VHDL Quick Start

Edward Gatt
Modeling Digital Systems

- VHDL is for writing models of a system
- Reasons for modeling
  - requirements specification – design needs to meet specifications which maybe incomplete or ambiguous and formal model is necessary to communicate requirements
  - formal modeling useful to communicate understanding of the function to the user. designer cannot predict all uses of the system. therefore presents model to user to check it against set of inputs – also useful in documentation
Modeling Digital Systems

• Reasons for modeling
  – testing using simulation and formal verification - systems can be designed from subsystems each with its own model of behaviour. Compare outputs/inputs from circuit to simulation – if they coincide the design is fine – otherwise need re-designing – process can be re-iterated until we arrive at the bottom level in the design hierarchy and the manufactured product can be verified to meet specifications
  – synthesis – modeling allows automatic synthesis of circuits – function is translated to circuitry saving human costs
Modeling Digital Systems

• Goal

  – most reliable design process, with minimum cost and time – allows optimisation – normally speed vs gate count compromise

  – avoid design errors!
Domains and Levels of Modeling

- Structural
- Geometric
- Functional

(high level of abstraction)
(low level of abstraction)

“Y-chart” due to Gajski & Kahn
Domains and Levels of Modeling

Structural

Functional

Algorithm (behavioral)

Register-Transfer Language

Boolean Equation

Differential Equation

Geometric

“Y-chart” due to Gajski & Kahn
Domains and Levels of Modeling

• Behavioural Model - Function of the entire system may be described by an algorithm – similar to programming

  eg. loop
  for each data input loop
  read the value on the input
  scale the value using a scale factor
  end loop
  wait for 10 ms;
  end loop;
Domains and Levels of Modeling

• Register-Transfer Level (RTL)
  – Storage of data is represented using register variables and transformations are represented by arithmetic and logical operations
  eg. \[ \text{MAR} \leftarrow \text{PC}, \text{memory}_\text{read} \leftarrow 1 \]
  \[ \text{PC} \leftarrow \text{PC} + 1 \]
  \[ \text{wait until ready} = 1 \]
  \[ \text{IR} \leftarrow \text{memory}_\text{data} \]
  \[ \text{memory}_\text{read} \leftarrow 0 \]

• Boolean Algebra – Truth Tables

• Differential Equations – Transistor Behaviour
Domains and Levels of Modeling

“Y-chart” due to Gajski & Kahn
Domains and Levels of Modeling

• Processor Memory Switch (PMS)
  – Describing a system as interconnections of Processing Elements – Memory Components – Input/Output Devices

![Diagram]

PMS Model for a Controller
Domains and Levels of Modeling

• Register Transfer Level

• System can then be translated to gates and transistor implementation
Domains and Levels of Modeling

“Y-chart” due to Gajski & Kahn
Domains and Levels of Modeling

• Floor Planning

Edward Gatt

VHDL
Domains and Levels of Modeling

• Geometric Level of Abstraction
  – Standard Library Cells are used to implement the Registers and Data Transformation Units and must be placed in the areas allocated in the Chip Floor Plan
  – Stick Diagrams

Use of Stick Diagrams to Implement Gate Layout Floorplanning
Domains and Levels of Modeling

• Geometric Level of Abstraction
  – Polygons for Layout Masks
Digital Circuit Design

- Schematic Entry
- HDL
  - VHDL
  - Verilog
- State Diagrams
VHDL

- **VHDL or VHSIC Hardware Description Language** is commonly used as a design-entry language for field-programmable gate arrays and application-specific integrated circuits in electronic design automation of digital circuits.

- VHDL is a fairly general-purpose language, although it requires a simulator on which to run the code. It can read and write files on the host computer, so a VHDL program can be written that generates another VHDL program to be incorporated in the design being developed. Because of this general-purpose nature, it is possible to use VHDL to write a test bench that verifies the functionality of the design using files on the host computer to define stimuli, interacts with the user, and compares results with those expected.

- The key advantage of VHDL when used for systems design is that it allows the behaviour of the required system to be described (modeled) and verified (simulated) before synthesis tools translate the design into real hardware (gates and wires).
VHDL

• VHDL allows the description of a concurrent system (many parts, each with its own sub-behaviour, working together at the same time).

• When a VHDL model is translated into the "gates and wires" that are mapped onto a programmable logic device such as a CPLD or FPGA, then it is the actual hardware being configured, rather than the VHDL code being "executed" as if on some form of a processor chip.

• To start coding in VHDL, one needs a simulation tool. While very few open source VHDL simulators exist today, most commercial vendors offer free, but often limited, versions of their software.

