Introduction to VHDL

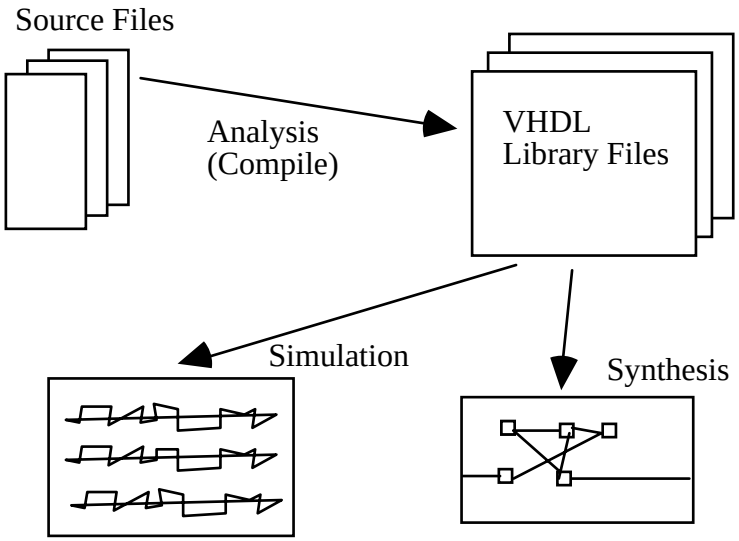
HDL - Hardware Description Language

A language that allows the description of hardware for documentation, simulation, synthesis, ...

To use HDLs you need a CAD system
Major CAD systems support VHDL, Verilog, SystemC

Cadence - Leapfrog (VHDL, Verilog)
Mentor Graphics – ModelTech is subsidiary
Model Technology
– ModelSim – VHDL, Verilog, System C
Altera (VHDL compilation - FPGA native simulation)
Xilinx (VHDL compilation - FPGA native simulation)

ALL SYSTEMS

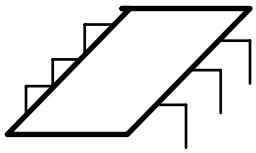


A First Example

Desire to do a VHDL Description of a full adder.

All devices consist of

an interface



and an operational part.

Interface - **THE INPUTS AND OUTPUTS**
Operational Part - THE **FUNCTIONAL BEHAVIOR**

VHDL Entity Design Unit:

```
ENTITY   unit_name   IS
      [port_clause]
END unit_name;
```

For a Full Adder would have:

```
ENTITY  full_adder  IS
      PORT(  a,b,cin  :  IN  BIT;
                 sum :  OUT BIT;
                 cout :  OUT BIT);
END full_adder;
```

THE PORT portion is named a PORT CLAUSE

What is specified in the clause are signals that have scope over all architectures of this entity.

Signals/Port Modes/Types

PORT(a,b,cin: IN BIT; sum: OUT BIT; cout: OUT BIT);

Signals: The names referenced in the PORT CLAUSE are signals.

a, b, cin, sum, cout these represent the wires of the physical unit.

SIGNALS are objects that have both a value and a time component.

Port Modes:

In this example you just have inputs and outputs. The Port Mode gives the direction of the signal and also specifies the direction of signal transfer.

Mode:

IN - signal can only be used (i.e., can only be read or can only appear on the right-hand-side of an equation).

OUT - signal value can only be written. Cannot be seen or used in the design as it is an output and therefore external.

INOUT - signal can be both written to (assigned to) and read (used). However, most signals of this type are

connected to busses and therefore this signal mode requires a resolved signal.

BUFFER - signal value can be written to and used internally in the design.

BASIC TYPES (built in – part of the standard)

TYPE BIT - your typical binary type with values of ‘0’ and ‘1’.

