Introduction to VHDL

Dr. Adnan Shaout

The University of Michigan-Dearborn

Objective

- Quick introduction to VHDL
  - basic language concepts
  - basic design methodology
  - examples

VHDL

Very Hard Difficult Language

jk -- VHDL

VHSIC Hardware

Description Language

VHSIC --

Very High Speed Integrated Circuits

Modeling Digital Systems

- VHDL is for coding models of a digital system...
- Reasons for modeling
  - requirements specification
  - documentation
  - testing using simulation
  - formal verification
  - synthesis
  - class assignments
- Goal
  - most ‘reliable’ design process, with minimum cost and time
  - avoid design errors!

Basic VHDL Concepts

- Interfaces -- i.e. ports
- Behavior
- Structure
- Test Benches
- Analysis, simulation
- Synthesis

VHDL --

- VHDL is a programming language that allows one to model and develop complex digital systems in a dynamic environment.

- Object Oriented methodology for you C people can be observed -- modules can be used and reused.

- Allows you to designate in/out ports (bits) and specify behavior or response of the system.

VHDL Intro.--

- Oh yeah, For all you C people --forget everything you know...
- Well, not EVERYTHING ...
- But VHDL is NOT C ...
  There are some similarities, as with any programming language, but syntax and logic are quite different; so get over it !!

  -obviously, this was a painful transition for me.

3 ways to DO IT -- the VHDL way

- Dataflow
- Behavioral
- Structural

Kindof BORING sounding huh??

well, it gets more exciting with the details !!

:)

Modeling the Dataflow way

- uses statements that defines the actual flow of data.....

  such as,

  ```
  x  <= y          -- this is NOT less than equl to
                   -- told you its not C
  ```

  this assigns the boolean signal x to the value of boolean signal y... i.e. x = y

  this will occur whenever y changes....

Jumping right in to a Model -- e.g. 1

- lets look at a d - flip-flop model -- doing it the dataflow way..... ignore the extra junk for now --

```
entity dff_flow is
    port (      d       :in   bit;
                prn     :in   bit;
                clrn    :in   bit;
                q       :out  bit;
                qbar    :out  bit;
          );
end dff_flow;
architecture  arch1 of dff_flow  is
begin
    q  <=  not prn     Or   (clrn  And  d);      % this is  the DATAFLOW %
    qbar <= prn And  (not clrn  Or not d);     %  STUFF    %
end arch1;
```

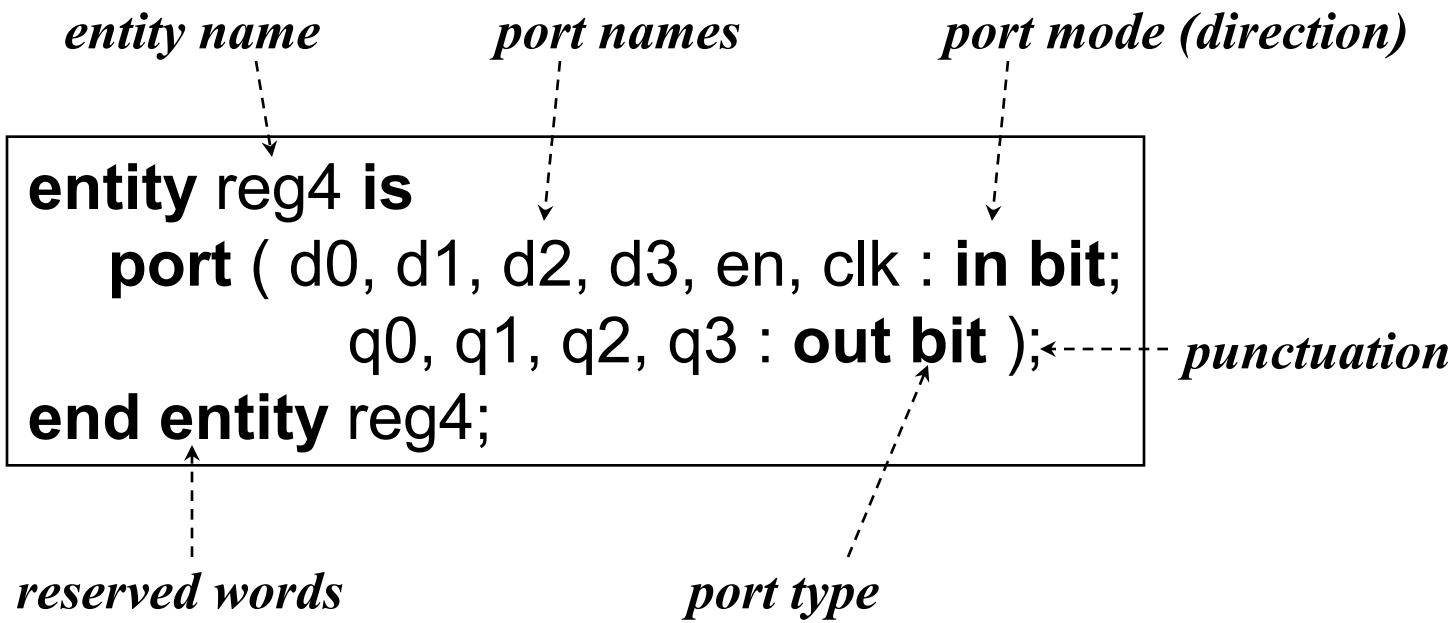
```
library ieee; use ieee.std_logic_1164.all;

entity fulladd is
        port(A1,A2,Cin: IN std_logic;
                Sum, Cout: OUT std_logic);
end fulladd;

Architecture a of fulladd is
Begin
        process(A1,A2,Cin)
        Begin
            Sum <= Cin XOR A1 XOR A2;
            Cout <= (A1 AND A2) OR (Cin AND (A1 XOR A2));
        end process;
end a;
```

