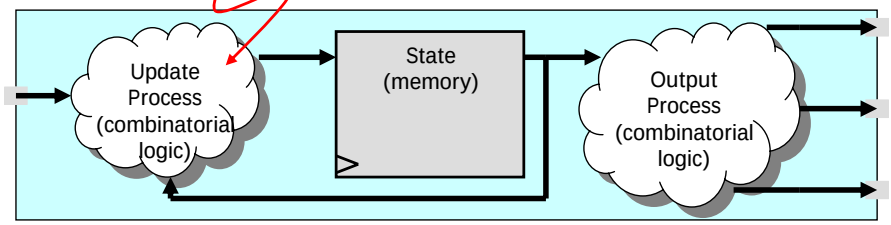
- **Simulation:** It is the process of verifying the proper functionality of the VHDL description of a system (similar to **executing** a program in C++ or Java)
- **Synthesis:** It is the process of translating the VHDL description of a system into simple gates (similar to **compiling** a program in C++ or Java)

Traffic Light in VHDL

- By using VHDL we can “forget” Boolean algebra, we can describe the functionality
- In our FSM, **`next_state`** is a function of **`state`** and **`button`**

```
update_process : PROCESS(state,button)

BEGIN
   IF (state = green) THEN
      IF (button = '1') THEN next_state <= orange;
                        ELSE next_state <= green;
      END IF;
   ELSIF (state = orange) THEN next_state <= red;
   ELSIF (state = red) THEN next_state <= red_orange;
                       ELSE next_state <= green;
   END IF;
END PROCESS update_process;
```

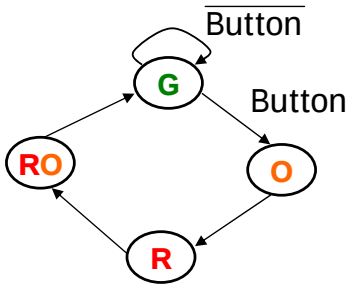


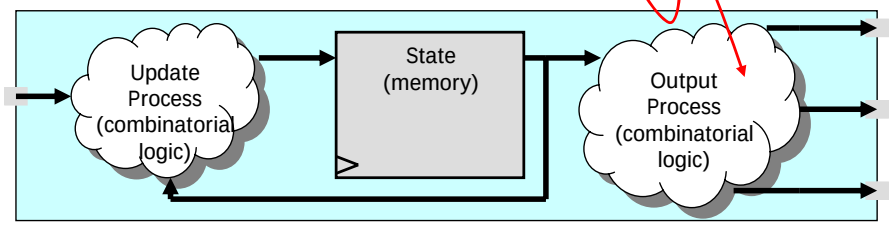


Traffic Light in VHDL

- The outputs are a function of only the current **`state`** (Moore FSM)

```
output_process : PROCESS (state)

BEGIN
   IF (state = green) THEN Green_Light <= '1';
                      ELSE Green_Light <= '0';
   END IF;
   IF (state = orange OR
       state = red_orange) THEN Orange_Light <= '1';
                           ELSE Orange_Light <= '0';
   END IF;
   IF (state = red OR
       state = red_orange) THEN Red_Light <= '1';
                           ELSE Red_Light <= '0';
   END IF;
END PROCESS output_process;
```

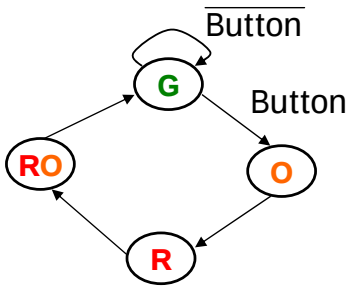


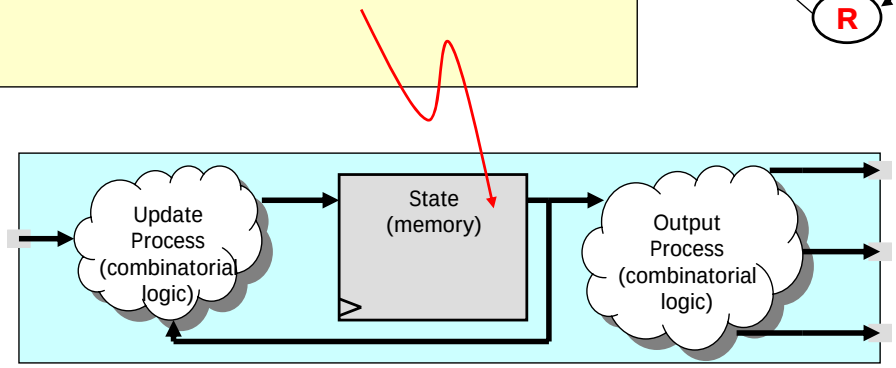


Traffic Light in VHDL

- And finally, the “memory” of the state has to be modelled
- Note that the new `state` only depends on `next_state` and `clk`

```
state_process : PROCESS (clk, next_state)

BEGIN
   IF (clk'event AND clk='1') THEN state <= next_state;
   END IF;
END PROCESS state_process;
```

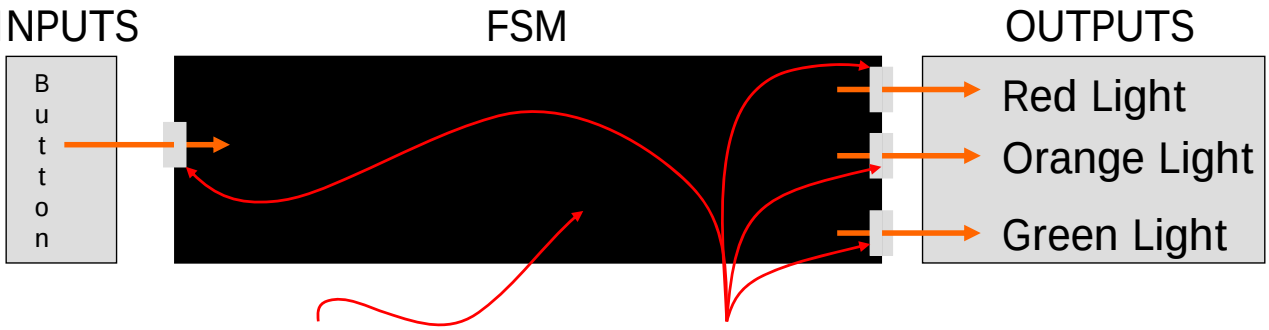




Basic Syntactical Rules

- VHDL is a **case insensitive** language
- VHDL has a free formatting
- Each command sequence should be **terminated by a “;”**
- Remarks can be placed by using **“--” in front of the remark**
- A remark terminates at the end of the line

Entity



The "black box" is called **ENTITY** in VHDL

The signals going into, or coming from the **ENTITY** are called ports, and are described in the **PORT** section

```
ENTITY fsm IS
   PORT ( B  : IN  std_logic;  -- Button input
          RL : OUT std_logic;  -- Red light output
          OL : OUT std_logic;  -- Orange light output
          GL : OUT std_logic); -- Green light output
END fsm;
```


Syntax of the Entity

Each ENTITY may have a library definition

We will always use the ones described here
(similar to **`#include <stdlib.h>`** in C/C++)

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

ENTITY <entity_name> IS
   PORT ( … );
END <entity_name>;
```

Each **ENTITY** may have a port section

Each **ENTITY** requires a unique name

Syntax of the Port Section

Each **PORT** requires a unique name

Each **PORT** requires a signal type

```
PORT ( <port_name> : <port_type> <signal_type>;
       <port_name> : <port_type> <signal_type>;
                          .
                          .
       <port_name> : <port_type> <signal_type>;
       <port_name> : <port_type> <signal_type>
     );
```

IMPORTANT NOTE:
The last port in the **PORT** section is not terminated by a ;

Each **PORT** requires a type, which can be:

IN => The port is an input port (Read Only)

OUT => The port is an output port (Write Only)

INOUT => The port is bidirectional (Read and Write)

We will seldom (maybe never...) use **INOUT** ports

(Signal) Types in VHDL

- VHDL knows various types, like:
  - Real
  - Integer
  - Time
  - ...

We will seldom (or never...) use standard types

- In this course we are only going to use the following VHDL types:
  - **STD_LOGIC** → This is the type holds a one bit quantity (see it as being one wire)
  - **STD_LOGIC_VECTOR ((n-1) DOWNTO 0)** → This type holds a set of n-bits (see it as being a collection of *n* wires)
- Furthermore we are going to use own defined types, which can come in handy in FSMs (as we will later see):
  - **TYPE state_type IS (Green, Orange, Red, RedOrange)**
  - **TYPE hex_type IS (1,2,3,4,5,6,7,8,9,A,B,C,D,E,F)**

STD_LOGIC and STD_LOGIC_VECTOR

- The **STD_LOGIC** and each element (“bit”) of the **STD_LOGIC_VECTOR** can hold nine values:
  - ‘U’ = uninitialized
  - ‘X’ = forcing unknown
  - ‘0’ = forcing 0
  - ‘1’ = forcing 1
  - ‘Z’ = high impedance
  - ‘W’ = weak unknown
  - ‘L’ = weak 0 (pull-down)
  - ‘H’ = weak 1 (pull-up)
  - ‘-’ = don’t care
- In this course we will **ONLY** use and consider the values ‘U’, ‘X’, ‘0’, ‘1’ and ‘Z’ !

Our Entity

We of course need also a clock input for our memory elements

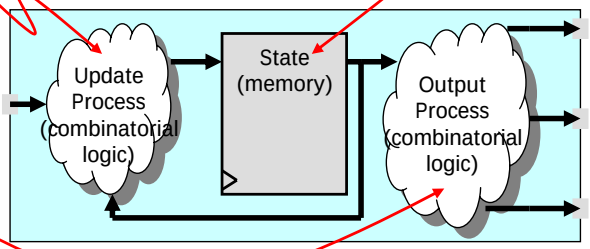
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY fsm IS
   PORT ( CLK          : IN  std_logic;  -- Clock input
          Button       : IN  std_logic;
          Red_Light    : OUT std_logic;
          Orange_Light : OUT std_logic;
          Green_Light  : OUT std_logic);
END fsm;
```

BUT: Where is the functionality of this black-box?

