CLEMSON UNIVERSITY

M.C. Smith

Adapted from:
"VHDL Quick Start" by Peter J. Ashenden

VHDL INTRO (PART I)

Objective

CLEMSON UNIVERSITY

- Introduction to VHDL
  - basic language concepts
  - basic design methodology
- Some Suggested Resources for
  - self-learning for more depth
  - references for projects

1. *The Internet*
2. *The Student's Guide to VHDL*
3. *The Designer's Guide to VHDL*
4. *Introductory VHDL From Simulation to Synthesis*
5. *VHDL A Starter's Guide*

Why Use VHDL?

- Quick Time-to-Market
  - Allows designers to quickly develop designs requiring tens of thousands of logic gates
  - Provides powerful high-level constructs for describing complex logic
  - Supports modular design methodology and multiple levels of hierarchy
- One language for design and simulation
- Allows creation of device-independent designs that are portable to multiple vendors.
  → Good for ASIC Migration
  - Allows user to pick any synthesis tool, vendor, or device

Hardware Description Languages

- Register transfer level (RTL) design describes the logic between a pair of register stages
  - Useful in describing functionality that will be mapped to FPGAs
- VHDL and Verilog
  - Support RTL hardware descriptions
  - Used to describe concurrency at the hardware level
  - We can write complex logic expressions and let the tools map them to gates
  - Differ mainly in syntax
    (VHDL derived from Ada; Verilog from C)

VHDL vs. Verilog History

VHDL

- Developed by DoD in early 80s as means for Contractors to Describe Designs-Funded VHSIC
- 1987 IEEE ratified 1076 and DoD mandated VHDL(F-22) and EDA vendors created tools.
- 1993 - IEEE 1076 '93
- 1996 Commercial Sim and Synthesis tools become available and 1164 pkg enables multi value logic

Verilog

- 1983 -Gateway founded by Genrad's HDL and HILO simulator author.Releases Verilog HDL and Simulator
- 1985 Enhanced Verilog-XL-used for high end designs -Fast Simulator - interpretive-no need to precompile
- 1990 Cadence buys Gateway-nearly all ASIC foundries used XL as Golden Simulator
- 1995 IEEE 1364

VHDL vs. Verilog Compilation/Data Types/High Level Constructs/Verbosity/Ease

VHDL

- Many E-A pairs may reside in single system file
- User Can define Data Types-Powerful
- High Level Modeling w/ Package, Config, Generate
- Strongly Typed Language - models must be precisely coded-often longer code
- Less intuitive but much more powerful constructs

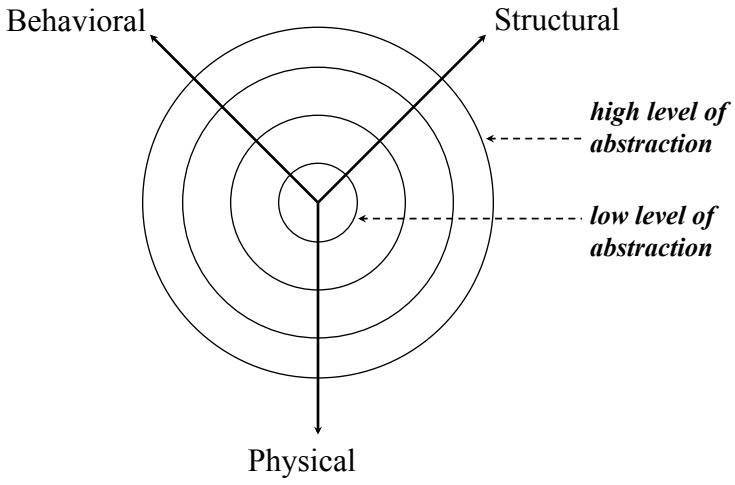
Verilog

- Order of Code is crucial to obtaining desired output
- Simple Data Types are controlled by language
- No Equivalent High Level Modeling Constructs
- Verilog has looser structure-can lead to unwanted and unidentified errors-more concise code
- May be easiest to grasp - but prone to create unwanted results

