

Types, Resolution Functions, and Testbeds

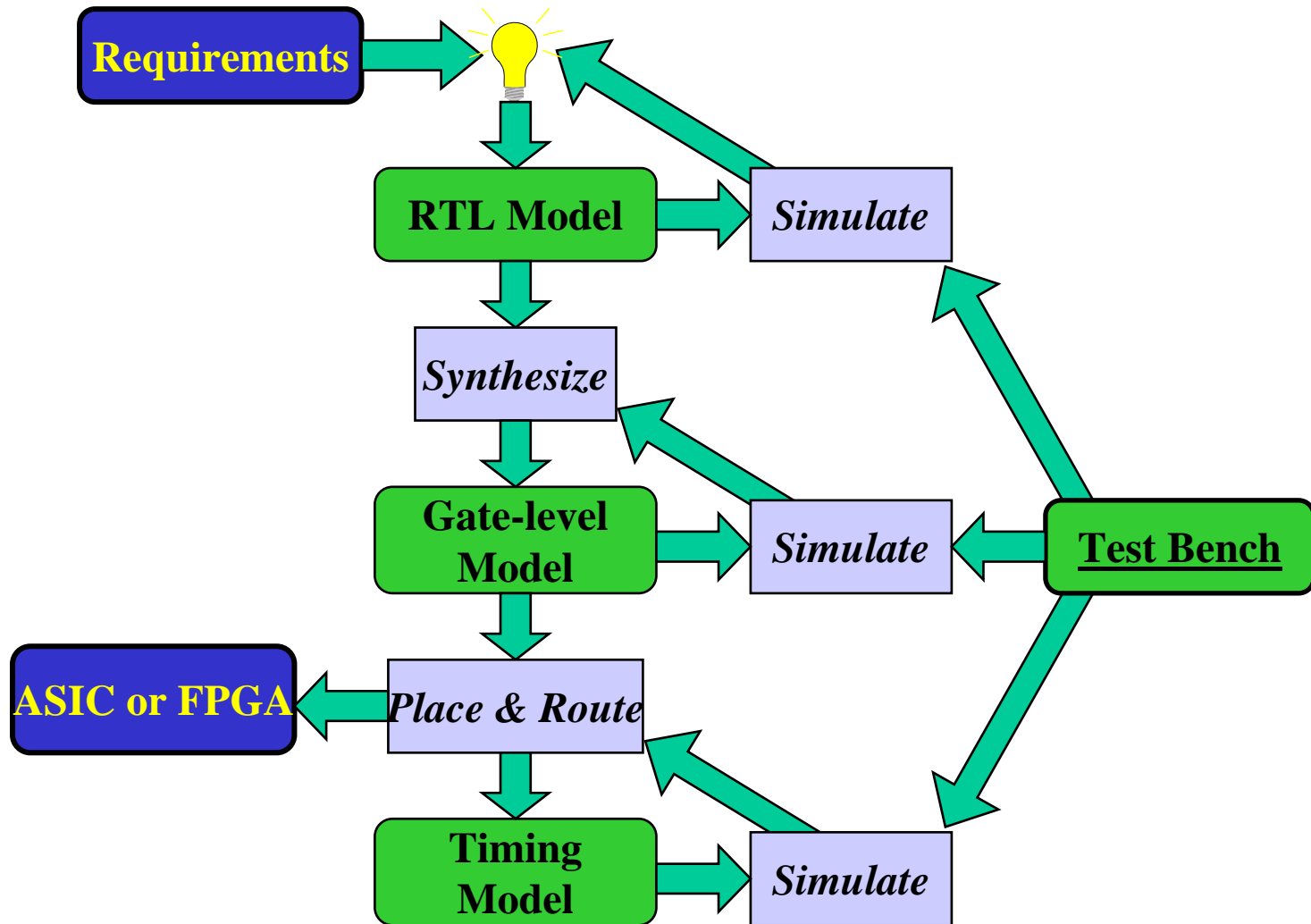
Ashenden:

Chapter 4 "Composite Data Types and Operations"

Chapter 5 "Basic Modeling Constructs"

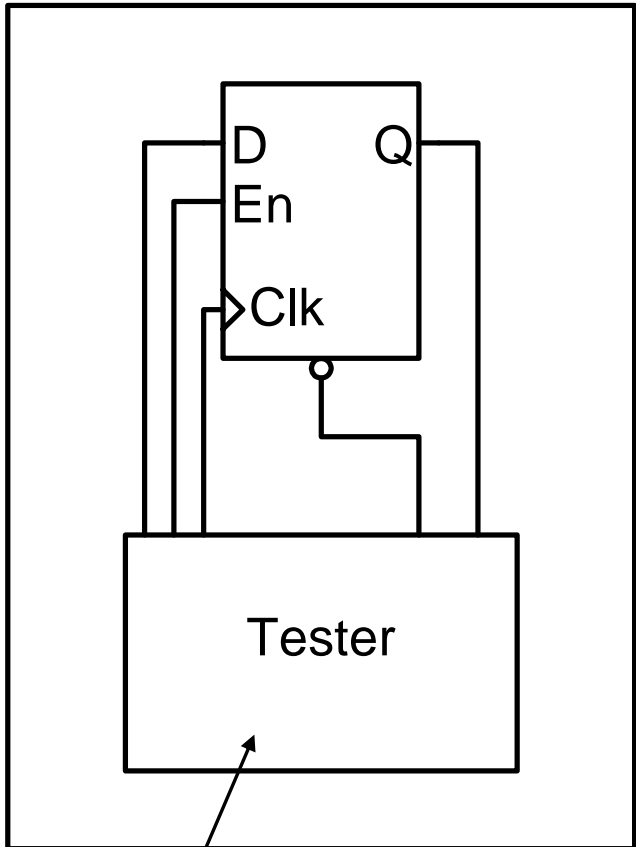
Chapter 11 "Resolved Signals"

How Does the Testbench Fit In?



Simple Testbed Example

Testbench



In this simple case "tester" will be a set of process statements in the testbench architecture.

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

entity dff_tb is
end entity dff_tb;

architecture model of dff_tb is
  --components
  component dff is
    port ( d      : in std_logic;
          clk     : in std_logic;
          resetn  : in std_logic;
          enable  : in std_logic;
          q       : out std_logic );
  end component dff;
  --Signals
  signal clk      : std_logic := '0';
  signal resetn  : std_logic := '0';
  signal enable  : std_logic := '0';
  signal d       : std_logic;
  signal q       : std_logic;
begin
  .
  .
  .
end architecture model;
```

Simple Testbed Example (2)

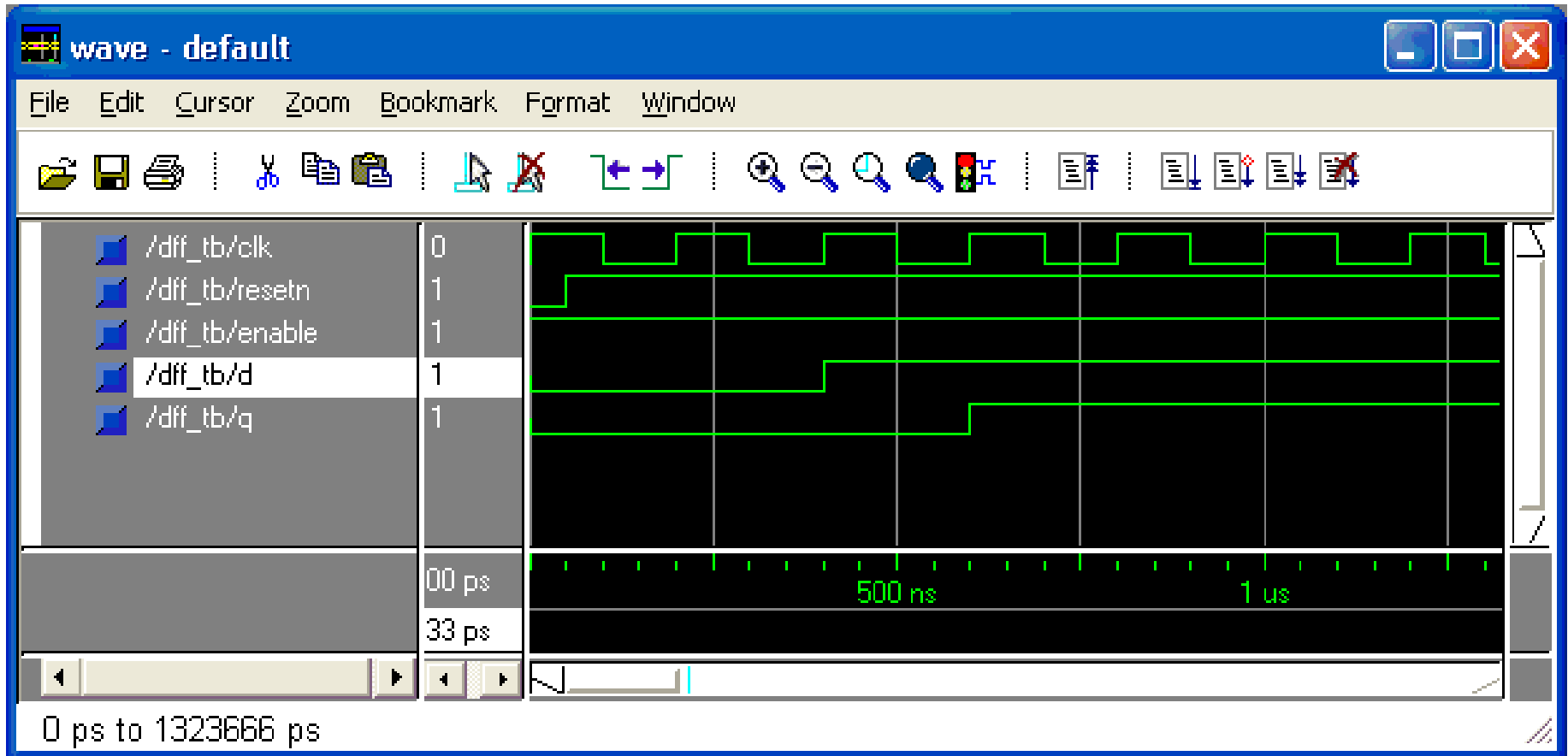
```
begin
  dut : dff port map
    ( d => d, clk => clk, resetn => resetn,
      enable => enable, q => q );
  gen_clk : process is
  begin
    clk <= '1';
    wait for 100 ns;
    clk <= '0';
    wait for 100 ns;
  end process gen_clk;
  set_resetn : process is
  begin
    resetn <= '0';
    wait until clk'event and clk = '1';
    wait for 50 ns;
    resetn <= '1';
    wait;
  end process set_resetn;
  a_test : process is
  begin
    d <= '0';
    wait until clk'event and clk = '1';
    wait until clk'event and clk = '1';
    wait until clk'event and clk = '1';
    d <= '1';
    wait until clk'event and clk = '1';
    --will this give an error?
    assert '1' = q
      report "ERROR, D did not propogate to Q."
      severity error;
    wait;
  end process a_test;
end architecture model;
```

Create a clock

Toggle the resetn

We can **assert** certain conditions to be true, simplifying the job of inspecting the waveforms.

Simple Testbed Example (3)



Q: From the waveform and the previous code, was the assert thrown?

A: No...why?

test.do (test script for Modelsim)

