EE-334
Digital System Design

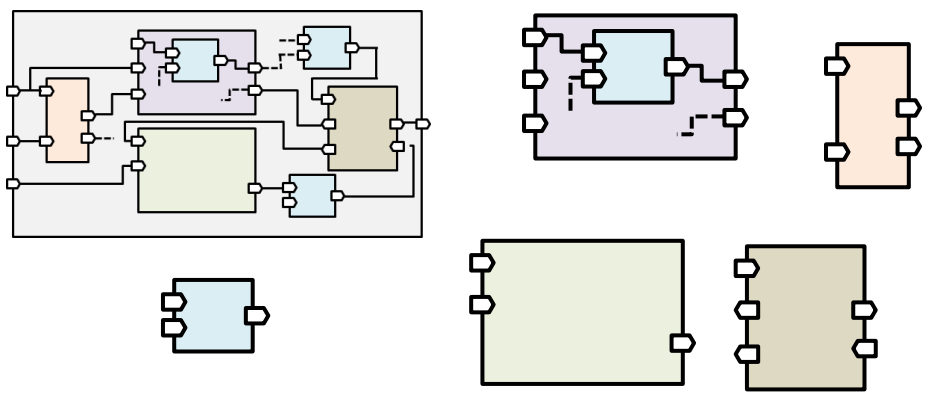
Custom Digital Circuits

VHDL for Synthesis –
Basic Constructs and Correspondences

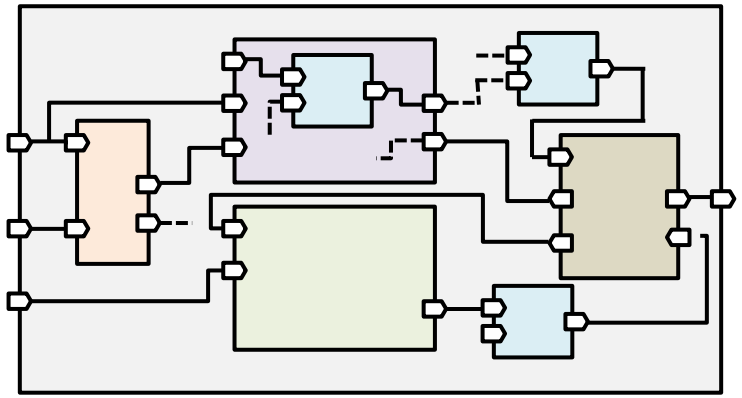
Andreas Burg

Hierarchy and Instantiation

- **Hierarchy is required** for modularity, abstraction, and **to deal with complexity**
- **Blocks used later** in a hierarchy **are** called **COMPONENTS**
  - **Components can** themselves **be hierarchical** (include instances of other components)
  - **Signals** declared in a component are **only visible within the component**
  - The interface of a component is defined by its ENTITY
- Each "**appearance/use**" **of a component** is **called an INSTANCE**
  - There can be **multiple instances of the same component**
  - **Instances are connected through** the **ports** of their components



all
separate
VHDL files

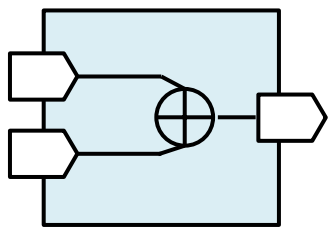


Components vs. Functions

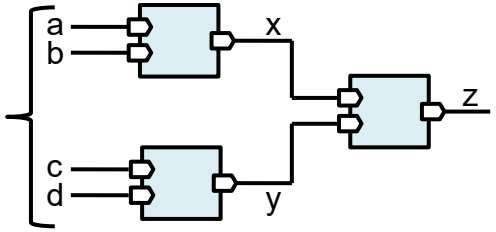
- Only **distant analogy to functions** in programming: **similar purpose** (reuse), **BUT very different physical presence**

Hardware (VHDL)

Component definition



Component **instantiations**



Co-exist in parallel

Software (C)

```
Int my_fct(int a, int b)
{
        return a+b;
}

Main() {
        …
        x = my_fct(a, b);
        y = my_fct(c, d);
        z = my_fct(x, y);
}
```

Function definition

Function calls

Sequential execution

Instantiating Components in VHDL (1/2)

- **Components need to be declared** before instantiating/using them in the architecture of another component
  - **Declaration in the preamble of the architecture** in which they are used
  - Alternative: declaration in a package (not discussed here)
- **Component** declaration & **name must match** the **corresponding entity name**

Entity declaration of a component in a file,
e.g., entity_name.vhd

```
ENTITY entity_name IS
          GENERIC (
               generic_1_name : generic_1_type;
               generic_2_name : generic_2_type
          );
          PORT (
               port_1_name : port_1_dir port_1_type;
               port_2_name : port_2_dir port_2_type
          );
END entity_name;
```

```
ARCHITECTURE architecture_name OF other_entity_name IS

          -- component declaration
          COMPONENT component_name IS
               GENERIC (
                    generic_1_name : generic_1_type;
                    generic_2_name : generic_2_type
                     );
               PORT (
                    port_1_name : port_1_dir port_1_type;
                    port_2_name : port_2_dir port_2_type
                    );
          END COMPONENT;

BEGIN
          -- VHDL statements
END architecture_name;
```

Instantiating Components in VHDL (2/2)

- Declared **components can be instantiated in the architecture body**

- **Instantiation(s)**
  - defines the instance name
  - **connects ports** of an instance of the components **to signals**
  - **Defines the generics** (parameters) based on expressions that can be evaluated at compile-time

```
ARCHITECTURE architecture_name OF other_entity_name IS
        -- signal declarations
        SIGNAL port_1_1_signal, port_2_1_signal : port_1_type;
        SIGNAL port_1_2_signal, port_2_2_signal : port_2_type;
        -- component declaration
        COMPONENT component_name IS
            GENERIC ( … );
            PORT ( … );
        END COMPONENT;
BEGIN
        -- Component instantiation
        instance_1_name : component_name
            GENERIC MAP (
                generic_1_name => CONSTANT_EXP_1_1,
                generic_2_name => CONSTANT_EXP_1_2
            )
            PORT MAP (
                port_1_name => port_1_1_signal,
                port_2_name => port_1_2_signal
            );
        instance_2_name : component_name
            GENERIC MAP ( CONSTANT_EXP_2_1, CONSTANT_EXP_2_2
)
            PORT MAP ( port_2_1_signal, port_2_2_signal );
END architecture_name;
```

