VHDL Introduction
VHDL

- VHSIC Hardware Description Language
  - it *is* a programming language
  - it *it not* a programming language
    » it’s a hardware description language

- You *might* write a dBase program in VHDL
  - we will stick with *synthesizable* VHDL
    » we want to build hardware in the end
    » synthesis tools only take a subset of VHDL
VHDL Elements

-- comments look like this

◆ identifiers

◆ numbers

◆ characters: ‘0’

◆ strings: “0011” or “abc”
VHDL Types

- integers
  - easily converted to bit strings and back
- floating point
- enumeration types
- arrays
- subtypes
  - restricted version of another type
    » subtype address is integer range 0 to 65535
Other VHDL Elements

◆ attributes
  – built-in
    » clk’event
    » vec’high
    » xyz’range
  – user-defined
  – one way to give synthesis directives

◆ entities/architectures

◆ configurations
Vector Types Tutorial

-- The std_logic_arith package defines the SIGNED and UNSIGNED types
-- In this architecture type conversions are used on plain
-- STD_LOGIC_VECTOR's to force them to be compared as SIGNED if an
-- UNSIGNED compare is not wanted.
lifetime ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity demoB is
derial archiecture SIGNED_convert of demoB is
  signal a : STD_LOGIC_VECTOR(3 downto 0);
signal b : STD_LOGIC_VECTOR(3 downto 0);
signal c : STD_LOGIC;
signal d : STD_LOGIC;
begin
  a <= "1111";
b <= "0100";
  c <= '1' when (SIGNED(a) < SIGNED(b)) else '0';
  d <= '1' when (a < b) else '0';
end SIGNED_convert;
Vector Types (part 2)

-- By using package std_logic_signed we force STD_LOGIC_VECTORs to be treated
-- as SIGNED by default. Type conversions are now needed to make them be treated as UNSIGNED.
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;
entity demoC is
end;
architecture use_signed of demoC is
  signal a : STD_LOGIC_VECTOR(3 downto 0);
  signal b : STD_LOGIC_VECTOR(3 downto 0);
  signal c : STD_ULOGIC;
  signal d : STD_ULOGIC;
begin
  a <= "1111";
  b <= "0100";

  -- this is true, due to the extra USE stmt above
  c <= '1' when (a < b) else '0';
  -- this is false, the type conversions make 'a' be viewed as a 15.
  d <= '1' when (UNSIGNED(a) < UNSIGNED(b)) else '0';
end use_signed;
Vector Types (part 3)

-- Using the SIGNED type it is also possible to have SIGNED without
-- type conversions.
library iee;
use iee.std_logic_1164.all;
use iee.std_logic_arith.all;
entity demoD is
end;
architecture SIGNED_type of demoD is
  signal a : SIGNED(3 downto 0);
  signal b : SIGNED(3 downto 0);
  signal c : STD_ULOGIC;
begin
  a <= "1111";
  b <= "0100";
  -- this is true, due to their inherent types.
  c <= '1' when (a < b) else '0';
end SIGNED_type;
Entity Declarations

entity add4 is -- 4 bit adder
port( a : in std_logic_vector(3 downto 0);
b : in std_logic_vector(3 downto 0);
s : out std_logic_vector(3 downto 0));
end add4;

entity adder is -- generic adder
generic (width: positive);
port( a : in std_logic_vector(width-1 downto 0);
b : in std_logic_vector(width-1 downto 0);
s : out std_logic_vector(width-1 downto 0));
end adder;
More Entity Declarations

entity adder is -- another generic adder
port( a : in std_logic_vector;
     b : in std_logic_vector;
     s : out std_logic_vector);
end adder;
architecture foo of adder is
begin
  ... a'high ...
  ... b'range
end;

-- no ports, a container for a testBench
entity testBench is
end testBench;
Architectures

- The *body* or *implementation* of the entity.

- Can have more than one
  - specify which one to use
    » using configurations (ugh!)
    » to the tool
Architecture Examples

architecture first of add4 is
  begin
  s <= a + b; -- works with std_logic_vector or integer
  end;

architecture second of add4 is
  signal c: std_logic_vector(2 downto 0);
  begin
  s(0) <= a(0) XOR b(0);
  c(0) <= a(0) AND b(0);
  s(1) <= a(1) XOR (b(1) XOR c(0));
  c(1) <= .....
  end add4;
Yet More on Architectures

- local declarations
  - types
  - constants
  - signals (wires)

- Examples:
  - subtype addr is …
  - constant a : integer := 12;
  - signal d : bit := ‘0’;
Concurrent Statements

- \( s(0) \leq a(0) \oplus b(0); \)
- \( q \leq a \text{ when } \text{sel}='0' \text{ else } b; \)
- \text{process}(a, b, sel) begin
  if \( \text{sel}='0' \) then
    \( q \leq a; \)
  else \( q \leq b; \)
end process;

Each of these lines executes any time one of its inputs changes…
(what are its inputs?)