Declaration for the type is

```
TYPE BIT is (‘0’,’1’)
```

USE of SIGNALS

```
a <= ‘0’;
b <= x AND y  OR  z;
```

Note that the value is either ‘0’ ir ‘1’

Architectural Design Unit

Specifies operational part

ARCHITECTURE ***identifier*** **OF** ***entity_id*** **IS**

[declarations]

BEGIN

[architecture_statement_part]

END [*identifier*];

[architecture_statement_part] -

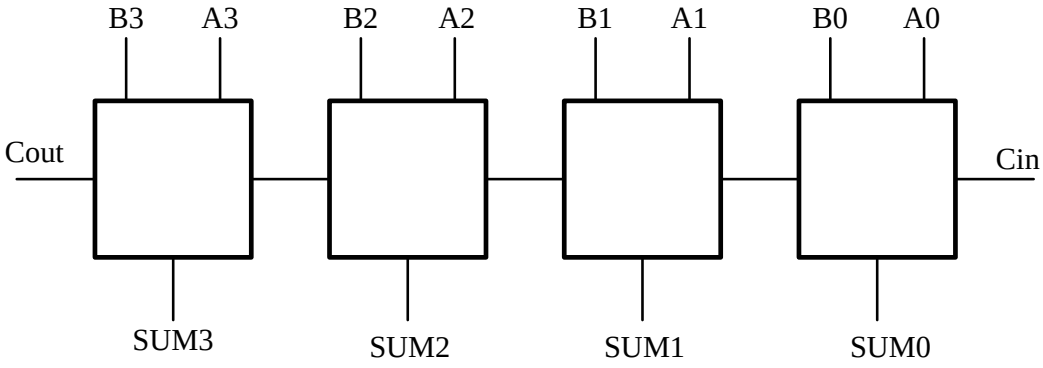
Any concurrent statement of the language

Example dataflow architecture of full adder:

```
ARCHITECTURE one OF fulladder  IS
BEGIN
      sum <= a XOR b XOR cin;
      cout <= (a AND b) OR (a AND cin) OR (b AND cin);
END;
```

Statements between BEGIN and END are concurrent signal assignment statements.

Consider a 4 bit Adder



This is the hardware to be modeled in VHDL

Multibit Adder Example

Can model multibit adder using a dataflow style
1) Bit vectors for ports and individual signals internally
2) Bit vectors for ports and bit vectors internally.

The Entity Design Unit:

```
ENTITY mb_adder IS
    PORT (a,b : IN BIT_VECTOR(3 downto 0);
          cin : IN BIT;  cout : OUT BIT;
          sum: OUT BIT_VECTOR(3 downto 0));
END mb_adder;
```

The First Dataflow Architecture : (bit operation)

```
ARCHITECTURE one OF mb_adder IS
  SIGNAL c : BIT_VECTOR (4 downto 0);
BEGIN
  c(0) <= cin;
  sum(0) <= a(0) XOR b(0) XOR c(0);
  sum(1) <= a(1) XOR b(1) XOR c(1);
  sum(2) <= a(2) XOR b(2) XOR c(2);
  sum(3) <= a(3) XOR b(3) XOR c(3);
  c(1) <= (a(0) AND b(0)) OR (a(0) AND c(0)) OR
          (b(0) AND c(0));
  c(2) <= (a(1) AND b(1)) OR (a(1) AND c(1)) OR
          (b(1) AND c(1));
  c(3) <= (a(2) AND b(2)) OR (a(2) AND c(2)) OR
          (b(2) AND c(2));
  c(4) <= (a(3) AND b(3)) OR (a(3) AND c(3)) OR
          (b(3) AND c(3));
    Cout <= c(4);
END one;
```

A Second Dataflow Architecture: (vector operations)

```
ARCHITECTURE two OF mb_adder IS
  SIGNAL c : BIT_VECTOR (4 downto 0);
BEGIN
  c(0) <= cin;
  sum <= a XOR b XOR c(3 downto 0);
  c(4 downto 1) <= (a(3 downto 0) AND b(3 downto 0))
      OR (a(3 downto 0) AND c(3 downto 0)) OR
                  (b(3 downto 0) AND c(3 downto 0));
  Cout <= c(4);
END two;
```

Architecture “two” use vector operation on the signals.