• Furthermore, it is highly recommended to use a synthesis tool even when you do not plan to test your code on real hardware. The reason for this is that you can often see the graphical representation (i.e. gates) of your code after synthesis. Seeing what kind of hardware correspond to your code is a very important step in learning any HDL and becoming a good designer.
Design Flow

Design Entry

Functional Simulation

Vendor
Library Elements

{Basic Gates; Flip Flop; Complex Gates}

Synthesis

{Parasitic Cap Extraction}

Place and Route

Post Layout Simulation

{Check Timing Constraints}

Schematic
VHDL Architectures

• VHDL architectures are divided into two main categories:
  
  – structural: with full circuit details {netlist form}
  
  – functional: description of the functionality of the circuit
    – no need of circuit details
Constructs in VHDL

• There are three main constructs in VHDL:
  – Component Declaration
  – Component Instantiation
  – Component Configuration
Component Declaration

- Define Component name and input/output ports

```vhdl
component component_name

port ( A,B: In BIT; Y: Out BIT);

end component
```

VHDL logic types:
- BIT \{1,0\}
- BIT_VECTOR
- STD_LOGIC
- STD_LOGIC_VECTOR

More versatile types

Can have different values other than \{1,0\}
e.g. u, w, x, z
- u – unresolved
- z – tri-state
- x – don’t care
Component Instantiation

- Instance = “occurrence”

These statements represent the use of a component. They specify:

- A unique name for each instance of the component
- How the ports of the component are to be connected to the rest of the signals.

signal names

A1
B1

component port names

A
B
Y

Y1

signal names

have to be declared before hand
Component Instantiation

instance_name: component_name

port map (A => A1, B => B1, Y=> Y1);

component port names

wire/signal names

signal names

component ports
Component Configuration

- Maps component instantiations to pre-compiled VHDL library design units.

- Component Configurations are ‘named’ to allow multiple configurations to exist for a single design, allowing design alternatives and modifications to be explored in parallel.

```vhdl
configuration config_name of entity_name is
  for component_label: component_name
    use entity library_name.entity_name(architecture_name);
  end for
end config_name
```

“ALL” refers to all instances

Low Level vs High level Description
Example 1: Half Adder Structural VHDL Description

Entity Name: Halfadder
Architecture Name: Structural
Example 1: Half Adder Structural VHDL Description

```vhdl
entity Halfadder is
    port (A1,B1: In BIT; Sum,Carry: Out BIT);
end Halfadder;

architecture structural of Halfadder is

component Xor2
    port (A,B: IN Bit; Y: OUT Bit);
end component;

component Nand2
    port (A,B: IN Bit; Y: OUT Bit);
end component;

signal icarry: BIT;
```

In this section the components to be used are declared.
Example 1: Half Adder Structural VHDL Description

begin

Gate1: Xor2

port map (A => A1, B => B1, Y => sum);

Gate2: Nand2

port map (A => A1, B => B1, Y => icarry);

Gate3: Nand2

port map (A => icarry, B => icarry, Y => carry);

end structural;

in this section the different components are connected together via the port map

component instantiation

Gate1: Xor2

Gate2: Nand2

Gate3: Nand2
Example 1: Half Adder Structural VHDL Description

configuration Halfadderconfig of Halfadder is

for structural

for Gate1:Xor2
use entity work.xorgate2(simple);
end for;

for ALL:Nand2
use entity work.xorgate2(simple);
end for;

end Halfadderconfig;

Assume that Xor2 and Nand2 are pre-compiled in library called ‘work’.

<table>
<thead>
<tr>
<th>Component Name</th>
<th>Entity Name</th>
<th>Architecture Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xor2</td>
<td>XorGate2</td>
<td>simple</td>
</tr>
<tr>
<td>Nand2</td>
<td>NandGate2</td>
<td>simple</td>
</tr>
</tbody>
</table>

component configuration
Testing VHDL description using a test bench

Entity Name: HalfadderTop
Architecture Name: Structural
Testing VHDL description using a test bench

entity HalfadderTop is
-- no i/o ports
end HalfadderTop;

architecture structural of HalfadderTop is

component Halfadder is
(A1, B1: In BIT; Sum, Carry: Out BIT);
end component;

component Testbench is
(A, B: Out BIT);
end component;

signal DriveA1, DriveB1, MonitorSum, MonitorCarry: Bit;

continues …
Testing VHDL description using a test bench

… continues

begin

HA: Halfadder

port map (A1=>DriveA1, B1=>DriveB1, sum=>MonitorSum, Carry=>MonitorCarry);

TB: Testbench

port map (A=>DriveA1, B=>DriveB1);

end structural;
VHDL Functional Description

• Functional descriptions can be made using:
  – Concurrent VHDL statements
  – Sequential VHDL statements

• Concurrent VHDL statements: no implied sequence/timing i.e. statements are executed in parallel.

