“A Brief Introduction to VHDL”

Tuesday, Thursday - Sept 10,12, 2013

Prof. Ted Szymanski
Dept. of ECE
McMaster University

5 Classes of VHDL statements

- A VHDL design includes 5 types of statements

- *LIBRARY* ; a file which includes ***predefined*** hardware component declarations, provided by Altera, IEEE, etc
- *PACKAGE* ; a file which includes shared data type definitions in one file
- *ENTITY* ; specifies a logic block with its exact I/O ports
- *ARCHITECTURE* ; (a) specifies what an entity does (a ‘**behavioural**’ description) or (b) what an entity is composed of (a ‘**structural**’ description) or (c) specifies a combined behavioral & structural descriptions (a ‘dataflow’ description)
- *CONFIGURATION* ; statement which specifies which versions of entities to use; a version control system
- These classes of statements can be in the same file, or separate files which can be compiled separately, but the order must be preserved for the compiler.

IEEE Libraries and Packages

- The library “ieee” contains several packages of definitions standardized by IEEE, which can be used in all IEEE-certified VHDL environments;
- We’ll always use the ieee definitions

Library	Package	Contents
ieee	std_logic_1164	standard data types (bit, byte..)
ieee	std_logic_arith	signed and unsigned numbers, converters
ieee	std_logic_signed	signed numbers only
ieee	std_logic_unsigned	unsigned numbers only
STD	STANDARD	very basic types (ie BIT)
STD	TEXTIO	definitions for user I/O, printing messages

Defining the Signal Types
(IEEE’s Std_logic_1164 Package)

- The type ***std_ulogic*** consists of 9 symbolic values;

```
TYPE  std_ulogic  IS  (     ‘U’, -- Uninitialized
                  ‘X’, -- Forcing Unknown
                  ‘0’, -- Forcing 0
                   ‘1’, -- Forcing 1
                  ‘Z’, -- High impedance
                  ‘W’, -- Weak unknown
                  ‘L’, -- Weak 0
                  ‘H’, -- Weak 1
                  ‘-’, -- Don’t care   );
```

- The type “***std_logic***” uses the same 9 values, after ***resolution*** (when multiple values are driven onto a wire, one value “wins”)
- ***Std_logic*** allows modelling of 3-state logic, weak signals, etc

Type Converters between Signal Types

- BIT_VECTOR is different from STD_LOGIC_VECTOR
- SIGNED is different from UNSIGNED
- UNSIGNED'("1010") represents +10
- SIGNED'("1010") represents -6
- library "ieee", package "ieee.std_logic_arith", provides several built-in type converter functions:
    - CONV_INTEGER (signal/variable, #bits)
    - CONV_UNSIGNED (signal/variable, #bits)
    - CONV_SIGNED (signal/variable, #bits)
    - CONV_STD_LOGIC_VECTOR (signal/variable, #bits)
    - (The parameter "#bits" can be optional)

Entities & Architectures

- ***Entity*** = hardware module with a unique name
- ***Entity*** declaration specifies the exact ***Input*** and ***Output*** ports and the ***data types*** on those ports
- Optional "***generic***" statements (may include various default values, timing parameters, useful during simulation)

- ***Architecture*** statement = specifies what is inside each Entity, using a ***behavioural***, ***structural*** ***or*** ***dataflow*** description.
- ***Entity*** may have several ARCHITECTURE statements; you can experiment with several ways of specifying an entity, and you can select which design to use in a VHDL simulation;

Process Statement = Sequential VHDL

- Outside a process, VHDL statements execute **in parallel**, and in a **random order** like the hardware that they model, so their order is not important
- Inside a process, VHDL statements execute **sequentially**, so their order is important
- Use a process to generate large Combinational Logic Blocks, or Finite-State-Machines
- Use a process to implement an **algorithm of sequential steps** using VHDL statements; you cannot do this outside a process

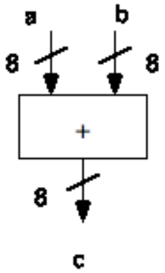
- A **PROCESS** has a list of signals to which it is sensitive = 'the sensitivity list'
- A process is 'activated' (wakes up) whenever any one of the sensitive inputs changes
- A process remains 'asleep' when its sensitive inputs do not change -> efficient simulation
-

Signals and Variables: "<=" versus ":="