```
# Create the library and analyze the VHDL
```

```
vlib work
```

```
vcom -explicit -93 "dff.vhd"
```

```
vcom -explicit -93 "test.vhd"
```

```
# Load and start the simulation
```

```
# To restart without re-analyzing, etc., just type "restart -f"
```

```
vsim -t 1ps -lib work test_vhd
```

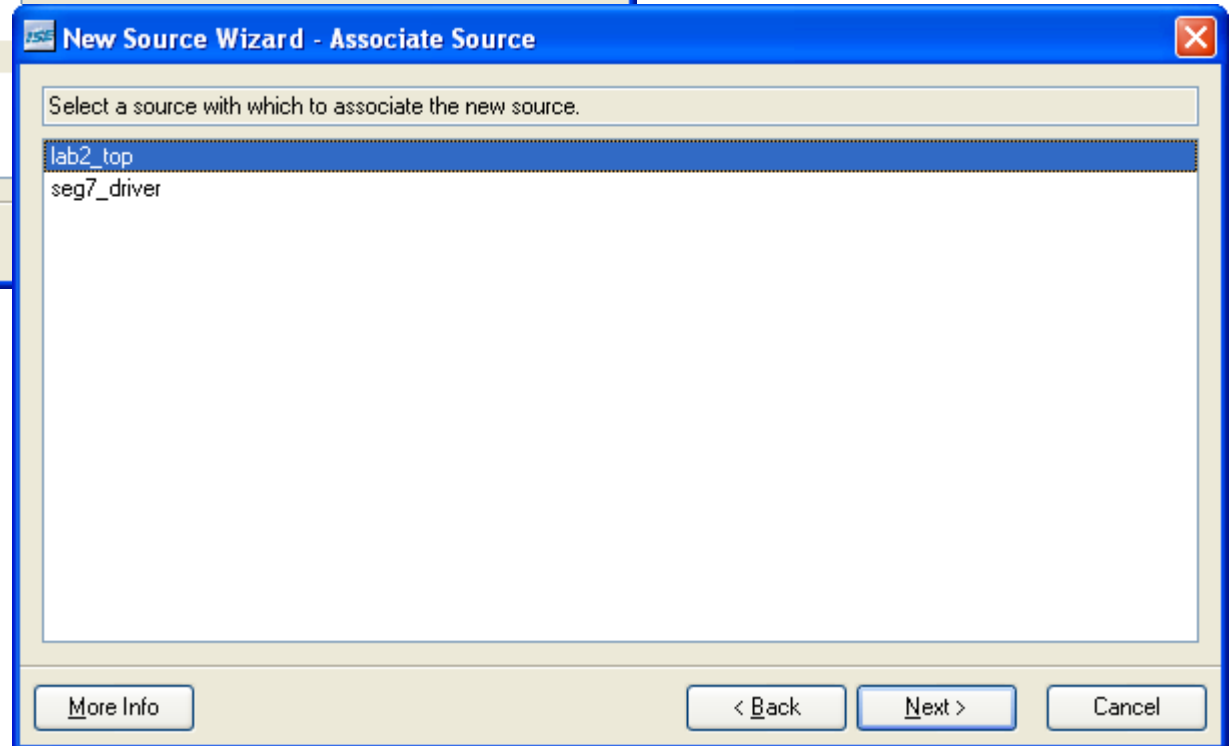
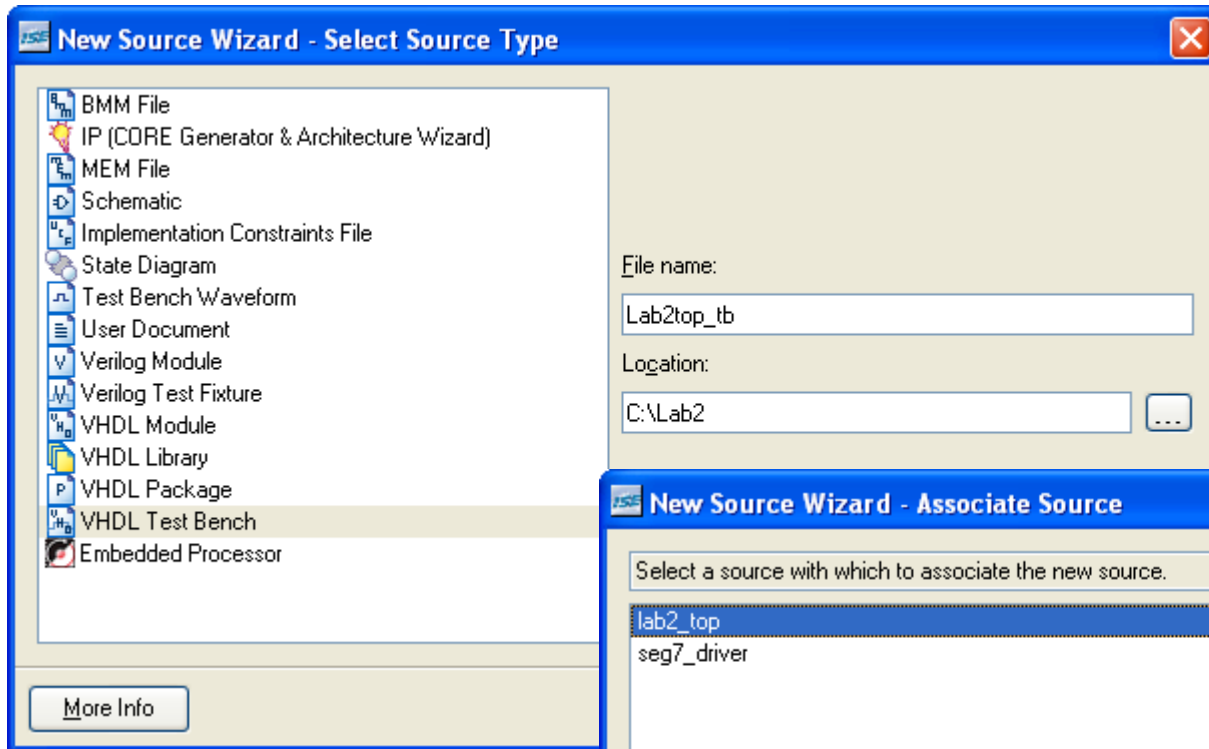
```
do wave.do
```

```
view structure
```

```
view signals
```

```
# to run, type "run" followed by the amount of time to run  
  (remember the units, ns, ps, etc.)
```

Testbed Creation in ISE



File Created Automatically

```
ENTITY lab2top_tb IS
END lab2top_tb;

ARCHITECTURE behavior OF lab2top_tb IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT lab2_top
    PORT(
        clk50 : IN  std_logic;
        rst   : IN  std_logic;
        sliderSwitches : IN  std_logic_vector(3 downto 0);
        Seg7  : OUT std_logic_vector(6 downto 0);
        anodes : OUT std_logic_vector(3 downto 0)
    );
    END COMPONENT;

    --Inputs
    signal clk50 : std_logic := '0';
    signal rst   : std_logic := '0';
    signal sliderSwitches : std_logic_vector(3 downto 0) := (others => '0');

        --Outputs
    signal Seg7 : std_logic_vector(6 downto 0);
    signal anodes : std_logic_vector(3 downto 0);

    -- Clock period definitions
    constant clk50_period : time := 1us;

BEGIN

    -- Instantiate the Unit Under Test (UUT)
    uut: lab2_top PORT MAP (
        clk50 => clk50,
        rst   => rst,
        sliderSwitches => sliderSwitches,
        Seg7  => Seg7,
        anodes => anodes
    );
```

File Created Automatically (cont)

```
-- Clock process definitions
clk50_process :process
begin
    clk50 <= '0';
    wait for clk50_period/2;
    clk50 <= '1';
    wait for clk50_period/2;

end process;