Array Types

- To describe **hardware** we often **need BUSSES** (groups of signals or constants)
- **Arrays are** defined by declaring **a custom type**
- **Custom types** are **declared in** the **architecture preamble**
  - **Elements** from left-to-right **can be counted "from-low-to-high" or "from-high-downto-low"**
  - **Array types**, once declared, **can** again **serve as basis for new arrays** to build 2+D arrays

```
ARCHITECTURE architecture_name OF other_entity_name IS
        -- type declarations
        TYPE array_type_name_1_1D IS ARRAY (low TO high) of base_type;
        TYPE array_type_name_2_1D IS ARRAY (high DOWNTO low) of base_type;
        TYPE array_type_name_3_2D IS ARRAY (INTEGER RANGE <>) of array_type_name_2;

        -- signal declaration
        SIGNAL signal_1_name : array_type_name_1;
        SIGNAL signal_2_name : array_type_name_3(low TO high);
BEGIN
        -- VHDL statements
END architecture_name;
```

Counting high-to-low

Range defined here

- **Array size declaration can be deferred until type is used**

STD_LOGIC_VECTOR with DOWNTO Index

- **STD_LOGIC_VECTOR**: heavily used in hardware modelling
  - Array type that comes with STD_LOGIC type
  - **Defined in IEEE.STD_LOGIC_1164 package**
  - **STD_LOGIC_VECTORS for binary numbers** are **typically declared** "**from-high-downto-low**"
    - Correlates better to the established weighted number representation
      - MSB on the left (low index)
      - LSB on the right (high index)

```
        "0 1 0 0 0" = 8'dec
Bit idx. 4 3 2 1 0
        (4 downto 0)
```

    - **Ideally, use only DOWNTO**

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.all;

ENTITY entity_name IS
    …
END entity_name;

ARCHITECTURE architecture_name OF other_entity_name IS
        -- signal declaration
        SIGNAL signal_bus_name : STD_LOGIC_VECTOR(low TO high);
        SIGNAL bin_signal_name : STD_LOGIC_VECTOR(high DOWNTO low);
BEGIN
        -- VHDL statements
END architecture_name;
```

Operations on Array Types

- **Accessing from** and **assigning to** **array elements or ranges**

```
target_array_object(index) <= base_type_object
target_base_type_object <= target_base_type_array(index)
target_array_object(index_range) <= from_array_object(index_range)
index_range = low TO high | high DOWNTO low
```

- **Assigning array aggregates** (collections of elements) to an array
  - Multiple options exist to “fill” an array (assign multiple elements together)

```
target_array_object <= (value_1, value_2, …);
target_array_object <= (idx_1=>value_1, idx_2=>value_2, …);
target_array_object <= (idx_1=>value_1, idx_2|idx_3=>value_2, …);
```

Operations on Array Types

- **Filling an array:** `OTHERS` refers to all still unassigned elements in the aggregator

```
target_array_object <= (idx_1=>value_1, OTHERS=>value_2); -- fill all remaining
target_array_object <= (OTHERS=>value_1);                 -- fill all elements
```

- **Assignments to arrays with character elements**
  - **Array literal values** can be places **in double quotes**

```
target_array_object(index_range) <= "…" -- e.g., "0100-1"
```

  - **STD_LOGIC_VECTOR** is based on STD_LOGIC which is a character type, i.e., '0', '1', 'X', '-', …

```
target_array_object(index_range) <= "010-10-"
```

- **Concatenation of arrays and array elements**

```
target_array_object(index_range) <= base_type_object_1 & base_type_object_1
```

Example Operations on STD_LOGIC_VECTOR

- Signal declarations of STD_LOGIC_VECTOR

```
-- signal declaration of std_logic_vectors
SIGNAL AxD, BxD, CxD, DxD, ExD : STD_LOGIC_VECTOR(8-1 DOWNTO 0);
SIGNAL QxS : STD_LOGIC;
```

- Assignments

```
QxS <= '1';
AxD <= "10010101";
BxD <= "-0-001-1";
CxD <= (OTHERS => '0');
```

- Concatenation and indexing

```
BxD <= AxD(7-1 downto 0) & '0';                -- SHIFT LEFT
CxD <= '0' & AxD(8-1 downto 1);                -- SHIFT RIGHT
DxD <= AxD(7-1 downto 0) & AxD(8-1);           -- ROTATE LEFT
ExD <= QxS & QxS & '1' & "00001";
```

Types for Arithmetic: UNSIGNED/SIGNED

- Built-in type **INTEGER supports basic arithmetic** operations,
  BUT the integer type **is not sufficiently generic** for optimized hardware
  - INTEGER type has a fixed width of 32 bit (often too large or sometimes too small)
  - INTEGER represents only signed numbers having half the range
  - Any overflow or underflow will trigger an ERROR in the simulation rather than a wrap-around

- **Need for a** more **flexible type** with variable length **for signed and unsigned**
- **`IEEE numeric_std` package:** define integer as array of `std_logic`
- **Two new data types: `UNSIGNED`, `SIGNED`**
- The **array interpreted as** an unsigned or signed **binary numbers**
  - `UNSIGNED` are represented as standard binary
  - `SIGNED` vectors are represented using two's complement
  - Array elements can be accessed and assigned as in a std_logic_vector

UNSIGNED/SIGNED Data Types (Best practice)

- SIGNED and UNSIGNED data types represent numbers. Therefore
  - Corresponding **signals** best **carry the suffix xD** to indicate the numeric type of data

<Signal Name>xD

`MyInputAxDI, MySIGNALxD, AccuREGxDN, AccuREGxDP`

- SIGNED and UNSIGNED use weighted binary digits, counted from right to left
  - **Use DOWNTO** bit order to place the LSB (bit-0) on the right and the MSB (bin-N) on the left

```
SIGNAL <Signal Name>xD : UNSIGNED(8-1 DOWNTO 0);
SIGNAL ExSIGNALxD : SIGNED(8-1 DOWNTO 0);
```

- When declaring a signal, it is often useful to immediately see the number of bits
  - Since bits are counted from zero (0), it is often more clear to write