A process is like one big concurrent statement. It executes whenever one of its inputs changes…
(what are its inputs?)

It executes sequentially once it is awakened for execution.

- always concurrently executing
Concurrent vs. Sequential Code

- **Sequential Code**
  - processes
  - procedures
  - functions

- **Concurrent Code**
  - assignments
    - simple
    - conditional
    - selected

For every concurrent signal assignment there is a corresponding sequential construct.

The opposite it NOT true.
Example Assignments

\[ a <= b; \]

\[
q <= \begin{cases} 
  a & \text{when sel="00"} \\
  b & \text{when sel="01"} \\
  c & \text{when sel="10"} \\
  d & \text{else}
\end{cases}
\]

\[
\begin{align*}
\text{process}(b) & \\
& \begin{align*}
& \text{begin} \\
& a <= b; \\
& \text{end process;}
\end{align*}
\end{align*}
\]

\[
\begin{align*}
\text{process}(a, b, c, d, \text{sel}) & \\
& \begin{align*}
& \text{begin} \\
& \text{if sel = "00" then} \\
& \quad q <= a; \\
& \text{elsif sel = "01" then} \\
& \quad q <= b; \\
& \text{elsif sel = "10" then} \\
& \quad q <= c; \\
& \text{else} \\
& \quad q <= d; \\
& \text{end if;}
\end{align*}
\end{align*}
\]
More Assignments

with sel select
  q <= a when "00" else
  b when "01" else
  c when "10" else
  d;

process(a, b, c, d, sel)
begin
  case sel is
    when "00" => q <= a;
    when "01" => q <= b;
    when "10" => q <= c;
    when others => q <= d;
  end case;
end process;
Synthesis Note

- As much as you don’t want to…
  - think hardware in your head
  - “Just what will this construct synthesize to”

- NOT a general programming language
  - recursion???
What in the world is this?

process(b)
begin
for i in b’range loop
  a(i) <= not(b(i));
end loop;
end process;

* It’s just a shorthand for: \texttt{a <= not(b);} where \texttt{a} and \texttt{b} are vectors.
  – No timing or sequentiality is implied!!!
Processes

- Code executes sequentially
  - last assignment to a signal takes precedence
  - can have `wait()` statements
    » useful for synchronous logic
A Loadable Register

process(clk, load, dataIn)
begin
if clk'event and clk = '1' then
  if load = '1' then
    dataOut <= dataIn;
  end if;
end if;
end if;
end process;

◆ What happens when load='0'???
Synthesis Note

- Use synthesizable idioms
  - synthesizers are not going to recognize just anything

- SYNOPSYS VHDL Programming Guide
  » lists their idioms
A Clearable, Loadable Register

process(clk, load, clr, dataIn)
begin
if clk’event and clk = ‘1’ then
  if load = ‘1’ then
    dataOut <= dataIn;
  elsif clr = ‘1’ then -- this is a synchronous clear!
    dataOut <= (others => '0'); -- all zeroes!
  end if;
end if;
end if;
end process;

◆ Which takes priority?
◆ How to write it so they have equal priority? (be careful!!!)
#2 Clearable, Loadable Register

process(clk, load, clr, dataIn)
begin
if clk’event and clk = '1' then
  if load = '1' then
    dataOut <= dataIn;
  end if;
  if clr = '1' then
    dataOut <= (others => '0');
  end if;
end if;
end process;

Which takes priority?
Synthesis Note

- If priority NOT inferred a MUX results.
- If priority inferred, extra logic results.

if sel="00" then ...
elsif sel="01" then ...
elsif sel="10" then...
else ...

Is priority inferred?
...really?

if load = ‘1’ then ...
elsif clr = ‘1’ then ...

If you know they are mutually exclusive then you can tell SYNOPSYS to do a MUX...

