Essential VHDL for ASICs

A brief introduction to design with VHDL for ASIC design.

Roger Traylor

9/7/01

Version 0.2

Revision Record

rev 0.1 : Initial rough entry of material. 9/7/01 RLT
rev 0.2 : Started entering JMMs revisions 4/15/02

HDL Design

Traditionally, digital design was done with schematic entry.

In today's very competitive business environment, building cost-effective products in an quick fashion is best done with a top down methodology utilizing hardware description languages and synthesis.

```
shift_register:
 PROCESS (clk_50, reset_n, data_ena, serial_data, parallel_data)
   BEGIN
     IF (reset_n = '0') THEN
       parallel_data <= "00000000";
     ELSIF (clk_50'EVENT AND clk_50 = '1') THEN
       IF (data_ena = '1') THEN
         parallel_data(7) <= serial_data;          --input gets input data
         FOR i IN 0 TO 6 LOOP
          parallel_data(i) <= parallel_data(i+1); --all other bits shift down
         END LOOP;
       ELSE
          parallel_data <= parallel_data;
       END IF;
     END IF;
   END PROCESS shift_register;
```

synthesis
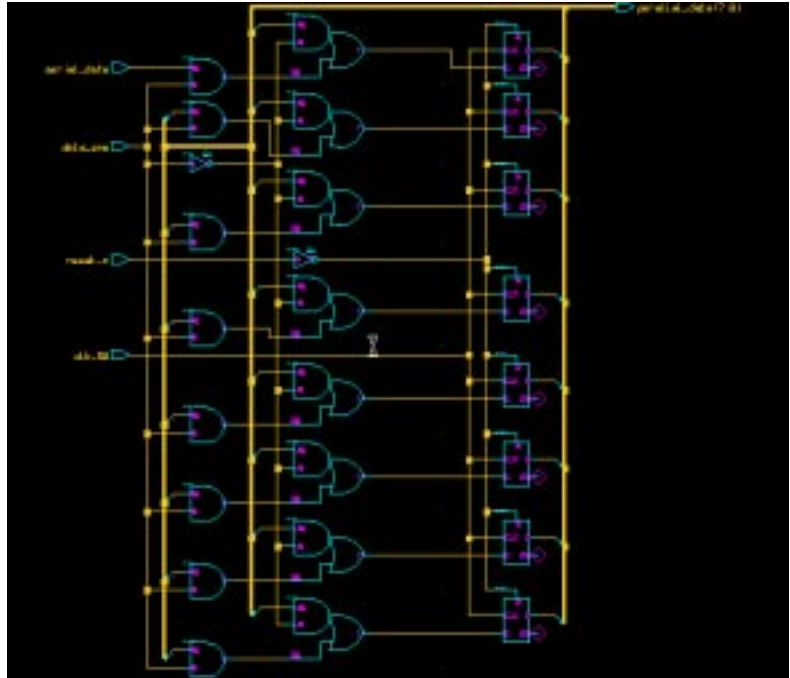
HDLs - Motivation

Increased productivity
> shorter development cycles, more features, but........
> still shorter time-to-market, 10-20K gates/day/engineer

Flexible modeling capabilities.
> can represent designs of gates or systems
> description can be very abstract or very structural
> top-down, bottom-up, complexity hiding (abstraction)

Design reuse is enabled.
> packages, libraries, support reusable, portable code

Design changes are fast and easily done
> convert a 8-bit register to 64-bits........
> four key strokes, and its done!
> exploration of alternative architectures can be done quickly

Use of various design methodologies.
> top-down, bottom-up, complexity hiding (abstraction)

Technology and vendor independence.
> same code can be targeted to CMOS, ECL, GaAs
> same code for: TI, NEC, LSI, TMSC....no changes!

Enables use of logic synthesis which allows a investigation of the area and timing space.
> ripple adder or CLA?, How many stages of look ahead?

HDLs can leverage software design environment tools.
> source code control, make files

Using a standard language promotes clear communication of ideas and designs.
> **schematic standards?... what's that... a tower of Babel.**

HDLs - What are they? How do we use them?

A Hardware Description Language (HDL) is a programming language used to model the intended operation of a piece of hardware.

An HDL can facilitate:
abstract behavioral modeling
-no structural or design aspect involved
hardware structure modeling
-a hardware structure is explicitly implied

In this class we will use an HDL to describe the structure of a hardware design.