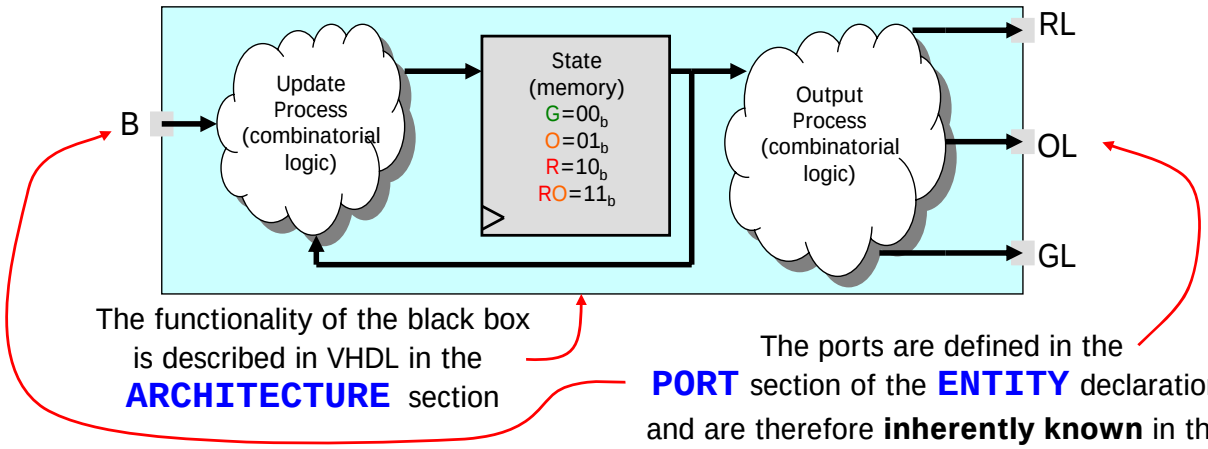
Functionality

```
ARCHITECTURE functional_level OF fsm IS

   TYPE fsm_state_t IS (green,orange,red,red_orange);
   SIGNAL state      : fsm_state_t;
   SIGNAL next_state : fsm_state_t;

BEGIN
update_process : PROCESS(state,button)
BEGIN
   IF (state = green) THEN
      IF (button = '1') THEN next_state <= orange;
                        ELSE next_state <= green;
      END IF;
   ELSIF (state = orange) THEN next_state <= red;
   ELSIF (state = red) THEN next_state <= red_orange;
                       ELSE next_state <= green;
   END IF;
END PROCESS update_process;

output_process : PROCESS (state)
BEGIN
   IF (state = green) THEN Green_Light <= '1';
                      ELSE Green_Light <= '0';
   END IF;
   IF (state = orange OR
       state = red_orange) THEN Orange_Light <= '1';
                           ELSE Orange_Light <= '0';
   END IF;
   IF (state = red OR
       state = red_orange) THEN Red_Light <= '1';
                           ELSE Red_Light <= '0';
   END IF;
END PROCESS output_process;
```

```
state_process : PROCESS (clk, next_state)
BEGIN
   IF (clk'event AND clk='1') THEN state <= next_state;
   END IF;
END PROCESS state_process;

END functional_level;
```



Architecture

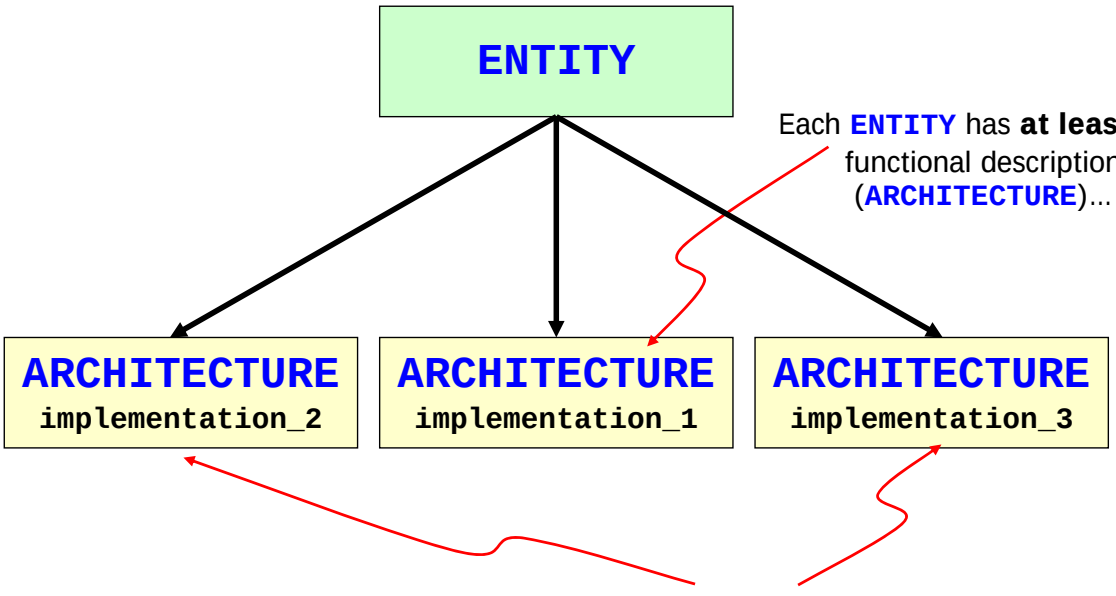


The functionality of the black box is described in VHDL in the **ARCHITECTURE** section

The ports are defined in the **PORT** section of the **ENTITY** declaration, and are therefore **inherently known** in the **ARCHITECTURE**

```
ARCHITECTURE implementation_1 OF fsm IS
…
BEGIN
…
END implementation_1;
```

Architecture and Entity Relationship



Syntax of the Architecture

Each **ARCHITECTURE** definition can contain declarations (Similar: `int loop;` in C++)

Each **ARCHITECTURE** section is **referenced** to it's corresponding **ENTITY** definition **by the name** of this **ENTITY**

```
ARCHITECTURE <implementation_name> OF <entity_name> IS

  [Declaration Section]

BEGIN
  [Body]
END <implementation_name>;
```

The body of the **ARCHITECTURE** contains the actual functionality

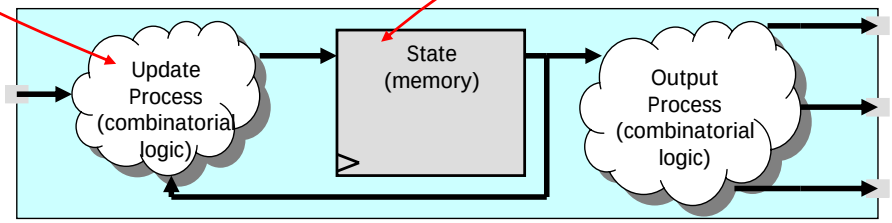
Each **ARCHITECTURE** section has it's own unique implementation name

Declaration Section

- The declaration section of an **ARCHITECTURE** can contain:
  - SIGNAL declarations. Signals are the “wires” or “state” of the circuit.
  - CONSTANT definitions. Constants are fixed value “wires” or fixed value “state” within the circuit.
  - COMPONENT declarations. Components give us the possibility to design hierarchical.
  - FUNCTION definitions. Functions can be used for actions which are often required in the functional description
  - PROCEDURE definitions. Similar to Functions also procedures can be used.
- In this course we will only use the signal, constant, and eventually the component.
- Each of this topics will be defined later on in this course.

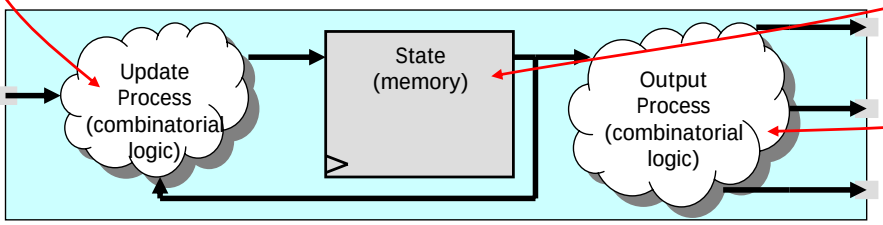
Body of an Architecture

- The body of an architecture consists of:
  - Implicit processes
  - Explicit processes
  - Component instantiations
- Each of these topics will be used in this course and will be introduced later on
- Each of these parts execute in parallel (in contrast with software programming languages where instructions are executed sequentially!)



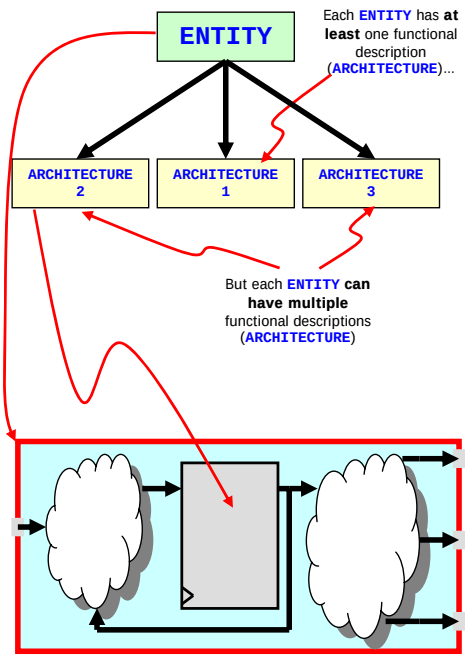
Creation of the Body Structure

- Most VHDL code written and used nowadays by designers is written following the **RTL (=Register Transfer Level)** principle. In this course we will only design using this principle.
- The RTL principle consists of separating the sequential processes (e.g., memory, flip-flops and latches) from the combinatorial processes. Finally we connect the different processes by wires.
- How do we do this:
  - Identify the combinatorial and sequential elements.
  - Write the VHDL code for all these elements.
  - Control that no memory action is introduced in the combinatorial elements (to be elaborated later on in this course).
  - Connect the elements by using wires (e.g., signals).