Getting complicated if-then-elses right can be a nightmare… (maybe you want a case)
FSM Design - Moore Machine

IFL: \texttt{process}(\texttt{currentState, in1, in2})
\begin{verbatim}
begin
nextState <= currentState; -- default
case currentState is
  when stateA =>
    nextState <= stateY;
  when stateB =>
    if in1='0' AND in2='1' then
      nextState <= stateC;
    else
      nextState <= stateD;
    end if;
end case;
end process;
\end{verbatim}

OFL: \texttt{process}(\texttt{currentState})
\begin{verbatim}
begin
  out1 <= '0'; -- default value
  case currentState is
    when stateA =>
      out1 <= '1';
    when stateB =>   ..
    when stateC =>   ...
  end case;
end process;
\end{verbatim}

\texttt{stateReg: process}(\texttt{clk})
\begin{verbatim}
begin
  if clk'event and clk = '1' then
    currentState <= nextState;
  end if;
end process;
\end{verbatim}

How would you code IFL and OFL in same process?

© Brent E. Nelson 1998
FSM Design - Mealy Machine

IFL: process(currentState, in1, in2)
begin
nextState <= currentState; -- default
out1 <= '0'; -- default
case currentState is
  when stateA =>
    nextState <= stateY;
    out1 <= '1';
  when stateB =>
    if in1='0' AND in2='1' then
      nextState <= stateC;
      out1 <= '1';
    else
      nextState <= stateD;
    end if;
end case;
end process;

stateReg: process(clk)
begin
  if clk'event and clk = '1' then
    currentState <= nextState;
  end if;
end process;
Mealy Machine - Are You Sure?

IFL: process(currentState, in1, in2)
begin
nextState <= currentState; -- default
out1 <= '0';
case currentState is
when stateA =>
nextState <= stateY;
out1 <= '1';
when stateB =>
if in1='0' AND in2='1' then
nextState <= stateC;
out1 <= '1';
else nextState <= stateD;
end if;
end case;
end process;

stateReg: process(clk)
begin
if clk'event and clk = ‘1’ then
currentState <= nextState;
end if;
end process;

I lied, it is a mixed machine.
Is there anything wrong with this?
Nope...
FSM Design

- Use enumerated types for states

- Code in (at least) two processes
  - combinational
    » mix Mealy and Moore - depending on timings
  - next state register
UART Block Diagram

- **Bit Counter**
  - clearBit
  - incBit
  - bit10
  - bit8

- **FSM**
  - clearCnt
  - cnt7
  - cnt15

- **Timer**
  - reset
  - send

- **Shift Register**
  - serialDataIn
  - parallelDataIn (from host)
  - ‘0’ ‘1’
  - shiftSin
  - load
  - shift

- **parallelDataOut (to Host)**

© Brent E. Nelson 1998
The UART Design - One Method

architecture first of UART is
signal …;
begin
comb: process(…)
comb2: process(…)
nextstate: process(…)
shift: process(…)
parity: process(…)
betCode: process(…)
timer: process(…)
end first;

The FSM input forming logic
The FSM output forming logic
The FSM state register
The PISO/SIPO shift register
The parity generator (combinational)
The bit counter and divide down timers
How About Hierarchy?

architecture hierarchical of add4
  signal c : bit_vector(3 downto 0);
  signal gnd : bit;
  component adder
    port (a : in bit; b : in bit; cin : in bit; s : out bit; cout : out bit);
  end component;
  for all:adder use entity work.add(gates);  -- Tell simulator which one to use
begin
  gnd <= '0';  -- GND not needed in VHDL 92 (which we don’t have yet)
  X0: adder port map (a => a(0), b => b(0), cin => gnd, s => s(0), cout => c(0));
  X1: adder port map (b => b(1), cin => c(0), s => s(1), a => a(1), cout => c(1));
  X2: adder port map (a(2), b(2), c(1), s(2), c(2));
  X3: adder port map (a => a(3), b => b(3), cin => c(2), s => s(3), cout => c(3));
end hierarchical;

• Note all the different ways to specify parameter lists...
Hierarchy with Generates

architecture gen of add4 is
    ... put component decl here...
begin
G0: for i in 0 to 3 GENERATE
    if i=0 then
        X: adder port map (a => a(0), b => b(0); cin => gnd; s => s(0); cout => c(0));
    else
        Y: adder port map (a => a(i), b => b(i); cin => c(i-1); s => s(0); cout => c(i));
    end if;
END GENERATE;

-- Don’t quote me on the syntax...
Hierarchy - contd.

