CPEG 422/622
EMBEDDED SYSTEMS DESIGN

Chengmo Yang

chengmo@udel.edu

Evans 201C

LECTURE 9
MULTIPLIER TESTBENCH

UPDATED SYLLABUS

	Before	Now
• **Projects**	**40%**	40%
• **Homeworks**	**10%**	20%
• **Quizzes**	**10%**	0
• **Midterm**	**15%**	20%
• **Final**	**25%**	20%

MIDTERM DATE

- April 13th, take home midterm

OUTLINE

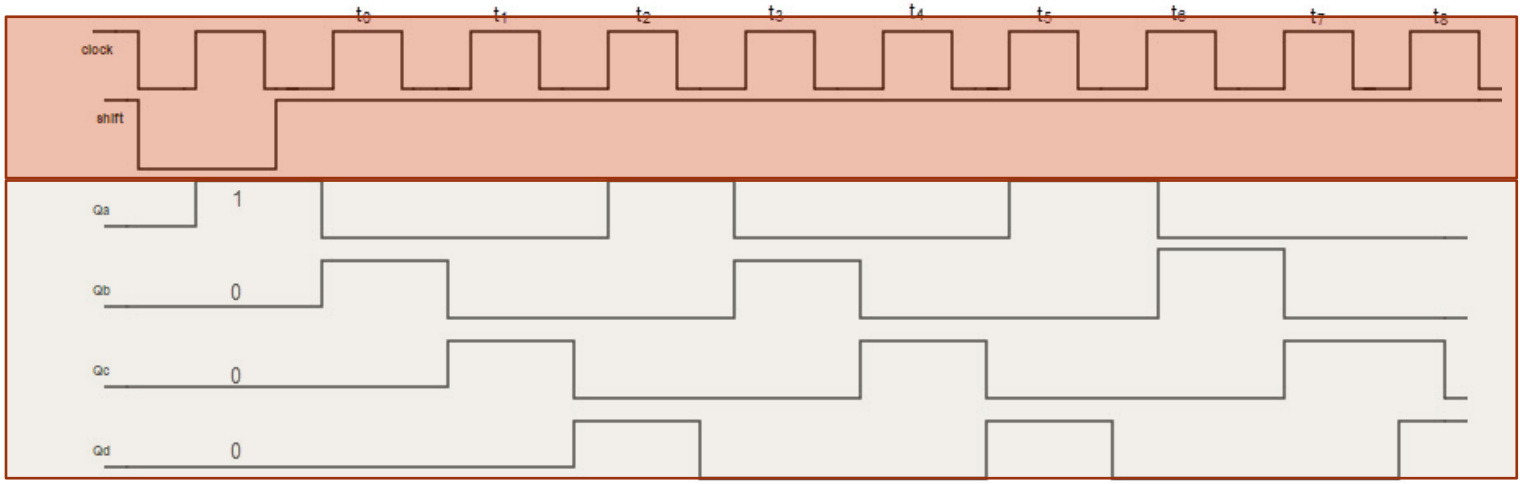
- Last Lecture:
  - Basic multiplication
  - Booth's multiplication
  - Multiplier design in VHDL

- This lecture:
  - Sequential circuit debugging
  - Test your multiplier
  - VHDL programming tips

SEQUENTIAL TEST IN TESTBENCH

- To test sequential logic in testbench, it always requires certain input sequences to set up or initialize the design for it to work properly.
- Functional correctness can also be observed in timing diagram (the waveform).