• Sequential VHDL statements: statements are executed one after the other as in a normal programming language {order is important}.

• Sequential VHDL statements are identified since they have to be in a process block.
The Process Block

process_label: **process** (sensitivity list)

**begin**
Sequential statements;
**end process** process_label;

- The sensitivity list is a list of signals which will trigger (start) the process. If no sensitivity list exists, then the process will continue to loop indefinitely: in this case the execution is typically controlled by **wait** statements forming part of the sequential statements in the **process block**. A process will therefore typically have a sensitivity list or **wait** statements, but it cannot have both. There can be more than one process running concurrently. However the statements in each process block would be executed sequentially.

- Changes in the value of any one of the signals in the sensitivity list will cause the process to be executed. The process will continue serial execution until the last statement has been executed. The process then suspends execution until another change occurs in one of the signals referenced in the sensitivity list.
Example 2: Half Adder Functional VHDL Description

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

entity Halfadder is
port (A1, B1: in std_logic; sum, carry: out std_logic);
end Halfadder;

architecture functional of Halfadder is

begin
sum <= A1 xor B1;
carry <= A1 and B1;
end functional;
```

Alternatively: `sum <= A1+B1;`

*where sum is a 2-bit vector*

`sum: out std_logic_vector(1 downto 0);`
library IEEE;
use IEEE.std_logic_1164.all;

entity Halfadder is
port (A1, B1: in std_logic; sum, carry: out std_logic);
end Halfadder;

architecture functional of Halfadder is

Adder: process (A1,B1)
begin
  sum <= A1 xor B1;
  carry <= A1 and B1;
end process Adder;

end functional;
Example 4: Latched Half Adder Functional VHDL Description

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

entity Halfadder is
port (A1, B1: in std_logic; sum, carry: out std_logic);
end Halfadder;

architecture functional of Halfadder is

Latch_Adder: process (Clk)
begin
If (Clk = '1') then
sum <= A1 xor B1;
carry <= A1 and B1;
end if;
end process Latch_Adder;

end functional;
```

Sequential VHDL Description

sequential logic
VHDL Basic Statements
Wait Statements

- Execution of a process suspends when a `wait` statement is encountered. Execution restarts when the condition of the `wait` statement is met.

- Syntax: `process` block
  : sequential statements
  : `wait` condition
  : sequential statements
`end process`
Type of Wait Statements

• (1) **wait**
  – wait forever: usually used within a test bench

• (2) **wait on** <signal list>
  – Waits for an event on any one of the signals (same purpose as sensitivity list)

• (3) **wait until** <condition>
  – Waits until condition becomes true e.g. **wait until** A = ‘1’;

• (4) **wait for** <time>
  – e.g. **wait for** 10ns; {typically this is not supported by synthesis tool}
Example for Wait Statements

Adder: process
begin
sum <= A xor B;
carry <= A and B;
wait on A,B;
end process Adder;

If in the wait statement we remove B, we would require memory to store value of B since a change in it will keep the adder idle -> it becomes a sequential logic circuit.
If statement

If condition \textbf{then}

sequential statements

\textbf{elsif} condition \textbf{then}

sequential statements \textit{optional}

\textbf{else}

sequential statements

\textbf{end if};
Case statement

- Allows selection of sequential statements to be executed, based on the value of a selection expression. Choice of a `case` statement must be unique. If not all possible values of the selector expression are listed, an `others` choice must be included.

- Example:

```vhdl
type select is array (3 downto 0) of BIT; -- defining a new type
signal operationselect: select;

case operationselect is
  when "0000" => sequential statements;
  when "0001" => sequential statements;
  when others => sequential statements;
end case;
```

The `NULL` statement signifies no operation. It is useful to explicitly show that no action is required for a particular choice in a case statement.

```
e.g. : 
  when "0101" => NULL;
```
Loop Statement

• Syntax:

    while condition loop

    sequential statements;

    end loop;

Boolean condition is checked each time before the loop is executed. If condition is true, loop is executed. If condition is false, execution continues with the next statement after the end loop.
Generic Loop

- A loop label is used in this case:

  Syntax: `loop_label: loop
           sequential statements;
           end loop;`

- `exit` and `next` statements can be used to control the flow of a generic loop.

  Syntax: `next loop_label when condition;
          exit loop_label when condition;`

  Infinite loop used in conjunction with a `wait` statement or `exit` statement

  The next statement terminates the execution of the current loop iteration and starts a subsequent iteration.