- Can put all configurations in a separate file
  - SYNOPSYS ignores them all anyway
    » uses last analyzed architecture for each entity

- Can pass in generics
  - define widths or other numeric values
My Coding Style

- Use hierarchy (structural design) for highest levels
  - chip, datapath, control
- Use behavioral design for lowest levels
  - counters, shifters, arithmetic

- Don’t use hierarchy for things like counters, shifters, …
  - rarely want exactly the same thing twice
  - get what you really want in every case
  - typing the code is often easier than component + instantiation

- Design from the top-down (in my head)
- Develop code from the bottom-up (in editor)
Synthesis Notes

- The synthesizer will optimize every block in its environment
  - timings of arriving inputs
  - demands on timings of outputs
  - loading dependent

- Using hierarchy for small pieces may not save much

- SYNOPSYS does well on up to about 5K gates in a block
  - if bigger than that, try again...
Synthesis Notes

- Synthesizer will choose \textit{smallest} implementation that meets timing

- Synthesis script is where you tell it the target speed and input arrival times.

- Must design according to recommendations – else it will get confused
Synthesis Notes

- Must design according to some recommendations they provide

- else it will not give you an optimal solution
What is arrival time of signal X?  \[= \text{clk-Q + wiring\_delay} \]

Put FF as last element (if possible) in module to simplify analysis.
Avoid combinational-only modules (if possible).
VHDL Simulation Timing Model

- Real timing
  \[a \leq b \text{ after } 12.6 \text{ ns; }\]

- Unit delay timing
  \[a \leq b;\]
  \[c \leq a;\]

  Which value of \( a \) does \( c \) get?
Unit Delay Simulation

simulate() {
    repeat {
        flag = 0;
        for_each_signal_S_on_changed_list {
            for_each_signal_T_on_S’s_fanout_list {
                newT = compute_new_value(T)
                if newT != T {
                    flag = 1;
                    put_T_on_changed_list;
                }
            }
        }
        remove_S_from_changed_list;
    } (until flag = 0);
    }

    A breadth-first traversal - maintains order.

    No timing information.

    Concludes when one pass through circuit results in an empty changed list.
Timing Simulation

- Instead of a changed_list
  - sorted event list

- When a signal changes in +5ns --
  - record that in event list

- Make events happen at proper times
VHDL Timing - does both

- all signals assignments take one $\Delta$ delay if no time specified
  
  a <= b;
  
  c <= a; -- c gets old value of a

- process repeats until circuit stabilizes
  – in unit delay mode

- time then advances to the next timed event
Delay Timings

- All signal assignments take one $\Delta$
  - unless an actual delay is specified
  - this is true in concurrent code
  - this is true in sequential (process) code
  - can be VERY confusing
Example of Δ Timing Subtleties

process(clk)
begin
if clk’event and clk = ‘1’ then
  a <= b;
  b <= a;
end if;
end process;

• So, what does the above code do?
  • both a and b could end up with the original b value
    OR
  • a and b could get swapped...
    which is it???? what does the hardware look like????
More $\Delta$ Timing Subtleties

architecture deltaProblems of classExample is
begin
a <= b when ctl = '1';
b <= a when ctl = '1';
end deltaProblems;

• So, what does the above code do?
  • a and b could get swapped...
    OR
  • ???
VHDL Timing: Do you use both?

- Not really - if you are synthesizing
  - synthesizers ignore timing specifications in code
  - you give synthesizers target timings as a part of the synthesis process, NOT in the VHDL code

- BUT - your testbench will use it:

  clk <= not(clk) after 10 ns;  -- a 50 MHz square wave
  reset <= ‘0’, ‘1’ after 25 ns, ‘0’ after 50 ns;  -- a reset pulse
VHDL Testbenches

- Patterns can be generated algorithmically or read from a file.
- Comparing to correct values is optional (can just view waveforms in simulator instead).
- Correct values to compare against can be generated algorithmically or from file...
Testbench Creation

- Create a nice general testbench
  - inputs come from an array of 1’s and 0’s
  - inputs come from a file

- Cannibalize it for all simulations you do
Variables

- Declared local to a process
- Use := to assign to
- Happens immediately, like in C code
  - no Δ delay

- Note in your text how they are often used
An Example With Variables

patternGenerator process
   variable cnt : integer := 0;
   begin
      for i in 0 to 9 loop
         wait until (clk'event and clk='1');
         cnt := cnt + 1;
         some_signal <= cnt;
      end loop;
   end loop;
end process;

A variable declaration
Increment it.
Assign it to some variable.
(It holds a value just like a signal - it just cannot be used outside the process and has a zero-delay assignment.)
More on Processes

- A process executes once on startup

- A process must have one of:
  - sensitivity list
  - wait statement

```pascal
process(a, b, c)
process
begin
wait until clk’event and clk='1'
do_something_here;
end process
```

There is an implied loop in the *wait* version
Final Thoughts

- Best way to learn VHDL is to write and simulate VHDL modules…

- Next best way is to study existing code…
  - SYNOPSYS manuals
  - Splash2 models