-- Stimulus process
stim_proc: process
begin
    -- hold reset state for 100ms.
    wait for 100ms;

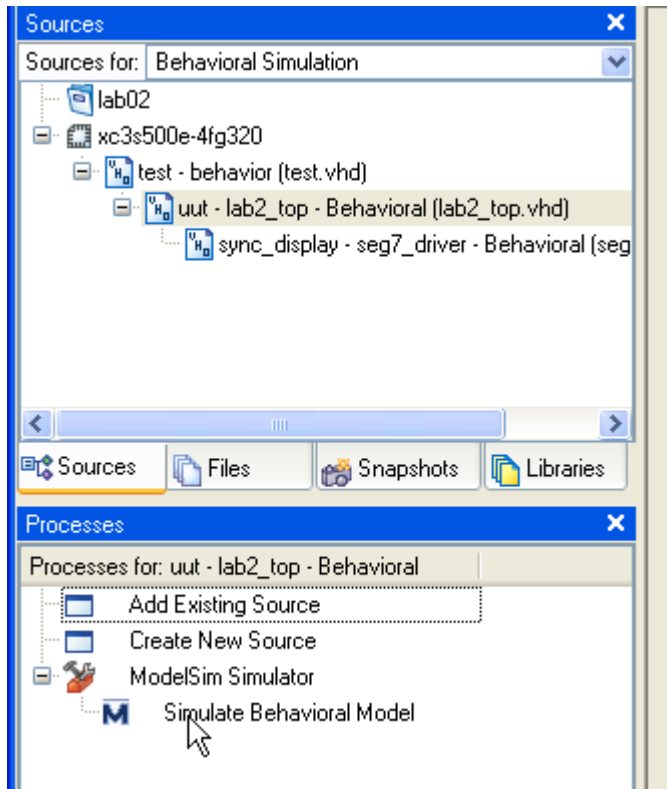
    wait for clk50_period*10;

    -- insert stimulus here

    wait;
end process;

END;
```

Simulating with Testbench



Runs modelsim with a “.do” file

```
## NOTE: Do not edit this file.  
## Autogenerated by ProjNav (creatfdo.tcl) on ...  
##  
vlib work  
vcom -explicit -93 "seg7_driver.vhd"  
vcom -explicit -93 "lab2_top.vhd"  
vsim -t lps -lib work lab2_top  
do {lab2_top_wave.fdo}  
view wave  
view structure  
view signals  
run 1000ns  
do {lab2_top.udo}
```

All you have to do is “run”

Summary

- No excuse not to simulate (for almost all of the exercises in the class)
 - At least visually inspect the results, auto-verifying answers is not always necessary
- Using a testbench will make this simulation **much** less painful
- When asking for help, you can point the instructors to a particular time in the simulation where something doesn't happen as you expect it to.

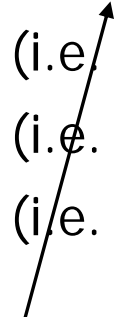
Types

- VHDL is a *strongly typed* language
 - Strongly typed means “every object may only assume values of its nominated type”
 - A language is strongly typed to **allow detection of errors at an early stage** of the design process, namely, when a model is analyzed (compiled).
 - Q: Why would a language not be strongly typed?
 - Each data flow (input, output, internal signal, etc.) has a type associated with it, and there can be no ambiguity
 - Must convert between types, even those that are very similar (somewhat analogous idea “casting”)
- In VHDL, there are some built-in types...
 - The “standard” types
- ...but many (even most) of what we use are additional types
 - Mainly `std_logic` and `std_logic_vector`


But, Before More Types...Objects

- To clarify vocabulary, an *object* is a named item in a VHDL model that has a value of a specified type."
- There are four (4) classes of objects:
 1. Constants (i.e. **constant** PERIOD : **time** := 100 ns;)
 2. Variables (i.e. **variable** cntInt : **integer**;
 3. Signals (i.e. **signal** notAClk : **bit**;))
 4. Files (i.e. **file** notCoveringTheseToday : **text**;))

I am an object



I am a type



Eight Type Classes Available

Type Class	Synthesizable?	Example
•Enumeration types	yes	std_ulogic
•Integer types	yes	integer
•Floating point types	no (only for calc)	real
•Physical Types	no	time
•Array Types	yes	std_logic_vector
•Record Types	yes	[user defined]
•Access Types	no	[a pointer to a type]
•File Types	a tiny bit	text

Standard Types and std_logic types

- There are a number of predefined VHDL types.
 - They are available in the library std.standard.all and the library std.textio.all
 - *Never* includes these at the top of a model (i.e. library std; use std.standard.all) as it may confuse the compiler
- In addition to these standard types, there are also standard types that have been added to the language to support the modelling of logic gates.
- These types are commonly known as the *std_logic* types
 - These are available in the library and package ieee.std_logic_1164
 - **These *do* need to be included** (Webpack includes them at the top of each new file automatically)

Standard and std_logic types (2)

Type	Type Class	Synthesizable
•Boolean	enumeration type	yes
•Bit	enumeration type	yes
•Character	enumeration type	yes*
•Severity_level	enumeration type	yes, but don't!
•Integer	integer type	yes
•Natural	subtype of integer	yes
•Positive	subtype of integer	yes
•Real	floating-point type	no
•Time	physical type	no
•String	array of character	yes
•Bit_vector	array of bit	yes
<hr/>		
•Std_ulogic	enumeration type	yes
•Std_logic	subtype of std_ulogic	yes
•Std_ulogic_vector	array of std_ulogic	yes
•Std_logic_vector	array of std_logic	yes

Operators Available

- Values, signals, and variables can be combined into expressions using *operators*.
- The operators available fall into five (5) categories:
 1. Boolean: not, and, or, nand, nor, xor, *xnor*
 2. Comparison: =, /=, <, <=, >, >=
 3. Shifting: *sll*, *srl*, *sla*, *sra*, *rol*, *ror*
 4. Arithmetic: sign +, sign -, abs, +, -, *, /, mod, rem, **
 5. Concatenation: &
- Note that the items in italics were not available in VHDL –87, while the items in red may not be fully synthesizable

Bit and Boolean

- Type **bit** is the built-in logical type:
 - **type bit is** ('0', '1');
 - This definition is known as an enumeration type
- Both the boolean and comparison operators apply to bit
- **Boolean** is the comparison type in VHDL
 - All comparison operators result in a boolean value, it is not possible to use any other type in a conditional test
 - Boolean is synthesizable, but does not make sense due to the availability of `std_logic`, `std_logic_vector`
 - Boolean is useful in modeling and testbenches, where it can represent what it is – whether something is true or false.
 - i.e.: **constant** use_mem : boolean := **true**;

Integer Types

- The type **integer** is the built-in numeric type which represents integral values
 - A VHDL implementation must allow an integer range that is $[-2147583647, 2147583647]$, this is the range for a 32-bit 1's complement
 - Most implementations presume a 2's complement representation, the range is the $[-2147583648, 2147583647]$
- All of the comparison and arithmetic operators are available for modeling
 - The comparison operators are synthesizable
 - However only the *sign +*, *sign -*, *abs*, *+*, *-*, and *** operators are fully synthesizable
- Integers in VHDL do not wrap:
 - If *x* is of type integer, and $x = 2147483647$, the assignment $y \leq x + 1$ would produce an error during simulation, not the value -2146483648 .
- Integer types are often (generally) used as an index

User-Defined Integer Types vs. Integer Subtypes

or

Examples of Strong Typing

- It is possible to define user defined integer types:
 - **type short is range -128 to 127;**
- Since VHDL is strongly typed, these types can not be copied to other integer types.
- Another way to create a new “type” would be to create a sub-type.
 - A subtype is a restricted range of a type
 - **subtype short is integer range -128 to 127**
- Two subtypes are already defined:
 - **subtype natural is integer range 0 to integer'high**
 - **subtype positive is integer range 1 to interger'high**

```
.
.
.
type short1 is range -128 to 127;
subtype short2 is integer range -128 to 127

signal a_natural : natural := 10;
signal a_integer : integer := 20;
signal a_short1  : short1  := 30;
signal a_short2  : short2  := 40;

begin

    a_integer <= a_natural; --fine
    a_natural <= a_integer; --fine
    a_short1  <= a_natural; --wrong!
    a_short2  <= a_natural; --fine
    a_short1  <= a_integer; --wrong!
    a_short2  <= a_integer; --fine
    a_integer <= a_short1; --wrong!
    a_integer <= a_short2; --fine

    --very interesting
    a_short2 <= a_short1; --wrong!