  The `exit` statement also terminates the current iteration and additionally also exists the loop.
For Loop

- Specifies a fixed number of iterations of the loop. Loop index does not have to be explicitly declared. Loop index only exists within the loop and disappears as soon as the loop is terminated.

- Example: `for I in 1 to 10 loop
  factorial = factorial*I;
  end loop;`

- Index type of enumerated type:

  Example: `type RGB is (Red,Green,Blue);
signal rgb_signal: RGB;
for RGB_Index in RGB loop
  -- sequential statements
  end loop;`
Nested Loops

outerloop:  For X in 1 to 100 loop

innerloop:  For Y in 1 to 100 loop

:: exit outerloop when (Y=1 and X=99);

::

:: end loop;

end loop;
Assert Statement

- Used to verify a condition and report if the condition is violated. The assertion statement will also specify the severity of the condition violation: note, warning, error, failure.

Syntax: `assert` condition

  `report “report expression”`

  `severity severity_level;`

Example: `assert (x < 32)`

  `report (“x out of limit”);`

  `severity error;`

This is printed if condition is not true
Conditional Assignment

with selector select

signal_name <= value when choice_1;
    <= value when choice_2;
    :   
    :   
    <= value when choice_n;
    <= value when others;

• The conditional assignment is very similar to a multiplexer implementation.
Expressions and Operators

Highest precedence of evaluation

- Miscellaneous: abs not …
- Multiplying: * / mod
- Signing: + –
- Adding: + – & {concatenation of 1D arrays}
- Relational Operators: = /= < <= > >=
- Logical Operators: and or nand nor xor

Lowest precedence of evaluation
Signals and Variables

- Signals: used for communication between concurrently executed VHDL blocks – will physically map into a wire (or bus).

- Variables: only valid with sequential code (higher level of abstraction) – are local to an individual process or function.

- **Signal Attributes:**
  
  - If s is a signal:
  
  - s’event : returns true if s changed (either ↑ or ↓)
  
  - s’last_event: returns the time elapsed since the previous transition on s.
Concurrent Statements

C <= A and B after 5 ns;
E <= C or D after 5 ns;

If delay is not specified, “delta” delay is assumed
C <= A and B;
E <= C or D;

Order of concurrent statements is not important
E <= C or D;
C <= A and B;

This statement executes repeatedly
CLK <= not CLK after 10 ns;

This statement causes a simulation error
CLK <= not CLK;
Code Examples - 2

entity FullAdder is
  port (X, Y, Cin: in bit; -- Inputs
        Cout, Sum: out bit); -- Outputs
end FullAdder;

architecture Equations of FullAdder is
begin -- Concurrent Assignments
  Sum <= X xor Y xor Cin after 10 ns;
  Cout <= (X and Y) or (X and Cin) or (Y and Cin) after 10 ns;
end Equations;
Code Examples -3

**Structural Description of 4-bit Adder**

```vhdl
entity Adder4 is
    port (A, B: in bit_vector(3 downto 0); Ci: in bit;  -- Inputs
          S: out bit_vector(3 downto 0); Co: out bit);  -- Outputs
end Adder4;
```
**Code Example - 4**

**Structural Description of 4-bit Adder**

```vhdl
architecture Structure of Adder4 is
component FullAdder
  port (X, Y, Cin: in bit; -- Inputs
       Cout, Sum: out bit); -- Outputs
end component;
signal C: bit_vector(3 downto 1);
begin
  -- instantiate four copies of the FullAdder
  FA0: FullAdder port map (A(0), B(0), Ci, C(1), S(0));
  FA1: FullAdder port map (A(1), B(1), C(1), C(2), S(1));
  FA2: FullAdder port map (A(2), B(2), C(2), C(3), S(2));
  FA3: FullAdder port map (A(3), B(3), C(3), Co, S(3));
end Structure;
```
VHDL Process - Timing

VHDL Processes

General form of Process
process(sensitivity-list)
begin
  sequential-statements
end process;

Process example
process (B, C, D)
begin
  A <= B;  -- statement 1
  B <= C;  -- statement 2
  C <= D;  -- statement 3
end process;

Simulation results

<table>
<thead>
<tr>
<th>time</th>
<th>delta</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>+0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>+0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>+1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>+2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>+3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

statements 1,2,3 execute; then update A,B,C
statements 1,2,3 execute; then update A,B,C
statements 1,2,3 execute; then update A,B,C
(no further execution occurs)

