EE 459/500 – HDL Based Digital Design with Programmable Logic

Lecture 5
Concurrent and sequential statements

Read before class:

Chapter 2 from textbook (continue to read from last lecture's topics)

Overview

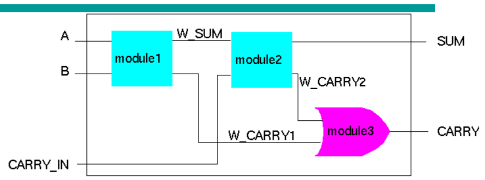
- Components → hierarchy
- Concurrency
- Sequential statements

Components

- **Structural model**: describe how it is composed of subsystems
  - Component declaration and instantiation
- A structural architecture describes the schematic by defining the interconnection of **components**
- Simplest components: associated with design entities describing AND, OR, etc. switching algebra operations; logic gates basically
- Use component statement in structural descriptions

Component declaration

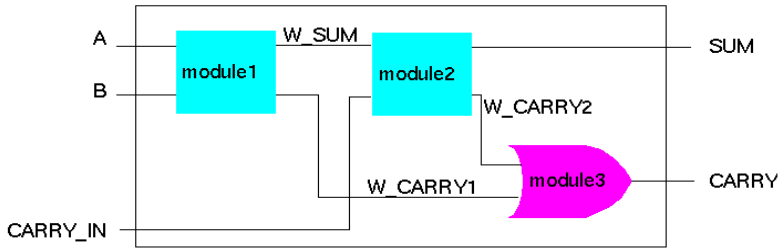
```
entity FULLADDER is
   port (A,B, CARRY_IN: in bit;
         SUM, CARRY:  out bit);
end FULLADDER;

architecture STRUCT of FULLADDER is

  component  HALFADDER
    port (A, B : in bit;
          SUM, CARRY : out bit);
  end component;

  component  ORGATE
    port (A, B : in bit;
          RES : out bit);
  end component;

  signal W_SUM, W_CARRY1, W_CARRY2 : bit;

begin
. . .
end STRUCT;
```



Full adder: 2 halfadders + 1 OR-gate

- In a component declaration, all module types, which will be used in the architecture, are declared.
- Their declaration must occur before the **begin** keyword of the architecture statement.
- The port list elements of the component are called **local elements,** they are **not signals**

Component instantiation → Hierarchy



Full adder: 2 halfadders + 1 OR-gate

- A **module** can be assembled out of several submodules → hierarchical model description
- A purely **structural architecture** does not describe any functionality and contains just a list of components, their instantiation and their interconnections

Component Declaration Format

The following is the FORMAT for declaring components.

COMPONENT component_name

PORT (clause) ;

END COMPONENT;

Note the similarity between component declaration statement and entity declaration statement. Both have a header, port clause, and end statement.

This similarity is not coincidental. Components are virtual design entities.

Component Instantiation

```
architecture STRUCT of FULLADDER is
   component HALFADDER
      port (A, B : in bit;
            SUM, CARRY : out bit);
   end component;

   component ORGATE
      port (A, B : in bit;
            RES : out bit);
   end component;

   signal W_SUM, W_CARRY1, W_CARRY2: bit;

begin   -- statements part

MODULE1:   HALFADDER
   port map( A, B, W_SUM, W_CARRY1 );

MODULE2:   HALFADDER
   port map ( W_SUM, CARRY_IN,
              SUM, W_CARRY2 );

MODULE3:   ORGATE
   port map ( W_CARRY2, W_CARRY1, CARRY );

end STRUCT;
```

- **Component instantiations** occur in the statements part of an architecture (after the keyword "begin").
- The choice of components is restricted to those that are already declared, either in the declarative part of the architecture or in a **package**.
- The connection of signals to the entity port:
  - Default: positional association, the first signal of the port map is connected to the first port from the component declaration.