Summary

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

ENTITY <entity_name> IS
   PORT ( <port_name> : <port_type> <signal_type>;
          <port_name> : <port_type> <signal_type>;
                              .
                              .
          <port_name> : <port_type> <signal_type>;
          <port_name> : <port_type> <signal_type>
        );
END <entity_name>;

ARCHITECTURE <implementation_name> OF <entity_name> IS

   [Declaration Section]

BEGIN
   [Body]
END <implementation_name>;
```



2

The body of an ARCHITECTURE

Signals as wires

Operators

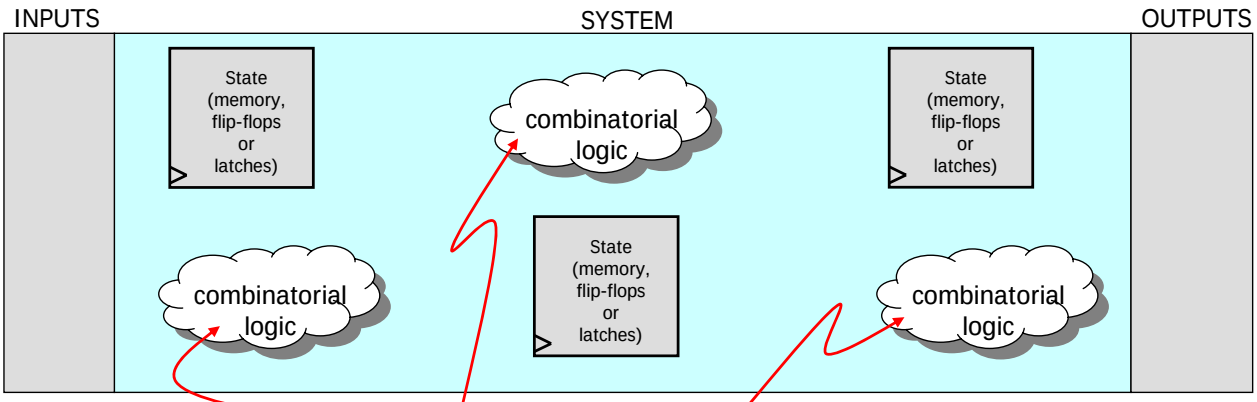
Statements

Events

Implicit processes

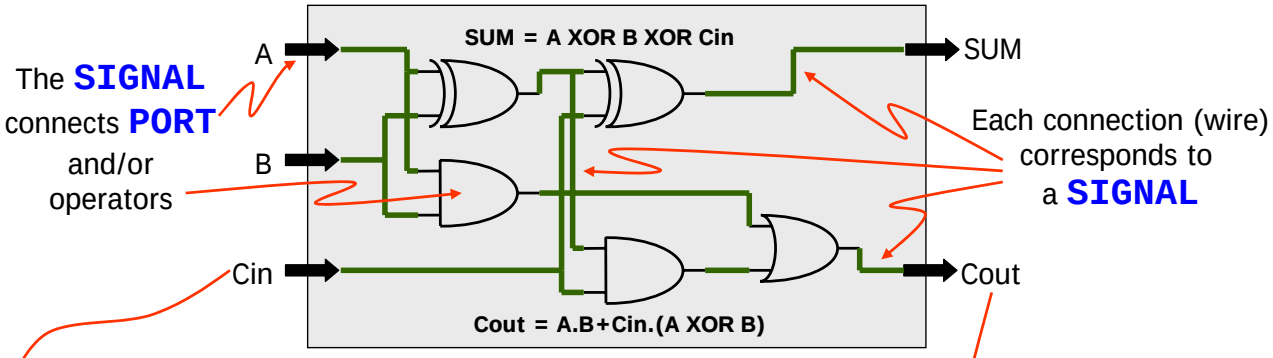
Remember RTL

- We will divide each system (ARCHITECTURE) into:
  - Combinatorial elements
  - Elementary sequential elements (memory, flip-flop and/or latches)
  - Interconnects (wires)
- Each combinatorial and sequential element form a PROCESS inside the ARCHITECTURE and all execute in parallel



Combinatorial Element: The Full Adder

- The Full Adder is a typical example of combinatorial logic



- Remember how the **entity** is defined:

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

ENTITY full_adder IS
   PORT ( A    : IN  std_logic;  -- A input
          B    : IN  std_logic;  -- B input
          Cin  : IN  std_logic;  -- Carry input
          SUM  : OUT std_logic;  -- Sum output
          Cout : OUT std_logic); -- Carry output
END full_adder;
```


Signal Syntax

Each **SIGNAL** has it's own unique name

Each **SIGNAL** has a signal type (see the previous lecture on these types)

```
SIGNAL <signal_name>[, <signal_name>,…] : <signal_type>;
```

And all **SIGNAL**'s are defined in the declaration section of the **ARCHITECTURE**

It is good practice, but not required, to prefix all signal names with "s_"

```
ARCHITECTURE <implementation_name> OF <entity_name> IS

   [Declaration Section]

BEGIN
   [Body]
END <implementation_name>;
```

Signal Properties

- Each **SIGNAL** may and must be **assigned only once**. Compare with an electrical wire, if we connect a wire to two outputs, we create a short circuit and “blow the fuse”
- A **SIGNAL** is assigned a value by

```
<signal_name> <= <source>;
```

- Each **SIGNAL** can be **used multiple times**. We can connect a wire to one or multiple loads, of course
- Each **SIGNAL** is global to its and only its **ARCHITECTURE**
- A **SIGNAL** which is assigned with a constant value, can be declared in the declaration section of the **ARCHITECTURE** as a **CONSTANT**

```
CONSTANT <constant_name> : <signal_type> := <value>;
```

It is good practice, but not required, to prefix all constant names with “c_”

Some synthesizers do not allow for constant declaration, and thus we will seldom use the constant

Constant Signal Assignments

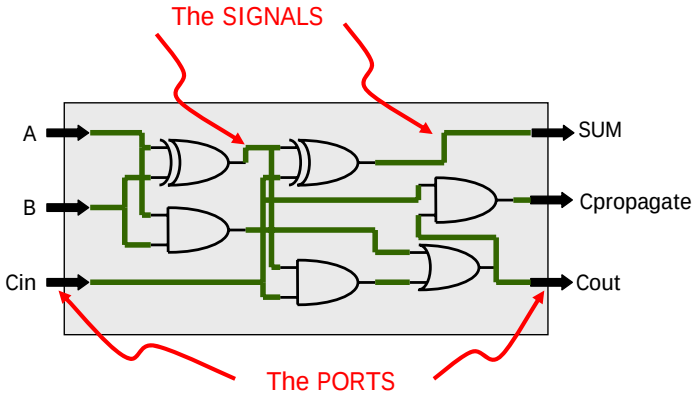
- We can assign a constant value to a signal of type **std_logic** by using **<signal_name> <= ’<value>’;**
  - Examples
    - `s_a <= ’0’;`
    - `s_a <= ’1’;`
    - `s_a <= ’X’;`
    - `s_a <= ’Z’;`
- We can assign a constant value to a signal of type **std_logic_vector** by using **<signal_name> <= ”<value>”;**
  - Examples

```
s_vect <= ”0101”;                  -- Assign binary value
s_vect <= X”F”;                    -- Assign hexadecimal value
s_vect <= to_stdlogicvector(-3,4); -- Assign integer –3 in 4
                                   -- bit representation
```

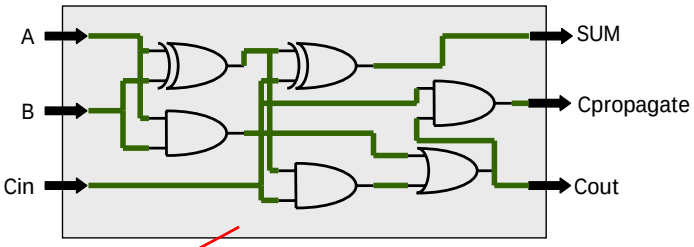
- We can use the macro **OTHERS** for signals of type **std_logic_vector**
  - Example
    - `s_vect <= ”0000”;` equals `s_vect <= (OTHERS => ’0’);`

Signals and Ports

- Each connection that is used in the body of the **ARCHITECTURE**, is called a **SIGNAL**
- **PORT**'s are defined in the **ENTITY** and can be used in the body of the **ARCHITECTURE**
- **PORT**'s have a slightly odd behavior:
  - Input **PORT**'s can **only** be **read**, but they **cannot** be **assigned** a value
  - Output **PORT**'s can **only** be **assigned** a value, but they **cannot** be **read**



Ports: A Typical Mistake




```
SUM        <= A XOR B XOR Cin;
Cout       <= (A AND B) OR (Cin AND (A XOR B));
Cpropagate <= Cin AND Cout;
```

NO

As Cout is an output port, it cannot be read! Hence this is not a correct implementation

```
SUM <= A XOR B XOR Cin;
s_x8 <= (A AND B) OR (Cin AND (A XOR B));
Cout <= s_x8;
Cpropagate <= s_x8 AND Cin;
```