Timing sequence example



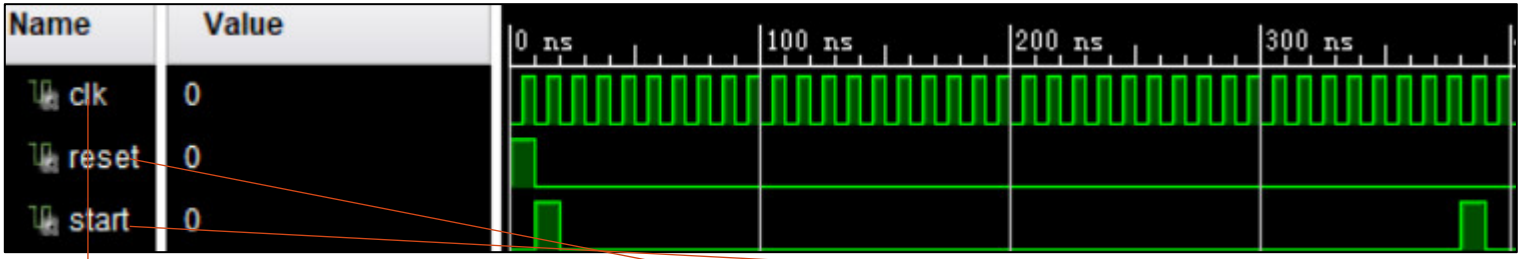
Input setup

Output response

SEQUENTIAL TEST IN TESTBENCH

- Generate specific input timing sequence:

1. Determine what exactly time sequence is required for test.
2. Each signal in one process.
3. Use wait statement to control.




```
clock_gen: process
begin
        clk <= '0';
        wait for 5ns;
        clk <= '1';
        wait for 5ns;
end process;
```

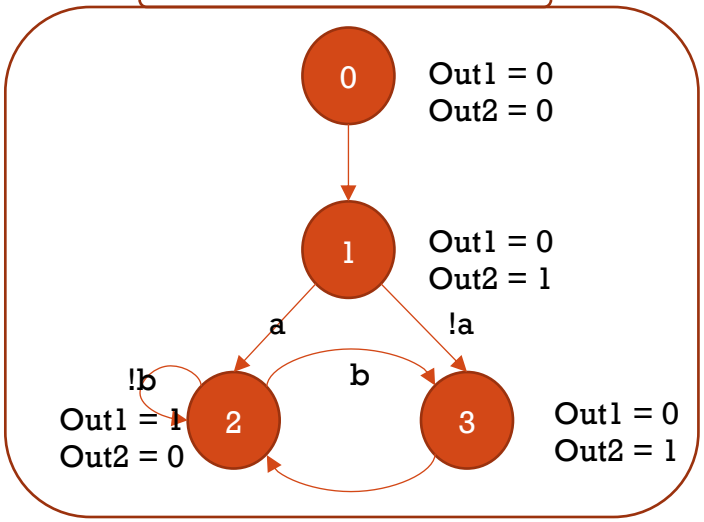
```
reset_gen: process
 begin
        reset <= '1';
        wait for 10ns;
        reset <= '0';
        wait;
 end process;
```

```
start_gen: process
 begin
        start <= '0';
        wait for 10ns;
        start <= '1';
        wait for 10ns;
        start <= '0';
        wait for 350ns;
 end process;
```

All executed concurrently!

SIMULATING AND DEBUGGING STATE MACHINES

Demo State Machine



Buggy Code

```
transition: process(clk, reset)
begin
    if (rising_edge(clk)) then
        if (reset = '1') then
            state <= "00";
        else
            state <= next_state;
        end if;
    end if;
end process;
```

```
next_logic: process(a, state)
begin
    case state is
    when "00" =>
        next_state <= "01";
    when "01" =>
        if (a = '1') then
            next_state <= "10";
        else
            next_state <= "11";
        end if;
    when "10" =>
        if (b = '1') then
            next_state <= "11";
        else
            next_state <= "10";
        end if;
    when "11" =>
        next_state <= "10";

    when others =>
        next_state <= "00";
    end case;

end process;
```

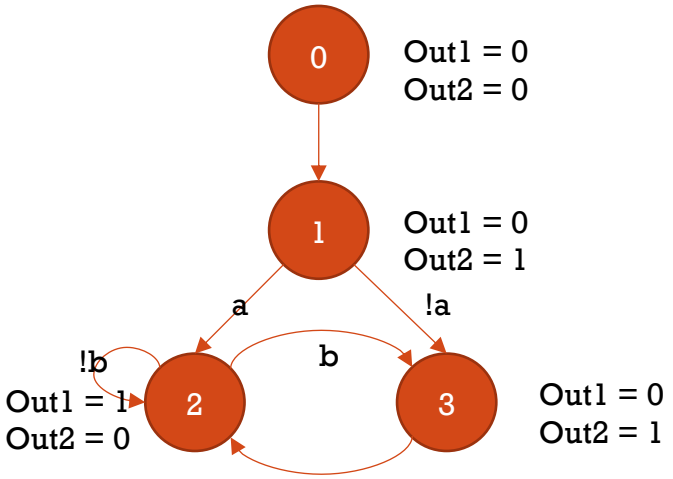
```
out_logic: process(state)
begin
    case state is
        when "00" =>
            out1 <= '0';
            out2 <= '0';
        when "01" =>
            out1 <= '0';
            out2 <= '1';
        when "10" =>
            out1 <= '1';
            out2 <= '0';
        when "11" =>
            out1 <= '0';
            out2 <= '1';
        when others =>
            out1 <= '0';
            out2 <= '0';
    end case;
end process;
```