```
SIGNAL <Signal Name>xD : UNSIGNED(#BITS-1 DOWNTO 0);
```

Arithmetic Operations on SIGNED and UNSIGNED

- **IEEE numeric_std defines many common operators**

overloaded operator	description	data type of operand a	data type of operand b	data type of result
abs a - a	absolute value negation	signed		signed
a * b a / b a **mod** b a **rem** b a + b a - b	arithmetic operation	unsigned unsigned, natural signed signed, integer	unsigned, natural unsigned signed, integer signed	unsigned unsigned signed signed
a = b a /= b a < b a <= b a > b a >= b	relational operation	unsigned unsigned, natural signed signed, integer	unsigned, natural unsigned signed, integer signed	boolean boolean boolean boolean

Arithmetic Operations on SIGNED and UNSIGNED

- **IEEE numeric_std defines many common operators and type conversions**

function	description	data type of operand a	data type of operand b	data type of result
shift_left(a,b) shift_right(a,b) rotate_left(a,b) rotate_right(a,b)	shift left shift right rotate left rotate right	unsigned, signed	natural	same as a
resize(a,b) std_match(a,b)	resize array compare '-'	unsigned, signed unsigned, signed std_logic_vector, std_logic	natural same as a	same as a boolean
to_integer(a) to_unsigned(a,b) to_signed(a,b)	data type conversion	unsigned, signed natural integer	 natural natural	integer unsigned signed

Type Conversions to/from SIGNED and UNSIGNED

- STD_LOGIC_VECTOR, UNSIGNED, and SIGNED are defined as arrays of std_logic
- **However**, they are **considered as different types**

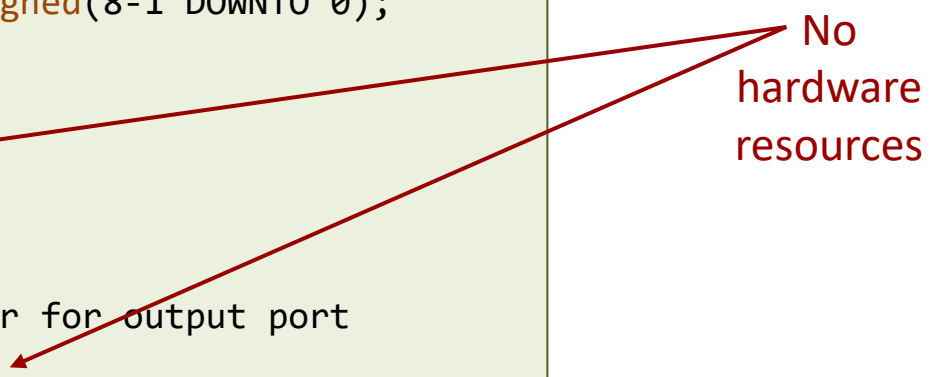
- **Type conversion functions needed** between these types
  - Second "length" argument required for source types without explicit length (INTEGER)
- **Type conversion requires NO hardware resources**

From	To	Cast/Function
std_logic_vector	unsigned	unsigned(std_logic_vector)
std_logic_vector	signed	signed(std_logic_vector)
unsigned	std_logic_vector	std_logic_vector(unsigned)
unsigned	signed	singed(unsigned)
signed	std_logic_vector	std_logic_vector(signed)
signed	unsigned	unsigned(signed)
unsigned	integer	to_integer(unsigned)
signed	integer	to_integer(signed)
integer	unsigned	to_unsigned(integer, no_of_bits)
integer	signed	to_signed(integer, no_of_bits)

Arithmetic Operations Example

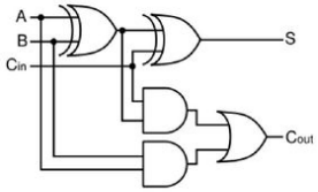
- Basic Arithmetic Operations: ENTITY ports are std_logic_vector

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.all;

ENTITY adder IS
     PORT (AxDI,BxDI        : IN  std_logic_vector(8-1 downto 0);
           CxDO             : OUT std_logic_vector(8-1 downto 0));
END adder;

ARCHITECTURE rtl OF adder IS
         -- signal declaration
         SIGNAL SgnAxD, SgnBxD, SgnCxD : signed(8-1 DOWNTO 0);
BEGIN
         -- Type conversion
          SgnAxD <= signed(AxDI);
          SgnBxD <= signed(BxDI);
         -- Arithmetic
          SgnCxD <= SgnAxD + SgnBxD;
         -- Convert back to std_logic_vector for output port
          CxDO <= std_logic_vector(SgnCxD);
END rtl;
```



The Problem of Describing Logic Efficiently

- **Combinational** logic is the work-horse of any digital circuit as it **performs the data manipulation** (algorithm/operations)



Truth Table

Inputs			Outputs	
A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

- **Truth/look-up tables are the most generic way to describe Boolean logic**, **but also the most tedious one**: difficult to formulate and to read/interpret

- Some more efficient means exist to describe specific types of logic:
  - Sometimes operations are naturally described in Boolean equations, but these quickly become difficult to create or understand
  - Arithmetic operations describe a frequently used specific subset of operations and we know how to map these to Boolean logic (from many early research works)

Need an efficient way to describe Boolean logic

The Problem of Describing Logic Efficiently

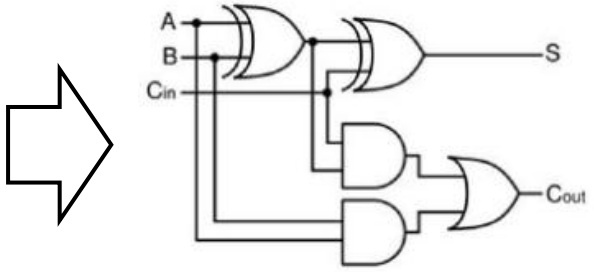
- **Combinational** logic is the work-horse of any digital circuit as it **performs the data manipulation** (algorithm/operations)

- **Truth/look-up tables (LUTs) are the most generic way to describe Boolean logic**
  - **Any** fully arbitrary **logic function can** always **be described by a truth/look-up table**
  - Inputs are the index into the LUT
  - LUT can have one or multiple outputs

- **Synthesis turns LUTs into Boolean functions (netlist of logic gates) that implement the corresponding logic**

Truth Table

Inputs			Outputs	
A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



LUT Realization of a Boolean Function (Logic)

- **Full adder** with three-inputs and two outputs, **realized as two LUTs**
  - **1-D LUTs defined as array constants** (std_logic_vector)
  - **LUT input used as index into the LUT**
    - Concatenate multiple inputs to a vector and convert to an integer-type to index into the LUT

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.all;

ENTITY FullAdder IS
     PORT (AxDI,BxDI, CINxDI          : IN  std_logic;
           SxSO, COUTxDO              : OUT std_logic);
END FullAdder;