We have to introduce an explicit signal s_x8, to make a correct implementation

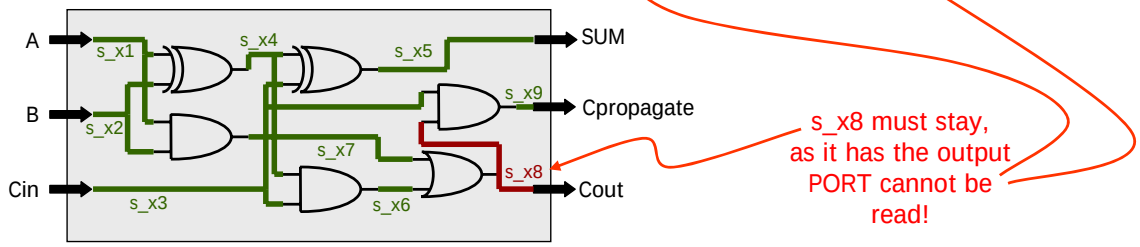
Different Implementations

Extensive signal usage:

```
s_x1 <= A;
s_x2 <= B;
s_x3 <= Cin;
s_x4 <= s_x1 XOR s_x2;
s_x5 <= s_x4 XOR s_x3;
SUM  <= s_x5;

s_x6 <= s_x1 AND s_x2;
s_x7 <= s_x3 AND s_x4;
s_x8 <= s_x6 OR s_x7;
Cout <= s_x8;

s_x9 <= s_x8 AND s_x3;
Cpropagate <= s_x9;
```

"Merging" of signals:

```
SUM <= A XOR B XOR Cin;


s_x8 <= (A AND B) OR (Cin AND (A XOR B));
Cout <= s_x8;


Cpropagate <= s_x8 AND Cin;
```



Operators

- To model the functionality of the system, VHDL provides several operators:
  - Logical:
    - `and or nand nor xor xnor not`
  - Comparison:
    - `= /= < <= > >=`
  - Concatenation:
    - `&`
  - Arithmetic:
    - `+ - * / mod rem`
  - Shifting:
    - `sll srl sla sra rol ror`
  - Sign:
    - `+ -`
  - Diverse:
    - `abs **`

← These operators we will use

← We will only use + and – of this set of operators

← These operators "make no sense" in std_logic and std_logic_vector types, as we can use the concatenation

← These operators we will not use

Sizes and Concatenation

- An operator can only work on signals of the same size

```
SIGNAL s_x1 : std_logic_vector(3 DOWNTO 0);
SIGNAL s_x2 : std_logic_vector(3 DOWNTO 0);
SIGNAL s_x3 : std_logic_vector(3 DOWNTO 0);
SIGNAL s_x4 : std_logic_vector(3 DOWNTO 0);
SIGNAL s_x5 : std_logic_vector(1 DOWNTO 0);
SIGNAL s_x6 : std_logic;

s_x3 <= s_x1 XOR s_x2; --This is correct as s_x1,s_x2 and s_x3 all contain 4 bits (wires)
s_x4 <= s_x1 OR s_x5;  --This is incorrect as s_x5 only contains 2 bits, whilst s_x1 and s_x4 contain 4
s_x6 <= s_x3 OR s_x4;  --This is incorrect as the result of s_x3 OR s_x4 is 4 bits, each operator does
                       --a bitwise operation (similar to | in C++; || does not exist in VHDL)
```

- We can increase the size of a signal by using the concatenation operator **&**, or the wire selection operation **( )**

```
SIGNAL s_x1 : std_logic_vector(3 DOWNTO 0);
SIGNAL s_x2 : std_logic_vector(3 DOWNTO 0);
SIGNAL s_x3 : std_logic_vector(3 DOWNTO 0);
SIGNAL s_x4 : std_logic_vector(3 DOWNTO 0);
SIGNAL s_x5 : std_logic_vector(1 DOWNTO 0);
SIGNAL s_x6 : std_logic;

s_x3 <= s_x1 XOR s_x2;          --This is correct as s_x1, s_x2 and s_x3 all contain 4 bits (wires)
s_x4 <= s_x1 OR ("00" & s_x5);  --This is correct as ("00" & s_x5), s_x1 and s_x4 all contain 4 bits
s_x6 <= s_x3(0) OR s_x4(2);     --This is correct as s_x6, s_x3(0) and s_x4(2) all contain 1 bit
```


Signed and Unsigned

- The types **std_logic** and **std_logic_vector** can hold signed, but also unsigned values, the synthesizer has no information on this
- To make clear to the synthesizer what implementation we want when we use arithmetic operators, we can use the type cast **signed(<signal_name>)** or **unsigned(<signal_name>)** in the **<source>** section (we cannot mix them!)
- Examples

Correct:

```
s_signed_add        <= signed( s_in1 )   + signed( s_in2);
s_unsigned_add      <= unsigned( s_in3 ) + unsigned( s_in4 );
s_signed_multiply   <= signed( s_in1 )   * signed( s_in2);
s_unsigned_multiply <= unsigned( s_in3 ) * unsigned( s_in4 );
```

NOT TO BE USED:

```
s_mixed_add         <= signed( s_in1 ) + unsigned( s_in3 );
s_mixed_multiply    <= signed( s_in1 ) * unsigned( s_in3 );
```

Special Operations

- We can perform shifting by using concatenation

```
SIGNAL s_x1 : std_logic_vector( 3 DOWNTO 0 );
SIGNAL s_x2 : std_logic_vector( 3 DOWNTO 0 );
SIGNAL s_x3 : std_logic_vector( 3 DOWNTO 0 );

s_x1 <= s_x2( 2 DOWNTO 0 ) & "0";      -- s_x1 is s_x2 << 1; hence s_x1 = s_x2*2…
s_x3 <= s_x2(3) & s_x2( 3 DOWNTO 1 );  -- s_x3 is signed(s_x2) >> 1; hence s_x3 = s_x2/2…
```

- We can use concatenation to extract the carry out

```
SIGNAL s_a   : std_logic;
SIGNAL s_b   : std_logic;
SIGNAL s_add : std_logic_vector( 1 DOWNTO 0 );
SIGNAL s_sum : std_logic;
SIGNAL s_cout: std_logic;

s_add  <= unsigned("0" & s_a) + unsigned("0" & s_b);
s_sum  <= s_add(0);
s_cout <= s_add(1);
```

- Important note:
  - For the operators **+** and **-**, the synthesizer does not know if a **`std_logic_vector`** is in a signed, or an unsigned representation
  - We use the typecasts **`unsigned(<std_logic_vector>)`** and **`signed(<std_logic_vector>)`** to tell the synthesizer the representation

Back to the Full Adder

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

ENTITY full_adder IS
   PORT ( A    : IN  std_logic;  -- A input
          B    : IN  std_logic;  -- B input
          Cin  : IN  std_logic;  -- Carry input
          SUM  : OUT std_logic;  -- Sum output
          Cprop: OUT std_logic;  -- Carry propagate
          Cout : OUT std_logic); -- Carry output
END full_adder;

ARCHITECTURE implementation_1 OF full_adder IS

   SIGNAL s_x8 : std_logic;

BEGIN
   SUM   <= A XOR B XOR Cin;
   s_x8  <= (A AND B) OR (Cin AND (A XOR B));
   Cout  <= s_x8;
   Cprop <= s_x8 AND Cin;
END implementation_1;
```

?

=

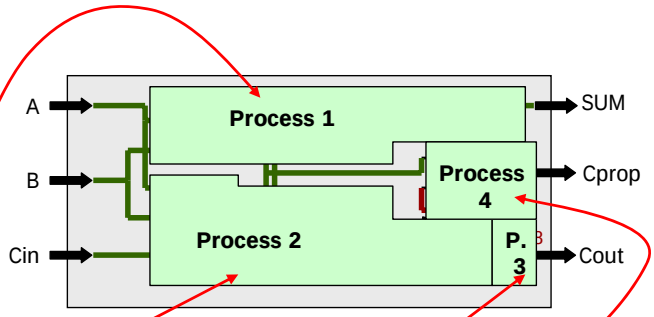
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY full_adder IS
   PORT ( A    : IN  std_logic;  -- A input
          B    : IN  std_logic;  -- B input
          Cin  : IN  std_logic;  -- Carry input
          SUM  : OUT std_logic;  -- Sum output
          Cprop: OUT std_logic;  -- Carry propagate
          Cout : OUT std_logic); -- Carry output
END full_adder;

ARCHITECTURE implementation_1 OF full_adder IS

   SIGNAL s_x8 : std_logic;

BEGIN
   Cprop <= s_x8 AND Cin;
   SUM   <= A XOR B XOR Cin;
   Cout  <= s_x8;
   s_x8  <= (A AND B) OR (Cin AND (A XOR B));
END implementation_1;
```

Remember that by definition:
All elements in the body of an **`ARCHITECTURE`** are either **`PROCESS`**'s or **`COMPONENT`**'s.
This architecture thus contains 4 (implicit) **`PROCESS`**'es

YES, both `full_adder`'s are equal!
As all **`PROCESS`**'es and **`COMPONENT`**'s execute in parallel, their order does not matter!
Note: For Java and C++ this is different, as they execute all lines sequentially!