Component Instantiation: Named Signal Association

```
entity FULLADDER is
  port (A,B, CARRY_IN: in bit;
        SUM, CARRY: out bit);
end FULLADDER;

architecture STRUCT of FULLADDER is

  component HALFADDER
     port (A, B : in bit;
           SUM, CARRY : out bit);
  end component;
  ...
  signal W_SUM, W_CARRY1, W_CARRY2 : bit;

begin

  MODULE1: HALFADDER
      port map ( A      => A,
                 SUM    => W_SUM,
                 B      => B,
                 CARRY => W_CARRY1 );
  ...
end STRUCT;
```

- Named association:
  - left side: "formals" (port names from component declaration)
  - right side: "actuals" (architecture signals)
  - Independent of order in component declaration

Syntax of the Component Instantiation Statement

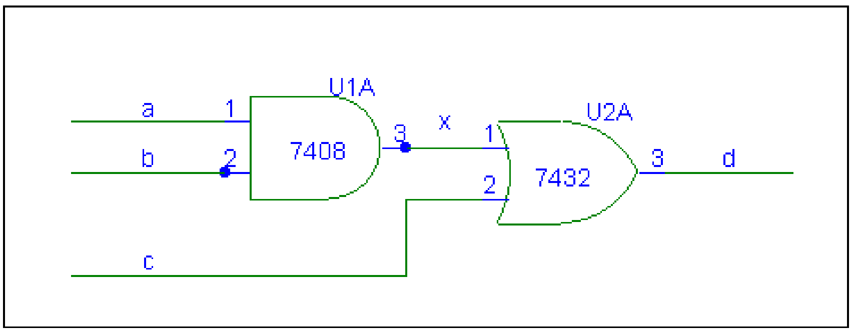
label : component_name
    [GENERIC MAP (association_list)]
    [PORT MAP (association_list)];

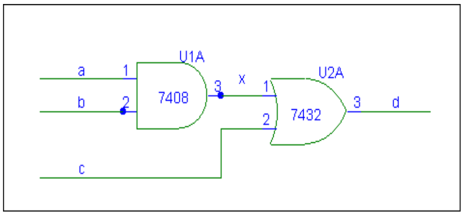
GENERIC MAP is optional if there are no generics declared within the entity declaration of the instantiated component or there is no need to override the declared generic.

PORT MAP describes how this actual component instance is connected to the rest of the system.

Example 1

Write a VHDL description for the circuit. Use component instantiation statement and an internal signal *x*.






```
-- These entity declarations may be done
-- in a separate file or package
-- AND gate entity
ENTITY and2 IS
    PORT ( p, q : IN STD_LOGIC;
           r : OUT STD_LOGIC);
END and2;
ARCHITECTURE my_arch OF and2 IS
BEGIN
    r <= p AND q;
END my_arch;

-- OR gate entity
ENTITY or2 IS
    PORT ( p, q : IN STD_LOGIC;
           r : OUT STD_LOGIC);
END or2;
ARCHITECTURE your_arch OF o2 IS
BEGIN
    r <= p OR q;
END your_arch;
```

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

-- Entity declarations
ENTITY example1IS
    PORT (a, b, c : IN STD_LOGIC;
          d : OUT STD_LOGIC);
END example1;

ARCHITECTURE arch1 OF example1 IS
  COMPONENT and2
      PORT ( p, q: IN STD_LOGIC;
             r : OUT STD_LOGIC);
  END COMPONENT;

  COMPONENT or2
      PORT ( p, q: IN STD_LOGIC;
             r : OUT STD_LOGIC);
  END COMPONENT;
-- Declare signals for interconnect ions
  SIGNAL x : BIT;