ARCHITECTURE rtl OF FullAdder IS
         CONSTANT S    : std_logic_vector(0 to 7) := "01101001";
         CONSTANT COUT : std_logic_vector(0 to 7) := "00010111";
BEGIN
         SxDO    <= S(to_integer(AxDI & BxDI & CINxDI));
         COUTxDO <= COUT(to_integer(AxDI & BxDI & CINxDI));
END rtl;
```

Inputs			Outputs	
A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

The Problem of Describing Logic Efficiently

- Some more efficient means exist to describe specific types of logic:
  - Sometimes operations are naturally described in Boolean equations
  - Arithmetic operations describe a frequently used specific subset of operations and we know how to map these to Boolean logic (from many early research works)

- However, **in most cases, LUTs are difficult to formulate and to read or interpret**

Need an efficient way to describe Boolean logic

Efficiently Describing Logic as Decision Trees

- Need a more systematic way to define a Boolean function based on “operational requirements”

- Completely **arbitrary functions are rare** in practice
  - Some inputs or combinations of some inputs render the output independent of all other inputs
  - In Karnaugh maps, we refer to this as “combining minterms”

- Such truth tables can be expressed efficiently and interpreted as decision diagrams
  - Interpretation as decision trees
  - Formulation based on if-then-elsif-elsif-...-else

	00	01	11	10
00	1	0	0	1
01	0	0	1	0
11	0	0	0	0
10	0	0	0	0

Designing Logic “with/as Multiplexers” (1/2)

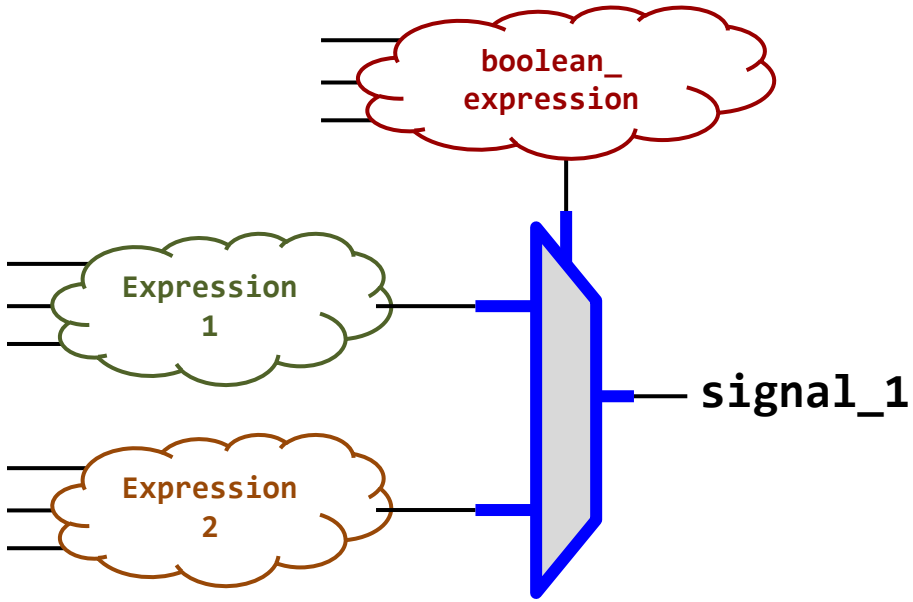
- Besides boolean and arithmetic operations, the **CONDITIONAL ASSIGNMENT** **is a fundamental building block for HDL-based hardware design**

- **The simplest conditional assignment corresponds to a 2-input MUX**
  - natural way to **combine 2 pieces of logic into one**

HDL
Pseudo-Code
Conditional Assignment

```
If boolean_expression then
        signal_1 <= expression_1
Else
        signal_1 <= expression_2
End
```

Hardware
2-input MUX



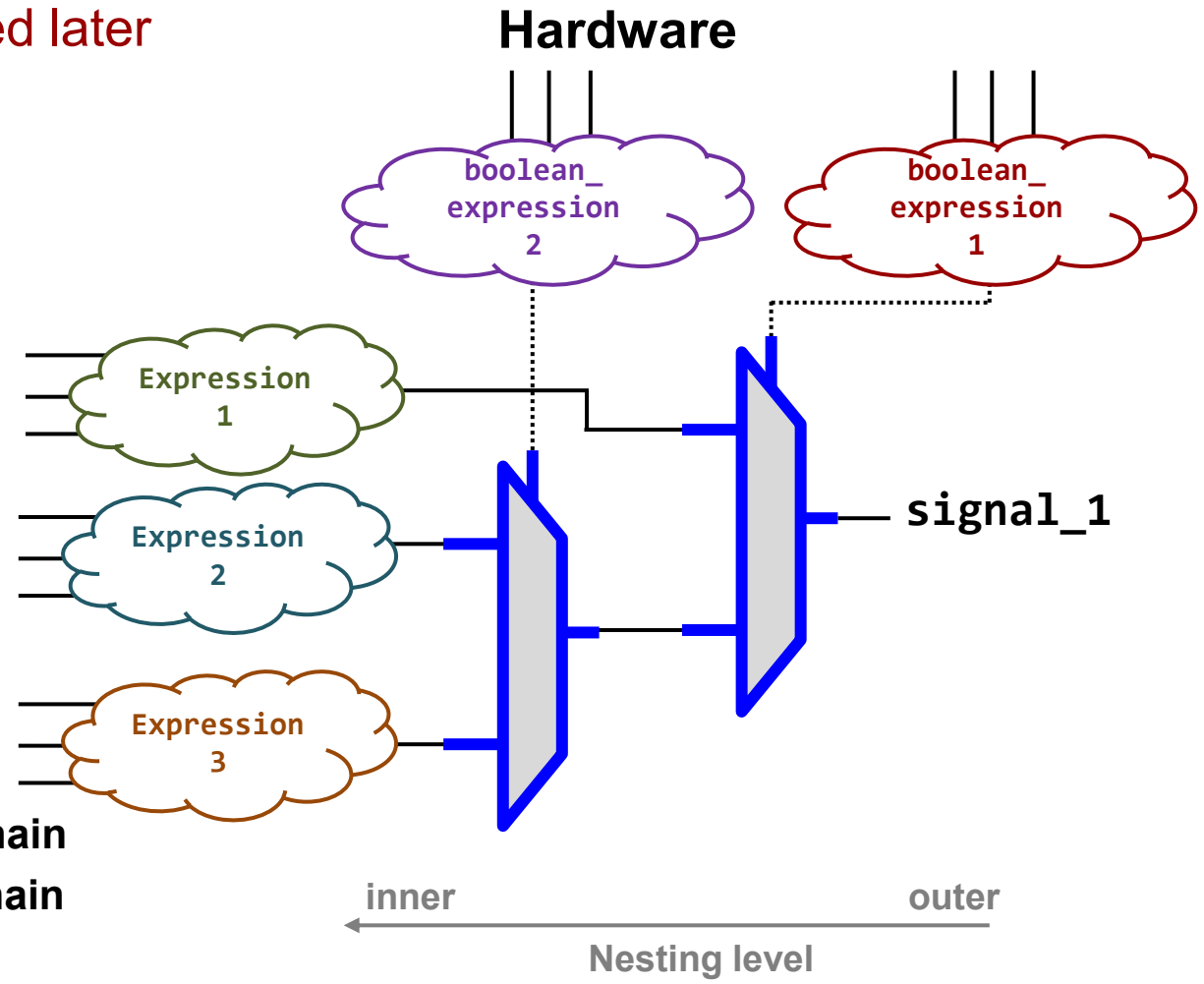
Designing Logic "with/as Multiplexers" (2/2)

- How to describe the **combination of multiple expressions?**
  - **Nested IF statements:** VHDL version described later

HDL Pseudo-Code