Back to the Full Adder

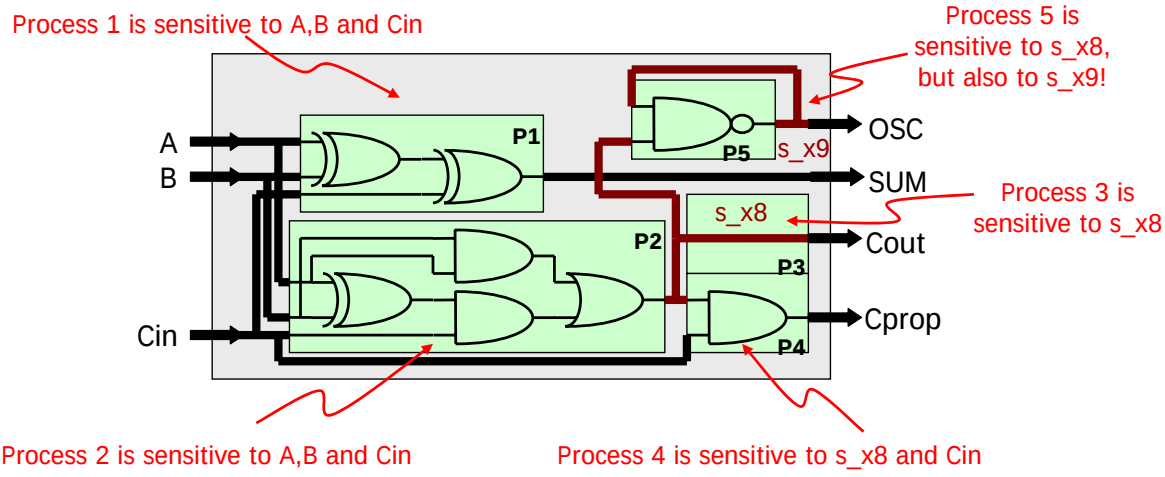
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY full_adder IS
   PORT ( A    : IN  std_logic;  -- A input
          B    : IN  std_logic;  -- B input
          Cin  : IN  std_logic;  -- Carry input
          SUM  : OUT std_logic;  -- Sum output
          Cprop: OUT std_logic;  -- Carry propagate
          Cout : OUT std_logic); -- Carry output
END full_adder;

ARCHITECTURE implementation_1 OF full_adder IS

   SIGNAL s_x8 : std_logic;

BEGIN
   SUM   <= A XOR B XOR Cin;
   s_x8  <= (A AND B) OR (Cin AND (A XOR B));
   Cout  <= s_x8;
   Cprop <= s_x8 AND Cin;
END implementation_1;
```



Intuition:
All elements in hardware will execute in parallel, hence the processes must also execute in parallel

Sensitivity and Trigger

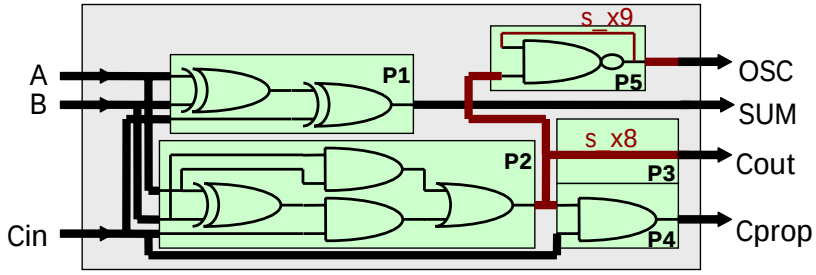
- A process is said to be *sensitive* to a wire, if and only if the output of the process can change in case the wire changes its value



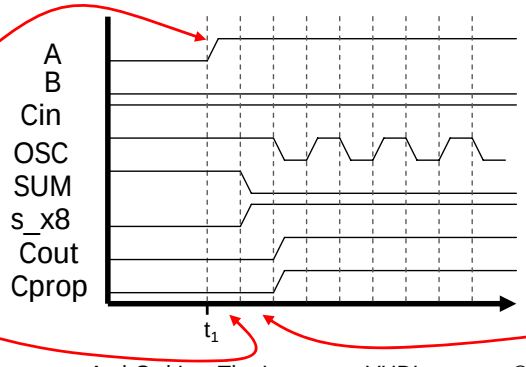
- A process is triggered if one or more wires, to which the process is sensitive, change its/their value

Delta

- Assume that every **PROCESS** has an input to output delay of δ
- If A, B, and/or Cin change(s) at time t_1, this is how simulation goes



Note: δ does not include the gate delays, as we are working with a zero delay model!



The change on A triggers process 1 and process 2

And this continues infinitely

After one δ process 1 and process 2 produce a stable output, and trigger processes 3, 4 and 5

After two δ process 3 and 4 produce stable outputs, but process 5 is triggered by s_x9

Delta

- We assumed the input and output delay to be δ, but in reality δ → 0
- Delta is defined as the number of δ steps we need to take until all **PROCESS**'es produce a stable output (no process is triggered):
  - **PROCESS** 1 and 2 produce a stable output after Delta = 1
  - **PROCESS** 3 and 4 produce a stable output after Delta = 2
  - **PROCESS** 5 does not produce a stable output, as it is triggered over and over again
- An **ARCHITECTURE** which does not produce a stable output on all its **PROCESS**'es, after a finite number of δ steps, is called instable, and cannot be simulated.
- We will only use stable **ARCHITECTURE**'s, hence Delta is finite

Notes:

1. We can of course model oscillators (instable architectures), by also modeling the gate delay, but this is outside the scope of this course
2. Instable architectures can be implemented in hardware, and are thus synthesizable

Processes and Components

- Each “element” or “box” in our **ARCHITECTURE BODY** is a process:

```
<process_name> : PROCESS ([Sensitivity List])

   [Declaration Section]

BEGIN
   [Process description]
END PROCESS <process_name>;
```

Explicit form

```
<signal_name> <= <source>;
```

Implicit form

- Or a component:

```
<component_reference> : <component_name>
                GENERIC MAP ( [generic mappings] )
                PORT MAP ( [Port connections] );
```


Implicit Processes

- We have seen **PROCESS**’es in the form:

```
<signal_name> <= <source>;
```

- This are so called **implicit PROCESS**’s
- Implicit **PROCESS**’es are by definition sensitive to all **SIGNAL**’s listed in the **<source>**, and thus are triggered by a change on these signals
- Implicit **PROCESS**’es are not embedded into a **PROCESS** container
- In implicit **PROCESS**’es the **<source>** consists of signals and operators
- There exists a special implicit **PROCESS**, the **WHEN … ELSE …;** construct

Implicit Constructs

- Syntax:

The **ELSE** part must always be present, as otherwise the **SIGNAL** is not always assigned a value!

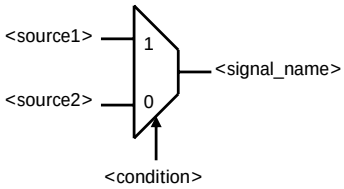
```
<signal_name> <= <source1> WHEN <condition> ELSE
                 <source2>;
<signal_name> <= <source1> WHEN <condition1> ELSE
                 <source2> WHEN <condition2> ELSE
                  …
                 <sourcen>;
```

- Example:

Question:
What is the hardware representing this construct?

```
next_state <= green  WHEN state = red_orange OR
                          (state = green AND
                           button /= '1') ELSE
              orange WHEN state = green AND
                          button = '1' ELSE
              red    WHEN state = orange ELSE
              red_orange;
```

Answer:



Summary

- The **SIGNAL**'s in the body of an **ARCHITECTURE** represent the wires of the system connecting the **PORT**'s and the operators
- The operators define the functionality of the system
- **SIGNAL**'s may only be assigned once
- Each element in the body of an **ARCHITECTURE** is a **PROCESS** or a **COMPONENT**
- **PROCESS**'es and **COMPONENT**'s execute in parallel

3

Sequential Logic

Explicit processes

Latches

Flip-flops and Registers

Components

Reminder
Processes and Components

- Each “element” or “box” in our **ARCHITECTURE BODY** is a process:

```
<process_name> : PROCESS ([Sensitivity List])

   [Declaration Section]

BEGIN
   [Process description]
END PROCESS <process_name>;
```

Explicit form

Implicit form

```
<signal_name> <= <source>;
```

- Or a component:

```
<component_reference> : <component_name>
                GENERIC MAP ( [generic mappings] )
                PORT MAP ( [Port connections] );
```

Explicit Process

- A **PROCESS** which is not implicit is called an **explicit PROCESS**
- An explicit **PROCESS** can be converted to one or multiple implicit one(s) if and only if it contains **only** combinatorial logic
- An explicit **PROCESS** is **not** inherently sensible to its inputs
- The body of an explicit **PROCESS** executes **in program order**

Explicit Process Syntax

Each **PROCESS** has a unique name

Each **PROCESS** can have a sensitivity list

```
<process_name> : PROCESS ([Sensitivity List])

   [Declaration Section]

BEGIN
   [Process body]
END PROCESS <process_name>;
```

Each **PROCESS** has a body

Each **PROCESS** can have a declaration section

Note:
We can also just write **END PROCESS;**

Sensitivity List

- Explicit **PROCESS**'s are not inherently sensitive to their inputs, to make the **PROCESS** sensitive to (and thus trigger on a change of) a **SIGNAL**, the name of this **SIGNAL** can be put in the sensitivity list:

```
example_process : PROCESS ( <signal_name>, … , <signal_name> )
```

- Example, a two input NAND gate:

```
SIGNAL s_a : std_logic;
SIGNAL s_b : std_logic;
SIGNAL s_q : std_logic;
```

We will **ALWAYS** put **all signals** which appear after **all** assignment commands (<= and :=) in the **sensitivity list** of an explicit process

```
Nand_gate1 : PROCESS ( s_a , s_b )
BEGIN
   s_q <= s_a NAND s_b;
END PROCESS nand_gate1;
```

?

=

```
Nand_gate2 : PROCESS ( s_a )
BEGIN
   s_q <= s_a NAND s_b;
END PROCESS nand_gate2;
```

In hardware both are equal, but **NOT** in simulation!
nand_gate2 will **only** be triggered by a change on **s_a**, and **not** by a change on **s_b**!
The synthesizer will issue a **warning** when it encounters such a construct

Declaration Section

- The declaration section of an explicit **PROCESS** can contain **VARIABLE** instantiations
- The declaration section of an explicit **PROCESS** can contain **SIGNAL** instantiations, which are local to this **PROCESS**
- The declaration section of an explicit **PROCESS** can contain **CONSTANT** instantiations, which are local to this **PROCESS**

Variables

- **VARIABLE**'s have a similar syntax as **SIGNAL**'s

```
VARIABLE <variable_name>[, <variable_name>,…] : <signal_type>;
```

It is good practice, but not required,
to prefix all variable names with "v_"

- **VARIABLE**'s can **only** exist inside an explicit **PROCESS**, and are local to this **PROCESS**

- **VARIABLE**'s can be assigned a value by using