    --however
    a_integer <= integer(a_short1);
    a_short2 <= short2(a_short1);
    a_short1 <= short1(a_short2);
```

Type conversion function that is automatically available for “closely related types”. Note that is is automatically defined for short1 and short2

Integer Types, Synthesis

- An integer is converted to a bus, with the size of the bus dependent on the range of the integer
 - The number of “wires” will be the number necessary to fully represent the range of the integer type
 - Therefore, to set the size of the bus, one can use the **subtype** to create an integer the correct size
- The synthesizer will determine if the integer type requires an unsigned representation or a 2's complement representation.
 - It is not always clear whether a synthesizer will choose signed or unsigned representations
 - If a calculation uses intermediate values, then the synthesizer may choose to use a signed representation, even if an unsigned integer signal is acted upon and assigned to another unsigned integer signal.
 - For example, a calculation $w \leq (x - y) + z$, where $y > x$, but $|x-y| < z$ would create a negative intermediate value as the result of the $(x-y)$ operation (this example presumes that w , x , y , and z are all unsigned integers).

A student example : use of integers in Lab2

```
proc_clock: process(clk50, rst)
    variable count : integer range 0 to 50000000;
begin
    if (rst = '1') then
        clk1Hz_en <= '0';
    elsif (rising_edge(clk50)) then
        if(count = 50000000) then
            clk1Hz_en <= '1';
            count := 0;
        else
            clk1Hz_en <= '0';
            count := count + 1;
        end if;
    end if;
end process proc_clock;
```

Integer Types, Example

```

library ieee;
use ieee.std_logic_1164.all;

entity int_example is
  port ( clk, resetn : in std_logic;
        whatToDo : in std_logic_vector(1 downto 0);
        num1, num2 : in integer;
        outNum : out natural );
end entity int_example;

```

```

architecture rtl of int_example is
begin
  process ( clk, resetn )
  begin
    if resetn = '0' then
      outNum <= 0;
    elsif clk'event and '1' = clk then
      if whatToDo = "00" then
        outNum <= num1*num2;
      --can't do this, divide can only shift
      --
      elsif whatToDo = "01" then
        outNum <= num1/num2;
      elsif whatToDo = "10" then
        outNum <= num1+num2;
      else
        outNum <= 0;
      end if;
    end if;
  end process;
end architecture rtl;

```

Resource Usage Report for int_example

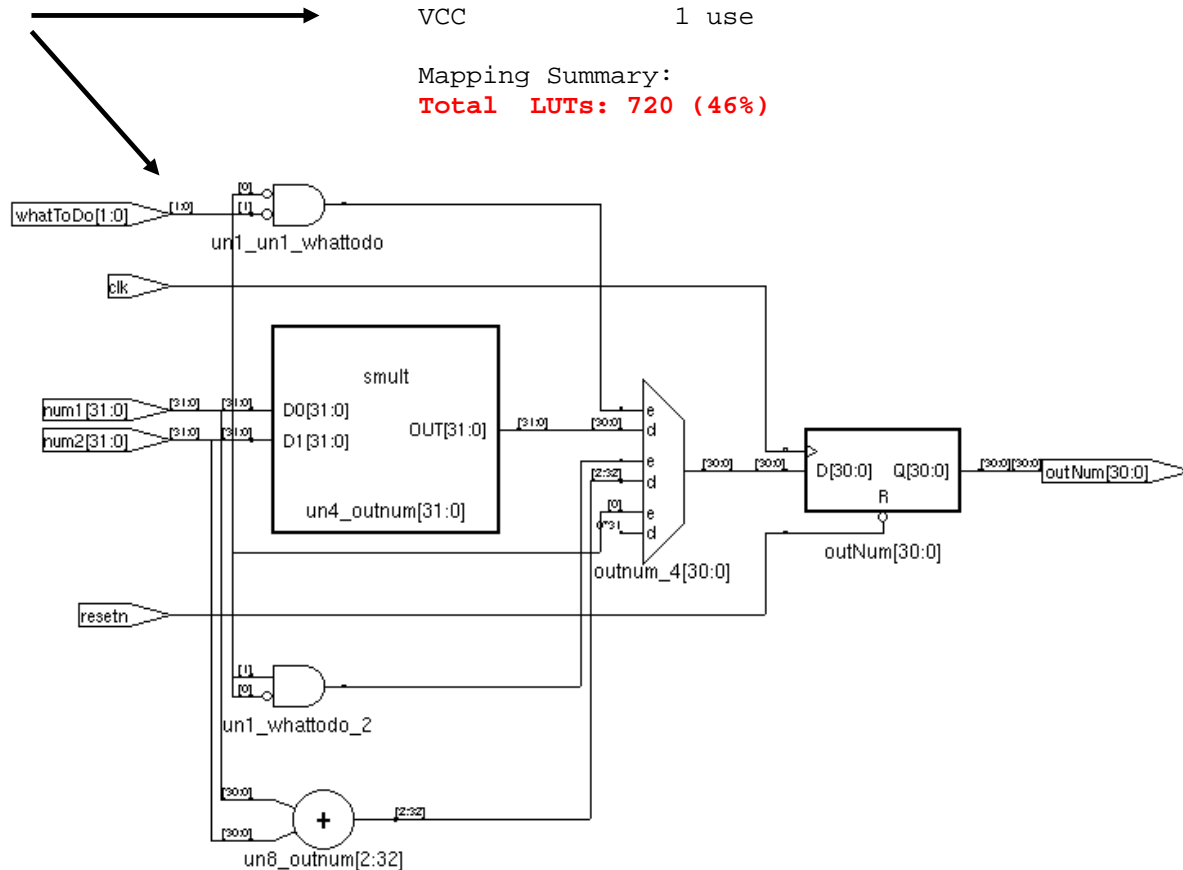
Mapping to part: xc2s50etq144-7

Cell usage:

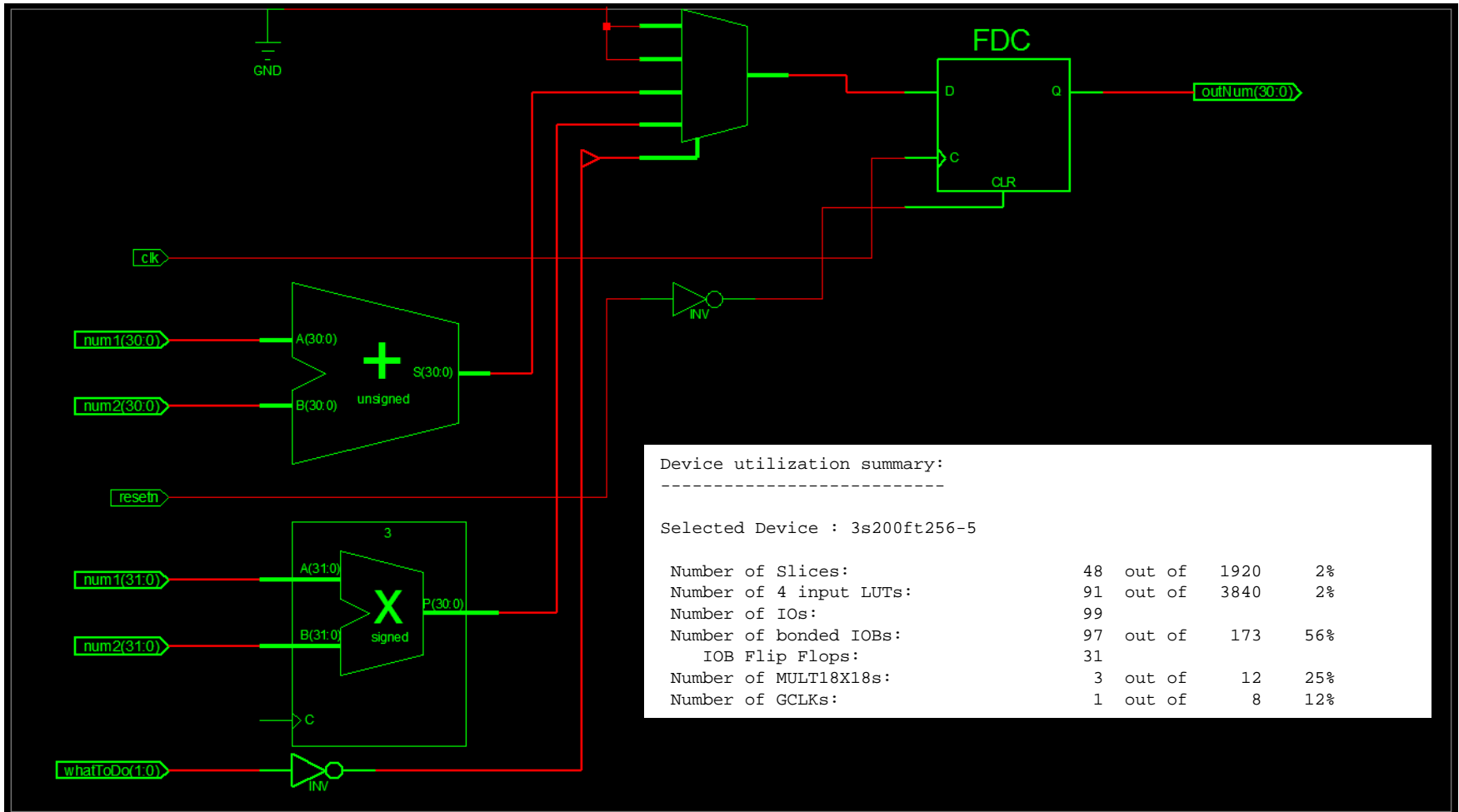
MUXCY_L	465 uses
XORCY	465 uses
MULT_AND	55 uses
FDC	31 uses
GND	1 use
VCC	1 use

Mapping Summary:

Total LUTs: 720 (46%)



Spartan-3



Device utilization summary:

Selected Device : 3s200ft256-5

Number of Slices:	48	out of	1920	2%
Number of 4 input LUTs:	91	out of	3840	2%
Number of IOs:	99			
Number of bonded IOBs:	97	out of	173	56%
IOB Flip Flops:	31			
Number of MULT18X18s:	3	out of	12	25%
Number of GCLKs:	1	out of	8	12%

Enumerated Types

- An **enumerated type** is a type composed of a set of literal values or character literals
- For example, a type of literal values would be:
 - **type** readStatesType **is** (grab_bus, strobe_read, read, release_bus);
- In contrast, a type of character values would be:
 - **type** tertiaryLogic **is** ('X', '?', '0', '1', '2')
- An enumerated type can also contain a combination of literals and character literals
 - **type** I_make_no_sense **is** ('A', B, 'C', D)
 - The one exception is the definition of the standard type **character**, which contains literals to represent control characters (for example, **HT** represents the horizontal tab character)

Enumerated Types (2)

- An enumerated type has the comparison operators already defined
 - This is because each symbol is assigned a *position number*
 - For example, the position numbers for our type tertiaryLogic would be 'x' = 0, '?' = 1, '0' = 2, '1' = 3, '2' = 4
 - The first (or leftmost) literal in the type is regarded as the smallest value, and the last (or rightmost) is regarded as the largest. For example:
 - 'x' > '0' evaluates to *false*
 - '2' > '0' evaluates to *true*
- It is possible to define the other operators by writing *overloaded functions*

Enumerated Type In Use