VIVADO SIMULATION

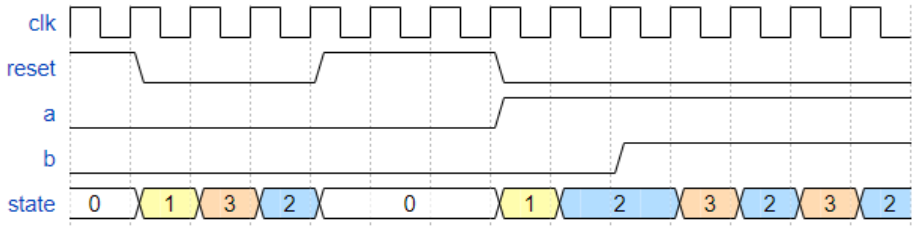
Simulation File

```
clk_proc: process
begin
    clk <= '1';
    wait for 5 ns;
    clk <= '0';
    wait for 5 ns;
end process;
```

```
stim: process
begin

    -----test rest-------
    rst <= '1';
    wait for 10 ns;
    rst <= '0';
    wait for 30ns;
    rst <= '1';
    wait for 30ns;
    rst <= '0';

    -----test a,b--------
    a <= '1';
    wait for 20ns;
    b <= '1';
    wait for 10ns;

    wait;


end process;
```



Expected Output



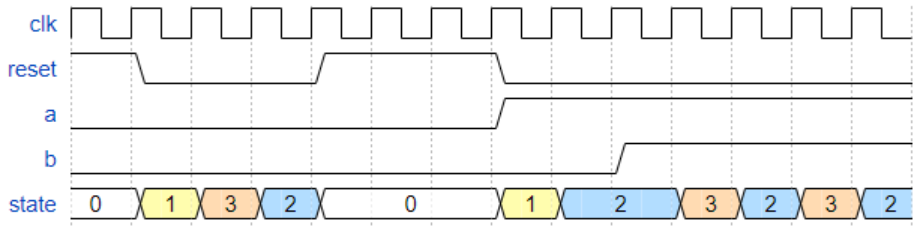
VIVADO SIMULATION

Simulation File

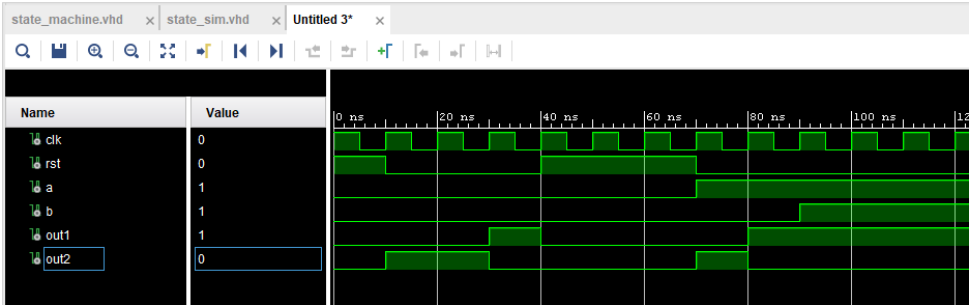
```
clk_proc: process
begin
    clk <= '1';
    wait for 5 ns;
    clk <= '0';
    wait for 5 ns;
end process;
```

```
stim: process
begin

    -----test rest-------
    rst <= '1';
    wait for 10 ns;
    rst <= '0';
    wait for 30ns;
    rst <= '1';
    wait for 30ns;
    rst <= '0';

    -----test a,b--------
    a <= '1';
    wait for 20ns;
    b <= '1';
    wait for 10ns;

    wait;


end process;
```