```
If boolean_expression_1 then
        signal_1 <= expression_1
Else
        If boolean_expression_2 then
                signal_1 <= expression_2
        Else
                signal_1 <= expression_3
        End
End
```

Nesting level



  - **Nesting level**
    - **encodes priority in the HDL description**
    - **defines the level of the MUX in the HW**
      - **Outer-most IF corresponds to last MUX in the chain**
      - **Inner-most IF corresponds to first MUX in the chain**

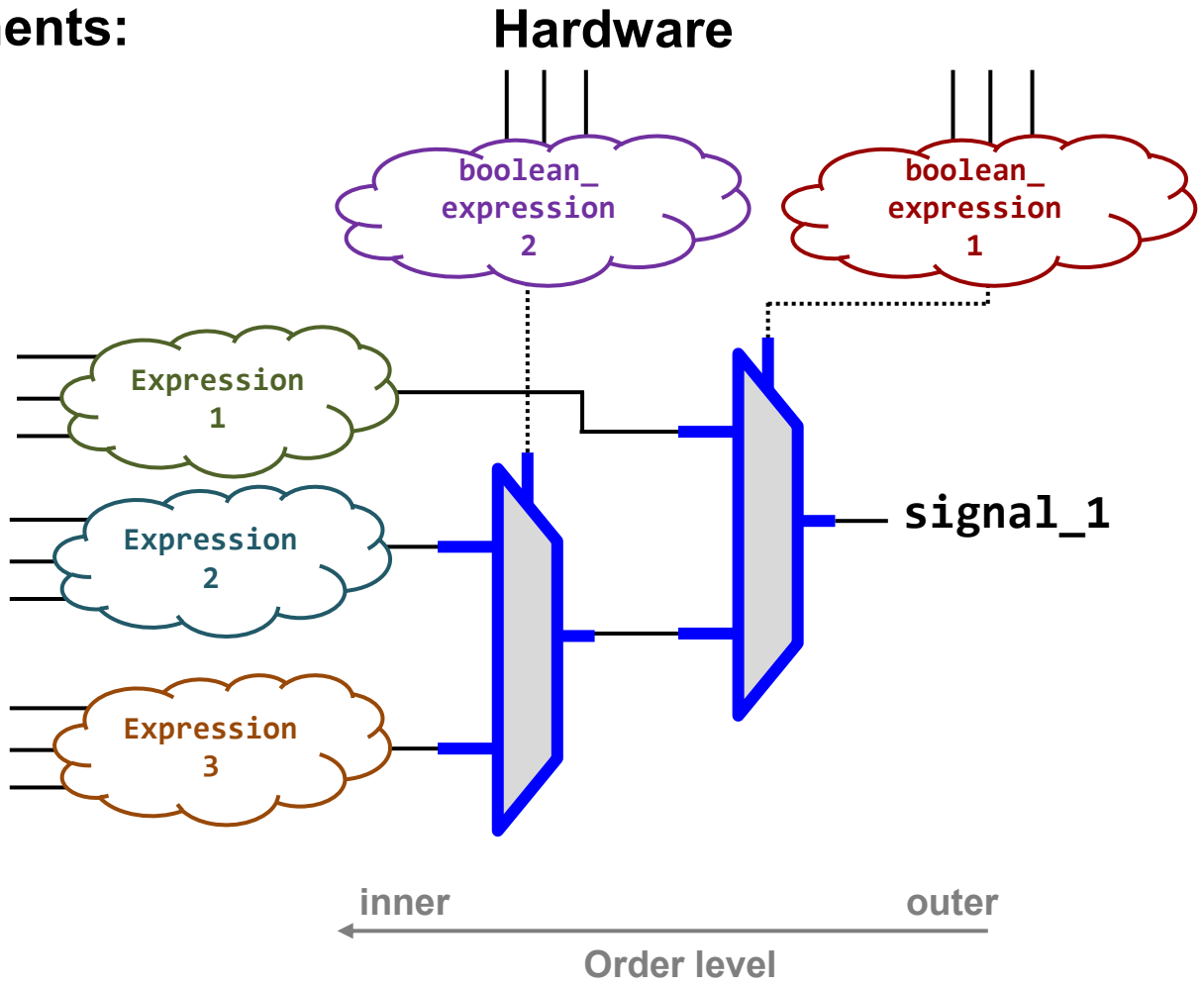
Designing Logic "with/as Multiplexers" (2/2)

- How to describe the **combination of multiple expressions?**
  - **Combined IF, ELSIF, ELSIF, … , ELSE statements:**

HDL Pseudo-Code