When we use an HDL, we will do so at what is called the *Register Transfer Language level (RTL).* At this level we are implying certain hardware structures when we understand apriori.

When programming at the RTL level, we are not describing an algorithm which some hardware will execute, we are describing a hardware structure.

Without knowing beforehand what the structure is we want to build, use of an HDL will probably produce a steaming pile (think manure) of gates which may or may not function as desired.

You must know what you want to build <u>before</u> you describe it in an HDL.

Knowing an HDL does not relieve you of thoroughly understanding digital design.

HDL's- VHDL or Verilog

We will use VHDL as our HDL.

VHDL
>   more capable in modeling abstract behavior
>   more difficult to learn
>   strongly typed
>   85% of FPGA designs done in VHDL

Verilog
>   easier and simpler to learn
>   weakly typed
>   85% of ASIC designs done with Verilog (1993)

The choice of which to use is not based solely on technical capability, but on:
>   personal preferences
>   EDA tool availability
>   commercial business and marketing issues

We use VHDL because
>   strong typing keeps students from getting into trouble
>   if you know VHDL, Verilog can be picked up in few weeks
>   If you know Verilog, learning VHDL can take several months

The Bottom line...Either language is viable.

VHDL - Origins

Roots of VHDL are in the Very High Speed Integrated Circuit (VHSIC) Program launched in 1980 by the US Department of Defense (DOD).

      The VHSIC program was an initiative by the DOD to extend integration levels and performance capabilities for military integrated circuits to meet or exceed those available in commercial ICs.

      The project was successful in that very large, high-speed circuits were able to be fabricated successfully. However, it became clear that there was a need for a standard programming language to describe and document the function and structure of these very complex digital circuits.

      Therefore, under the VHSIC program, the DOD launched another program to create a standard hardware description language. The result was the VHSIC hardware description language or VHDL.

The rest is history...

In 1983, IBM, TI and Intermetrics were awarded the contract to develop VHDL.

In 1985, VHDL V7.2 released to government.

In 1987, VHDL became IEEE Standard 1076-1987.

In 1993, VHDL restandardized to clarify and enhance the language resulting in VHDL Standard 1076-1993.

In 1993, development began on the analog extension to VHDL, (VHDL-AMS).

      Extends VHDL to non-digital devices and micro electromechanical components.   This includes synthesis of analog circuits.

Some Facts of Life (For ASIC designers)

The majority of costs are determined by decisions made early in the design process.
> "Hurry up and make all the mistakes. Get them out of the way!"

"Typical" ASIC project: concept to first silicon about 9 months.

95% of designs work as the specification states.

60% of designs fail when integrated into the system.
> The design was not the right one, but it "works".

Technology is changing so fast, the only competitive advantage is to learn faster than your competitors.

To design more "stuff" faster, your level of abstraction in design must increase.

Using HDLs will help to make digital designers successful. (and employed!)

VHDL Modeling

A VHDL models consist of an *Entity Declaration* and a *Architecture Body.*

**The entity defines the <u>interface</u>, the architecture defines the <u>function</u>.**

The entity declaration names the entity and defines the interface to its environment.

Entity Declaration Format:

```
ENTITY entity_name IS
    [GENERIC (generic_list);]
    [PORT (port_list);]
END ENTITY [entity_name];
```

There is a direct correspondence between a ENTITY and a block diagram symbol. For example:

```
ENTITY nand_gate IS
PORT(
        a : in   std_logic;
        b : in   std_logic;
        c : in   std_logic;
        z : out  std_logic);
END ENTITY nand_gate;
```


nand_gate
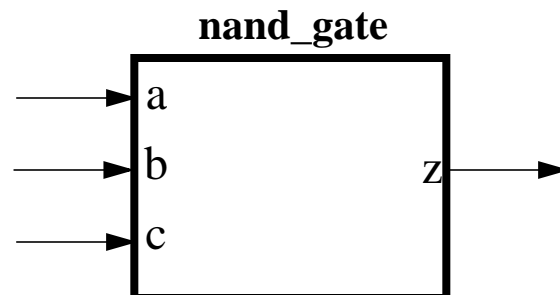
Port Statement

The entities *port* statement identifies the ports used by the entity to communicate with its environment

Port Statement Format:

```
PORT(
    name_list   : mode type;
    name_list   : mode type;
    name_list   : mode type;
    name_list   : mode type);
```