```
architecture state_machine of pulseGen is
    type states is (start, waitForSig, found);
    signal current_state, next_state : states;
begin
    state_logic : process(current_state, sig) is
    begin
        case current_state is
            when start =>
                result <= '0';
                if sig = (not edge) then
                    next_state <= waitForSig;
                else
                    next_state <= start;
                end if;
            when waitForSig =>
                result <= '0';
                if sig = (not edge) then
                    next_state <= waitForSig;
                else --we have had a transition
                    next_state <= found;
                end if;
            when found =>
                next_state <= start;
                result <= '0';
            when others =>
                next_state <= start;
        end case;
    end process;

    state_register : process(clk, resetn)
    begin
        if resetn = '0' then
            current_state <= start;
        elsif clk'event and clk = clkEdge then
            current_state <= next_state;
        end if;
    end process;
end architecture state_machine;
```

Arrays

- An array is a collection of elements, all of which are the same type.
- Arrays can either be *constrained* or *unconstrained*. For example:
 - An unconstrained array definition:
type std_logic_vector **is array** (natural range <>) **of** std_logic;
where the type would be constrained when the signal is declared
signal std_logic_vector(3 **downto** 0);
 - A constrained array definition:
type not_useful_vector **is array** (3 **downto** 0) **of** std_logic;
- The type not_useful_vector is not useful because it can not assigned to (or receive assignments from) a std_logic_vector, since VHDL is strongly typed.
- To create a constrained type of an array that *can* be used interchangeably (as long as the array sizes are the same) use subtypes:
 - **subtype** byte **is** std_logic_vector(3 **downto** 0);
- This subtype byte can now be used interchangeably with a std_logic_vector of length four (4).

std_logic_vector Array Assignments

```
--only defined signals that knowing their
--definition is necessary for the example
signal up    : std_logic_vector(1 to 4);
signal down  : std_logic_vector(4 downto 1);
Signal a     : std_logic_vector(0 to 3);
Signal b     : std_logic_vector(3 downto 0);
Signal long  : std_logic_vector(63 downto 0);
signal item  : natural range 0 to 3;
begin
    --this...
    down <= up
    --...is equivalent to this
    down(1) <= up(4);
    down(2) <= up(3);
    down(3) <= up(2);
    down(4) <= up(1);

    --we can index with the array indices
    --which is known as static indexing
    a(0) <= '1'

    --we can use the operators defined for
    --the individual element type of the array
    sum(0) <= a(0) xor b(0) xor cin;

    --we can dynamically index into an array
    --the array is accessed by the value of
    --item
    a(item) <= '0';

    --we can "slice" an array
    --remember the elements are assigned
    --left to right, regardless of the
    --indices, so this...
    b(1 downto 0) <= a(2 to 3);
    --...is equivalent to this
    b(1) <= a(2);
    b(0) <= a(3);
```

```
.
.
--aggregates
--copy "1000" to b
b <= (3 => '1', 2 => '0', 1 => '0', 0 => '0');

--another copy (copy "0001" to b)
b <= (0 => '1', 1 => '0', 2 => '0', 3 => '0');

--very strange, the indices refer to the
--aggregate formed, not the final array!
--repeat a different way (copy "0001" to b)
b <= (13 => '1', 14 => '0', 15 => '0', 16 => '0');

--but back to something useful (but still
--a bit strange given the previous
--examples)
b <= ('1', '0', '0', '0');
a <= (3 => '1', others => '0');

--concatenate the up and down arrays
both <= up & down;

--a little more fun, assign
--a std_logic_vector with hex values
long <= X"12ABDF2311562312";
.
.
.
```

Array Example

Define and unconstrained array to hold the ROM

Instantiate a **constant** object of type `std_logic_vector_vector` and fill it with the ROM contents

Index into the array – note how the `std_logic_vector` `pc_out` needs to be converted to an integer as the index

```
-----  
--Instruction "EEPROM"  
gen_instructions : process(clk, resetn, Q2, pc_out) is  
  
    type std_logic_vector_vector  
        is array (natural range <>)  
        of std_logic_vector(opLen - 1 downto 0);  
  
    variable opMem : std_logic_vector_vector(0 to 13) :=  
        ( "000000000000", --nop  
          "000000000000", --nop  
          "110001001010", --movlw  
          "000000000000", --nop  
          "111001110111", --andlw  
          "000000000000", --nop  
          "000000101001", --movwf  
          "000000000000", --nop  
          "000000000000", --nop  
          "000000000000", --nop  
          "000000000000", --nop  
          "000000000000", --nop  
          "000000000000", --nop  
          "000000000000", --nop  
          "000000000000");  
  
begin  
    if '0' = resetn then  
        nextOp <= "000000000000";  
    elsif clk'event and '1' = clk then  
        if '1' = Q2 then  
            nextOp <= opMem(conv_integer(unsigned(pc_out)));  
        end if;  
    end if;  
end process gen_instructions;
```

Attributes: Integer and Enumeration Types

- Enumerated and integer types have associated *attributes*.
- These are useful as a tool to make VHDL models, functions, procedure, and objects parameterizable.
- The following attributes apply to all of the scalar types (I.e., integers and enumerated types)
 - type'left
 - type'right
 - type'high
 - type'low
 - type'pred(value)
 - type'succ(value)
 - type'leftof(value)
 - type'rightof(value)
 - type'pos(value)
 - type'val(value)

```
type state is (main_green, main_yellow,
              farm_green, farm_yellow);
type short is range -128 to 127;
type backward is range 127 downto -128'

--You can apply the attributes to the types
--state'left    -> main_green,  state'right    -> farm_yellow
--short'left    -> -128,        short'right   -> 127
--backward'left -> 127,        backward'right -> -128

--note the differences using low rather than left!
--state'low     -> main_green,  state'high    -> farm_yellow
--short'low     -> -128,        short'high    -> 127
--backward'low  -> -128,        backward'high -> 127

--So, we see that the 'low value is the 'left value for an
--ascending range, and the 'right value for a descending
--range

--'pos and 'val convert from and enumerated value to it's
--position number, and vice-versa.

signal short1, short2 : short;
signal state1, state2 : state;

--pos returns a "universal integer" which means
--it can be assigned to any integer type
--assigns the position number of state1 to short1
short1 <= state'pos(state1);
--assigns the state at position number defined
--by short2 to state2
state2 <= state'val(short2);
--note that pos offers type conversion between
--integers and enumerated types

--finally...
--state'succ(main_green) -> main_yellow
--short'pred(0)          -> -1
--short'left(0)         -> -1
--backward'pred(0)      -> -1
--backward'leftof(0)    -> 1
```

Attributes: Arrays

- Arrays also have associated attributes – they are used to elicit information on the size, range, and indexing of an array.
- In general, it is (very) good practice to use attributes to refer to the size or range of an array signal.
 - This way, if the size of an array is changed due to a design change, then the VHDL statements that access the array will automatically adjust to the new size.
- The following attributes apply arrays
 - signal'left
 - signal'right
 - signal'high
 - signal'low
 - signal'range
 - signal'reverse_range
 - signal'length

```
-- converts a string into std_logic_vector
function to_std_logic_vector(s: string)
    return std_logic_vector is

    variable slv: std_logic_vector(s'high-s'low
                                   downto 0);

    variable k: integer;

begin
    k := s'high-s'low;
    for i in s'range loop
        slv(k) := to_std_logic(s(i));
        k      := k - 1;
    end loop;
    return slv;
end to_std_logic_vector;
```

Because the return `std_logic_vector` is defined this way, string can have any offset, i.e.:

```
variable input : string(3 to 5)
```

LED Decoder, Using Arrays

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

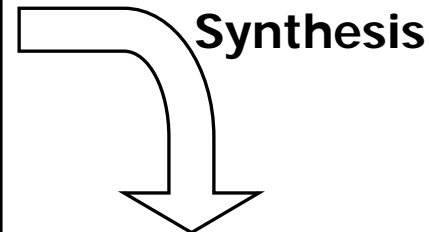
entity sev_seg is
  port (
    digit      : in std_logic_vector(4 downto 1);
    seg7       : out std_logic_vector(0 to 6)
  );
end entity sev_seg;

architecture rtl of sev_seg is

  type std_logic_vector_vector is array (natural range <>) of
    std_logic_vector(seg7'range);
  constant sev_seg_table : std_logic_vector_vector(0 to 15) :=
    ("1110111",
     "0100100",
     "1011101",
     "1101101",
     "0101110",
     "1101011",
     "1111011",
     "0100101",
     "1111111",
     "0101111",
     "0111111",
     "1111010",
     "1010011",
     "1111100",
     "1011011",
     "0011011");

begin
  seg7 <= sev_seg_table(conv_integer(digit));
end architecture rtl;
```