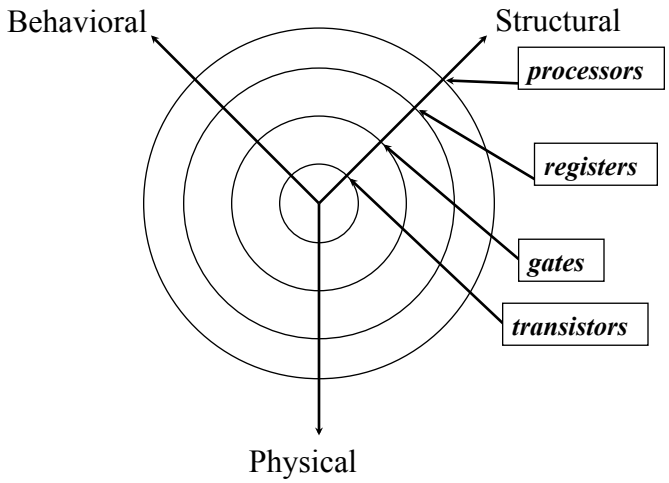
Modeling Digital Systems

- VHDL is useful for writing models of a system
- Reasons for modeling
  - requirements specification
  - documentation
  - testing using simulation
  - formal verification
  - **Synthesis!!!!!!!**
- Goal
  - most reliable design process, with minimum cost & time
  - avoid design errors!