This is legal but poor form:

```
ENTITY nand_gate IS
  PORT(a,d,e,f : in  std_logic;
    b,j,q,l,y,v : in  std_logic;
    w,k  : in  std_logic;
    z : out: std_logic);
END nand_gate;
```

This is much less error prone:
      Use one line per signal. This allows adequate comments.
      Capitalize reserved names.

```
ENTITY nand_gate IS
  PORT(
    a : IN  STD_LOGIC;   --a input
    b : IN  STD_LOGIC;   --b input
    c : IN  STD_LOGIC;   --c input
    z : OUT STD_LOGIC);  --nand output
END ENTITY nand_gate;
```
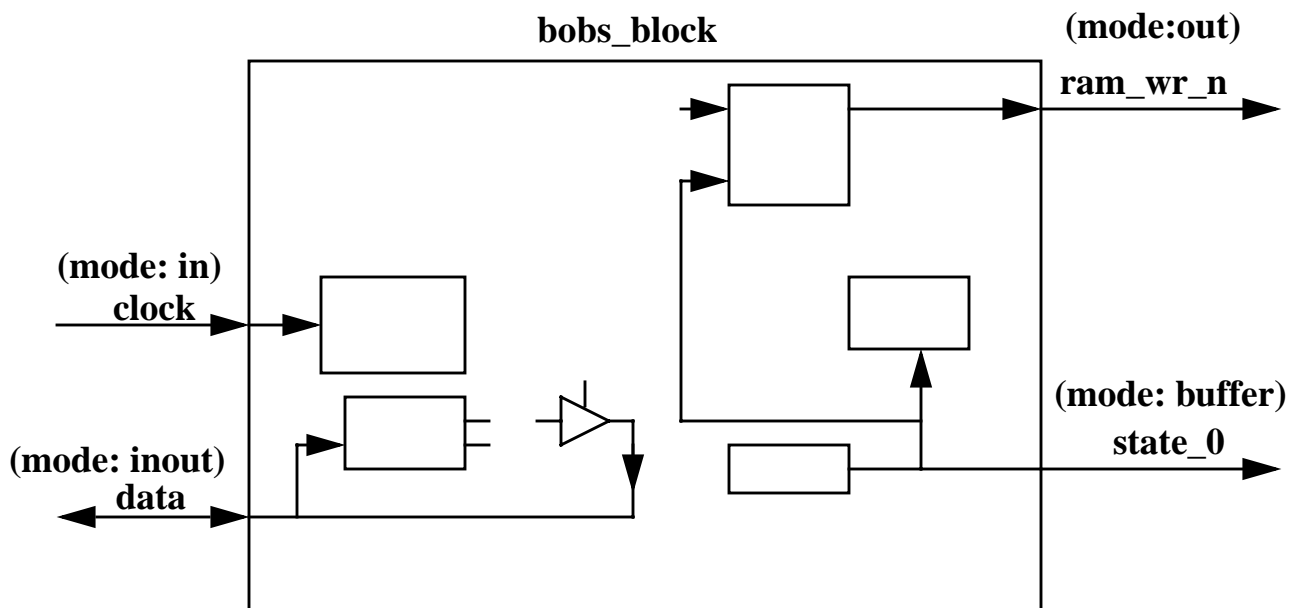
Port Mode:

Identifies the direction of data flow through the port.

The PORT statement is optional. At the top level, none is needed.

All ports must have an identified mode.

Allowable Modes:

- **IN**   **Flow is into the entity**
- **OUT**   **Flow is out of the entity**
- **INOUT**   **Flow may be either in or out**
- **BUFFER**   **An OUTPUT that can be read from**

Architecture Body

The architecture body describes the operation of the component.

Format:

```
ARCHITECTURE body_name OF entity_name IS
    --this is the ->declarative area<-
    --declare signals, variables, components,
    --subprograms
BEGIN
    --this is the ->statement area<-
    --in here go statements that describe
    --organization or functional operation of
    --the component
    --this is the "execution part" of the model
END [body_name]
```

The entity_name in the architecture statement must be the same as the entity declaration that describes the interface to the outside world.
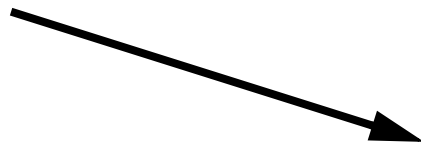
```
ENTITY entity_name IS



ARCHITECTURE body_name OF entity_name IS
```

The "body_name" is a user-defined name that should uniquely describe the particular architecture model.

```
ARCHITECTURE beh OF nand_gate IS

ARCHITECTURE struct OF nand_gate IS
```

Note: multiple architectures are allowed.