Modeling Interfaces

- *Entity* declaration
  - describes the input/output *ports* of a module



Modeling the Behavior way

- *Architecture body*
  - describes an implementation of an entity
  - may be several per entity
- *Behavioral* architecture
  - describes the algorithm performed by the module
  - contains
    - *process statements*, each containing
      - *sequential statements*, including
        - *signal assignment statements* and
        - *wait statements*

The Behavior way -- eg 2

```
architecture behav of reg4 is
begin
    process (d0, d1, d2, d3, en, clk)
        variable stored_d0, stored_d1, stored_d2, stored_d3 : bit;
    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 after 5 ns;
        q1 <= stored_d1 after 5 ns;
        q2 <= stored_d2 after 5 ns;
        q3 <= stored_d3 after 5 ns;
    end process;
end behav;
```

sensitivity list

notice := syntax used for equating values from signals...

simulates real-world propagation delays.

VHDL -- goofy syntax to know..

- Omit **entity** at end of entity declaration
- Omit **architecture** at end of architecture body
- Omit **is** in process statement header

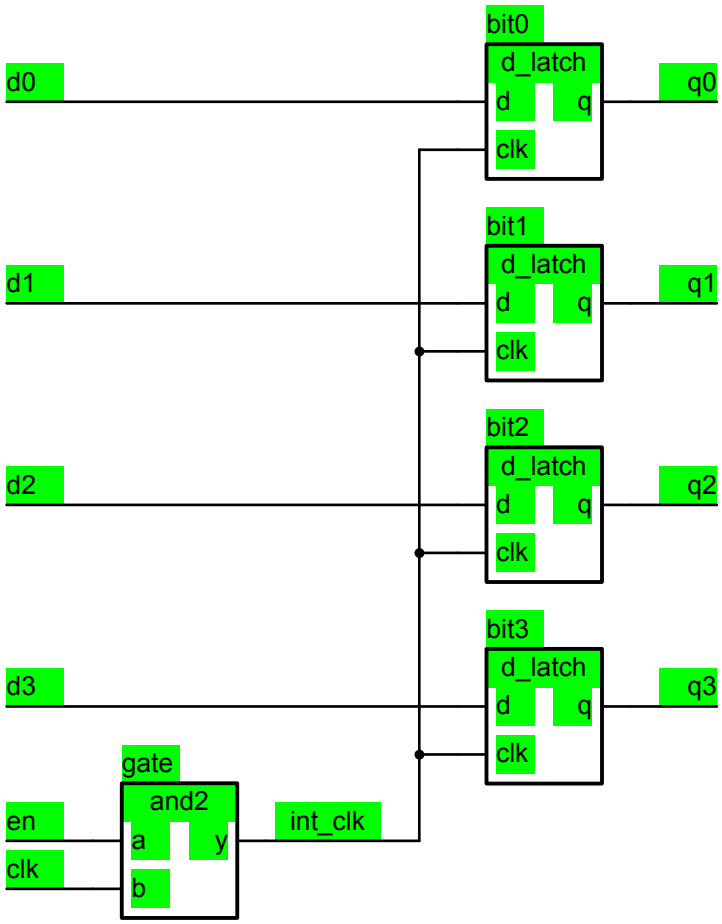
```
entity reg4 is
port ( d0, d1, d2  : in bit
       d3, en, clk : in bit;
   q0, q1, q2, q3 : out bit
      );
end reg4;
```

```
architecture behav of reg4 is
begin
    process (d0, ... )
        ...
    begin
        ...
    end process ;
end behav;
```

Modeling the Structurural way

- *Structural* architecture
  - implements the module as a composition of subsystems
  - contains
    - *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

Structural way -- e.g. 3



Structural way cont..