- In VHDL, "SIGNALS" are generally "wires" which carry electrical signal values between entities
- "SIGNALS" are used within VHDL structural descriptions as wires joining hardware entities
- When you write to a signal using **<=**, the value appears on the signal instantaneously, and may cause a "wave" of other connected logic and signals to change values (there are exceptions *)
- In VHDL, "VARIABLES" are defined only within architecture processes; they are locally visible and store temporary values used within an algorithm
- Generally, variables are intermediate results within a process not meant to be broadcast out beyond the process:
- When you write to a variable using **:=**, and copy result to an output port, the final value becomes visible on the output port only when the process exits and the clock in incremented
- SIGNALS and VARIABLES behave differently within a process ! This is a common source of errors

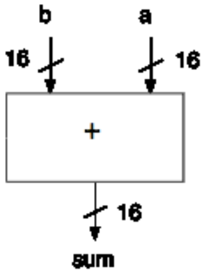
8 bit adder - Behavioural Entity

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_SIGNED.ALL;

entity  ADDER  is
         port (a,b     :  IN   STD_LOGIC_VECTOR(7 downto 0);
                sum   :  OUT STD_LOGIC_VECTOR(7 downto 0));
end Adder ;

architecture  adder_arch1  of  ADDER   is
signal  c1 : SIGNAL STD_LOGIC_VECTOR(7 downto 0);   - this signal is basically a 'wire'
begin
        c 1 <=  a + b;  --  creates an 8 bit adder
         c <= c1;   --   w     hen writing to a signal, use '<='
end adder_arch1 ;

-- or

architecture  adder_arch1  of  ADDER  is
begin
         sum  <=  a + b; -- when writing to an entity port, use '<='

end adder_arch1 ;
```



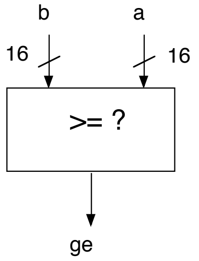
8 bit adder - Behavioural Entity

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.all;

entity adder is
  generic(width : integer := 16);
  port(
    a      : in std_logic_vector(width-1 downto 0)
    b      : in std_logic_vector(width-1 downto 0),
    sum    : out std_logic_vector(width-1 downto 0)
                                 );
end adder;

architecture rtl of adder is
begin

  process (a,b)
  begin
     sum <= a + b;
  end process;

end architecture rtl;
```



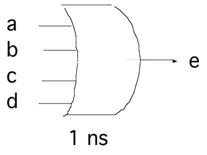
Comparator Entity

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.all;

entity cmp is
  generic(width : integer := 16);
  port(
    a      : in std_logic_vector(width-1 downto 0);
    b      : in std_logic_vector(width-1 downto 0);
    ge     : out std_logic  );

end cmp;

architecture cmp_arch1 of cmp is
begin
  -- set ge when a is greater or equal to b
  -- note: we can’t use ‘if-then-else’ stmt here
   ge <= '1' when (a >= b) else '0';
end architecture cmp_arch1;
```



4 bit OR gate

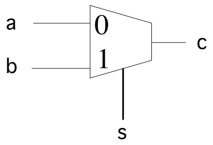
```
entity  OR4  is
        port ( a,c,b,d :  IN   BIT;
                e  : OUT BIT);
end OR4 ;

-- an architecture definition without using a process statement
architecture  OR4_a  of  OR4  is
begin
        -- an architecture body without a process
        if (a = '1') OR (b = '1') OR (c = '1') OR (d = '1') then
                e <= "1'  after  1  ns ;      -- write value directly to an output port
        else
                e <= '0'  after  1  ns ;
        end if ;
end OR4_a ;

-- an architecture definition which uses a process statement for efficient simulation
architecture  OR4_b  of  OR4  is
begin
        process  (a,b,c,d) ;  --  use a process for efficient simulation
        begin
                if (a = '1') OR (b = '1') OR (c = '1') OR (d = '1') then
                        e <= "1'  after  1  ns ;
                else
                        e <= '0'  after  1  ns ;
                end if ;
        end process ;
end OR4_b ;
```



VHDL rule: The ‘If-then-else’ statement must be in a process.

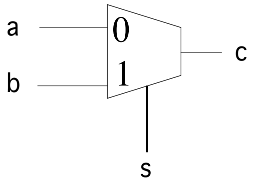
2-to-1 Multiplexer - Behavioural Entity