Concurrent Statements

A <= B;  -- statement 1
time  Δ  A  B  C  D
0    +0  1  2  3  0
10   +0  1  2  3  4  (statement 3 executes first)
10   +1  1  2  4  4  (then statement 2 executes)
10   +2  1  4  4  4  (then statement 1 executes)
10   +3  4  4  4  4  (no further execution occurs)

Edward Gatt

VHDL
**Code Example - 5**

```vhdl
entity DFF is
  port (D, CLK: in bit;
        Q: out bit; QN: out bit := '1');
-- initialize QN to '1' since bit signals are initialized to '0' by default
end DFF;

architecture SIMPLE of DFF is
begin
  process (CLK) -- process is executed when CLK changes
  begin
    if CLK = '1' then -- rising edge of clock
      Q <= D after 10 ns;
      QN <= not D after 10 ns;
    end if;
  end process;
end SIMPLE;
```
Code Example - 6

J-K Flip-flop Model

entity JKFF is
  port (SN, RN, J, K, CLK: in bit; Q: inout bit; QN: out bit := '1'); -- inputs
end JKFF;

architecture JKFF1 of JKFF is
begin
  process (SN, RN, CLK) -- see Note 2
  begin
    if RN = '0' then Q<= '0' after 10 ns; -- RN=0 will clear the FF
    elsif SN = '0' then Q<= '1' after 10 ns; -- SN=0 will set the FF
    elsif CLK = '0' and CLK'event then
      Q <= (J and not Q) or (not K and Q) after 10 ns; -- see Note 3
    end if;
  end process;
  QN <= not Q; -- see Note 5
end JKFF1;

Note 1: Q is declared as inout (rather than out) because it appears on both the left and right sides of an assignment within the architecture.
Note 2: The flip-flop can change state in response to changes in SN, RN, and CLK, so these 3 signals are in the sensitivity list.
Note 3: The condition (CLK = '0' and CLK'event) is TRUE only if CLK has just changed from '1' to '0'.
Note 4: Characteristic equation which describes behavior of J-K flip-flop.
Note 5: Every time Q changes, QN will be updated. If this statement were placed within the process, the old value of Q would be used instead of the new value.
Code Example - 7

MUX model using a *conditional signal assignment statement*:

\[
F <= I0 \text{ when } Sel = 0 \\
\text{else I1 when } Sel = 1 \\
\text{else I2 when } Sel = 2 \\
\text{else I3;}
\]

In the above concurrent statement, Sel represents the integer equivalent of a 2-bit binary number with bits A and B.

General form of conditional signal assignment statement:

\[
\text{signal}_\text{name} <= \text{expression1 when condition1} \\
\text{else expression2 when condition2} \\
\text{else expressionN;}
\]
If a MUX model is used inside a process, a concurrent statement cannot be used. As an alternative, the MUX can be modeled using a case statement:

```vhdl
  case Sel is
  when 0 => F <= I0;
  when 1 => F <= I1;
  when 2 => F <= I2;
  when 3 => F <= I3;
  end case;
```

The case statement has the general form:

```vhdl
  case expression is
  when choice1 => sequential statements1
  when choice2 => sequential statements2
  . . .
  [when others => sequential statements]
  end case;
```
Test Bench Code

VHDL Code for Simulation Example

```vhdl
entity simulation_example is
end simulation_example;

architecture test1 of simulation_example is
  signal A,B: bit;
begnin
  P1: process(B)
  begin
    A <= '1';
    A <= transport '0' after 5 ns;
  end process P1;

  P2: process(A)
  begin
    if A = '1' then B <= not B after 10 ns; end if;
  end process P2;
end test1;
```

- transport delay – used to model delays introduced by wiring – delays input by a specified time
Timings for Testbench

Signal Drivers for Simulation Example

After elaboration:
- time = 0
- Queued values: '0' @ 0, '1' @ ∆
- Current value: '0'

After initialization:
- time = 0
- Queued values: '0' @ 5, '1' @ ∆
- Current value: '0'

Simulation step:
- time = ∆
- Queued values: '0' @ 5, '1' @ 10
- Current value: '1'