```
If boolean_expression_1 then
        signal_1 <= expression_1
Elsif boolean_expression_2 then
        signal_1 <= expression_2
Else
        signal_1 <= expression_3
End
```

Order



  - **Order**
    - **encodes priority in the HDL description**
    - **defines the level of the MUX in the HW**
      - **First IF corresponds to last MUX in the chain**
      - **Last IF corresponds to first MUX in the chain**

Conditional Assignments (MUXes) in VHDL (1/2)

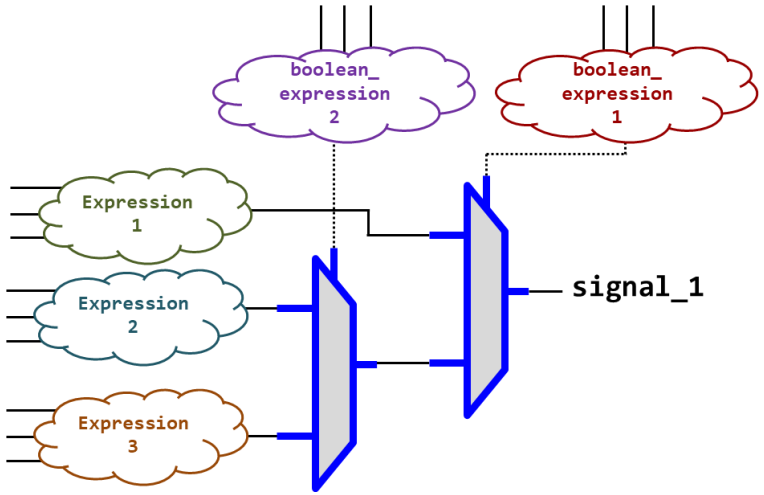
- Due to their importance and power to describe logic in an intuitive manner, **VHDL provides many options for conditional assignments**

  - **WHEN-ELSE** statement **for multiple** different **binary conditions (IF, ELSIF, ELSIF, … ELSE)**

```
target_signal <= expression_1 WHEN boolean_expression_1 ELSE
                 expression_2 WHEN boolean_expression_2 ELSE
                 …
                 expression_N;
```

Order= Priority

ELSE



- **Multiple conditions may be true at the same time**
  - **Order encodes the priority:** only first one is relevant

- **Note**: often `expression_x` is simply a *signal*

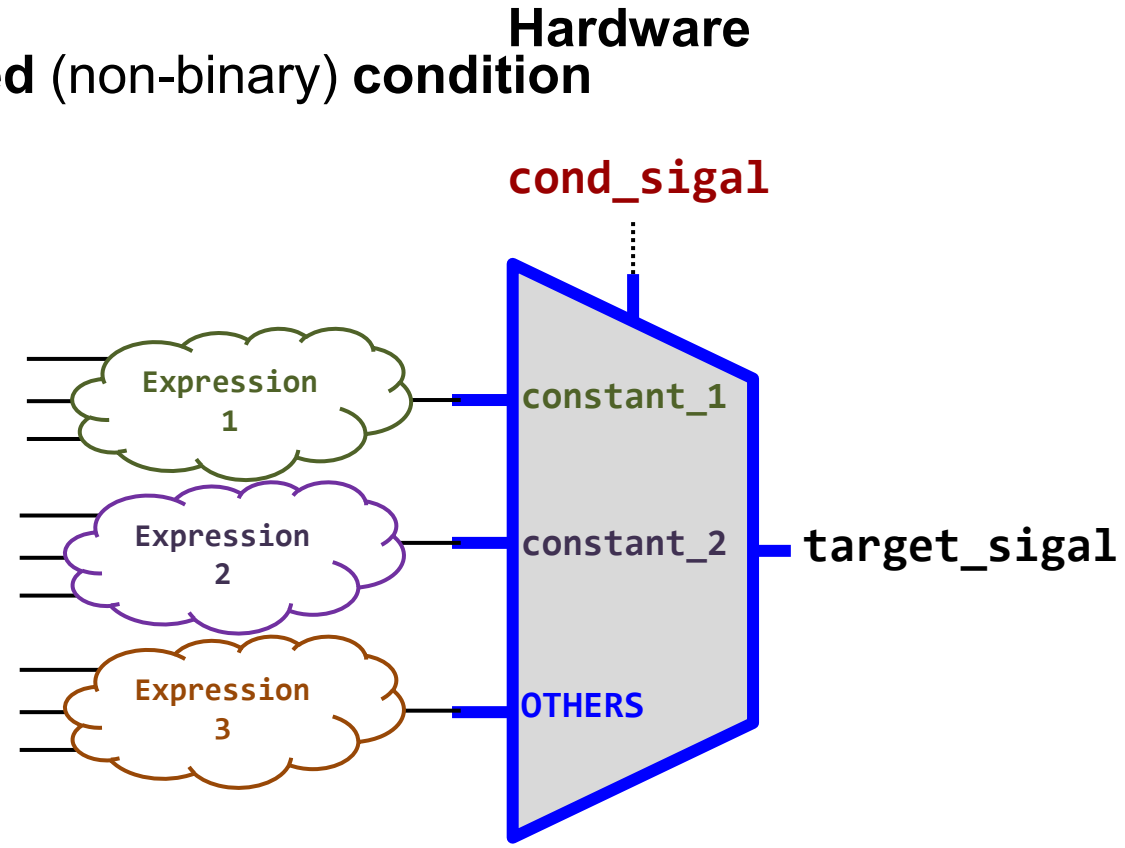
Conditional Assignments (MUXes) in VHDL (2/2)

- Due to their importance and power to describe logic in an intuitive manner, **VHDL provides many options for conditional assignments**

  - **WITH-SELECT** statement **for a single multi-valued** (non-binary) **condition**

```
WITH cond_signal SELECT
   target_signal <=
   expression_1 WHEN constant_1,
   expression_2 WHEN constant_2,
   …
   expression_N WHEN OTHERS;
```

ELSE



  - **Only one condition is true at the same time**
  - **Note**: often `expression_x` is simply a *signal*

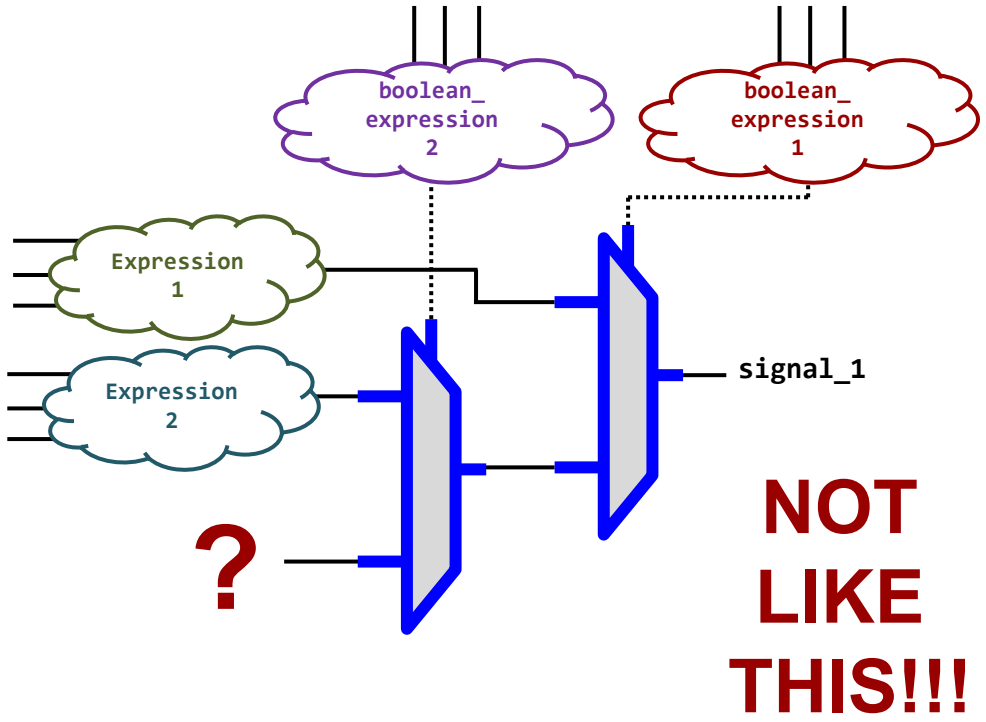
An Important Remark on Conditional Assignments

- Consider the pseudo-code below, which is common practice in Software