```
Entity  mux21 is
       Port ( a, b , s :   in   STD_LOGIC;
                      c     :  out      STD_LOGIC ) ;
End  entity  mux21;

-- an architecture definition without using a process statement
Architecture   mux21_arch1  of  mux21   is
begin
       c <= a  when  s = '0'  else   b;
end  mux21_arch;

-- using a process statement to generate combinational logic
Architecture   mux21_arch2  of   mux21  is
begin
       MUX : Process (a, b,  s)
       begin
              if  (s  = '1')  then
                            c <= b;
              else
                            c <= a;
              end
       end process  MUX;
end  architecture  mux21_arch2;
```



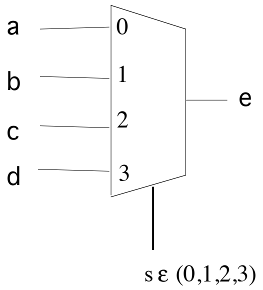
VHDL rule: 'If-then-else' statement must be in a process.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.all;

entity mux21 is
  generic(width : integer := 16);
  port(
    a       : in std_logic_vector(width-1 downto 0);
    b       : in std_logic_vector(width-1 downto 0);
    sel     : in std_logic;
    q       : out std_logic_vector(width-1 downto 0)
                               );
end mux;

architecture mux21_arch1 of mux21 is
begin
  q <= a when (sel = '0') else b;
end architecture rtl;
```



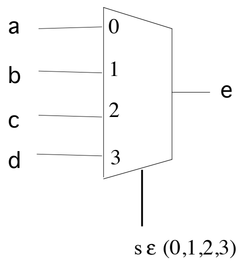
A 4-to-1 MUX - Behavioural Entity

```
entity  mux4  is
        port ( a,c,b,d :       IN   BIT;
                s  :            IN INTEGER RANGE 0 to 3;
                e  :            OUT BIT);
end mux4 ;

architecture  mux4_a  of  mux4  is
begin

        -- the 'with-select' construct usually synthesizes to a multiplexer

        with s select
                e <=  a when 0 ,
                        b when 1,
                        c when 2,
                        d when 3;

end mux4_a ;
```



Use the ‘With-Select’ statement

A 4-to-1 MUX - Behavioural Entity

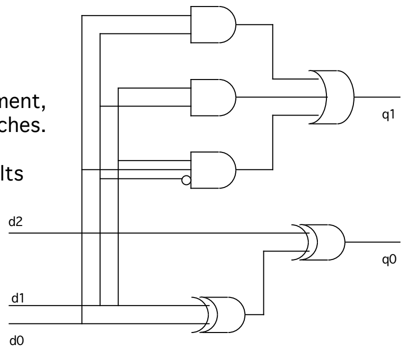


```
architecture mux4_arch1 of mux4 is
begin
        MUX: process(sel,d0,d1,d2,d3) is
        begin
                case sel is
                        when 0 =>
                                z <= d0;
                        when 1 =>
                                z <= d1;
                        when 2 =>
                                z <= d2;
                        when 3 =>
                                z <= d3;
                end case;
        end process MUX;
end architecture mux4_arch1;
```

```
-- introduce propagation delay, page 111 ashenden
architecture mux4_arch1 of mux4 is
begin
        MUX: process(sel,d0,d1,d2,d3) is
        begin
                case sel is
                        when 0 =>
                                z <= d0 after prop_delay;
                        when 1 =>
                                z <= d1 after prop_delay;
                        when 2 =>
                                z <= d2 after prop_delay;
                        when 3 =>
                                z <= d3 after prop_delay;
                end case;
        end process MUX;
end architecture mux4_arch1;
```