Note the power of this design method.
Carry ripple through repeated evaluation of the equation as whenever a signal on the right-hand-side changes, the equation is re-evaluated and a new value scheduled for assignment to the signal driven.

Operations on Type BIT

Consider the following declaration

SIGNAL *x, y* : BIT;

Logical Operations:

x AND y
x OR y
x NAND y
x NOR y
x XOR y
x XNOR y
NOT x

Aslo have shift operations
arithmetic shifts ASR
logical shifts LSL

Note: For AND OR NAND NOR the right operator is evaluated only if the value of the left operator is not sufficient to determine the result.

Assignment Operators:
For signals <=
For variables :=

Relational Operators:
(x = y) (x /= y) (x <= y) (x >= y)

(x = ‘1’) AND (y = ‘0’)

Structural Example

Again consider the full adder.

Before doing a structural description must have the components that are going to be wired together. These must be written and compiled into the library. Although not shown here, each would have an architecture describing its behavior.

Having the following entity declarations for the components:

```
ENTITY  and2  IS
  PORT (A,B : IN BIT;  Z : OUT BIT);
END and2;

ENTITY  xor2  IS
  PORT (A,B : IN BIT;  Z : OUT BIT);
END xor;

ENTITY  or3  IS
  PORT (A,B,C : IN BIT;  Z : OUT BIT);
END or3;
```

A Structural Description

```
ARCHITECTURE  structural  OF  full_adder  IS
--  Must declare the components that are to be used
COMPONENT  and2
  PORT (A,B : IN BIT;  Z : OUT BIT);
END COMPONENT  ;
COMPONENT  xor2
  PORT (A,B : IN BIT;  Z : OUT BIT);
END COMPONENT  ;
COMPONENT  or3
  PORT (A,B,C : IN BIT;  Z : OUT BIT);
END COMPONENT  ;
--  State which library to find them in and which architecture to use.
FOR ALL :  and2  USE ENTITY WORK.and2(behavioral);
FOR ALL :  xor2    USE ENTITY WORK.xor2(behavioral);
FOR ALL :  or3    USE ENTITY WORK.or3(behavioral);
--  Declare local signals required.
SIGNAL   addt. ct1, ct2, ct3  : BIT;
BEGIN
  G1:       xor2  PORT MAP(a,b,addt);
  G2:       xor2  PORT MAP(addt, cin, sum);

  G3:       and2  PORT MAP(a,b,ct1);
  G4:       and2  PORT MAP(a,cin,ct2);
  G5:       and2  PORT MAP(b,cin,ct3);

  G6:       or3 PORT MAP(ct1,ct2,ct3,cout);



END Structural;
```

Can use the Full Adder to Structurally Make a Multibit Adder

The Entity Design Unit:

```
ENTITY mb_adder IS
    PORT (a,b : IN BIT_VECTOR(3 downto 0);
          cin : IN BIT;  cout : OUT BIT;
          sum: OUT BIT_VECTOR(3 downto 0));
END mb_adder;
```

The Architecture

```
ARCHITECTURE  structural  OF  mb_adder  IS
--  Must declare the components that are to be used
    COMPONENT  full_adder
        PORT(  a,b,cin  :  IN  BIT;
               sum :  OUT BIT;
               cout :  OUT BIT);
    END COMPONENT;
    FOR ALL full_adder USE ENTITY
                              work.full_adder(structural);
    SIGNAL ic1,ic2,ic3  BIT;
BEGIN
    U0:  full_adder(a(0),b(0),cin,ic1,sum(0)):
    U1:  full_adder(a(1),b(1),ic1,ic2,sum(1)):
    U2:  full_adder(a(2),b(2),ic2,ic3,sum(2)):
    U3:  full_adder(a(3),b(3),ic3,cpit,sum(3)):
END structural;
```