```
If boolean_expression_1 then
        signal_1 <= expression_1
Elsif boolean_expression_2 then
        signal_1 <= expression_2
End
```

- HOWEVER, a **physical wire can never be "not assigned"** any value at all

  - **NOTE: If none of the conditions is met, no assignment is made** to `signal_1`

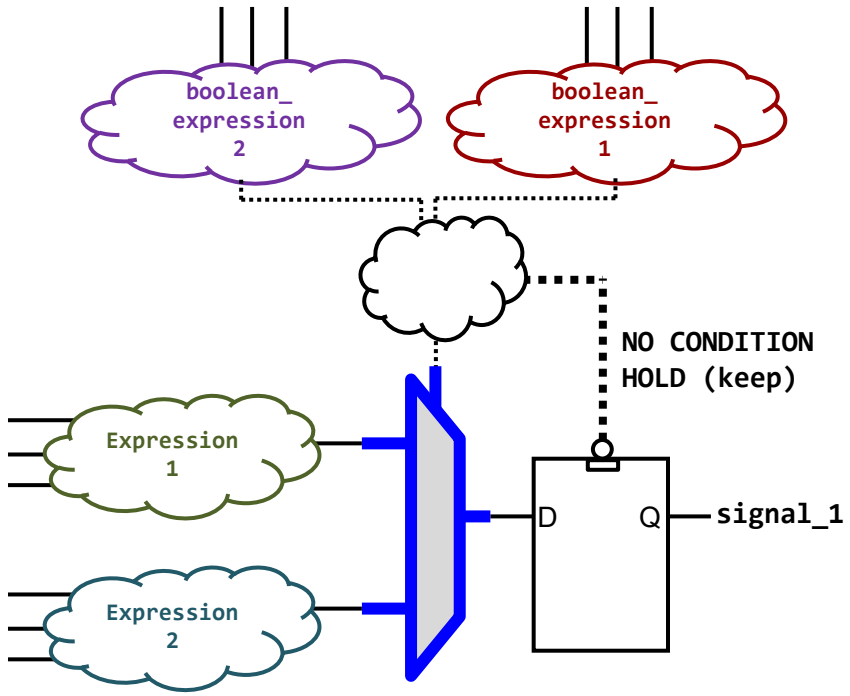


An Important Remark on Conditional Assignments

- Consider the pseudo-code below, which is common practice in Software

```
If boolean_expression_1 then
        signal_1 <= expression_1
Elsif boolean_expression_2 then
        signal_1 <= expression_2
End
```

  - **NOTE: If none of the conditions is met, no assignment is made** to `signal_1`



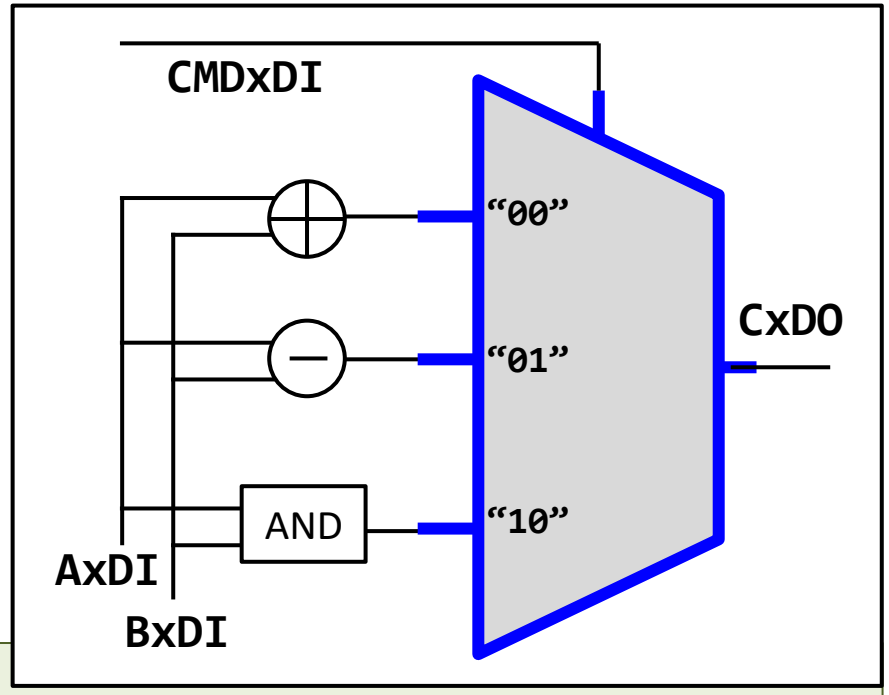
- HOWEVER, a **physical wire can never be "not assigned"** any value at all
- In VHDL, **signals preserve their state if no value is assigned**
  - Works perfectly in simulation, BUT

!
  - Is often not the desired behaviour
  - **Is NOT COMPATIBLE with the rules of synchronous design => issues later in the design**

RULE: every combinational conditional assignment must be complete

Example: A Simple ALU

- **Specification**: 8-bit ALU
  - Three operations: +, -, AND
  - Specified by CMDxSI (2-bit command)
  - For CMDxDI="11", the output DOES NOT MATTER




```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.all
ENTITY my_first_counter IS
  PORT (
    AxDI     : IN  std_logic_vector(8-1 DOWNTO 0);
    BxDI     : IN  std_logic_vector(8-1 DOWNTO 0);
    CMDxSI   : IN  std_logic_vector(2-1 DOWNTO 0);