An Algorithm in a Process - Count 1’s in a Vector

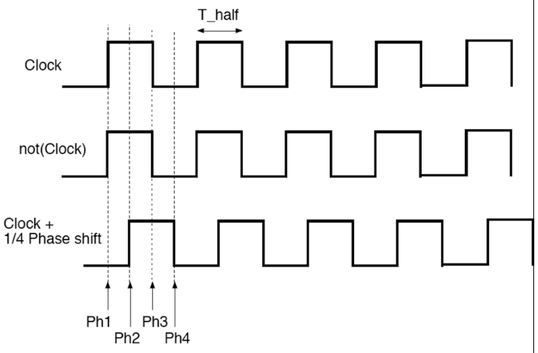
```
ENTITY  count_ones  is
        PORT (d :  IN   BIT_VECTOR (2  downto 0);
                q  : OUT INTEGER RANGE 0 to 3);
END count_ones ;


ARCHITECTURE   count_ones_arch  OF  count_ones  IS
BEGIN
        -- count the number of ones in vector d; the declaration of a
        -- sequential algorithm such as this one requires a process statement,
        -- but whenever you have a process, be careful about inferred latches.
        PROCESS (d)
                -- define a locally visible variable to store   temporary  results
                VARIABLE num_bits : INTEGER ;
        BEGIN
                num _bits := 0;

                FOR i IN  d'RANGE LOOP
                        IF d(i) = '1' THEN
                                num _bits := num_bits + 1;
                        END IF;
                END LOOP;

                q <= num_bits;

        END PROCESS ;
END count_ones_arch ;
```



• Here we use a process to specify an algorithm, which is synthesized to create combinational logic; no inferred memory is created here

Clock Generator Processes

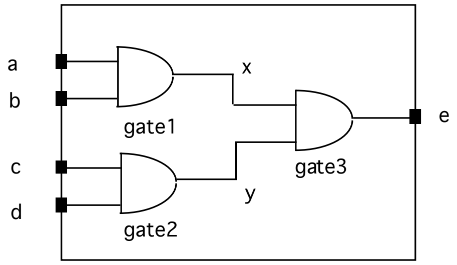
```
-- Ashenden text: clock generator process, page 110
clock_gen: process (clk) is
begin
    if clk = '0' then
        clk <= '1' after T_half, '0' after 2*T_half;
    end if;
end process clock_gen;


-- Ashenden text: clock generator process, page 116
clock_gen: process (clk) is
begin
    clk <= '1' after T_half, '0' after 2*T_half;
    wait for 2*T_half;
end process clock_gen;
```



Structural Descriptions

- VHDL entities can be specified ‘**structurally**’ rather than ‘**behaviourally**’
- Every entity can have a behavioural or structural definition, or both
- The structural description specifies the internal components and how they are interconnected, without specifying any behaviour
- Structural descriptions are useful to describe entities which the Synthesizer cannot synthesize - they are too complex and require too much ‘expert knowledge’
- For example, in the 4DM4 labs, we will build the parts of our CPU behaviourally, and then connect the parts structurally

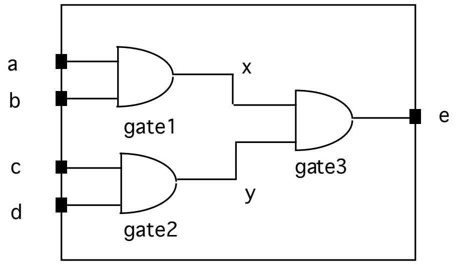
4-Input AND Gate - Structural Entity



Entity = bounding box
I/O ports denotes by boxes

```
LIBRARY    library_name
USE          library_name.package_ name.AND_2

ENTITY  AND_4  IS
PORT (a,b,c,d : IN BIT;
        e : OUT BIT);
END   AND_4


ARCHITECTURE  struct1  of  AND_4  IS

-- define internal signals (wires) between components
SIGNAL  x, y: BIT

-- list types of all components to be used
COMPONENT  AND_2  PORT   (  a, b: IN BIT, c: OUT BIT)
END COMPONENT

BEGIN

       -- instantiate 3 different components of type AND_2
       -- interconnect them using the internal signals and entity I/O ports

       gate1:  AND_2   PORT MAP ( a,b,  x); -- order of connections is important
       gate2:  AND_2   PORT MAP ( c,d,  y);
       gate3:  AND_2   PORT MAP ( x,y,  e);

END struct1;
```

4-Input AND Gate - Structural Entity (b)



Entity = bounding box
I/O ports denotes by boxes

```
ENTITY  AND_4  IS
PORT (a,b,c,d : IN BIT;
        e : OUT BIT);
END   AND_4


ARCHITECTURE  struct1  of  AND_4  IS

-- define internal signals (wires) between components
SIGNAL  x, y: BIT

-- list types of all components to be used
COMPONENT  AND_2  PORT  (     ai, bi : IN BIT, co : OUT BIT)
END COMPONENT

BEGIN

        -- here, we interconnect components using explicit connections
        --  syntax : source => destination  (check this)

        gate1:  AND_2   PORT MAP ( a=>ai,   b=>bi,       co=>x );
                                  -- order of connections is not important, inputs & outputs explicitly shown
        gate2:  AND_2   PORT MAP ( c=>ai,   d=>bi,       co=>y );
        gate3:  AND_2   PORT MAP ( x=>ai,   y=>bi,       co=>e );

END struct1;
```