At time = 5:
- '0' @ 10
- '0'
- A → B

At time = 10:
- '0' @ 15
- '1' @ 10 + ∆
- '0'
- A → B

At time = 10 + ∆:
- '0' @ 15
- '0' @ 20
- '1'
- A → B

At time = 15:
- '0' @ 20
- '1'
- A → B

Edward Gatt

VHDL
entity SM1_2 is
  port(X, CLK: in bit;  Z: out bit);
end SM1_2;

architecture Table of SM1_2 is
  signal State, Nextstate: integer := 0;
begin
  process(State,X)  --Combinational Network
  begin
    case State is
    when 0 =>
      if X='0' then Z<='1'; Nextstate<=1; end if;
      if X='1' then Z<='0'; Nextstate<=2; end if;
    when 1 =>
      if X='0' then Z<='1'; Nextstate<=3; end if;
      if X='1' then Z<='0'; Nextstate<=4; end if;
    when 2 =>
      if X='0' then Z<='0'; Nextstate<=4; end if;
      if X='1' then Z<='1'; Nextstate<=4; end if;
    when 3 =>
      if X='0' then Z<='0'; Nextstate<=5; end if;
      if X='1' then Z<='1'; Nextstate<=5; end if;
    when 4 =>
      if X='0' then Z<='1'; Nextstate<=5; end if;
      if X='1' then Z<='0'; Nextstate<=6; end if;
    end case;
  end process;

  table
    | PS | X = 0 | X = 1 |
    |----------|-------|-------|
    | S0 | S1   | S2   | 1   | 0   |
    | S1 | S3   | S4   | 1   | 0   |
    | S2 | S4   | S4   | 0   | 1   |
    | S3 | S5   | S5   | 0   | 1   |
    | S4 | S5   | S6   | 1   | 0   |
    | S5 | S0   | S0   | 0   | 1   |
    | S6 | S0   | -    | 1   | -   |
end architecture;
Code Example – 9 (cont/d)

```vhdl
when 5 =>
    if X='0' then Z<= '0'; Nextstate<=0; end if;
    if X='1' then Z<= '1'; Nextstate<=0; end if;
when 6 =>
    if X='0' then Z<= '1'; Nextstate<=0; end if;
    when others => null; -- should not occur
end case;
end process;

process(CLK)
begin
    if CLK='1' then
        State <= Nextstate;
    end if;
end process;
end Table;
```

---

Edward Gatt

VHDL

64
Process Using Variables

entity dummy is
end dummy;

architecture var of dummy is
signal trigger, sum: integer:=0;
begin
process
variable var1: integer:=1;
variable var2: integer:=2;
variable var3: integer:=3;
begin
    wait on trigger;
    var1 := var2 + var3; \( var1 = 2 + 3 = 5 \)
    var2 := var1; \( var2 = 5 \)
    var3 := var2; \( var3 = 5 \)
    sum <= var1 + var2 + var3; \( sum = 5 + 5 + 5 = 15 \) (after Δ)
end process;
end var;
Process Using Signals

entity dummy is
end dummy;

architecture sig of dummy is
  signal trigger, sum: integer:=0;
  signal sig1: integer:=1;
  signal sig2: integer:=2;
  signal sig3: integer:=3;
begin
  process
  begin
    wait on trigger;
    sig1 <= sig2 + sig3;  \[ sig1 = 2 + 3 = 5 \]  \textit{(after }\Delta\text{)}
    sig2 <= sig1;  \[ sig2 = 1 \]  \textit{(after }\Delta\text{)}
    sig3 <= sig2;  \[ sig3 = 2 \]  \textit{(after }\Delta\text{)}
    sum <= sig1 + sig2 + sig3;  \[ sum = 1 + 2 + 3 = 6 \]  \textit{(after }\Delta\text{)}
  end process;
end sig;
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_misc.all;
use IEEE.std_logic_unsigned.all;

entity ALU is
  port ( Accumulator_in: in STD_LOGIC_VECTOR (7 downto 0);
         Data_in : in STD_LOGIC_VECTOR (7 downto 0);
         Opcode_in : in STD_LOGIC_VECTOR (3 downto 0);
         Result_out : out STD_LOGIC_VECTOR (7 downto 0)
         );
end ALU;

architecture ALU_arch of ALU is
begin
  Main: process(Accumulator_in,Opcode_in, Data_in)
  begin
    case Opcode_in is
    when "0000" => Result_out <= Data_in;
    -- result = Data_in
    when "0001"=> Result_out <= Accumulator_in;
    -- result = accumulator_in
    when "0010"=> Result_out <= "00000000";
    -- result = accumulator_in + Data_in
    when "0011"=> Result_out <= "00000000";
    -- result = accumulator_in - Data_in
    when "0100"=> Result_out <= Accumulator_in and Data_in;
    -- result = accumulator_in and Data_in
• when "0101"=> Result_out <= Accumulator_in or Data_in;
• -- result = accumulator_in or Data_in
• when "0110"=> Result_out <= Accumulator_in xor Data_in;
• -- result = accumulator_in xor Data_in
• when "0111"=> Result_out <= not(accumulator_in);
• -- result = not(accumulator_in)
• when "1000"=> Result_out <= not(accumulator_in);
• -- result = not(Data_in);
• when "1001"=> Result_out <= "00000000"
• -- result = 0
• when "1010"=> Result_out <= "00000000"
• -- result = 8 LSBs of (accumulator_in * Data_in)
• when "1011"=> Result_out <= "00000000"
• -- result = 8 MSBs of (accumulator_in * Data_in)
• when "1100"=> Result_out <= accumulator_in nand Data_in
• -- result = accumulator_in nand Data_in
• when "1101"=> Result_out <= accumulator_in nor Data_in
• -- result = accumulator_in nor Data_in
• when "1110"=> Result_out <= accumulator_in xnor Data_in
• -- result = accumulator_in xnor Data_in
• when "1111"=> Result_out <= "00000000"
• -- result = Accumulator_in+1
• when others => Result_out <= "XXXXXXXX";
• end case;
• end process Main;
• end ALU_arc
Test Bench for ALU