BEGIN
    U1A : and2 PORT MAP ( a, b, x);
    U2A : or2 PORT MAP ( x, c, d);
END arch1;
```

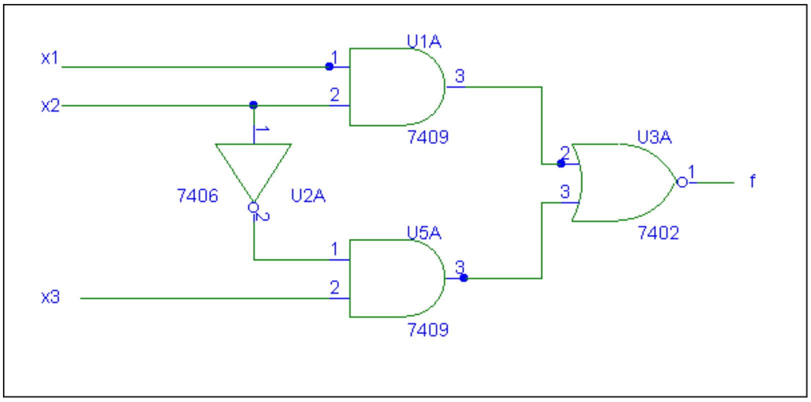
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
-- entity declaration begins
ENTITY example1 IS
        PORT (a, b, c : IN STD_LOGIC;
                  d : OUT STD_LOGIC);
END example1;
-- entity declaration ends. The PORT clause
declares signals {a, b, c, d} that interfaces the
module to the outside world.
```

```
ARCHITECTURE arch1 OF example1 IS
-- component declaration portion of architecture.
-- before a component is instantiated in a circuit, it must first be declared.
-- declared components: AND and OR gates with names "and2" and "or2".

COMPONENT and2
 PORT ( p, q : IN STD_LOGIC;
          r : OUT STD_LOGIC);
END COMPONENT;

COMPONENT or2
 PORT ( p, q : IN STD_LOGIC;
          r : OUT STD_LOGIC);
END COMPONENT;
```

```
-- signals declaration portion of architecture.
-- declare signals to interconnect logic operators or modules.
-- our circuit has an internal signal named x which is used by
-- both components; this signal should also be declared prior
-- to its usage in the architecture body.

SIGNAL x : BIT;

-- component instantiation portion of architecture
-- component instantiation statement connects logic
-- operators or modules to describe the schematic or structure.
BEGIN
      U1A : and2 PORT MAP ( a, b, x );
      U2A : or2 PORT MAP ( x, c, d );
END arch1;
```

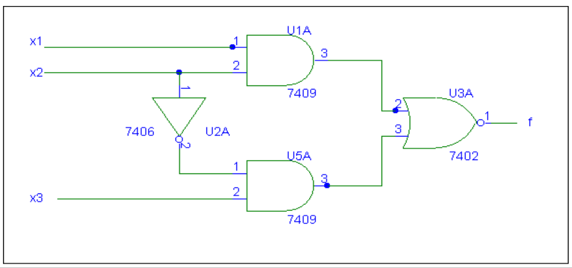
Example 2

Write a VHDL code for the given circuit. The inputs are x1, x2, x3 and the output is f



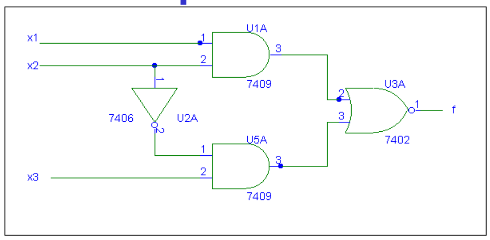
Example 2: behavioural description

```
entity example2 is
        port (x1, x2, x3: in bit;
              f: out bit);
end example2;

-- "my_behavioral" is user defined
architecture my_behavioral of example2 is
begin
        f <= (x1 and x2) nor (not x2 and x3);
end my_behavioral;
```

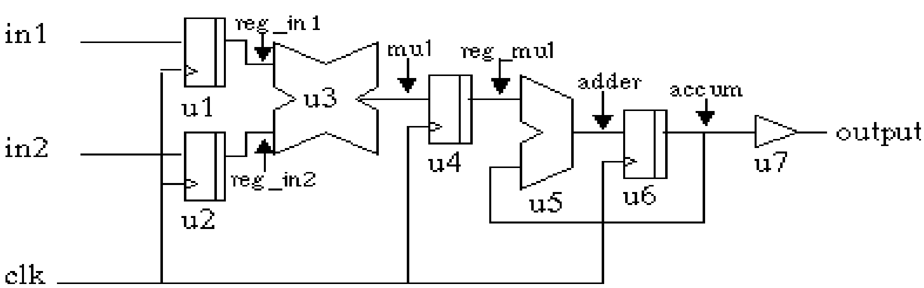


Example 2: structural description

```
entity example2 is
    port (x1, x2, x3: in bit;
          f: out bit);
end example2;

architecture structure of example2 is
  component and is
    port (a, b: in bit, f: out bit);
  end component and;

  component nor is
    port (a,b: in bit; f: out bit);
  end component nor;

  component inv is
    port (a: in bit; f: out bit);
  end component inv;

  signal x2bar,u1aout,u5aout: bit;

  begin
    u1a: component and port map(x1,x2,u1aout)
    u2a: component inv port map(x2,x2bar);
    u3a: component nor port map(u1aout,u5aout,f);
    u5a: component and port map (x2bar,x3,u5aout);
  end structure;
```



Example 3: Multiply-accumulator



A two-input multiply accumulator device: multiply two 16 bit inputs. Internal signals are: reg_in1, reg_in2, mul, reg_mul, adder, accum.