Signals and Variables: “<=“ versus “:=“

- In VHDL, “SIGNALS” are generally “wires” which carry electrical signal values between entities
- “SIGNALS” are often used within VHDL structural descriptions as wires joining separate hardware entities
- When you write a new value to a signal using **<=**, it appears on that signal instantaneously, and may cause a “wave” of other connected gates and signals to change values
- In VHDL, “VARIABLES” are defined only within architecture processes; they are locally visible and store temporary values used within a sequential algorithmic description of an entity
- Generally, variables are intermediate results within a process not meant to be broadcast out beyond the process
- When you write a new value to a variable using **:=**, it becomes visible outside the process only once the process exits and the clock is incremented !!
- SIGNALS and VARIABLES behave differently within a process ! This is a common source of errors

Processes and Clock Edges

- Combinational Logic versus Finite State Machines (FSM)
- process is NOT sensitive to rising or falling edges of a signal, => it will usually synthesize to combinational logic (with no memory elements)
- ie the process contains an algorithmic description of a complex function to be performed in combinational logic, and the synthesis engine will attempt to create the optimized combinational logic (ie using karnaugh maps, etc)

- If the process IS sensitive to rising or falling edges of any signal, the process will usually synthesize to a FSM with inferred memory elements
- ie the process contains an algorithmic description of a FSM, and the synthesis engine will attempt to create the optimized FSM
- How does the synthesis engine know when to create a latch / memory ?
- The existance or necessity of Memory (latches, D Flip-flops, registers) is implied in your VHDL statements: this is called "inferring memory"

Inferring Registers with Signals

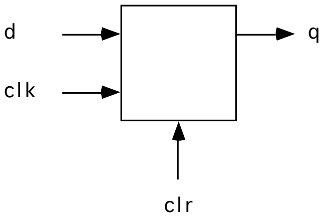
- Writing to a SIGNAL (using <=) within a process triggered by a clock event will creat an "inferred" register/latch/D flip flop
- When Simulating your VHDL: the signal assumes the written value **only upon exit** of the process and after simulated time has been incremented; subsequent VHDL statements executed within the process see the original signal value, until the process is exitted and time is incremented !

- With ALTERA's synthesis engine, registers may be inferred in these cases :
- Within a process triggered by a clock event
- Within any process with a WAIT statement
- By incompletely specified mappings ie (when you use a "with .. select" statement, but do not specify all the possible cases)

Inferring Registers with Variables

- Writing to a VARIABLE (using **:=**) within a process triggered by a clock event also creates an 'inferred" register/latch/D Flip-flop
- When Simulating your VHDL: the variable assumes the written value **right away,** so that you can work with it; subsequent VHDL statements in the same VHDL process will see the new value right away
- With ALTERA's synthesis engine, registers may be inferred in these cases:
- Within a process triggered by a clock event
- Within any process with a WAIT statement
- By incompletely specified mappings.

Registers and Latches - Behavioural Models



```
ENTITY  dff  IS
PORT (d,  clk,  clr :  IN BIT;
        q:  OUT BIT);
END dff;

-- edge triggered  dff register with active-high clock
-- and asynchronous clear

ARCHITECTURE  arch1  OF   dff  IS
BEGIN
        PROCESS (clr,  clk)    -- process is sensitive to a clock
        BEGIN
                IF clr = '1' THEN
                        q <= '0';
                ELSIF  clk'EVENT  AND   clk = '1' THEN
                        q <= d;
                ENDIF;

        -- these statements infer a memory element, since the assignment to 'q'
        -- occurs within a process sensitive to  clk'EVENT; According to the VHDL,
        -- input 'd' can change asynchronously, yet the code requires that 'q'
        --  only changes on rising clock edges, so a memory element is inferred

        END PROCESS;
END arch1;
```

Synthesizing Registers

- The above D flop-flop RTL code can be easily modified to synthesize variations:

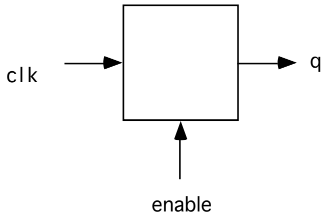
    - register with active-high clock, register with active-high clock and asynchronous clear, register with active-low clock and asynchronous clear, etc.