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

notice semicolon placements -- odd as it is, omit from last statement

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

architecture basic of d_latch is
begin
    process (clk, d)
    begin
        if clk = '1' then
            q <= d after 2 ns;
        end if;
    end process;
end basic;
```

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

architecture basic of and2 is
begin
    process (a, b)
    begin
        y <= a and b after 2 ns;
    end process ;
end basic;
```

Structural way...

- Declare corresponding components in register architecture body

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

Structural way..

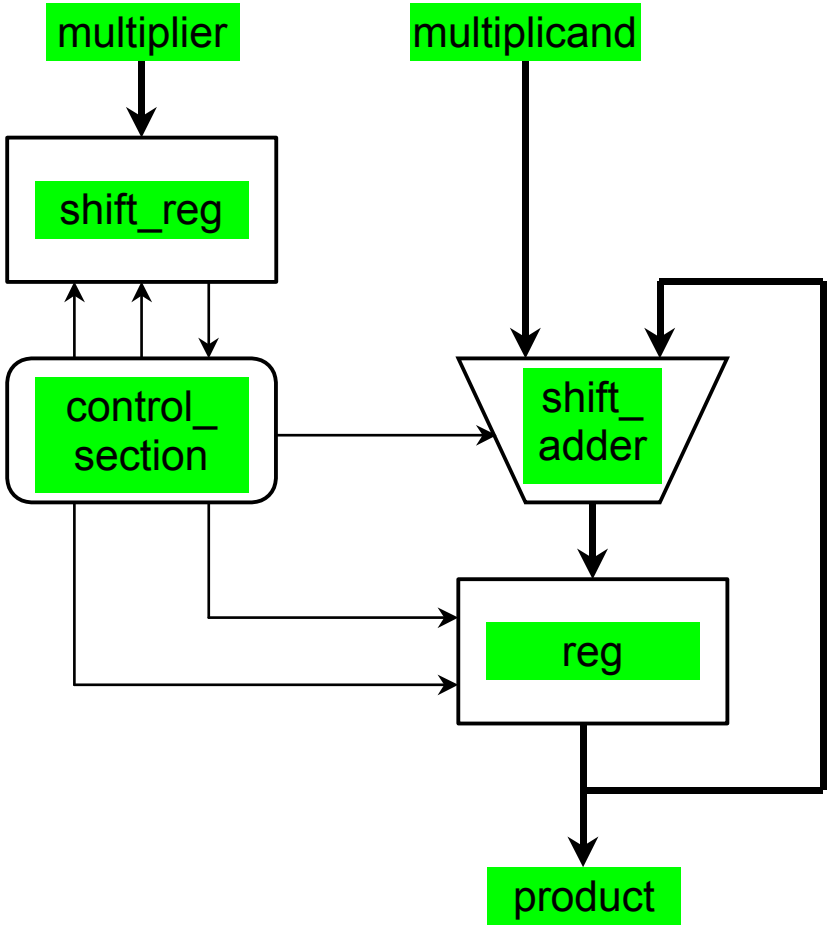
- Now use them to implement the register

```
...
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 (RTL) Model
  - data path described structurally
  - control section described behaviorally

Mixed Example



Mixed Example

```
entity multiplier is
    port ( clk, reset : in bit;
           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 : bit;
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;