Expected Output

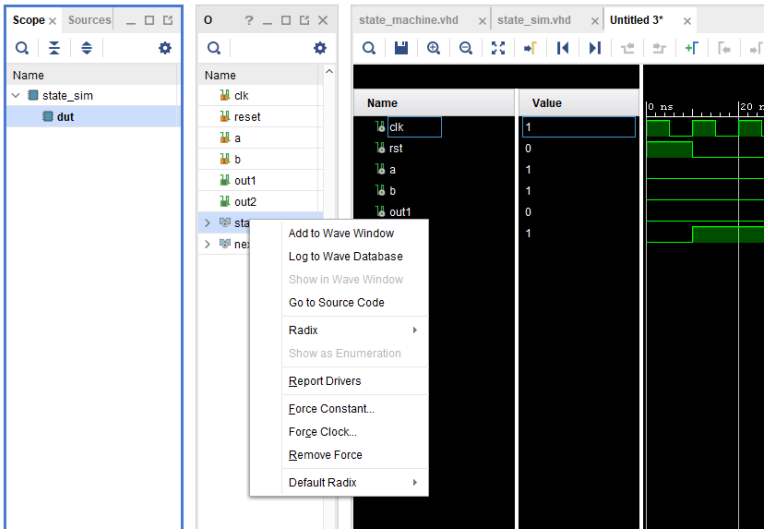


Actual Output



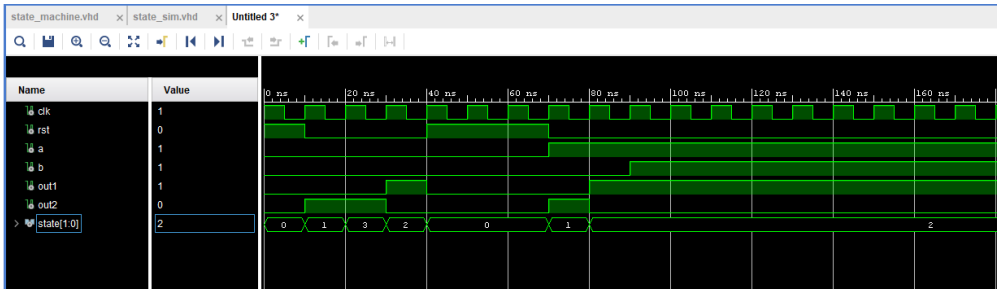
No State! Hard to debug!

ADD SIGNALS TO WAVE WINDOW



- Select 'Scope' in left window. Expand device and select component of interest.
- Select signal of interest in middle window, right-click, and 'Add to Wave Window'.

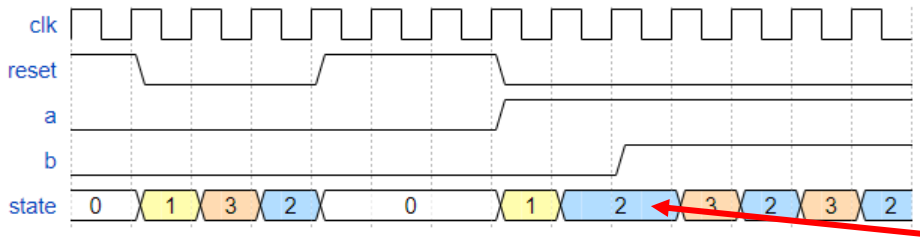
Rerun Simulation



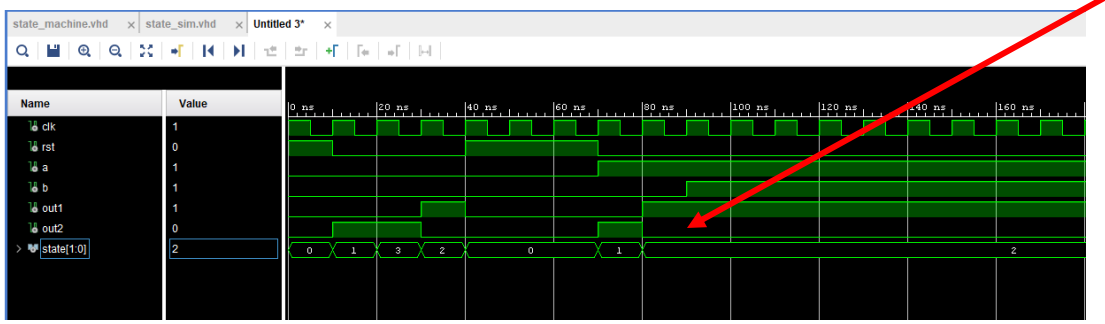
We can see state!