- library IEEE;
- use IEEE.std_logic_1164.all;
- use IEEE.std_logic_arith.all;
- use IEEE.std_logic_misc.all;
- use IEEE.std_logic_unsigned.all;
- entity alu_tb is
  - port (  
      Accumulator_out: out std_logic_vector (8 downto 1);  
      Data_out : out std_logic_vector (8 downto 1);  
      Opcode_out : out std_logic_vector (4 downto 1)  
  );
- end alu_tb;
- architecture alu_tb_arch of alu_tb is
  - begin
  - operation_1: PROCESS
    - begin
      - wait for 0ns; Opcode_out <= "0000";
      - wait for 10 ns; Opcode_out <= "0001";
      - wait for 10 ns; Opcode_out <= "0010";
      - wait for 10 ns; Opcode_out <= "0011";
      - wait for 10 ns; Opcode_out <= "0100";
    - end operation_1;
Test Bench for ALU (cont/d)

- wait for 10 ns; Opcode_out <= "0101";
- wait for 10 ns; Opcode_out <= "0110";
- wait for 10 ns; Opcode_out <= "0111";
- wait for 10 ns; Opcode_out <= "1000";
- wait for 10 ns; Opcode_out <= "1001";
- wait for 10 ns; Opcode_out <= "1010";
- wait for 10 ns; Opcode_out <= "1011";
- wait for 10 ns; Opcode_out <= "1100";
- wait for 10 ns; Opcode_out <= "1101";
- wait for 10 ns; Opcode_out <= "1110";
- wait for 10 ns; Opcode_out <= "1111";
- wait;
- end PROCESS operation_1;
- operation_2: PROCESS
  - begin
  - wait for 0ns; Data_out <= "00000001";
  - wait;
- end PROCESS operation_2;
- operation_3: PROCESS
  - begin
  - wait for 0ns; Accumulator_out <= "00001010";
- end PROCESS operation_3;
- end alu_tb_arch;
Top Level for ALU

- library IEEE;
- use IEEE.std_logic_1164.all;
- use IEEE.std_logic_arith.all;
- use IEEE.std_logic_misc.all;
- use IEEE.std_logic_unsigned.all;
- entity alu_tl is
  port ( Result : out std_logic_vector (8 downto 1)
  );
- end alu_tl;
- architecture alu_tl_arch of alu_tl is
- component alu
  port ( Accumulator_in: in std_logic_vector (8 downto 1);
  Data_in: in std_logic_vector (8 downto 1);
  Opcode_in: in std_logic_vector (4 downto 1);
  Result_out: out std_logic_vector (8 downto 1)
  );
- end component;
- component alu_tb is
  port ( Accumulator_out: out std_logic_vector (8 downto 1);
  Data_out: out std_logic_vector (8 downto 1);
  Opcode_out: out std_logic_vector (4 downto 1)
  );
- end component;
signal Accumulator : std_logic_vector(8 downto 1);
signal Data : std_logic_vector(8 downto 1);
signal Opcode : std_logic_vector(4 downto 1);
begin
  alu_1 : alu
  port map (  
    Accumulator_in => Accumulator,
    Data_in => Data,
    Opcode_in => Opcode,
    Result_out => Result
  );
  tb_1 : alu_tb
  port map (  
    Accumulator_out => Accumulator,
    Data_out => Data,
    Opcode_out => Opcode
  );
end alu_tl_arch;
Field Programmable Gate Arrays

- FPGAs are ICs that contain an array of identical blocks with programmable interconnections
- User can program the functions realised by each logic block and the connections between the blocks
- FPGAs offer easier design iterations
- Easier to correct mistakes that creep into the design
- Prototyping cost is reduced
FPGAs