Commenting Code

A double hyphen (--) indicates everything from that point on in that line is to be treated as a comment.

```
ARCHITECTURE example OF xor_gate IS
    --The following is a silly example of how
    --to write comments in VHDL.
BEGIN
    --comment from the beginning of a line
    a <= b XOR c; --or...comment from here on
    --
    --each line must have its own
    --comment marker unlike "C"
    --
END [body_name]
    --
    --
    --this is the end and there ain't no more!
```

Comments can be put anywhere except in the middle of a line of code.

Entity and Architecture for a NAND gate Model

```
--
--the following is a behavioral description of
--a three input NAND gate.
--
ENTITY nand3 IS
PORT(
        a  : IN   std_logic;
        b  : IN   std_logic;
        c  : IN   std_logic;
        z  : OUT  std_logic);
END ENTITY nand3;

ARCHITECTURE beh OF nand3 IS
 BEGIN
    z <= '1'  WHEN a='0' AND b='0' ELSE
         '1'  WHEN a='0' AND b='1' ELSE
         '1'  WHEN a='1' AND b='0' ELSE
         '0'  WHEN a='1' AND b='1' ELSE
         'X';
END ARCHITECTURE beh;
```

You can create VHDL source code in any directory.

VHDL source code file may be anything......but,
Use the name of the design entity with the extension ".*vhd*"

The above example would be in the file: nand3.vhd

Question: Why the 'X' in the above code?

Signal Assignment

The assignment operator (<=) is used to assign a waveform value to a *signal*.

Format:

```
target_object <= waveform;
```

Examples:

```
my_signal  <= '0'; --ties my_signal to "ground"
his_signal <= my_signal; --connects two wires

--vector signal assignment

data_bus <= "0010"; -- note double quote
bigger_bus  <= X"a5";  -- hexadecimal numbers
```

Declaring Objects

Declaration Format:

```
OBJECT_CLASS  identifier:  TYPE [:= init_val];
```

Examples:

```
CONSTANT   delay    :   TIME:= 10ns;
CONSTANT   size     :   REAL:=5.25;
VARIABLE   sum      :   REAL;
VARIABLE   voltage  :   INTEGER:=0;
SIGNAL     clock    :   BIT;
SIGNAL     spam     :   std_logic:='X';
```

Objects in the port statement are classified as signals by default.

Objects may be initialized at declaration time.

If an object is not initialized, it assumes the left-most or minimum value for the type

Naming Objects

Valid characters:

- **alpha characters (a-z)**

- **numeric characters (0-9)**

- **underscore (_)**

Names must consist of any number of alpha, numeric, or underline characters.

Underscore must be proceeded and followed by alpha or numeric characters.

The underscore can be used to separate adjacent digits in bit strings:
```
CONSTANT big_0 : STD_LOGIC_VECTOR(15 DOWNTO 0) :=
B"0000_0000_0000_0000";
```

Names are not case sensitive. (be consistent!, use lowercase!)

Coding hints:

Use good names that are meaningful to others. If your code is good, somebody else will want to read it.

Name signals by their function. For example, if you have a multiplexor select line that selects addresses, give it a name like "`address_select`" instead of "`sel_32a`".

Name blocks by their function. If a block generates control signals for a DRAM controller, call the block "`dram_ctl`" not something obscure like "`block_d`".

A Simple Example to Recap

```vhdl
-------------------------------------------------
--and-or-invert gate model
--Jane Engineer
--3/13/01
--version 0.5
-------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY aoi4 IS
PORT(
      a  : IN     std_logic;
      b  : IN     std_logic;
      c  : IN     std_logic;
      d  : IN     std_logic;
      z  : OUT    std_logic);
END ENTITY aoi4;

ARCHITECTURE data_flow OF aoi4 IS
  SIGNAL  temp1, temp2  :  std_logic;
  BEGIN
    temp1 <= a AND b;
    temp2 <= c AND d;
    z      <= temp1 NOR temp2;
END ARCHITECTURE data_flow;
```