DEBUGGING WAVEFORMS

Expected Output



Actual Output



Major difference in state changes when 'b' changes to '1'

DEBUGGING CODE

We know there are issues with waveforms when 'b' is asserted.

Identify where 'b' is utilized

Recall, process runs when signal in sensitivity list changes

Process doesn't run when 'b' changes. We may get lucky that 'state' or 'a' changes when 'b' changes, but this isn't guaranteed. We must add 'b' into sensitivity list.

TIP: any signals that are read in a process should be in the sensitivity list

Buggy Code

```
transition: process(clk, reset)
begin
    if (rising_edge(clk)) then
        if (reset = '1') then
            state <= "00";
        else
            state <= next_state;
        end if;
    end if;
end process;
```

```
next_logic: process(a, state)
begin
    case state is
    when "00" =>
        next_state <= "01";
    when "01" =>
        if (a = '1') then
            next_state <= "10";
        else
            next_state <= "11";
        end if;
    when "10" =>
        if (b = '1') then
            next_state <= "11";
        else
            next_state <= "10";
        end if;
    when "11" =>
        next_state <= "10";

    when others =>
        next_state <= "00";
    end case;

end process;
```

```
out_logic: process(state)
begin
    case state is
        when "00" =>
            out1 <= '0';
            out2 <= '0';
        when "01" =>
            out1 <= '0';
            out2 <= '1';
        when "10" =>
            out1 <= '1';
            out2 <= '0';
        when "11" =>
            out1 <= '0';
            out2 <= '1';
        when others =>
            out1 <= '0';
            out2 <= '0';
    end case;
end process;
```

CHECKING FIX

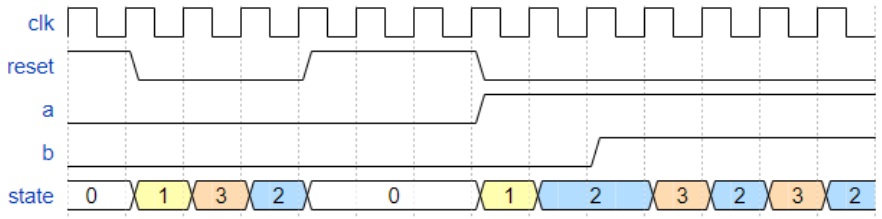
Fixed Code

```
transition: process(clk, reset)
begin
    if (rising_edge(clk)) then
        if (reset = '1') then
            state <= "00";
        else
            state <= next_state;
        end if;
    end if;
end process;
```

```
next_logic: process(a, b, state)
begin
    case state is
    when "00" =>
        next_state <= "01";
    when "01" =>
        if (a = '1') then
            next_state <= "10";
        else
            next_state <= "11";
        end if;
    when "10" =>
        if (b = '1') then
            next_state <= "11";
        else
            next_state <= "10";
        end if;
    when "11" =>
        next_state <= "10";

    when others =>
        next_state <= "00";
    end case;

end process;
```

```
out_logic: process(state)
begin
    case state is
        when "00" =>
            out1 <= '0';
            out2 <= '0';
        when "01" =>
            out1 <= '0';
            out2 <= '1';
        when "10" =>
            out1 <= '1';
            out2 <= '0';
        when "11" =>
            out1 <= '0';
            out2 <= '1';
        when others =>
            out1 <= '0';
            out2 <= '0';
    end case;
end process;
```

Expected Output



Simulated Output

OUTLINE

- This lecture:
  - Sequential circuit debugging
  - **Test your multiplier**
  - VHDL programming tips