```
<variable_name> := <source>;
```

Note:
We need **:=** and **not <=**

Back to the Full Adder

Using implicit **PROCESS**'es:

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

ENTITY full_adder IS
   PORT ( A    : IN  std_logic;  -- A input
          B    : IN  std_logic;  -- B input
          Cin  : IN  std_logic;  -- Carry input
          SUM  : OUT std_logic;  -- Sum output
          Cprop: OUT std_logic;  -- Carry propagate
          Cout : OUT std_logic); -- Carry output
END full_adder;

ARCHITECTURE implementation_1 OF full_adder IS

   SIGNAL s_x8 : std_logic;

BEGIN
   SUM   <= A XOR B XOR Cin;
   s_x8  <= (A AND B) OR (Cin AND (A XOR B));
   Cout  <= s_x8;
   Cprop <= s_x8 AND Cin;
END implementation_1;
```

=

Using one explicit **PROCESS**:

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

ENTITY full_adder IS
   PORT ( A    : IN  std_logic;  -- A input
          B    : IN  std_logic;  -- B input
          Cin  : IN  std_logic;  -- Carry input
          SUM  : OUT std_logic;  -- Sum output
          Cprop: OUT std_logic;  -- Carry propagate
          Cout : OUT std_logic); -- Carry output
END full_adder;

ARCHITECTURE implementation_2 OF full_adder IS

BEGIN
   adder : PROCESS( A , B , Cin )
      VARIABLE v_x8 : std_logic;
   BEGIN
      SUM   <= A XOR B XOR Cin;
      v_x8  := (A AND B) OR (Cin AND (A XOR B));
      Cout  <= v_x8;
      Cprop <= v_x8 AND Cin;
   END PROCESS adder;
END implementation_2;
```

Back to the Full Adder

With implicit **PROCESS**'s, we have seen that:

```
BEGIN
   SUM   <= A XOR B XOR Cin;
   s_x8  <= (A AND B) OR (Cin AND (A XOR B));
   Cout  <= s_x8;
   Cprop <= s_x8 AND Cin;
END implementation_1;
```

=

```
BEGIN
   Cprop <= s_x8 AND Cin;
   SUM   <= A XOR B XOR Cin;
   Cout  <= s_x8;
   s_x8  <= (A AND B) OR (Cin AND (A XOR B));
END implementation_1;
```

What about the explicit **PROCESS**'s:

```
BEGIN
   adder : PROCESS( A , B , Cin )
      VARIABLE v_x8 : std_logic;
   BEGIN
      SUM   <= A XOR B XOR Cin;
      v_x8  := (A AND B) OR (Cin AND (A XOR B));
      Cout  <= v_x8;
      Cprop <= v_x8 AND Cin;
   END PROCESS adder;
END implementation_2;
```

≠

```
BEGIN
   adder : PROCESS( A , B , Cin )
      VARIABLE v_x8 : std_logic;
   BEGIN
      Cprop <= v_x8 AND Cin;
      SUM   <= A XOR B XOR Cin;
      Cout  <= v_x8;
      v_x8  := (A AND B) OR (Cin AND (A XOR B));
   END PROCESS adder;
END implementation_2;
```

These two implementations are **not** equal!

Due to the fact that these are now **inside** the body of the process, they are not any longer **separate processes**, but **statements**. And statements execute **in program order** inside the body of an explicit process, and **not in parallel**! We will go into more detail

Statements and Signals

- Note that the **<=** has a different meaning in an implicit **PROCESS**, and when used in a statement
  - Using **<=** in an implicit process means **assign immediately**
  - Using **<=** in a statement means **schedule an assignment**
- The explicit process has some "strange" behavior when it comes to signals

```
example : PROCESS( A , B , C )
BEGIN
   C <= A OR B;
   IF (A AND B) = '1' THEN
      C <= '0';
   END IF;
   X2 <= C;
END PROCESS example;
```

The process "fixes" the values of the signals when it is triggered

The statements within the process **schedule** new assignments to a signal

The process assigns the new values of the signals when it is finished

Hence, X2 is assigned the **old** value of C

- Note further that **X2** and **C** are only **assigned once** a value, namely **at the end of the PROCESS**

Statements and Variables

- It's getting even more "strange", as **VARIABLE**'s are assigned immediately in explicit **PROCESS**'es, hence **:=** means **assign immediately**
- Thus

IMPORTANT:
:= can **ONLY** be used with variables!

```
example : PROCESS( A , B , C )
   VARIABLE v_x : std_logic;
BEGIN
   v_x := A OR B;
   IF (A AND B) = '1' THEN
      v_x := '0';
   END IF;
   X2 <= v_x;
   c  <= v_x;
END PROCESS example;
```

The difference with Signals and Variables, is that **v_x** is now **assigned** immediately a value, and not just **scheduled** a new assignment!

Explicit Constructs

- Similar to the **WHEN … ELSE …;** construct, we have the **IF … END IF;** construct in an explicit process:

```
IF <condition> THEN [Statement(s)];
END IF;

IF <condition> THEN [Statement(s)];
               ELSE [Statement(s)];
END IF;

IF <condition> THEN [Statement(s)];
ELSIF <condition2> THEN [Statement(s)];
                   ELSE [Statement(s)];
END IF;
```

This version should **only** be used to model sequential logic (we will see more in the next lecture)

Each of these sections should contain at least one, but can contain multiple statements

Explicit Constructs

- Nested **IF … END IF;** constructs can be replaced by the **CASE … END CASE;** construct:

The **<case_variable>** can be a **SIGNAL**, **PORT**, or a **VARIABLE**

```
CASE <case_variable> IS
   WHEN <value1> => [Statement(s)];
   WHEN <value2> => [Statement(s)];
   WHEN <value3> => [Statement(s)];
         …
   WHEN OTHERS   => [Statement(s)];
END CASE;
```

Each of these sections should contain at least one, but can contain multiple statements

The **OTHERS** case covers for all situations not covered by the previous **WHEN** lines (similar **default:** in C++).
It is not required, but **highly recommended** to use!

- Note, the **CASE … END CASE;** construct has no equivalent implicit form!

Size Macros

- Assume following code

```
s_bitwise_and(0) <= s_input_vector(0) AND s_bit;
s_bitwise_and(1) <= s_input_vector(1) AND s_bit;
s_bitwise_and(2) <= s_input_vector(2) AND s_bit;
s_bitwise_and(3) <= s_input_vector(3) AND s_bit;
```

- We can simplify this code by using a macro
  - For implicit process usage

```
Bitwise_and : FOR n IN 3 DOWNTO 0 GENERATE
   s_bitwise_and(n) <= s_input_vector(n) AND s_bit;
END GENERATE;
```

  - For explicit process usage

```
FOR n IN 3 DOWNTO 0 LOOP
   s_bitwise_and(n) <= s_input_vector(n) AND s_bit;
END LOOP;
```

Note:
These macros do **not** equal to a for loop as used in C++, they merely help us to compact our code

- Syntax

```
FOR <variable> IN <value1> DOWNTO <value2> LOOP
   [Statement(s)]
END LOOP;
```

Explicit Process usage

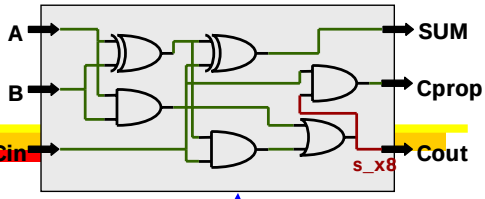
```
<id> : FOR <variable> IN <value1> DOWNTO <value2> GENERATE
   [Statement(s)]
END GENERATE;
```

Implicit Process usage

Summary

- We now have all the tools to create combinatorial logic:
  - **SIGNAL's** → the wires of the system
  - **Operators** → to model the functionality of the system
  - **PROCESS'es** → which execute in parallel, and can be of two forms
    - Implicit: These processes are not contained in a **PROCESS** container and are in the form **<signal_name> <= <source>**
    - Explicit: These processes are in a **PROCESS** container, and the statements in these processes execute in program order
  - **Statements** → help to express conditional logic (= multiplexing)
    - Implicit usage only: The **WHEN … ELSE …** construct
    - Explicit usage only: The **IF … END IF;** and **CASE … END CASE;** constructs
  - **Size Macros** → helps to compact our code
    - Implicit usage: The **GENERATE** macro
    - Explicit usage: The **LOOP** macro

Summary



All processes are defined in the body of the architecture

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

ENTITY full_adder IS
   PORT ( A    : IN  std_logic;  -- A input
          B    : IN  std_logic;  -- B input
          Cin  : IN  std_logic;  -- Carry input
          SUM  : OUT std_logic;  -- Sum output
          Cprop: OUT std_logic;  -- Carry propagate
          Cout : OUT std_logic); -- Carry output
END full_adder;

ARCHITECTURE implementation_1 OF full_adder IS

   SIGNAL s_x8 : std_logic;

BEGIN

   SUM   <= A XOR B XOR Cin;
   Cout  <= s_x8;
   Cprop <= s_x8 AND Cin;

   make_s8 : PROCESS( A , B , Cin )
   BEGIN
      s_x8  <= (A AND B) OR (Cin AND (A XOR B));
   END PROCESS make_s8;

END implementation_1;
```

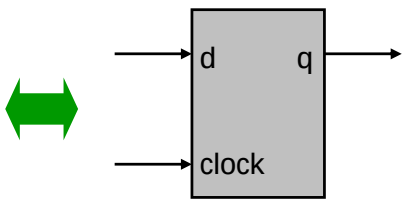
The order of the process (implicit and explicit) does not matter as they execute in **parallel**

We have **three** implicit processes

We have **one** explicit process

Latches

```
latch : PROCESS ( clock , d )
BEGIN
   IF (clock = '1') THEN
      q <= d;
   END IF;
END PROCESS latch;
```