Two serial-to-parallel registers u1 and u2 convert the input bit stream of the data into a 16-wide data. Multiplier u3 sends its output to the adder u5 via register u4.

*Assume for the moment that these components are found in a package called **my_beautiful_package**, located in the library **WORK**. More on packages and libraries later!*

*The **USE** statement makes all the components in this package visible to our design.*

```
LIBRARY WORK;

USE WORK.my_beautiful_package.ALL;

ENTITY mac IS

  GENERIC (tco : time := 10 ns);
  PORT ( in1, in2 : IN BIT_VECTOR (15 DOWNTO 0);
              clk : IN BIT;
              output : OUT BIT_VECTOR(31 DOWNTO 0);

END mac;
```

```
ARCHITECTURE structure_is_cool OF mac IS
-- component declaration part, components include a register called reg,
-- an adder called adder, a multiplier called multiply, a buffer called buf
COMPONENT reg
    GENERIC ( width : integer := 16);
    PORT ( d   : IN  BIT_VECTOR (width-1 DOWNTO 0);
             clk : IN  BIT;
              q   : OUT BIT_VECTOR (width-1 DOWNTO 0));
END COMPONENT;

COMPONENT adder
    PORT ( port1, port2 : IN  BIT_VECTOR (31 DOWNTO 0);
              output        : OUT BIT_VECTOR (31 DOWNTO 0));
END COMPONENT;

COMPONENT multiply
     PORT ( port1, port2 : IN BIT_VECTOR (15 DOWNTO 0);
               output        : OUT BIT_VECTOR (31 DOWNTO 0));
END COMPONENT;

COMPONENT buf
     PORT ( input   : IN BIT_VECTOR (31 DOWNTO 0);
               output : OUT BIT_VECTOR (31 DOWNTO 0));
END COMPONENT;
```

```
-- signals declaration portion of architecture

SIGNAL reg_in1, reg_in2 : BIT_VECTOR (15 DOWNTO 0);
SIGNAL mul, reg_mul, adder, accum : BIT_VECTOR (31 DOWNTO 0);

-- components instantiation and logic interconnection

BEGIN

 u1: reg GENERIC MAP(16)  PORT MAP ( in1, clk, reg_in1);
 u2: reg GENERIC MAP(16)  PORT MAP ( in2, clk, reg_in2);
 u3: multiply PORT MAP (reg_in1, reg_in2, mul);
 u4: reg GENERIC MAP(32) PORT MAP (mul, clk, reg_mul);
 u5: adder PORT MAP (reg_mul, accum, adder);
 u6: reg GENERIC MAP(32) PORT MAP (adder, clk, accum);
 u7: buf PORT MAP (accum, output);

END structure;
```

Overview

- Components → hierarchy
- Concurrency
- Sequential statements

Classification

- VHDL provides mainly two types/classes of statements that can be used to assign logic values to signals.
  - **Concurrent Statements**
    - The term concurrent means that the VHDL statements are executed only when associated signals change value. There is no master procedural flow control, each concurrent statement executes when driven by an event.
  - **Sequential Statements**
    - Most statements found in programming languages such as BASIC, PASCAL, C, C++, etc. execute in a sequential fashion. Sequential statements execute only when encountered by the procedural flow of control and the textual order in which statements appear determines the order in which they execute.

Concurrency

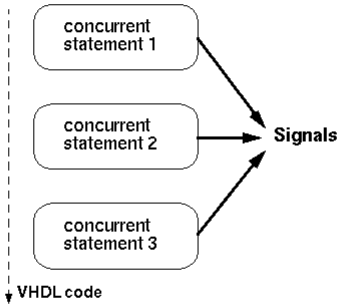
- VHDL concurrent statements execute in a ***concurrent*** fashion (all at the same time, concurrently or simultaneously). Individual statements execute only when "associated" signals change value.
- There is no master, procedural flow of control; each concurrent statement execute in a *nonprocedural stimulus/response*.

```
ENTITY example1 IS
      PORT ( x1, x2, x3 : IN BIT;
             f : OUT BIT);
END example1;

ARCHITECTURE logicFunc OF example1 IS
 SIGNAL a1, b2:  BIT;

 BEGIN -- Concurrent signal assignment statements
    a1 <= x1 AND x2;
    b1 <= NOT x2 AND x3;
    f <= a1 NOR b1;
END logicFunc;
```

Concurrent statements!