TEST THE 24-BIT INTEGER MULTIPLIER

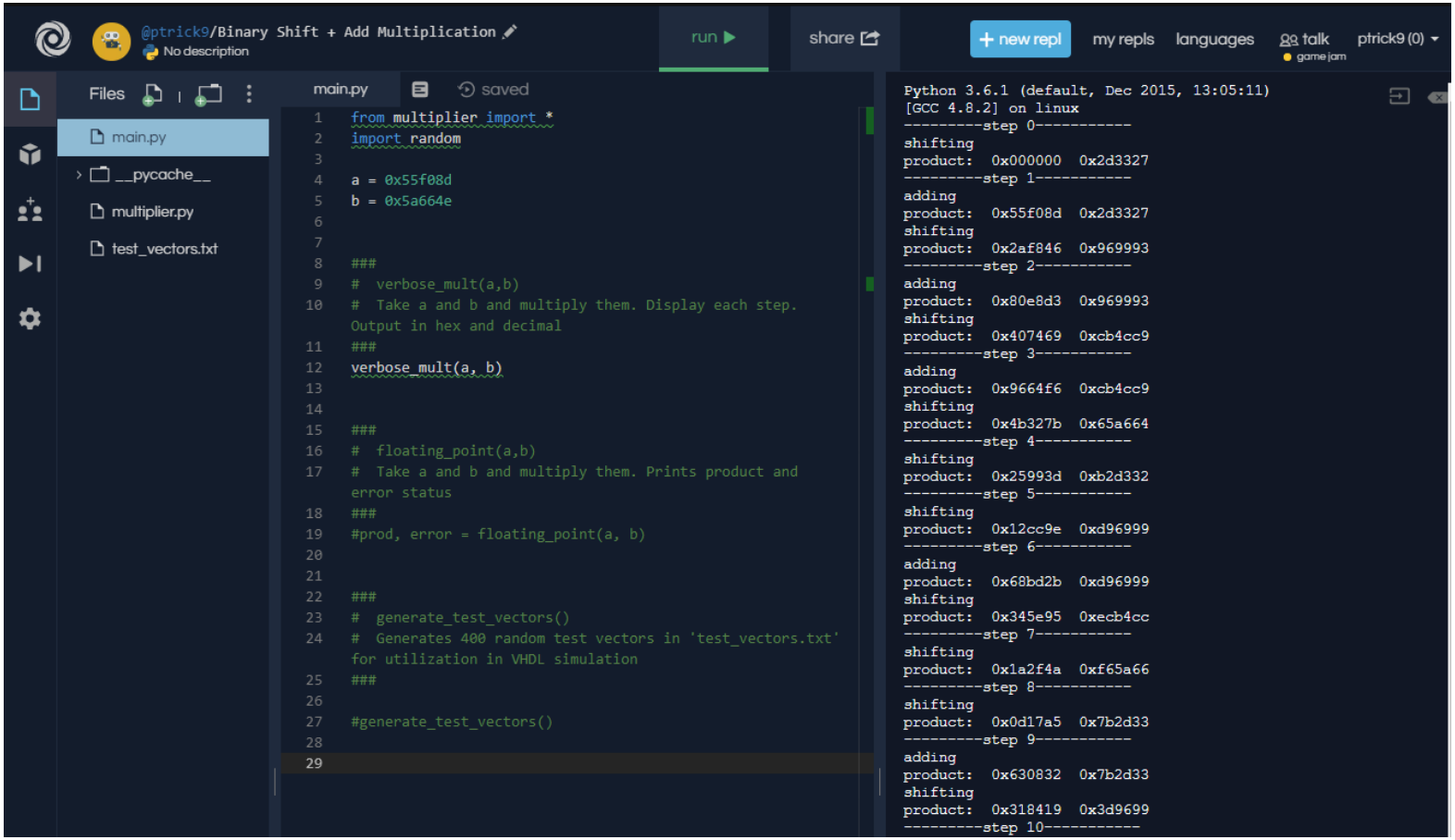
- Use three 16-bit LFSRs
  - One for A [15:0], one for B [15:0], one for A [23:16] and B [23:16]
- Use new libraries

  ieee.std_logic_signed.all:
  Support conversion of vector to signed number

  ieee.numeric_std.All:
  Provide arithmetic functions for vectors for both signed and unsigned

  ieee.std_logic_textio:
  Provide user defined input/output stream, such as printing on screen
- Use the Python helper to debug state-by-state