• Disadvantages:
  FPGAs are less dense than traditional gate arrays
  A lot of resources is spent to achieve programmability
  Programmable points have resistances and capacitances
  FPGAs are slower than traditional gate arrays
Organisation of FPGAs

• The interior of FPGAs typically contain three elements that are programmable:
  – Programmable logic blocks
  – Programmable input/output blocks
  – Programmable routing resources
Organisation of FPGAs
Organisation of FPGAs

- Arrays of programmable logic blocks are distributed within the FPGA
- Logic blocks are surrounded by (I/O) interface blocks
- I/O blocks are on the periphery on the chip
- They connect the logic signals to the FPGA pins
- Space between logic blocks are used to route connections between the logic blocks
Programmable Logic Blocks

• Programmable Logic Blocks are created by using multiplexers, look-up tables, AND-OR or NAND-NAND arrays.
• Programming means changing the input or control signals to the multiplexers, changing look-up table contents or selecting/not selecting particular gates in AND-OR gate blocks.
• For a programmable interconnect, programming means making or breaking specific connections.
• This is required to interconnect various blocks in the chip and to connect specific I/O pins to specific logic blocks.
Programmable I/O blocks

- Programmable I/O blocks denote blocks which can be programmed to be input, output or bi-directional lines.
- They can also be programmed to adjust the properties of their buffers such as inverting/non-inverting, tristate, passive pull-ups or even adjust the slew rate.
FPGA Programming Technologies

- Several techniques have been used to achieve the programmable interconnections between FPGAs
  - Static RAM programming technology
  - EPROM/EEPROM/flash programming technology
  - Antifuse programming technology
SDRAM Programming Technology

- The SDRAM programming technology involves creating reconfigurability by bits stored in SRAM cells
- The logic blocks, I/O blocks and interconnect can be programmed by using configuration bits stored in SRAM
- Reconfigurable logic blocks can be easily implemented as look-up tables
- Problem is that system is volatile
SRAM Programming Technology

- 16 SRAM cells can implement any 4 variable function
- Programmable interconnect can also be achieved by SRAM
- The key idea is to use pass transistors to create switches and then control them using SRAM content
- SDRAM requires 6 transistors – therefore number of transistors to provide a connection is high

Edward Gatt

VHDL
EPROM/EEPROM Programming Technology

- EPROM cells used to control programmable connections
- A transistor with 2 gates is used – floating gate and control gate
- Transistor turned off by injecting charge on the floating gate using a high voltage between the control gate and drain of the transistor
- Charge increases the threshold voltage of the transistor and it switches off
- Charge can be removed by exposing the floating gate to ultraviolet light
EPROM

- EPROM technology slower than SRAM
- EPROM switches have high ON resistance and high static power consumption
- EEPROM technology is similar to EPROM but removal of gate charge can be done electrically
Antifuse Programming Technology

- Antifuse programming element changes from open to closed when a high voltage is applied.
- They are often built using dielectric layers between N+ diffusion and polysilicon layers or by amorphous silicon between metal layers.
- Antifuses are normally OFF and put ON when programmed.
- Process is irreversible and not reprogrammable but consumes little area.
Programmable Logic Block Arrays

- LUT-based FPGAs use four-variable look-up tables plus a flip-flop as the basic element and then connect several of them in various topologies.
Programmable Logic Block Arrays

• Some FPGAs use multiplexers as the basic block.
• Any combinational logic can be implemented using multiplexers alone.
• E.g., a 4-to-1 multiplexer can generate any two-input function.
• If inverted inputs can be provided, the same multiplexer can generate any three-input function.
Programmable Interconnects

Interconnects in Symmetric Array FPGAs

- **General Purpose Interconnect**: Switch Matrices provide interconnections between routing wires connected to the switch matrix.
Programmable Interconnects

Interconnects in Symmetric Array FPGAs

- **Direct Interconnects**: Many FPGAs provide special connections between adjacent logic blocks. These interconnects are fast as they do not go through the routing matrix.
Programmable Interconnects

Interconnects in Symmetric Array FPGAs

- **Global Lines:** For purposes like high fan-out and low-skew clock distribution, most FPGAs provide routing lines that span the entire width of the device/height of the device.

- **Interconnects in Row-Based FPGAs:** In devices that are row based, there are rows of logic blocks and there are channels of switches to enable connections between logic blocks.
Programmable I/O Blocks

• The I/O pads are connected to programmable input/output blocks, which facilitate connecting the signals from FPGA logic blocks to the external world in desired forms and formats.

• I/O blocks on modern FPGAs allow use of the pin as input and/or output, in direct (combinational) or latched forms, in tristate true or inverted forms and with a variety of I/O standards.