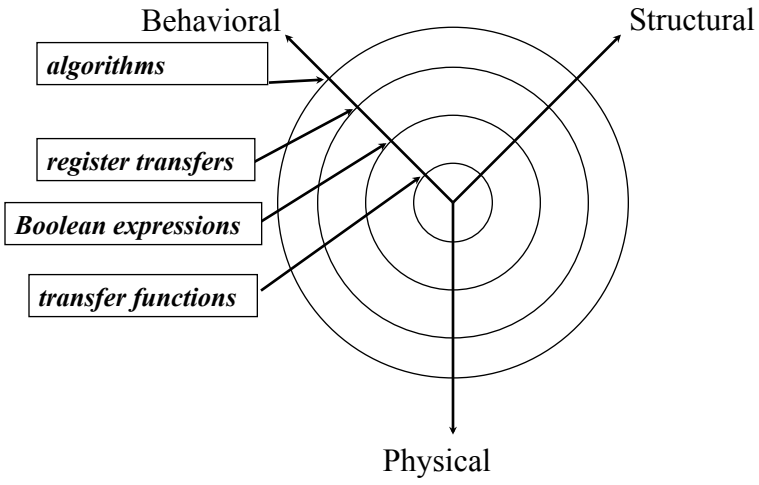
Domains and Levels of Modeling



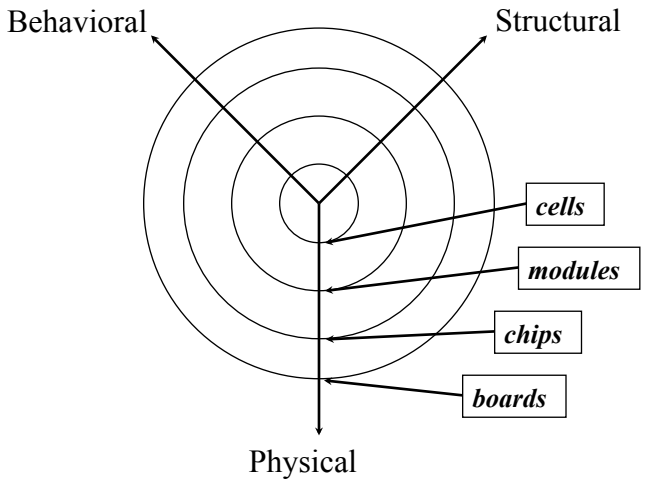
Domains and Levels of Modeling



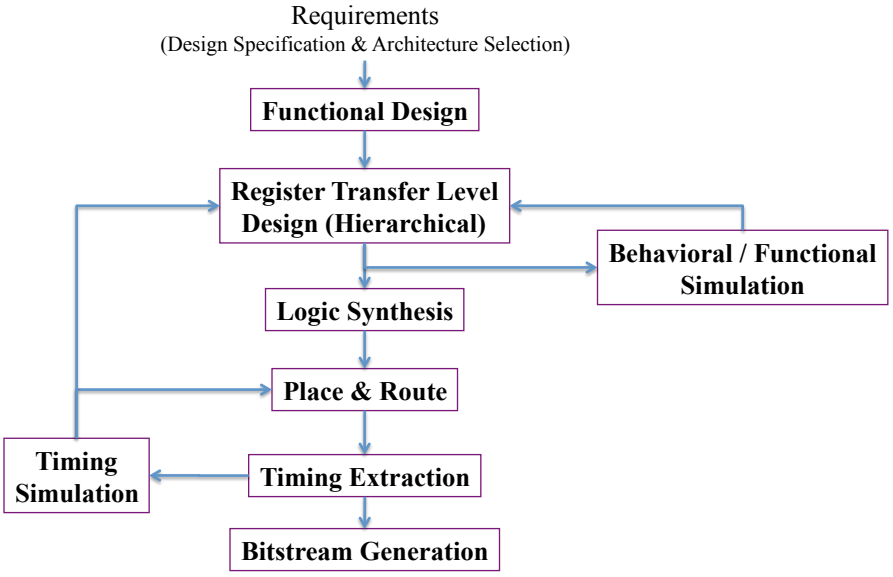
Domains and Levels of Modeling



Domains and Levels of Modeling



Synthesis Design Flow for FPGAs



VHDL Introduction

- Designer writes a logic circuit description in VHDL source code
- VHDL compiler translates this code into a logic circuit
- Representation of digital signals in VHDL
  - Logic signals in VHDL are represented as a data object
  - VHDL includes a data type called ***BIT***
  - BIT objects can assume only two values: 0 and 1

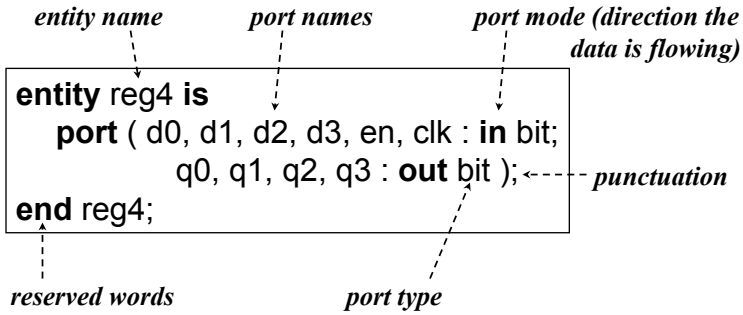
We will use a more advanced datatype in our designs…

Basic VHDL Concepts

- VHDL code is composed of a number of entities
  - Used to describe the interface of the component
  - Can be primitive objects or complex objects
- Architectures are associated with each entity
  - Used to describe Behavior or Structure
- Test Benches
- Analysis
  - Check for design errors (syntax and semantic)
  - Check each design unit separately
- Simulation
- Synthesis

Simple VHDL: Entity

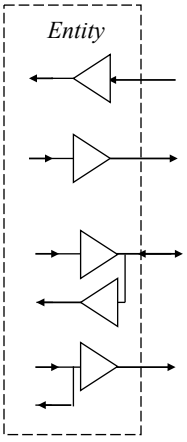
- First step in writing VHDL code is to declare the input and output signals
- Done using a construct called an ***entity***
  - describes the input/output ports of a module



Port Modes

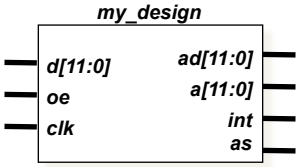
- A port's MODE indicates the direction that data is transferred:

- **IN** Data goes into the entity only
- **OUT** Data goes out of the entity only (and is not used internally)
- **INOUT** Data is bi-directional (goes into and out of the entity)
- **BUFFER** Data that goes out of the entity and is also fed-back internally



Exercise #1: The Entity - A Walk through

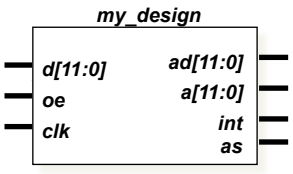
- Write an entity declaration for the following:
  - Port D is a 12-bit bus, input only
  - Port OE and CLK are each input bits
  - Port AD is a 12-bit, three-state bi-directional bus
  - Port A is a 12-bit bus, output only
  - Port INT is a three-state output
  - Port AS is an output also used internally