    - A designer can use any of these registers in a design; there is no need to design at a lower level, since the synthesis engine will perform the structural design and optimization.

- You can also use predefined descriptions of latches/registers in the vendor's library, but it is just as easy to define your own

Counters - Behavioural Models

```
ENTITY enable_ counter  IS
    PORT(clk,  enable,  load  : IN BIT;
              d : IN   INTEGER RANGE 0 to 255;
              q : OUT  INTEGER RANGE  0 TO 255);
END enable_counter;

- a synchronous,  loadable  up-counter

ARCHITECTURE   arch1   OF   enable_ counter  IS
BEGIN
    PROCESS(clk)
       VARIABLE count:  INTEGER RANGE 0 TO 255;
    BEGIN

    -- the  assignment to variable  'count'  within  a process infers one  register
    IF (clk'EVENT  AND  clk  = '1') THEN
       IF load  = '1'  THEN
              count  := d;
       ELSE
              IF enable  = '1' THEN
                     count  := count  + 1;
              END IF;
      END IF;
    END IF;

    q <= count;    -- assign count  to output  port --
    END PROCESS;
END arch1;
```



Synthesizing Counters

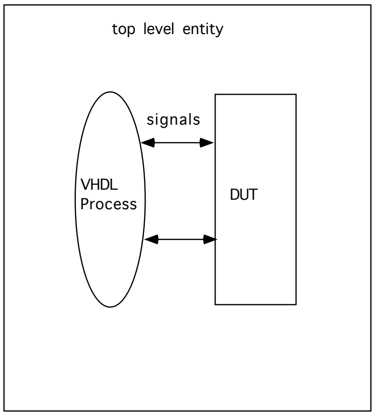
- The Synthesis tool will synthesize an appropriate counter given behavioural code.
- The above RTL code can be easily modified to synthesize variations:

  enabled counter, synchronous load counter, synchronous clear counter, up/down counter, etc.
- A designer can use the RTL versions of these counters in a design; there is no need to design at a lower level, since the synthesis engine will perform the structural design.

Common Error 1: Inferred Memory by Incomplete Specifications

```
architecture  OR4_a  of  OR4  is
begin
        if (a = '1') OR (b = '1') OR (c = '1') OR (d = '1') then
                e <= "1'  after  1  ns ;
        -- else        -- an incomplete specification
        --        e <= '0'  after  1  ns ;
        end if ;

        -- suppose we incompletely specify 'e'; a memory element may be inferred
        -- by default, VDHL assumes 'e' remains unchanged, which necessitates memory

end OR4_a ;


architecture  mux4_a  of  mux4  is
begin

        -- the 'with-select' construct usually synthesizes to a multiplexer
        with s select
                e <=  a when 0,
                        b when 1,
                        c when 2;
                        -- d when 3;  -- this line is omitted, so we have an incomplete specification
                        -- this incompletely specified multiplexer will infer memory
                        -- since by default e must remain unchanged when s = 3
end mux4_a ;
```

Testing Your Design - Test Benches



- Design a top-level entity with no external I/O signals to test a device
- Consists of a <u>dataflow</u> (mixed behavioral & structural) representation
- “Device Under Test” (DUT) will be synthesized onto the FPGA, and interconnected to stimulus signals from TestBench VHDL process
- Stimulus signals controlled by Testbench behavioral VHDL process, to sequentially apply the stimulus as needed (recall processes execute sequentially)

Test Bench for mux41

```
Library ieee;
Use ieee.std_logic_1164.all;
Use work.all;  -- contains definition of mux41

-- top level entity with no I/O signals
Entity test_mux_entity is
End  test_mux_entity;

Architecture   test_mux   of   test_mux_entity  is
Begin

-- declare DUT
component mux41
          port (a,b,c,d : IN STD_LOGIC_VECTOR(7 downto 0);
                 s :  in integer range 0 to 3;
                 z : OUT STD_LOGIC_VECTOR(7 downto 0);
end component;

-- declare signals to connect to DUT
signal w1, w2, w3, w4, y: STD_LOGIC_VECTOR(7 downto 0);
signal  select : INTEGER range 0 to 3;

begin

     -- connect signals to the DUT  IO ports; use explicit mapping in this example
     -- note the order: component port names map to signals
DUT: mux41 port map(a->w1,b->w2,c->w3,d->w4,s->select,z->y);
```