Concurrent Statements

- VHDL provides several types of concurrent statements:
  - Signal assignment statement
    - Simple Assignment Statement
    - Selected Assignment Statement
    - Conditional Assignment Statement
  - Component instantiation statement
  - Generate statement
  - Process statement (its declaration, not what is inside the process!)
  - Concurrent assertion statement
  - Procedure statement
  - Block statement



Signal assignments: data objects

- Data objects hold a value of specified type. They belong to one of three classes:
  - Constants
  - Variables
  - Signals
- **Constants** and **variables** are typically used to aid with modelling the behaviour of the circuit
- **Signals** are typically used to model wires and flip-flops
- Must be declared before they are used

Constants

- A constant holds a value that cannot be changed within the design description.
- **Constant** must be declared in *Entity, Architecture, Process, Package.*
  - A constant defined in a package can be referenced by any entity or architecture for which the package is used.
  - **Local property:** A constant declared in an entity/architecture/process is visible only within the local environment
- **Example:**
  - **constant** RISE_TIME: TIME := 10 ns;

  -- declares a constant RISE_TIME of type TIME, with a value of 10 ns
  - **constant** BUS_WIDTH: INTEGER := 8;

  -- declares a constant BUS_WIDTH of type INTEGER with a value of 8

Constants

```
ENTITY example IS

   CONSTANT width : integer :=8;

   PORT ( input : IN bit_vector (width-1 DOWNTO 0);
             output: OUT bit_vector (width-1 DOWNTO 0);
END example;
```

- The above constant represents the width of a register.
- The identifier **width** is used at several points in the code. To change the width requires only that the constant declaration be changed and the code recompiled.

Signals

- **Signals** represent or model logic signals or wires in a real circuit. Signals can also represent the state of a memory

- There are three places in which signals can be declared in a VHDL code
  - Entity declaration
  - Declarative part of an architecture
  - Declarative part of a package

Signals

- A signal has to be declared with an associated type:
  - **SIGNAL** signal_name : type_name;
- The signal's type_name determines the legal values that the signal can have and its legal use in VHDL code
- **Signal types:**
  - bit, bit_vector, std_logic, std_logic_vector, std_ulogic, signed, unsigned, integer, numeration, boolean

Signals – Example 1

- SIGNAL Ain : BIT_VECTOR (1 TO 4);
- Note:
  - The syntax "lowest_index TO highest_index" is useful for a multi-bit signal that is simply an array of bits.
  - In the signal Ain, the most-significant (left-most) bit is referenced using lowest_index, and the least-significant (right-most) bit referenced using highest index.
- Example:
  - The signal "Ain" comprises 4 bit objects.
  - The assignment statement   Ain <= "1010"

    results in: Ain(1) = 1, Ain(2) = 0, Ain(3) = 1, Ain(4) = 0

Signals – Example 2

- SIGNAL My_Byte: BIT_VECTOR (7 downto 0);
- Note:
  - The signal "My_Byte" has eight bit objects.
  - The assignment statement:

    My_Byte <= "10011000";

    results in: My_Byte(7)=1, My_Byte(6)=0, My_Byte(5)=0, My_Byte(4)=1, My_Byte(3)=1, My_Byte(2)=0, My_Byte(1)=0, My_Byte(0)=0

Simple Signal Assignment Statement

- A simple signal assignment statement is used for a logic or an arithmetic expression
- General format is:
  - **Signal name <= Expression;**
  - **<=** : VHDL assignment operator.
  - It is the only operator which can be used to assign a waveform to a signal.
- Example:

```
f <= (x1 AND x2) NOR (NOT x2 AND x3);
```

Simple Signal Assignment Statement

```
ENTITY example1 IS
        PORT ( x1, x2, x3 : IN BIT;
                 f : OUT BIT);
END example1;

ARCHITECTURE logicFunc OF example1 IS

BEGIN
     -- Simple signal assignment statement
     f <= (x1 AND x2) NOR (NOT x2 AND x3);
END logicFunc;
```

Variables

- A **VARIABLE**, unlike a SIGNAL, does not necessarily represent a wire in a circuit.
- Variables can be used in sequential areas only
  - The scope of a variable is the process or the subprogram.
  - A variable in a subprogram does not retain its value between calls.
- Variable assignment is immediate, not scheduled.
- More info later (during processes discussion).