These constructs are not just useful for modeling: using types, constants, unconstrained arrays, attributes, and type casting we can make a very simple LED decoder, with the same synthesis results as our "with/select" statement.



```
Synthesizing Unit <sev_seg>.
  Related source file is
  Found 16x7-bit ROM for signal <seg7>.
  Summary: inferred    1 ROM(s).
```

Multi-Valued Logic Types

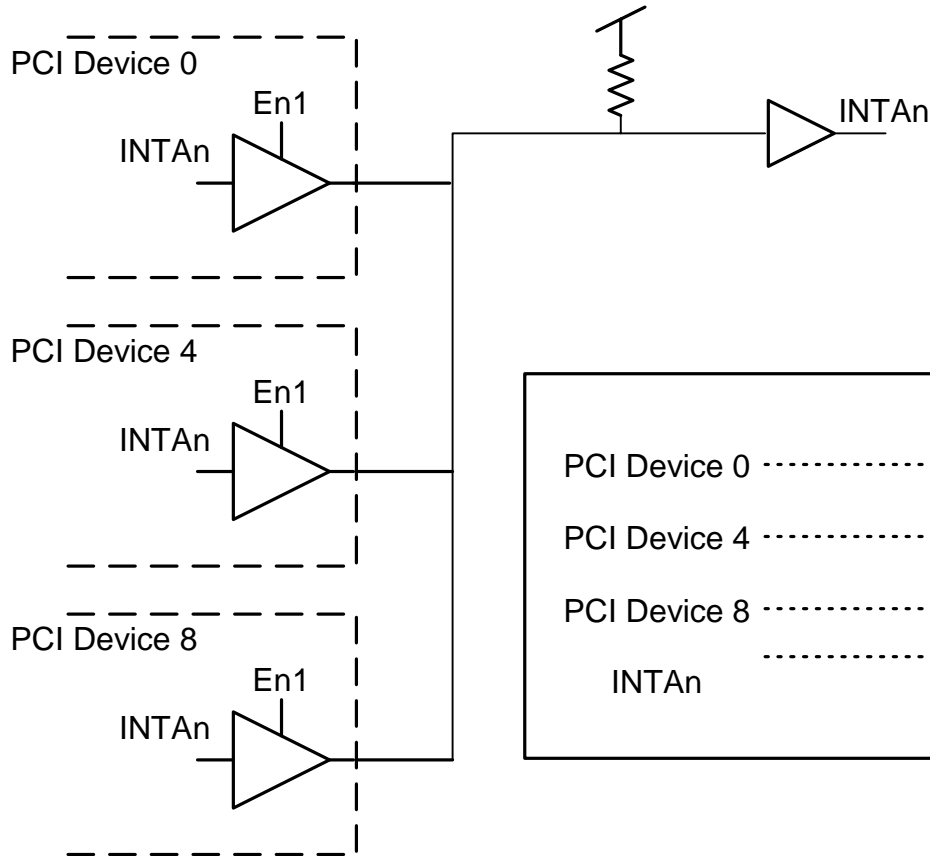
- A **multi-valued logic type** is a logic type that includes values that do not exist in the real world, but are useful in simulation.
 - These values are referred to as *metalogical* values.
 - An example is 'U'. 'U' stands for an undefined value, which can not occur in a circuit – all nodes will have a voltage.
- Note that the concept of a multi-valued logic type is particular to synthesis – the simulator is able to treat them the same as all other enumerated types.
 - A synthesizer needs to be able to identify multi-valued logic types so that they can be represented by a single wire, rather than a bus of wires representing the enumeration encoding of all the metalogical values as unsigned integers

std_ulogic

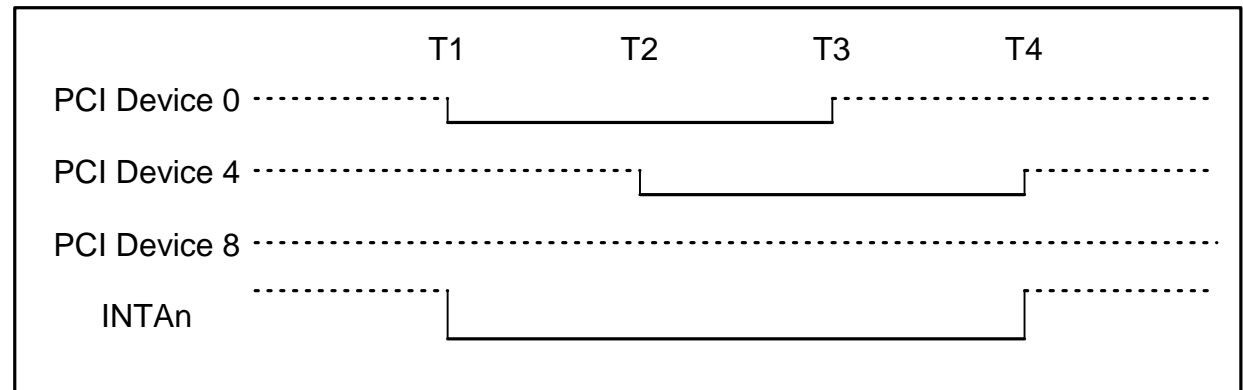
- std_ulogic is the base type of the multi-valued logic standard available in VHDL
- The **attribute** statement is used to tell the synthesizer to recognize std_ulogic as a multi-valued logic type, not as a pure enumerated type (**note that this is implementation dependent**)
- Std_ulogic does not do everything needed, because it is not *resolved*.

```
type std_ulogic is (  
  'U', --Uninitialized (an input port)  
  'X', --Forcing unknown (FF not initied)  
  '0',  
  '1',  
  'Z', --high impedance (tri-state buff)  
  'W', --weak unknown  
  'L', --weak 0 (pull-down resistor)  
  'H', --weak 1 (pull-up resistor)  
  '-', --don't care  
);  
attribute mv_encoding of std_ulogic  
  : type is "UU01ZUUUU"
```

An Example Requiring "Resolved" Signals



Our simulator and synthesizer needs to handle this situation, where *multiple* signals can drive a single node.



```
IntAn <= PCI_DEV0;  
IntAn <= PCI_DEV4;  
IntAn <= PCI_DEV8;  
IntAn <= 'H';
```

The solution is std_logic

- The type std_logic is a resolved subtype of std_ulogic...
 - **subtype** std_logic **is** resolved std_ulogic;
- ...where “resolved” is a function that specifies what occurs if a signal has multiple sources
 - The resolution function must take a single parameter that is a one-dimensional unconstrained array of values of the signal type, and must return a value of the signal type.
 - The resolution function must be a *pure function*, that is, it can have no side effects
 - Finally, the resolution function must be commutative; that is the result should be independent of the order of the values
- When the design is simulated, the resolution function is called whenever any of the resolved signal’s sources is active. The function is passed an array of all the current source values, and the result it returns is used to update the signal value.

std_ulogic Resolution Function

```
-----
-- local types
-----
TYPE stdlogic_1d IS ARRAY (std_ulogic) OF std_ulogic;
TYPE stdlogic_table IS ARRAY(std_ulogic, std_ulogic) OF std_ulogic;
-----
-- resolution function
-----
CONSTANT resolution_table : stdlogic_table := (
--
--      | U   X   0   1   Z   W   L   H   -   |   |
--      -----
--      ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
--      ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
--      ( 'U', 'X', '0', 'X', '0', '0', '0', '0', '0', 'X' ), -- | 0 |
--      ( 'U', 'X', 'X', '1', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
--      ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
--      ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
--      ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
--      ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
--      ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ) -- | - |
);

FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic IS
    VARIABLE result : std_ulogic := 'Z'; -- weakest state default
BEGIN
    -- the test for a single driver is essential otherwise the
    -- loop would return 'X' for a single driver of '-' and that
    -- would conflict with the value of a single driver unresolved
    -- signal.
    IF (s'LENGTH = 1) THEN RETURN s(s'LOW);
    ELSE
        FOR i IN s'RANGE LOOP
            result := resolution_table(result, s(i));
        END LOOP;
    END IF;
    RETURN result;
END resolved;
```

Two Examples

```
--TRISTATE A single element tristate buffer.

library ieee;
use ieee.std_logic_1164.all;

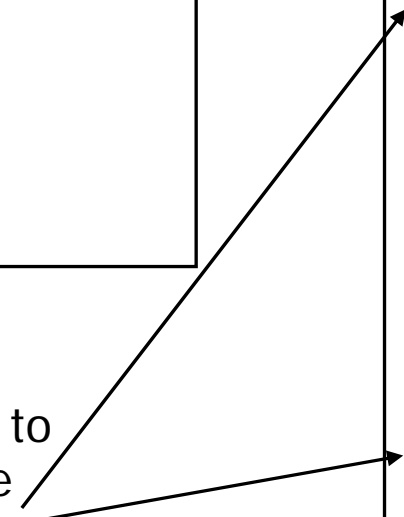
entity tristate is
  port( d : in std_logic;
        en : in std_logic;
        q : out std_logic );
end entity tristate;

architecture mux of tristate is
begin
  process (d, en)
  begin
    if en = '1' then
      q <= d;
    else
      q <= 'Z';
    end if;
  end process;
end architecture mux;
```

```
.
.
.
--part of a testbed

init : process do_read is
begin
  wait until readSig = '1';
  addr <= currentReadAddr;
  readStrobe <= '0';
  wait for 100 ns;
  readStrobe <= '1';
  wait for 100 ns;
  readStrobe <= '0';
end process init;
.
.
.
init : process do_write is
begin
  wait until writeSig = '1';
  addr <= currentWriteAddr;
  data <= currentData;
  writeStrobe <= '0';
  wait for 100 ns;
  writeStrobe <= '1';
  wait for 100 ns;
  writeStrobe <= '0';
  data <= (others => '0');
end process init;
.
.
.
```

These processes need to tri-state when they are through, or contention will occur.