Test Bench for mux41

```
        -- create a behavioral process to generate the stimuli
        -- “wait for” construct seems to work in processes without a sensitivity list
        -- ALTERA may or may not support the “wait’ statement;
        -- their CAD system is updated several times a year : check this code first
        --
waveform: process is
            constant interval : time := 25 ns;
            begin
            w1 <= '00010001";
            w2 <= "00100010";
            w3 <= "00110011";
            w4 <= "01000100";

            select <= 0;
            wait for interval;  -- Altera  may or may not support the ‘wait’ construct
            select <= 1;
            wait for interval;
            select <= 2;
            wait for interval;
            select <= 3;
            wait;    -- waits forever, test ends
end process waveform;

end architecture test_mux;
```

Test Bench for mux41

```
        -- create a behavioral process to generate the stimuli
        -- if the ‘wait’ statement does not work in Altera, try this
        --
waveform: process is
            constant interval : time := 25 ns;
            begin
            w1 <= '00010001";
            w2 <= "00100010";
            w3 <= "00110011";
            w4 <= "01000100";

            select <= 0;

            select <= 1 after 25 nsec;

            select <= 2 after 25 nsec;

            select <= 3 after 25 nsec;

            wait;    -- waits forever, test ends
end process waveform;

end architecture test_mux;
```

Library of Parameterized Modules (LPMs)

- Vendors often supply VHDL descriptions for common modules.
- These usually synthesize optimally.
- Some typical parameterized modules you might use include:

Description

parameterized tri-state buffers

parameterized counters, adders, subtractors

parameterized dual ported , multi-ported RAM

parameterized multipliers, DSP functions

- Altera has several RAM LPMs that you use: synchronous or asynchronous RAM, ROM, dual-ported RAM, etc

Example: Altera 256x8 RAM LPM

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
LIBRARY lpm;                         -- invoke Altera's lpm library
USE lpm.lpm_components.All;
LIBRARY WORK;
USE work.ram_constants.All;          -- your own package of constants

ENTITY my_ram256x8 IS
        PORT (data: IN STD_LOGIC_VECTOR(DATA_WIDTH-1 downto 0);
                address: IN STD_LOGIC_VECTOR(ADDR_WIDTH-1 downto 0);
                we, inclock, outclock : IN STD_LOGIC;
                q : OUTSTD_LOGIC_VECTOR(DATA_WIDTH-1 downto 0));
END my_ram256x8;

-- see the ALTERA manuals (online or in the lab) for details
ARCHITECTURE my_ram256x8_instance OF my_ram256x8 IS
BEGIN
        -- create one component instance of type LPM_RAM_DQ
        inst_1: LPM_RAM_DQ
        -- pass your own parameters to fix the address & data bus widths
        GENERIC_MAP (     lpm_widthad => ADDR_WIDTH,
                          lpm_width  => DATA_WIDTH);
        -- connect your own entity IO ports to the ports of the LPM RAM entity
        PORT MAP (data => data, address => address, we => we,
                        inclock => inclock, outclock => outclock, q => q);
END my_ram256x8_instance;
```

Megafunctions and Cores

- Vendors usually license / sell more complex “megafunctions” including:

  Description

  - phase-locked loops
  - NTSC video control signal generator (for TV)
  - Universal Asynchronous Receiver/Transmitters
  - Programmable DMA Controllers
  - Ethernet / network adaptors

- These megafunctions are vendor-specific. They are usually not parameterizable, and are more complex than basic lpm functions.

- ARM licenses the ARM microprocessor core, which can be synthesized on most FPGAs

Summary

- Introduction to IEEE Standard VHDL - “**V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage”
- Libraries, Packages, Entities, Architectures, Configurations
- Behavioral vs. Structural Descriptions in VHDL
- Synthesis of hardware from VHDL behavioural descriptions
- Libraries of Parameterized Modules, Megafunctions & Cores
- Introduction to test benches
- For details, see the textbook, “Fundamentals of Digital Logic, with VHDL Design”, Brown and Vranesic
- Hopefully, you will now be be able to develop moderately complex hardware designs using VHDL