    process (clk, reset)
        -- variable declarations for control_section
        -- …
    begin
        -- sequential statements to assign values to control signals
        -- …
    end process;
end mixed;
```

Test Bench your Model

- Testing a design by simulation
- Use a *test bench* model
  - a Model that uses your Model
  - apply test sequences to your inputs
  - monitors values on output signals
    - either using simulator
    - or with a process that verifies correct operation
    - or logic analyzer

Analysis

- Check for syntax and logic errors
  - syntax: grammar of the language
  - logic: how your Model responds to stimuli
- Analyze each *design unit* separately
  - entity declaration
  - architecture body
  - …
  - put each design unit in a separate file -- *helps a lot.*
- Analyzed design units are placed in a *library*
  - make sure your Model is truly OOP

Simulation

- Discrete event simulation
  - time advances in discrete steps
  - when signal values change—*events* occur
- A processes is *sensitive* to events on input signals
  - specified in wait statements
  - resumes and schedules new values on output signals
    - schedules *transactions*
    - *event* on a signal if value changes

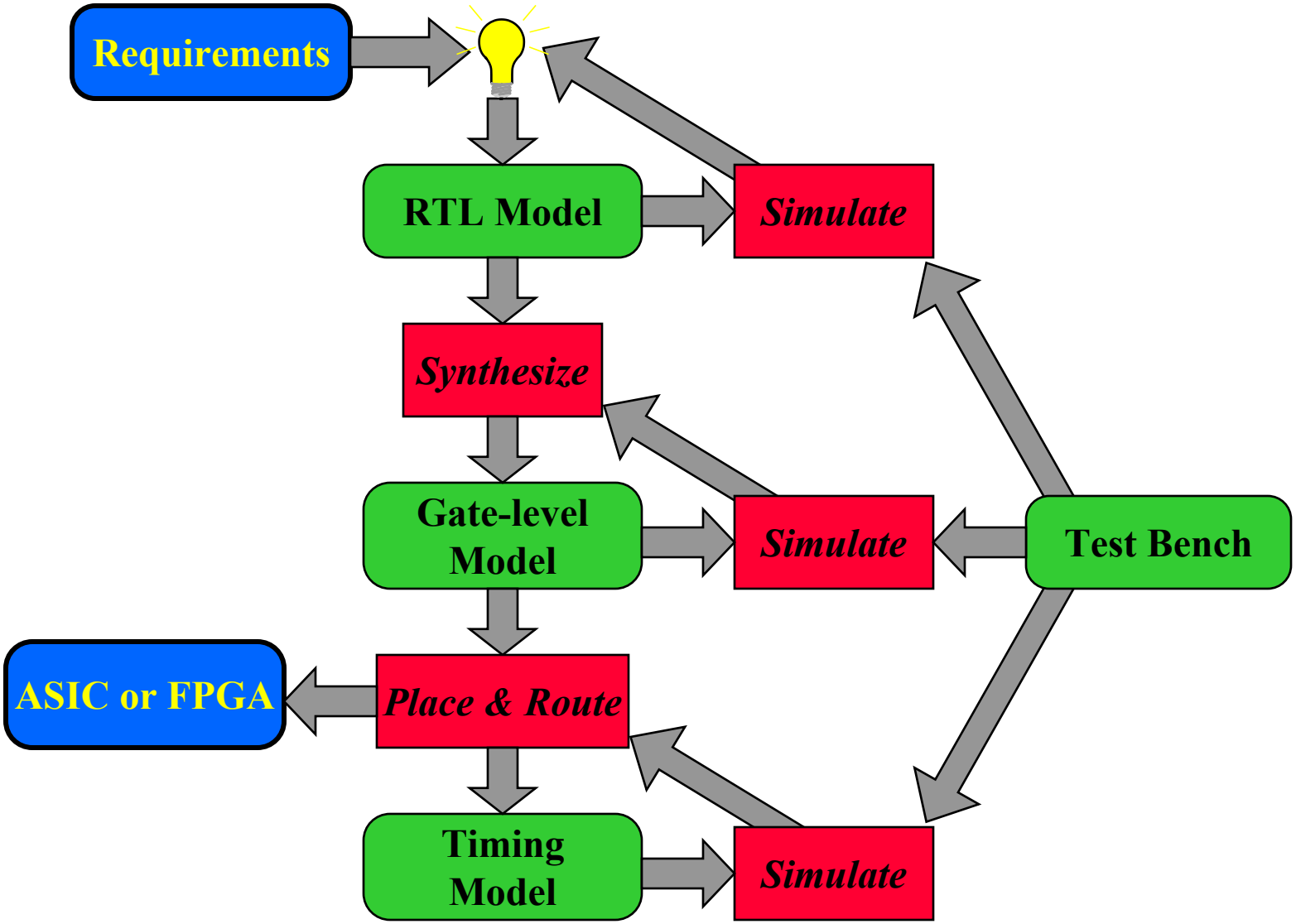
Simulation Algorithm

- Initialization phase
  - each signal is given its initial value
  - simulation time set to 0
  - for each process
    - activate
    - execute until a wait statement, then suspend
      - execution usually involves scheduling transactions on signals for later times

Simulation Algorithm

- Simulation cycle
  - advance simulation time to time of next transaction
  - for each transaction at this time
    - update signal value
      - event if new value is different from old value
  - for each process sensitive to any of these events, or whose "wait for …" time-out has expired
    - resume
    - execute until a wait statement, then suspend
- Simulation finishes when there are no further scheduled transactions

Basic Design Methodology



VHDL -- conclusion...

- Thats it !! in review -- replay presentaion

- Now for first asignment design a computer
  - Memory access
  - processor
  - data/address bus
  - display

- Always remember to use this knowledge for GOOD...