Operators

- **Logical: not, and, or, nand, nor, xor, xnor**
- **Arithmetic:**

Operator	Definition
+	addition
-	Subtraction
*	Multiply
/	divide
**	Exponentiation
MOD	modulus
REM	remainder
&	Concatenation

- **Relational:**

Operator	Definition
=	equal
/=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

Miscellaneous Operators

Operator	Definition
ABS	Absolute Value
SLL	Shift left logical
SRL	Shift right logical
SLA	Shift left arithmetic
SRA	Shift right arithmetic
ROL	Rotate left
ROR	Rotate right

Generate Statements

- **Generate Statements**: describe regular and/or slightly irregular structure by automatically generating component instantiations instead of manually writing each instantiation.
  - E.g., if we implement the three-state buffers for a 32-bit bus using component instantiation statement, we will have to instantiate the three-state buffer component 32 times. In such cases, a ***generate*** statement is preferred.
- There are two variants of the generate statement:
  - FOR GENERATE statement
    - Provides a convenient way of repeating either a logic equation or a component instantiation.
  - IF GENERATE statement

Generate Statements Formats

- The syntax of **GENERATE** statement:

```
Label : generation_scheme GENERATE
    [concurrent_statements]
END GENERATE [label];
```

Where **generation_scheme**:

FOR generate_specification

or

IF condition

- The beginning delimiter: GENERATE.
- The ending delimiter: END GENERATE.
- A label is required for the generate statement and is optional when used with the END GENERATE statement.

Example: 16-bit register

A 16-bit wide bus is to be connected to a 16-bit register. Create such a register using a series of 1-bit FFs. Utilize the GENERATE VHDL construct to do it.

```
LIBRARY work;
USE WORK.my_beautiful_package.all;

ENTITY reg16 IS
   PORT ( input  : IN STD_LOGIC_VECTOR (0 to 15);
          clock  : IN STD_LOGIC;
          output : OUT STD_LOGIC_VECTOR (0 to 15);
END reg16;
```

```
ARCHITECTURE bus16_wide OF reg16 IS

 COMPONENT dff
     PORT ( d, clk : IN STD_LOGIC,
             q : OUT STD_LOGIC);
 END COMPONENT;

BEGIN

-- "i" is the counter and does not need to be
-- declared. It will automatically increase by 1
-- for each loop through the generate statement.

G1 : FOR  i  IN  0 to 15  GENERATE
     dff1: dff PORT MAP (input (i), clock, output(i));
END GENERATE G1;
END bus16_wide;
```

Overview

- Components → hierarchy
- Concurrency
- Sequential statements

Sequential Statements

- Executed according to the order in which they appear.
- Permitted only within processes.
- Used to describe algorithms.
- There are six variants of the sequential statement, namely:
  - PROCESS Statement
  - IF-THEN-ELSE Statement
  - CASE Statement
  - LOOP Statement
  - WAIT Statement
  - ASSERT Statement

Process Statement

- **PROCESS** statement:
  - basic building block for behavioral modeling of digital systems.
  - concurrent shell in which a sequential statement can be executed.
    - appears inside an architecture body, and it encloses other statements within it.
    - IF, CASE, LOOP, and WAIT statements can appear only inside a process.
    - All statements with a process are executed sequentially when the process becomes active.

Process Statement Format

```
[Process_label] : PROCESS [(sensitivity_list)] [is]
      Process_declarative_region
BEGIN
      Process_statement_region
END PROCESS [Process_label]
```

- The optional label allows for a user-defined name for the process.
- The keyword **PROCESS** is the beginning delimiter of the process.
- The **END PROCESS** is the ending delimiter of the process statement.

Process Statement Sensitivity List

- Sensitivity list: contains the signals that trigger the process.
- The process statement begins to execute if any of the signals sensitivity list contains an event.
- Once activated by a sensitivity list event, the process statement executes statements in a sequential manner.
- Upon reaching the end of the process, execution suspends until another event occurs from the sensitivity list.