Fuzzy Logic Homework Assignment

Fuzzy logic is a superset of conventional (Boolean) logic that has been extended to handle the concept of incomplete truth -- truth values between "completely true" and "completely false". For this assignment, you will create a new unresolved enumerated type, called `fuzzy_uloic` and its resolved sub-type, `fuzzy_logic`. This type could be used to create a VHDL model of a fuzzy logic system.

Value	Weighting
<code>unInit</code>	n.a.
<code>contention</code>	n.a.
<code>definitelyTrue</code>	n.a.
<code>probablyTrue</code>	3
<code>maybeTrue</code>	1
<code>noDecision</code>	0
<code>maybeFalse</code>	-1
<code>probablyFalse</code>	-3
<code>definitelyFalse</code>	n.a.

The literal values above resolve with the following rules:

1. If one of the signals assigned to a node is `unInit`, the resolved signal is `unInit`.
2. If one of the signals assigned to a node is `contention`, and none of the signals driven into the node are `unInit`, the result is `contention`.
3. If a node is driven by a `definitelyFalse` and a `definitelyTrue`, and none of the signals driven into the node are `unInit`, the result is `contention`.
4. If a node is driven by a `definitelyTrue`, and none of the conditions 1-3 are met, then the result is `definitelyTrue`.
5. If a node is driven by a `definitelyFalse`, and none of the conditions 1-3 are met, then the result is `definitelyFalse`.
6. If none of the above conditions are met, then the result is calculated by summing up the weighted value for each of the literal values:

If the sum is 0, then the result is `noDecision`

If the sum is between [1,2], then the result is `maybeTrue`

If the sum is between [-2, -1], then the result is `maybeFalse`

If the sum is between [3, 6], then the result is `probablyTrue`

If the sum is between [-3,-6], then the result is `probablyFalse`

If the sum is > 6 , then the result is `definitelyTrue`

If the sum is < -6 , then the result is `definitelyFalse`

Fuzzy Logic Testbench

```
use work.fuzzy_logic.all;

entity use_fuzzy is
end use_fuzzy;

architecture test of use_fuzzy is
    constant clockLen : time := 10 ns;
    constant fuzzy_result_length : integer := fuzzy_logic'pos(fuzzy_logic'right) -
                                                fuzzy_logic'pos(fuzzy_logic'left) + 1;

    signal fuzzyResult, fuzzy1, fuzzy2, fuzzy3 : fuzzy_logic := noDecision;
begin

    --Connect three drivers to the fuzzyResult node.
    fuzzyResult <= fuzzy1;
    fuzzyResult <= fuzzy2;
    fuzzyResult <= fuzzy3;

    gen_fuzzy1 : process
    begin
        for i in fuzzy_logic range fuzzy_logic'left to fuzzy_logic'right loop
            fuzzy1 <= i;
            wait for clockLen;
        end loop;
    end process gen_fuzzy1;

    gen_fuzzy2 : process
    begin
        for i in fuzzy_logic range fuzzy_logic'left to fuzzy_logic'right loop
            fuzzy2 <= i;
            wait for clockLen * fuzzy_result_length;
        end loop;
    end process gen_fuzzy2;

    gen_fuzzy3 : process
    begin
        for i in fuzzy_logic range fuzzy_logic'left to fuzzy_logic'right loop
            fuzzy3 <= i;
            wait for clockLen * fuzzy_result_length * fuzzy_result_length;
        end loop;

    end process gen_fuzzy3;

end test;
```

Fuzzy Logic

```
PACKAGE fuzzy_logic IS

-----
-- logic state system (unresolved)
-----
TYPE fuzzy_ulogic IS ( uninit,
                      contention,
                      definatelyTrue,
                      probablyTrue,    --3
                      maybeTrue,      --1
                      noDecision,     --0
                      maybeFalse,     --1
                      probablyFalse,  --3
                      definatelyFalse ---infinity
                      );

-----
-- unconstrained array of fuzzy_ulogic for use with the resolution function
-----
TYPE fuzzy_ulogic_vector IS ARRAY ( NATURAL RANGE <> ) OF fuzzy_ulogic;

-----
-- resolution function
-----
FUNCTION resolved ( f : fuzzy_ulogic_vector ) RETURN fuzzy_ulogic;

SUBTYPE fuzzy_logic IS resolved fuzzy_ulogic;

END fuzzy_logic;

PACKAGE BODY fuzzy_logic IS

FUNCTION resolved ( f : fuzzy_ulogic_vector ) RETURN fuzzy_ulogic IS
    variable aFalse      : boolean := false;
    variable aTrue       : boolean := false;
    variable aContention : boolean := false;
    variable aUnInit     : boolean := false;
    variable theSum      : integer := 0;
```

Fuzzy Logic (2)

- Use one loop
- Loop through all of the inputs, keeping track of sum and any important “higher order” values
- Afterwards, apply the given fuzzy logic algorithm
- Cannot solve with only a table like std_logic
 - What if six maybeFalse drivers drove a node?

```
FOR i IN f'RANGE LOOP
    if f(i) = unInit then
        aUnInit := true;
    elsif f(i) = contention then
        aContention := true;
    elsif f(i) = definatelyTrue then
        aTrue := true;
    elsif f(i) = definatelyFalse then
        aFalse := true;
    elsif f(i) = probablyTrue then
        theSum := theSum + 3;
    elsif f(i) = maybeTrue then
        theSum := theSum + 1;
    elsif f(i) = maybeFalse then
        theSum := theSum - 1;
    elsif f(i) = probablyFalse then
        theSum := theSum - 3;
    end if;
END LOOP;

if aUnInit then
    return unInit;
elsif aContention or (aTrue and aFalse) then
    return contention;
elsif aTrue then
    return definatelyTrue;
elsif aFalse then
    return definatelyFalse;
elsif theSum < 1 and theSum > -1 then
    return noDecision;
elsif theSum >= 1 and theSum <= 2 then
    return maybeTrue;
elsif theSum >= -2 and theSum <= -1 then
    return maybeFalse;
elsif theSum > 2 and theSum <= 6 then
    return ProbablyTrue;
elsif theSum >= -6 and theSum < 2 then
    return ProbablyFalse;
elsif theSum > 6 then
    return DefinatelyTrue;
elsif theSum < 6 then
    return DefinatelyFalse;
end if;
END resolved;
```

compile.do

```
#This must be run from the directory that contains the package and the testbed code. The VHDL file
#that contains the package is compiled below -- if the filename is not my_fuzzy_logic.vhd,
#change the compilation file.
```

```
#This is run by starting Modelsim XE, changing directory to where this file and the source code is
#located, and then typing
```

```
# do compile.do
# in the workspace window. This should:
# * create a library (deleting any old libraries),
# * compile your design (stopping if there are any errors),
# * load the design,
# * load the waveform,
# * and run the simulator.
```

```
#Delete the old work directory, and create a new one
```

```
onerror {resume}
vdel work
vlib work
```

```
#Compile the files
```

```
onerror {abort}
vcom -93 -explicit my_fuzzy_logic.vhd
vcom -93 -explicit use_fuzzy.vhd
vsim use_fuzzy
```

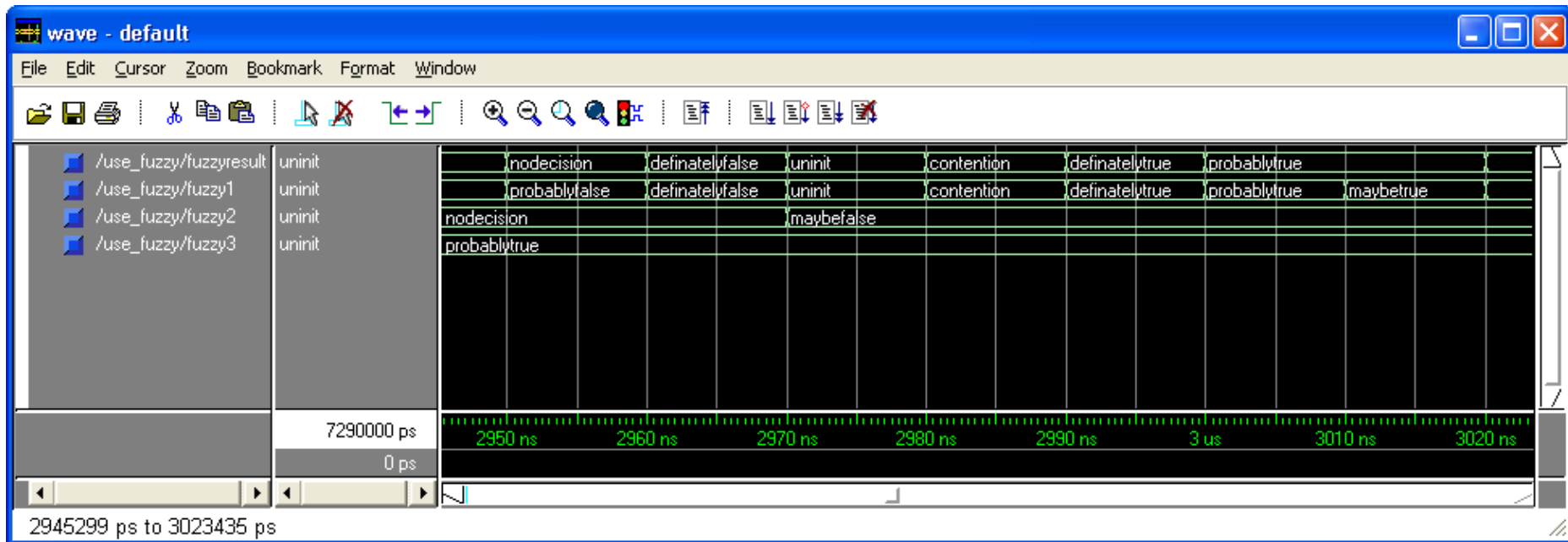
```
#Open the waveform viewer, and load some signals
```

```
onerror {resume}
view wave
quietly WaveActivateNextPane {} 0
add wave -noupdate -format Literal /use_fuzzy/fuzzyresult
.
.
.
configure wave -childrowmargin 2
```

```
#Run for the necessary amount of time
```

```
onerror {abort}
run 7290 ns
```