MULTIPLIER PYTHON HELPER



Provides detailed 24-bit multiplication

https://repl.it/@ptrick9/Signed-Booth-Mulitplication

MATH IN THE TESTBENCH

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_signed.all;
use ieee.numeric_std.ALL;
use ieee.std_logic_textio.all; --includes hwrite
use STD.textio.all; --allows for writing to line


check: process(done, clk)----check result correctness at each ending point of one computing
variable s: line;
begin
    if(done = '1' and clk = '0') then
       if (Product/=A*B) then---check if product is not the same as expected value.
            write(s, time'image(now));
            write(s, string'("Mult failure: "));
            hwrite(s, Product);
            write(s, string'(" \= "));
            write(s, integer'image(to_integer(signed(A))));
            write(s, string'(" * "));
            write(s, integer'image(to_integer(signed(B))));
            writeline(output, s);

       end if;
    end if;
end process;
```

Ensure math libraries are used

Simply check if A*B = Product! (/= is not equal)

Report any issues

ERROR REPORTING IN THE TESTBENCH

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_signed.all;
use ieee.numeric_std.ALL;
use ieee.std_logic_textio.all; --includes hwrite
use STD.textio.all; --allows for writing to line

check: process(done, clk)----check result correctness at each ending poi
variable s: line;
begin
    if(done = '1' and clk = '0') then
       if (Product/=A*B) then---check if product is not the same as expected value.
            write(s, time'image(now));
            write(s, string'("Mult failure: "));
            hwrite(s, Product);
            write(s, string'(" \= "));
            write(s, integer'image(to_integer(signed(A))));
            write(s, string'(" * "));
            write(s, integer'image(to_integer(signed(B))));
            writeline(output, s);

       end if;
    end if;
end process;
```

Ensure IO library is included

Create Line Variable, type 'line' comes from textio

Formatting the line

Write line to output

OUTLINE

- This lecture:
  - Sequential circuit debugging
  - Test your multiplier
  - **VHDL programming tips**
    - Signal vs. variable
    - Procedure vs. function

VARIABLES

- Variables are useful in keeping track of values within the context of a process, a function, or a procedure

Declaration:

Syntax:

```
variable <variable_name> is : type;
```

Examples:

```
variable count_v: integer range 0 to 15;
variable data_v: std_logic_vector(7
downto 0);
variable condition_v: boolean;
```

Note: must be inside a process, function, or procedure

Assignment:

Syntax:

```
<variable_name> := <expression>
   -- the expression must be of a form whose
   result matches
   the type of the assigned variable
```

Examples:

```
boolean_v := true;
temp_v(3 downto 0) := sl_vector_signal(7 downto
4);
```

Note: you can assign variables to signals and vice versa

SIGNAL VS VARIABLE

Signal	Variable
Can be considered **wires** in a schematic that can have current and future values	Used to model the behavior of a circuit, used in processes, procedures and functions
Declared **outside** process	Declared **inside** process/function/ procedure, and are "**local**"
Use **<=** for **concurrent** assignment	use **:=** for **sequential** assignment
Use **:=** for initialization	Use **:=** for initialization
Updated *after a delta delay*	Updated **instantly**

SIGNAL VS VARIABLE – EXAMPLE

A process using variables

```
architecture VAR of EXAMPLE is
        signal TRIGGER, RESULT: integer := 0;
begin
        process
                variable var1: integer :=1;
                variable var2: integer :=2;
                variable var3: integer :=3;
        begin
                wait on TRIGGER;
                var1 := var2;
                var2 := var1 + var3;
                var3 := var2;
                RESULT <= var1 + var2 + var3;
        end process;
end VAR
```

A process using signals

```
architecture SIGN of EXAMPLE is
        signal TRIGGER, RESULT: integer := 0;
        signal sig1: integer :=1;
        signal sig2: integer :=2;
        signal sig3: integer :=3;
        begin
        process
        begin
                wait on TRIGGER;
                sig1 <= sig2;
                sig2 <= sig1 + sig3;
                sig3 <= sig2;
                RESULT <= sig1 + sig2 + sig3;
        end process;
end SIGN;
```

Q: What is the value RESULT when exiting process for the first time?

SIGNAL VS VARIABLE – EXAMPLE

A process using variables	A process using signals

1. var1, var2 and var3 are computed sequentially;
2. their values are updated instantaneously after the TRIGGER signal arrives.
Based on 1 and 2 we have:
var1 = 2;
var2 = 2+3 = 5;
var3 = 5;
 	
3. RESULT, which is a signal, is computed using the new values of the variables.
4. RESULT is updated when exiting the process.
Based on 3 and 4 we have
RESULT = 2 + 5 + 5 = 12; -- after exiting the process
	1. sig1, sig2 and sig3 are also computed sequentially;
	2. their values are not updated until exiting the process.
	Based on 1 and 2 we have:
	sig1 = 2; -- -- after exiting the process
	sig2 = 1+3 = 4; -- after exiting the process
	sig3 = 2; -- after exiting the process
	
	3. RESULT, which is a signal, is computed using the current values of the signals.
	4. RESULT is updated when exiting the process.
	Based on 3 and 4 we have
	RESULT = 1 + 2 + 3 = 6; -- after exiting the process

Q: What is the value RESULT when exiting process for the **second** time?