Exercise #1: Solution

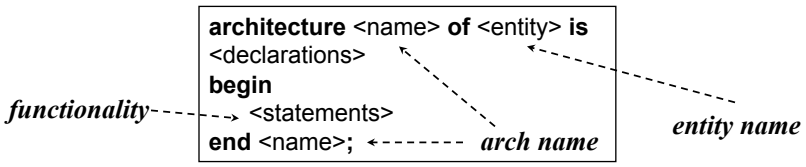
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY my_design IS PORT (
   d:        IN std_logic_vector(11 DOWNTO 0);
   oe, clk:  IN std_logic;
   ad:       INOUT std_logic_vector(11 DOWNTO 0);
   a:        OUT std_logic_vector(11 DOWNTO 0);
   int:      OUT std_logic;
   as:       BUFFER std_logic);
END my_design;

-- In this example, VHDL keywords
-- are highlighted in bold, CAPITALS;
-- however, VHDL is not case sensitive:
-- clock, Clock, CLOCK all refer to the
-- same signal, -- means a comment
```



Simple VHDL: Architecture Body

- The entity specifies the inputs and outputs for a circuit, but does not describe the circuit function
- Circuit functionality is specified using a VHDL construct called an ***architecture***
  - Describes an implementation (content or function) of an entity
  - May be several per entity
  - Can be behavioral, data-flow, structural, or combination



Modeling Behavior

- *Behavioral* architecture
  - Describes the algorithm performed by the module
  - Contains *process statements*, each containing
    - *sequential statements*, including
      - *signal assignment statements* and
      - *wait statements (explicit or implicit)*

Implicit wait

```
architecture behav of xor_gate is
begin
    new_input : process(a,b)
    begin
        if (a = b)
            f <= 0   after prop_delay;
        else
            f <= 1   after prop_delay;
    end process;
end behav;
```

Only for modeling, not synthesis

Uses High-Level programming constructs, like in C or Pascal

Behavioral Example

```
architecture behav of reg4 is
begin
    storage : process
        variable stored_d0, stored_d1, stored_d2, stored_d3 : std_logic;
    begin
        if en = '1' and clk = '1' then
            stored_d0 := d0;
            stored_d1 := d1;
            stored_d2 := d2;
            stored_d3 := d3;
        end if;
        q0 <= stored_d0;
        q1 <= stored_d1;
        q2 <= stored_d2;
        q3 <= stored_d3;
        wait on d0, d1, d2, d3, en, clk;
    end process;
end behav;
```

Explicit wait

Modeling Data-flow Architecture

- ***Consists entirely of boolean equations***
- *References only* ***port signals***

```
entity xor_gate is
    port (
      a : IN std_logic;
      b : IN std_logic;
      f : OUT std_logic );
end xor_gate;
```

```
architecture dataflow of xor_gate is
begin
    f  <=  (a AND NOT b)  OR  (NOT a AND b);
end dataflow;
```

High-Level programming constructs or rtl/boolean-like descriptions

Sometimes lumped in with behavioral

Obviously not structural

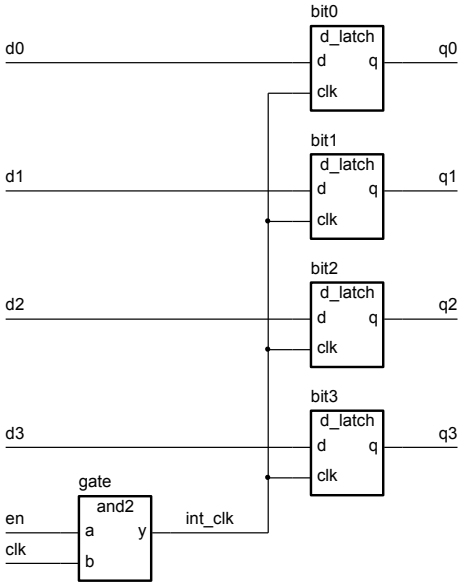
Modeling Structural Architecture

- Defines an entity using **components**
- Internal signals connect components
  - *signal declarations*, for internal interconnections
    - the entity ports are also treated as signals
  - *component instances*
    - instances of previously declared entity/architecture pairs
  - *port maps* in component instances
    - connect signals to component ports
  - *wait statements*

more like a netlist