Fuzzy Logic Testbed Output



Records

- A record is a collection of elements, each of which can be of any constrained type or subtype.
 - The only unconstrained types which cannot be used in a record are unconstrained arrays
- A record is declared as follows:
 - type complex is record
 areal : integer;
 imag : integer;
end record;
- As we would expect, to access a value of a record, use dot notation:
 - a.real <= 0;
- ~~• Records are generally very useful in testbenches, not in synthesizable hardware.
 - They are rarely used in synthesized models – in fact some synthesizers do not even support them.~~

Backup

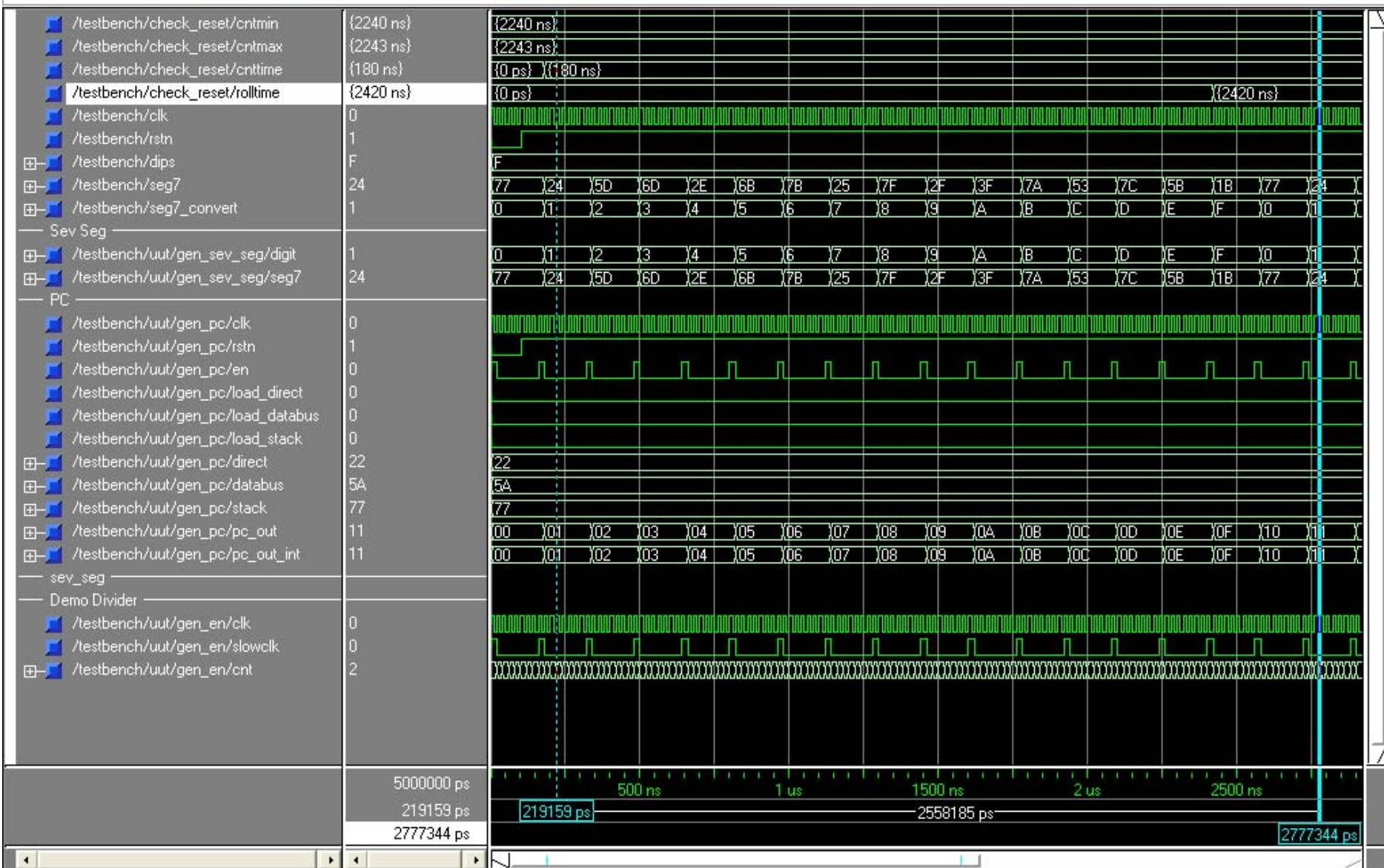
Stack Testbench

VHDL operates concurrently, and this can be taken advantage of when testing. Rather than writing a sequential set of test instructions (basically coding test vectors) we can write dynamic tests that have independent concurrent processes that stimulate and monitor the Device Under Test (DUT) independently. This gives great flexibility when testing.

This test will execute automatically whenever RSTn is set and removed.

```
--any time the reset is thrown this process will check
--it. although it does not currently take other signals
--into account (i.e. the dip switches) it could be added
check_reset : process
  --the *8 takes into account the enable signal
  variable cntMin : time := (maxCnt-2)*clk_period*8;
  variable cntMax : time := (maxCnt-2)*clk_period*8 + 3 ns;
  variable cntTime, rollTime : time := 0 ns;
begin
  --ignoring other values (H, L, etc.)
  --also presuming dips won't change in middle of test
  wait until RSTn = '1';
  wait until RSTn = '0' or seg7_convert = x"1";
  if seg7_convert = x"1" then --wasn't the reset
    cntTime := seg7_convert'delayed'last_event;
    wait until RSTn = '0' or seg7_convert = x"F";
    if seg7_convert = x"F" then --wasn't the reset
      rollTime := now;
      assert ((rollTime - cntTime) >= cntMin)
        and ((rollTime - cntTime) <= cntMax)
        report "ERROR, count is not at correct value."
          severity error;
      wait;
    end if;
  end if;
end process;
```

See handout for entire testbench.



'delayed Attribute

- “Note that we test the time since the last event on `seg7_convert`. When there is currently an event on a signal, the `'last_event` attribute returns the value 0 fs. In this case we determine the time since the previous event by applying the `'last_event` attribute to the signal delayed by 0 fs. We can think of this as an infinitesimal delay...”

Old Lab 3

Part1, General Purpose Register File

Introduction

One of the components of any microprocessor is a register file. On a microprocessor, this small set of memory locations is generally located in fast on-chip RAM, and is used to hold the intermediate results of calculations, loop counters, etc. On many small microprocessors with integrated peripherals (known as microcontrollers) some of these registers are meant to control the on-chip peripherals (these would be called special purpose registers.). In this assignment, we will concern ourselves with implementing only the general purpose registers which are for use by the programmer.

Our microprocessor will consist of thirty-two (32) memory locations, each 1 byte wide. Correspondingly, our general purpose register file (hereafter called **gpregfile** for General Purpose REGISTER FILE) has five (5) ports, defined as follows:

- **Port Definitions**

- **clk**: The **gpregfile** is synchronous, so the RAM is written on the rising edge of the **clk** when **WE** is high.
- **WE**: The write enable. The RAM at address **adr** will be written on the rising edge of the clock if **WE** is high.
- **ADR** : The 5-bit address to be written/read.
- **DIN** : Data to write (written if **WE** is high).
- **DOUT** : The data currently at address **ADR**. This can either be asynchronous or synchronous – meaning DOUT can either always reflect the contents of **gpregfile(ADR)**, or it may reflect the contents of **gpregfile(ADR)** where ADR is the address that was present at the last clock edge.

Lab 3 (cont.)

- **Instructions**

- 1) Create the entity, architecture, simulate and synthesize this model.
- 2) What are the synthesis results? Are the resources (i.e. flip-flops, logic elements) used by this design what you would expect? Are there any other ways you can think your gpregfile could have been implemented in the FPGA? What are they? Do you know why they were not used? Explain in detail.
- 3) Assemble a top-level design for demonstration : On the website is an entire project directory which is only missing the gpregfile. This project contains a ram initializer, which initializes the first 31 of the 32 bytes in RAM with 0x1a,0x1b,0x1c...etc. It works as follows:
 - After the reset (tied to button assigned reset tasks on the Spartan III board), the module sends a series of writes to the gpregfile in order to initialize its contents for demonstration.
 - After it is done writing (31 clocks), the module releases control of the address lines and allows it to be controlled by the switches on the Spartan III board.
 - The **DOUT** of the **gpregfile** should be tied to the your seg7_driver, such that as you step through the various addresses, you should see the contents displayed in hexadecimal on the display.