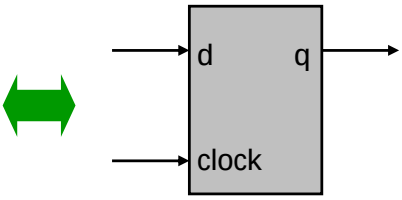


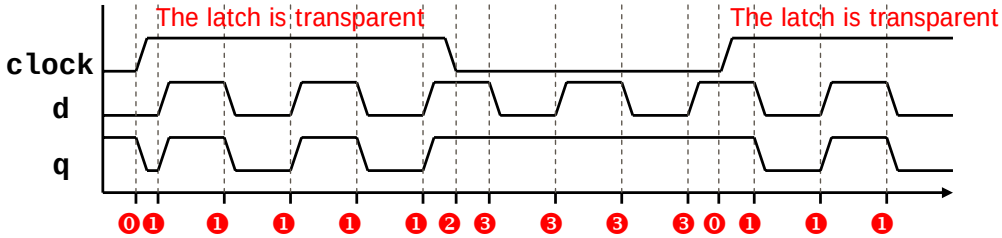
- If **clock** equals **1**, **q** will be scheduled to be the value of **d**. At the end of the **PROCESS**, **q** will be assigned the value of **d**
- If **clock** equals **0**, **q** will **not** be scheduled any value because the **IF** statement is skipped. Therefore at the end of the **PROCESS**, **q** will **keep** its previous value!
- We described a **memory** element, namely a latch

Latches

- Let's look at the simulation behavior

```
latch : PROCESS ( clock , d )
BEGIN
   IF (clock = '1') THEN
      q <= d;
   END IF;
END PROCESS latch;
```



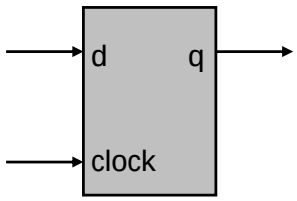


0 **clock** triggers the **PROCESS** and **clock** = **1** → **q** takes the value of **d**
1 an event on **d** triggers the **PROCESS** and **clock** = **1** → **q** takes the value of **d**
2 an event on **clock** triggers the **PROCESS** and **clock** = **0** → **q** **keeps** its previous value
3 an event on **d** triggers the **PROCESS** and **clock** = **0** → **q** **keeps** its previous value

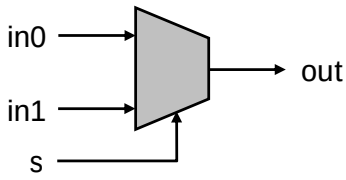
Latches vs. Muxes

- **Both** latches and multiplexers are described through **IF** statements

```
latch : PROCESS ( clock , d )
BEGIN
   IF (clock = '1') THEN
      q <= d;
   END IF;
END PROCESS latch;
```



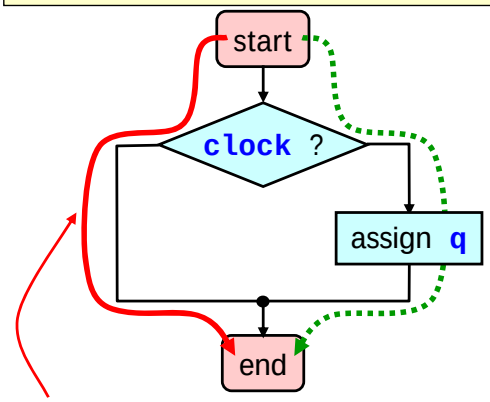

```
mux : PROCESS ( s , in0 , in1 )
BEGIN
   IF (s = '1') THEN
      out <= in1;
   ELSE
      out <= in0;
   END IF;
END PROCESS mux;
```



- The important difference is that in muxes there is **always a new value assigned** to signals or variables, whereas in latches sometimes nothing is assigned and they **keep their old value** (→ memory!)

Combinatorial or Sequential?

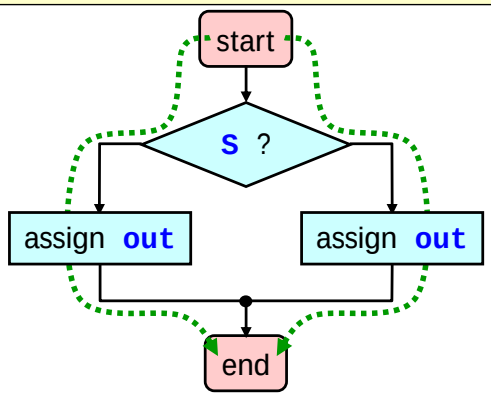
```
latch : PROCESS ( clock , d )
BEGIN
   IF (clock = '1') THEN
      q <= d;
   END IF;
END PROCESS latch;
```



At least one path between start and end of the process contains no assignment to **q**

→ q is the output of a sequential element

```
mux : PROCESS ( s , in0 , in1 )
BEGIN
   IF (s = '1') THEN
      out <= in1;
   ELSE
      out <= in0;
   END IF;
END PROCESS mux;
```



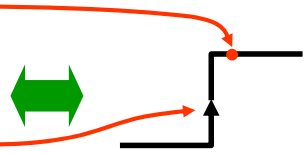
All paths between start and end of the process contain at least one assignment to **out**

→ out is the output a combinatorial element

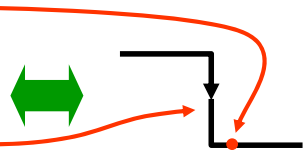
Testing for Edges

- A signal with an **'EVENT** modifier evaluates to true if there has been a high to low or low to high transition

```
IF (<signal_name>'EVENT AND <signal_name> = '1') THEN
    …
END IF;
```



```
IF (<signal_name>'EVENT AND <signal_name> = '0') THEN
    …
END IF;
```

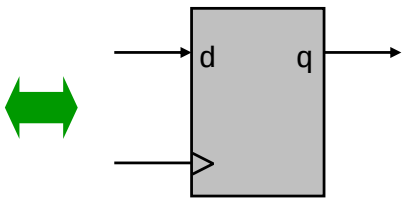


- An **EVENT** should **always** be used in combination with a test for the **current level**, because there exists **no hardware component** which is triggered by both transitions

```
IF <signal_name>'EVENT THEN
    …
END IF;
```

Flip-flops and Registers

```
flipflop : PROCESS ( clock , d )
BEGIN
   IF (clock'EVENT and clock = '1') THEN
      q <= d;
   END IF;
END PROCESS flipflop;
```

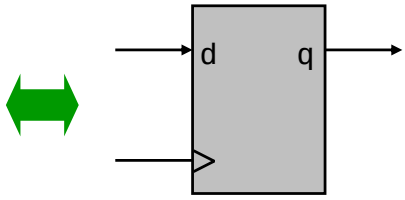


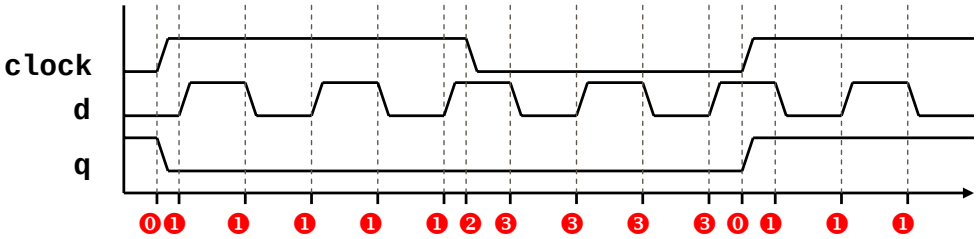
- If **there is a transition** and after it **clock** equals **1**, **q** will be scheduled to be the value of **d**, and at the end of the **PROCESS**, **q** will be assigned the value of **d**
- If there is no transition or **clock** equals **0**, **q** will **not** be scheduled any value because the **IF** statement is skipped. Therefore at the end of the **PROCESS**, **q** will **keep** its previous value!
- We also described a **memory** element, but this time it is a flip-flop

Flip-flops and Registers

- Let's look at the simulation behavior

```
flipflop : PROCESS ( clock , d )
BEGIN
   IF (clock'EVENT and clock = '1') THEN
      q <= d;
   END IF;
END PROCESS flipflop;
```



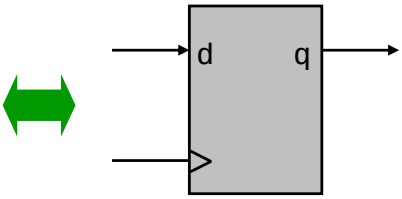


❶ **clock** triggers the **PROCESS** and there is a raising edge → **q** takes the value of **d**
❶ an event on **d** triggers the **PROCESS** but there is no event on **clock** → **q keeps** its previous value
❷ an event on **clock** triggers the **PROCESS** but it is not a raising edge → **q keeps** its previous value
❸ an event on **d** triggers the **PROCESS** but there is no event on **clock** → **q keeps** its previous value

A Bad Idea for a Flip-Flop

- One could think of exploiting the **sensitivity list**...

```
flipflop : PROCESS ( clock , d )
BEGIN
   IF (clock'EVENT and clock = '1') THEN
      q <= d;
   END IF;
END PROCESS flipflop;
```

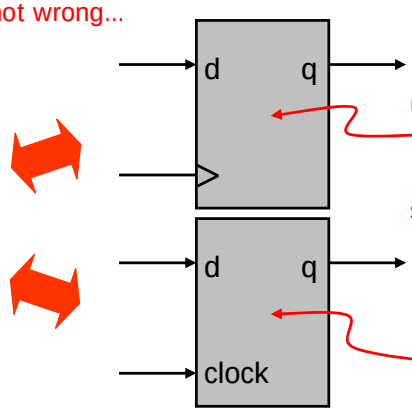


By the way, this is **useless** but not wrong...