```
architecture structure of xor_gate is
    component nor_gate
        port (x, y: IN std_logic;
              z: OUT std_logic);
    end component;
    component and_gate
        port (x, y: IN std_logic;
              z: OUT std_logic);
    end component;
    signal s1, s2: std_logic;
begin
    nor1: nor_gate PORT MAP (s1, s2, f);
    and1: and_gate PORT MAP (a, b, s1);
    nor2: nor_gate PORT MAP (a, b, s2);
end structure;
```


Structure Example



Structure Example

- First declare D-latch and and-gate entities and architectures

```
entity d_latch is
    port ( d, clk : in std_logic;
           q : out std_logic);
end d_latch;

architecture basic of d_latch is
begin
    latch_behavior : process
    begin
        if clk = '1' then
            q <= d after 2 ns;
        end if;
        wait on clk, d;
    end process latch_behavior;
end basic;
```

```
entity and2 is
    port ( a, b : in std_logic;
           y : out std_logic);
end and2;

architecture basic of and2 is
begin
    and2_behavior : process
    begin
        y <= a and b after 2 ns;
        wait on a, b;
    end process and2_behavior;
end basic;
```


Structure Example

- Declare corresponding components in register architecture body

```
architecture struct of reg4 is
    component d_latch
        port ( d, clk : in std_logic;  q : out std_logic);
    end component;
    component and2
        port ( a, b : in std_logic;  y : out std_logic);
    end component;
    signal int_clk : std_logic;
...
```

Structure Example

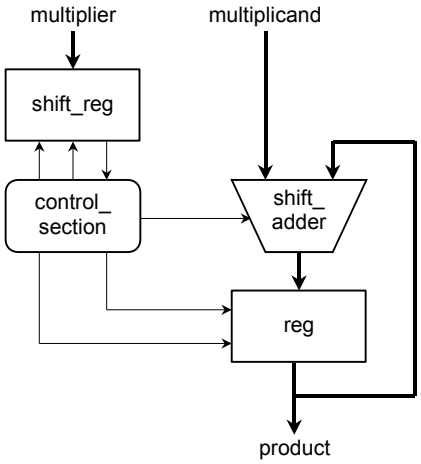
- Now use them to implement the register

```
...
begin
    bit0 : d_latch
        port map ( d0, int_clk, q0 );
    bit1 : d_latch
        port map ( d1, int_clk, q1 );
    bit2 : d_latch
        port map ( d2, int_clk, q2 );
    bit3 : d_latch
        port map ( d3, int_clk, q3 );
    gate : and2
        port map ( en, clk, int_clk );
end struct;
```

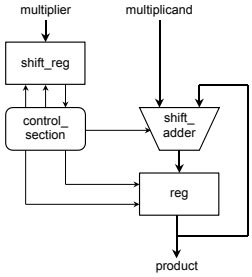

Mixed Behavior and Structure

- An architecture can contain both behavioral and structural parts
  - process statements and component instances
    - collectively called concurrent statements
  - processes can read and assign to signals
- Example: register-transfer-level model
  - data path described structurally
  - control section described behaviorally

Mixed Example



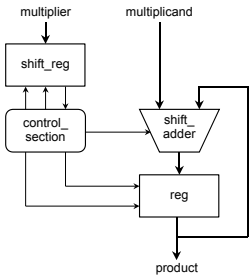
Mixed Example




```
entity multiplier is
    port ( clk, reset : in std_logic;
           multiplicand, multiplier : in integer;
           product : out integer );
end multiplier;

architecture mixed of mulitplier is
    signal partial_product, full_product : integer;
    signal arith_control, result_en, mult_bit, mult_load : std_logic;
begin
    arith_unit : entity work.shift_adder(behavior)
        port map ( addend => multiplicand,  augend => full_product,
                   sum => partial_product,
                   add_control => arith_control );
    result : entity work.reg(behavior)
        port map ( d => partial_product,  q => full_product,
                   en => result_en,  reset => reset );
    ...
```

Mixed Example




```
…
multiplier_sr : entity work.shift_reg(behavior)
    port map ( d => multiplier,  q => mult_bit,
               load => mult_load,  clk => clk );
product <= full_product;

control_section : process is
    -- variable declarations for control_section
    -- …
begin
    -- sequential statements to assign values to control signals
    -- …
    wait on clk, reset;
end control_section;
end mixed;
```


IEEE 1076 Types

- VHDL is a strongly typed language (you cannot assign a signal of one type to the signal of another type)
  - bit - a signal of type bit that can only take values of '0' or '1'
  - bit_vector - a grouping of bits (each can be '0' or '1')