- **Process_declarative_region** may include:
  - type declaration
  - constant declaration
  - variable declaration

(Note: no signal declaration)

- **Process_Statement region** may include:
  - -- signal assignment statement
  - -- variable assignment statement
  - -- IF statement
  - -- CASE statement
  - -- LOOP statement
  - -- WAIT statement

Example 1

```
PROCESS (clock)
BEGIN
  -- toggles clock every 50 ns
  clock <= not clock after 50 ns;
END PROCESS;
```

- This process is sensitive to the signal "clock". When an event occurs on clock, the process will execute.
- Within the process_statement_region of the process is a simple signal assignment statement. This statement inverts the value of clock after 50 ns.

Variables (see also Appendix A, at the end)

```
architecture RTL of XYZ is
  signal A, B, C : integer range 0 to 7;
  signal Y, Z    : integer range 0 to 15;
begin
  process (A, B, C)
    variable M, N : integer range 0 to 7;
  begin
    M := A;
    N := B;
    Z <= M + N;
    M := C;
    Y <= M + N;
  end process;
end RTL;
```

- **Variables** can be only defined in a **process or subprogram**.
  - Variables are only accessible within this process.
- In a process, the last signal assignment to a signal is carried out when the process execution is suspended. Value assignments to variables, however, are carried out immediately.
- '**<=**' signal assignment
- '**:=**' variable assignment

Summary

- Component instantiation facilitates hierarchical structural VHDL description
- Concurrency is a big deal in VHDL
- Sequential statements → inside processes

Appendix A: Variables vs. Signals

- There are three main differences between variable and signal assignments:

1) Syntax: for the variable assignment '**:=**', for the signal assignment operator '**<=**'

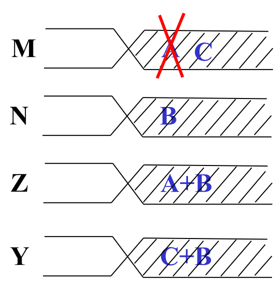
2) Timing: Variables are assigned immediately, while signals are assigned at a future **delta time**.

3) Range: Variables are used for local processes and signals are used to pass information among concurrent statements.

Variables vs. Signals

```
...
signal A, B : integer;
signal C    : integer;
signal Y, Z : integer;

begin
  process (A, B, C)
   variable M, N: integer;
  begin
    M := A;
    N := B;
    Z <= M + N;
    M := C;
    Y <= M + N;
  end process;
...
```

M, N: variables

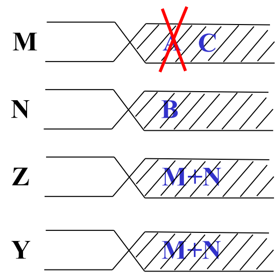


- The 2nd adder input is connected to C

Variables vs. Signals

```
...
signal A, B : integer;
signal C    : integer;
signal Y, Z : integer;
signal M, N : integer;
begin
  process (A,B,C,M,N)
  begin
    M <= A;
    N <= B;
    Z <= M + N;
    M <= C;
    Y <= M + N;
  end process;
...
```

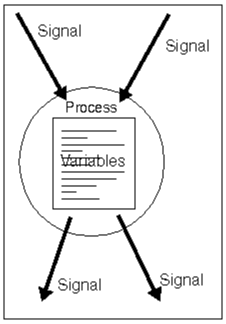
M, N: signals



- Signal values are assigned after the process execution
- Only the last signal assignment is carried out
  - M <= A; is overwritten by M <= C;
- The intermediate signals have to be added to the sensitivity list, as they are read during process execution.

Use of Variables

- Variables are suited for the implementation of algorithms.
- A variable behaves like you would expect in a software programming language.
- They can be used for local storage inside processes.
- Since all variables scope is only within the current PROCESS where the variable is declared, it is always necessary to assign the final values of variables to signals if they are used outside of the process.



Variables: Example

```
-- Parity Calculation
entity PARITY is
 port (DATA: in bit_vector(3 downto 0);
        ODD: out bit);
end PARITY;

architecture RTL of PARITY is

begin
  process (DATA)
    variable TMP : bit;
  begin
    TMP := '0';
    for I in DATA'low to DATA'high loop
       TMP := TMP xor DATA(I);
    end loop;
    ODD <= TMP;
  end process;

end RTL;
```

- While a scalar signal can always be associated with a wire, this is not valid for variables.
- In the example, FOR LOOP is executed four times. Each time the variable TMP describes a different line of the resulting hardware. The different lines are the outputs of the corresponding XOR gates.