    CxDO     : OUT std_logic_vector(8-1 DOWNTO 0)
  );
END my_first_counter;
…
```

```
…
ARCHITECTURE rtl OF my_first_counter IS
        -- signal declaration
        SIGNAL SgnCxD          : SIGNED(8-1 DOWNTO 0);
BEGIN
        WITH CMDxSI SELECT
          SgnCxD <=
          SIGNED(AxDI) + SIGNED(BxDI) WHEN "00",
          SIGNED(AxDI) - SIGNED(BxDI) WHEN "01",
          SIGNED(AxDI AND BxDI)       WHEN "10",
          "--------" WHEN OTHERS;

        -- Output assignment with type conversion
        CxDO <= std_logic_vector(SgnCxD);
END rtl;
```

How to Implement Registers in a Clean Way?

- **Synchronous designs requires** the notion of **a state**
  - In VHDL, signals preserve their state (have a state) when nothing is assigned to them
  - This **preservation of states can be exploited to describe registers**, **but** it **must be done with care**

- **Objective**: stick to the **rules of synchronous design**
  - **ONLY the clock triggers a state transition**
  - **Use ONLY positive edge triggered FlipFLops (no latches)**
  - ➔ Incomplete combinational conditional assignments are not the solution to create registers

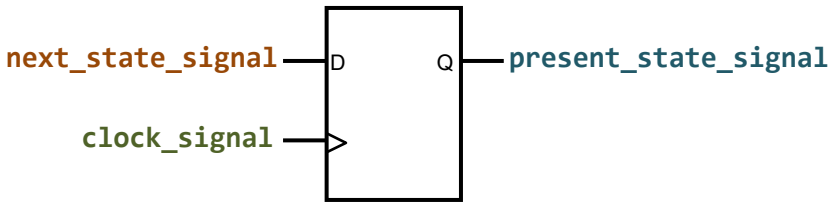
- **Clean solution:**

  Describe registers EXPLICITLY with a well controlled template

Describing Edge Triggered Registers in VHDL

- Two ingredients:
  - A conditional statement that is TRUE when an edge (transition) occurs: `clock_signal'event`
    - Positive edge: `clock_signal'event AND clock_signal = '1'`
  - A special process template that is well understood and clean

- **Assign input signal** of a FlipFlop (`next_state_signal`) to the FlipFlop output signal (`present_state_signal`)
- Conditional **assignment** is incomplete and only **triggers on rising edge of the clock**

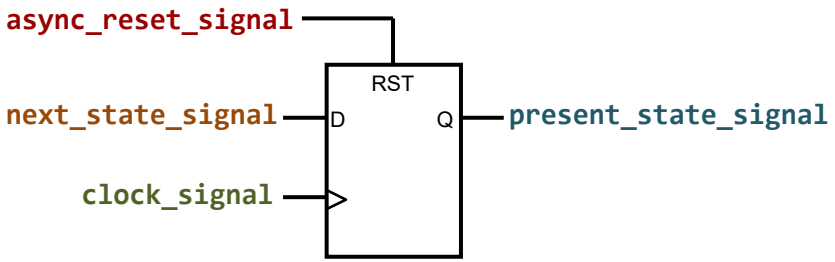



```
ARCHITECTURE architecture_name OF other_entity_name IS
        -- signal declaration
        SIGNAL next_state_signal_1,next_state_signal_2:state_signal_type;
        SIGNAL present_state_signal_1 : state_signal_type;
        SIGNAL present_state_signal_2 : state_signal_type;
BEGIN

        -- Clocked Process, generating a FlipFlop behavior
        p_seq: PROCESS (clock_signal) IS
        BEGIN  -- process name: p_seq
                IF clock_signal'EVENT AND clock_signal = '1' THEN
                        present_state_signal_1 <= next_state_signal_1
                                                | expression;
                        present_state_signal_2 <= next_state_signal_2
                                                | expression;
                END IF;
        END PROCESS p_seq;
END architecture_name;
```

TWO registers

Describing Registers with Asynchronous Reset

- An **asynchronous reset triggers** also a **state transition**, independent of clock
  - The **asynchronous reset** typically **takes precedence** over the clock
    - **Asynchronous reset (async_reset_signal) assigns a constant** to the FlipFlop output signal (present_state_signal)
    - Conditional **assignment** is still incomplete and **now triggers on rising edge of the clock and on the reset signal**
    - Reset can be low- or high-active




```
ARCHITECTURE architecture_name OF other_entity_name IS
        -- signal declaration
        SIGNAL next_state_signal    : state_signal_type;
        SIGNAL present_state_signal : state_signal_type;
BEGIN

        -- Clocked Process, generating a FlipFlop behavior
        p_seq: PROCESS (clock_signal, async_reset_signal) IS
        BEGIN  -- process name: p_seq
                IF async_reset_signal = '0|1' THEN
                        present_state_signal <= constant;
                ELSIF clock_signal'EVENT AND clock_signal = '1' THEN
                        present_state_signal <= next_state_signal
                                                | expression;
                END IF;
        END PROCESS p_seq;

END architecture_name;
```

Example: A Simple Counter (Overflowing)

- **Specification**: 8-bit counter, overflowing (wrap-around)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.all
ENTITY my_first_counter IS
  PORT (
    CLKxCI  : IN std_logic;
    RSTxRBI : IN std_logic;
    CNTxDO  : OUT std_logic_vector(8-1 DOWNTO 0)
  );
END my_first_counter;
…
```

```
…
ARCHITECTURE rtl OF my_first_counter IS
        -- signal declaration
        SIGNAL CNTxDN : UNSIGNED(8-1 DOWNTO 0);
        SIGNAL CNTxDP : UNSIGNED(8-1 DOWNTO 0);
BEGIN
        -- Counting/incrementing (Combinational Logic)
        CNTxDN <= CNTxDP + 1;

        -- Clocked Process, generating a FlipFlop behavior
        p_seq: PROCESS (CLKxCI, RSTxRBI) IS
        BEGIN  -- process name: p_seq
                IF RSTxRBI = '0' THEN
                        CNTxDP <= (OTHERS => '0');
                ELSIF CLKxCI'EVENT AND CLKxCI = '1' THEN
                        CNTxDP <= CNTxDN;
                END IF;
        END PROCESS p_seq;

        -- Output assignment with type conversion
        CNTxDO <= std_logic_vector(CNTxDP);
END rtl;
```