This looks like a **latch**...

```
bad_flipflop : PROCESS ( clock )
BEGIN
   IF (clock = '1') THEN
      q <= d;
   END IF;
END PROCESS bad_flipflop;
```

NO



Simulators understand **this**...

...but most synthesizers ignore sensitivity lists and understand **this**!

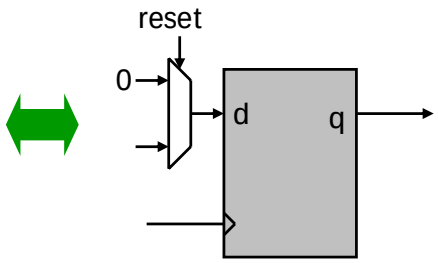
...**but** it is **only** executed if **clock** changes!
(normally a latch has **d** too here...)

Reset

- A register does not have an initial state, and therefore needs a reset
- Two types of reset
  - **Synchronous** reset

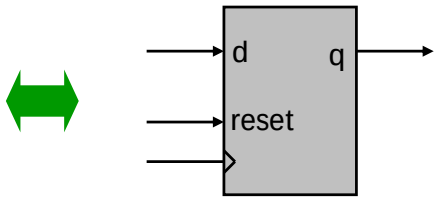
Note: We will always use registers with a reset, either synchronous or asynchronous

```
flipflop : PROCESS ( clock , reset , d )
BEGIN
   IF (clock'EVENT and clock = '1') THEN
      IF (reset = '1') THEN q <= '0';
                       ELSE q <= d;
      END IF;
   END IF;
END PROCESS flipflop;
```



  - **Asynchronous** reset

```
flipflop : PROCESS ( clock , reset , d )
BEGIN
   IF (reset = '1') THEN q <= '0';
   ELSIF (clock'EVENT and clock = '1') THEN
      q <= d;
   END IF;
END PROCESS flipflop;
```



Registers, Signals, and Ordering

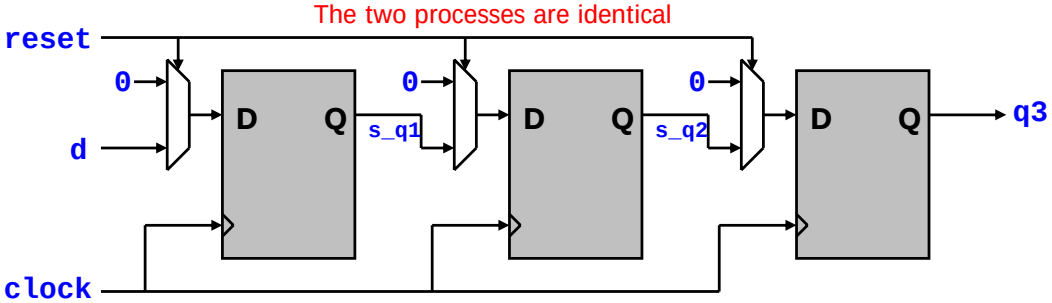
```
Delay_line : PROCESS ( clock , reset , d ,
                       s_q1 , s_q2 )
BEGIN
   IF (clock'EVENT AND clock = '1') THEN
      IF (reset = '1') THEN s_q1 <= '0';
                            s_q2 <= '0';
                            q3   <= '0';
                       ELSE q3   <= s_q2;
                            s_q2 <= s_q1;
                            s_q1 <= d;
      END IF;
   END IF;
END PROCESS Delay_line;
```

=

```
Delay_line : PROCESS ( clock , reset , d ,
                       s_q1 , s_q2 )
BEGIN
   IF (clock'EVENT AND clock = '1') THEN
      IF (reset = '1') THEN s_q1 <= '0';
                            s_q2 <= '0';
                            q3   <= '0';
                       ELSE s_q1 <= d;
                            s_q2 <= s_q1;
                            q3   <= s_q2;
      END IF;
   END IF;
END PROCESS Delay_line;
```

YES!

The two processes are identical



Registers, Variables, and Ordering

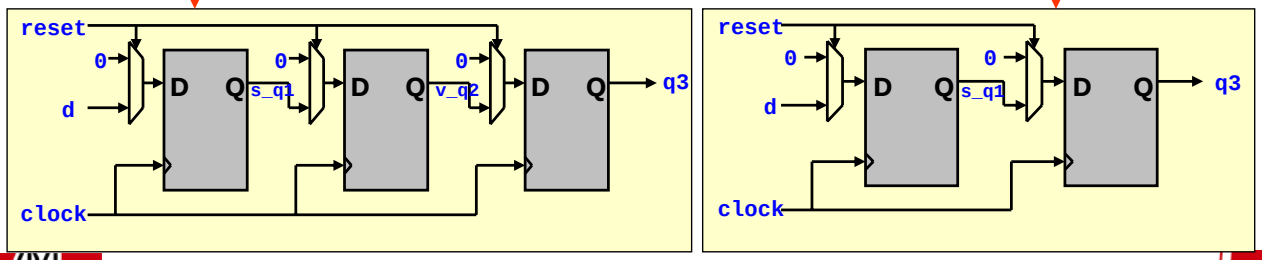
```
Delay_line : PROCESS ( clock , reset , d ,
                       s_q1 )
   VARIABLE v_q2 : std_logic;
BEGIN
   IF (clock'EVENT AND clock = '1') THEN
      IF (reset = '1') THEN s_q1 <= '0';
                            v_q2 := '0';
                            q3   <= '0';
                       ELSE q3   <= v_q2;
                            v_q2 := s_q1;
                            s_q1 <= d;
      END IF;
   END IF;
END PROCESS Delay_line;
```

≠

```
Delay_line : PROCESS ( clock , reset , d ,
                       s_q1 )
  VARIABLE v_q2 : std_logic;
BEGIN
   IF (clock'EVENT AND clock = '1') THEN
      IF (reset = '1') THEN s_q1 <= '0';
                            v_q2 := '0';
                            q3   <= '0';
                       ELSE s_q1 <= d;
                            v_q2 := s_q1;
                            q3   <= v_q2;
      END IF;
   END IF;
END PROCESS Delay_line;
```

NO!

The two processes are **different!**



Components

- We can introduce a hierarchy by using **COMPONENT**'s
- A **COMPONENT** is a reference to an **ENTITY**
- The syntax of a **COMPONENT** is

```
COMPONENT <entity_name>
   PORT ( … );
END COMPONENT;
```

Each **COMPONENT** requires a reference to the corresponding **ENTITY**

Each **COMPONENT** has the same port section as it's corresponding **ENTITY**

- The **COMPONENT** is defined in the declaration section of an **ARCHITECTURE**

Usage of Components

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

ENTITY full_adder IS
   PORT ( A    : IN  std_logic;  -- A input
          B    : IN  std_logic;  -- B input
          Cin  : IN  std_logic;  -- Carry input
          SUM  : OUT std_logic;  -- Sum output
          Cout : OUT std_logic); -- Carry output
END full_adder;

ARCHITECTURE implementation OF full_adder IS
BEGIN

   SUM   <= A XOR B XOR Cin;
   Cout  <= (A AND B) OR (Cin AND (A XOR B));

END implementation;
```

The reference to the **ENTITY** **full_adder** with the same **PORT** section

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

ENTITY adder IS
   PORT ( A    : IN  std_logic_vector( 3 DOWNTO 0);
          B    : IN  std_logic_vector( 3 DOWNTO 0);
          Cin  : IN  std_logic;
          SUM  : OUT std_logic_vector( 3 DOWNTO 0);
          Cout : OUT std_logic);
END adder;

ARCHITECTURE implementation OF adder IS
   COMPONENT full_adder
   PORT ( A    : IN  std_logic;  -- A input
          B    : IN  std_logic;  -- B input
          Cin  : IN  std_logic;  -- Carry input
          SUM  : OUT std_logic;  -- Sum output
          Cout : OUT std_logic); -- Carry output
   END COMPONENT;

   SIGNAL s_carries : std_logic_vector( 4 DOWNTO 0);
BEGIN
   s_carries(0) <= Cin;
   fas : FOR n IN 3 DOWNTO 0 GENERATE
     fa : full_adder
          PORT MAP ( A    => A(n),
                     B    => B(n),
                     Cin  => s_carries(n),
                     SUN  => SUM(n),
                     Cout => s_carries(n+1) );
   END GENERATE fas;
END implementation;
```

Each **COMPONENT** needs a unique identifier

And here we connect the **PORT**'s of the component with the **SIGNAL**'s to form the new functionality

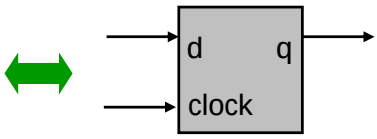
Summary
Signals, Variables, and Parallelism

- For a **SIGNAL**, the **<=** has a different meaning in an implicit **PROCESS**, and when used in a statement (explicit **PROCESS**)
  - Using **<=** in an implicit **PROCESS** means **assign immediately**
  - Using **<=** in an explicit **PROCESS** means **schedule an assignment**
- A **VARIABLE** can **only** exist in an explicit **PROCESS**, and is assigned a value by using **:=**, where
  - Using **:=** means **assign immediately**
- Implicit and explicit **PROCESS**'es are defined inside the **body** of the **ARCHITECTURE**, and execute **in parallel**
- Statements are defined inside the **body** of an explicit **PROCESS**, and execute **in program order**

Summary
Sequential Components

- Sequential elements can only be made by using an explicit **PROCESS**
- Memory elements can only be introduced in case the **PROCESS** contains a non-assigned path, otherwise a multiplexer is described
- There are two kinds of memory elements:
  - Latches:

```
latch : PROCESS ( clock , d )
BEGIN
   IF (clock = '1') THEN
      q <= d;
   END IF;
END PROCESS latch;
```



  - Flip-flops:

```
flipflop : PROCESS ( clock , d )
BEGIN
   IF (clock'EVENT and clock = '1') THEN
      q <= d;
   END IF;
END PROCESS flipflop;
```