```
SIGNAL   a: BIT_VECTOR(0 TO 3);     -- ascending range
SIGNAL   b: BIT_VECTOR(3 DOWNTO 0); -- descending range
         a <= "0111"; -- double quotes used for vectors
         b <= "0101";
```

This means that:

a(0) = '0'	b(0) = '1'
a(1) = '1'	b(1) = '0'
a(2) = '1'	b(2) = '1'
a(3) = '1'	b(3) = '0'

IEEE 1076 Types (cont.)

CLEMSON UNIVERSITY

- INTEGER
  - useful as index holders for loops, constants, generics, or high-level modeling
- BOOLEAN
  - can take values 'TRUE' or 'FALSE'
- ENUMERATED
  - has user defined set of possible values, e.g.,
  - TYPE traffic_light IS (green, yellow, red);

IEEE 1164

CLEMSON UNIVERSITY

- A package created to solve the limitations of the BIT type
- Nine values instead of just two ('0' and '1')
- Allows increased flexibility in VHDL coding, synthesis, and simulation
  - Can assume several different values [0, 1, Z (high impedance), - (don't care), etc]
- STD_LOGIC and STD_LOGIC_VECTOR are used instead of BIT and BIT_VECTOR when a multi-valued logic system is required
- STD_LOGIC and STD_LOGIC _VECTOR must be used when tri-state logic (Z) is required
- To be able to use this new type, you need to add 2 lines to your code:

  LIBRARY ieee;

  USE ieee.std_logic_1164.ALL;

std_logic and std_logic_vector

IEEE 1164 Types

- std_logic and std_logic_vector are the industry standard logic type for digital design
- Values for Simulation & Synthesis
  - '0' -- Forcing '0'
  - '1' -- Forcing '1'
  - 'Z' -- High Impedance
  - 'L' -- Weak '0'
  - 'H' -- Weak '1'
  - '-' -- Don't care
- Values for Simulation only (std_ulogic):
  - 'U' -- Uninitialized
  - 'X' -- Forcing Unknown
  - 'W' -- Weak Unknown

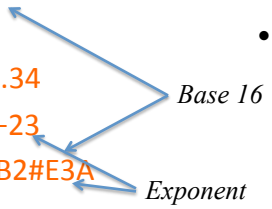
Not every simulatable construct is synthesizable or representable in hardware!!

Data Types

- Many data types in VHDL, but we will focus on a small subset for now:
  - STD_LOGIC
    - A single bit
  - STD_LOGIC_VECTOR
    - A bit vector, often used for a bus
  - INTEGER
    - Standard integer type that will often be used for generic parameters

VHDL Literals

- Numbers
  - 14364
  - 14_364
  - 16#FF03#
  - 16.3
  - 13_634.34
  - 1.293E-23
  - 16#3A.B2#E3A



- Bits
  - X"FF_AA"
  - B"1101_1010"
  - O"037"
- Characters
  - ' A'
  - ' _'
  - ''''
  - "This is a string"

Declarations - Signals

- Explicit declaration
  - CONSTANT bus_width : INTEGER := 32;
  - CONSTANT rise_delay : TIME := 20ns;
  - VARIABLE data_val :STD_LOGIC_VECTOR (7 DOWNTO 0);
  - VARIABLE sum : INTEGER RANGE 0 TO 100;
  - VARIABLE done : BOOLEAN;
  - SIGNAL clock : STD_LOGIC;
  - SIGNAL addr_bus : STD_LOGIC_VECTOR (31 DOWNTO 0);
- Implicit declaration
  - FOR count IN 1 TO 10 LOOP -- declares count
    sum := sum + count;
    END LOOP;

Declarations - Components

```
component fill_reg
  port(
    clk :in std_logic;
    rst :in std_logic;
    write :in std_logic;
    read :in std_logic;
    data_in :in std_logic_vector(3 downto 0);
    FFout :out std_logic;
    data_out:out std_logic_vector(31 downto 0));
end component;
```