FUNCTION DECLARATION

- Functions can be useful for repeated use of code.
- Always terminated by a **return** statement, return a value.

Syntax:

```
function <function_name> (
  -- function arguments
  <signal_1> : type; . . . < signal_n> :
  type
)
returns return_type is

  constant declarations
  variable declarations
  ...
begin

  statements
  ...

end <function_name>;
```

Example:

```
function sign_extend (
  narrow_bus: std_logic_vector(15 downto 0)

)
return std_logic_vector(31 downto 0) is

  variable output: std_logic_vector(31
  downto 0);

begin

  output(15 downto 0) <= narrow_bus(15
  downto 0);
    -- lower 16 are the same

  output(31 downto 16) <= (others =>
  narrow_bus(15));
    -- upper 16 are sign-extension of MSB

  return output;

end sign_extend;
```

FUNCTION

Declaration:

```
function identifier [input port declarations] return type is
      [variable declarations]
begin
      function statements
end identifier
```

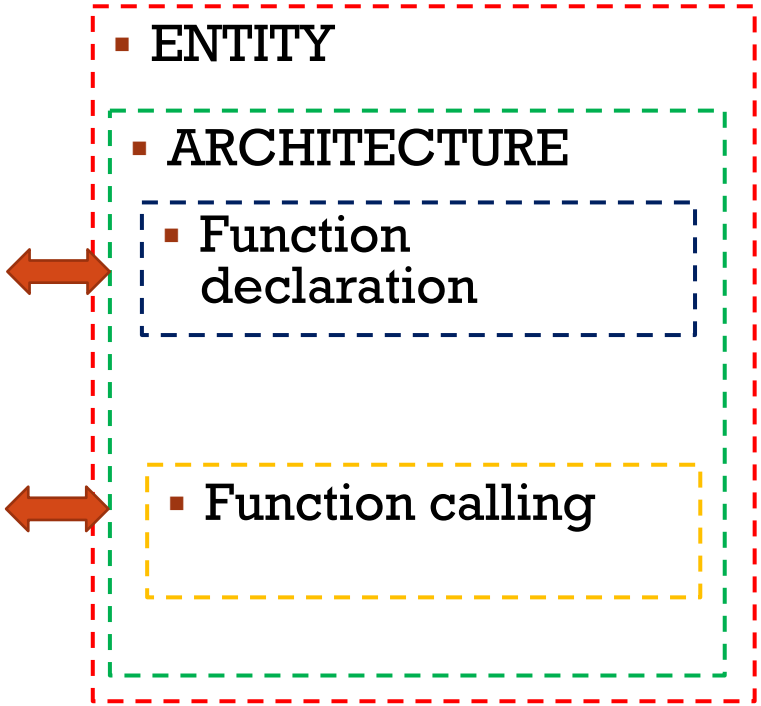
Call:

```
<signal> <=<function_name>[function_augments];
```

Example:

```
Func_Out <= identifier (input);
```

Function call should be put in a process.



PROCEDURE

- A procedure provides the ability to execute common pieces of code from several different places in a model. Same as function.

Syntax:

```
procedure identifier [input/output port declarations] is
      [variable declarations]
begin
      procedure statements
end identifier
```

PROCEDURE EXAMPLE

- This was used in the given 24-bit adder/subtractor testbench.
- Procedure declaration

(Print out error information)

parameter is string input

Runtime unit

```
procedure echo(arg : in string := "" unit: time:= ns) is
begin
    std.textio.write(std.textio.output, time'image(now) & arg & LF);
end procedure echo;
```

Current simulation time

input

Newline

A procedure defined in ieee.std_logic_textio.all;

- Procedure call

```
if (S/=A-B) then--check if diff is not the same as expected value.
    echo("Diff Error: " & integer'image(to_integer(signed(S))) & " /= " & integer'image(to_integer(signed(A))) &" - " & integer'image(to_integer(signed(B))) );
    end if;
```

Convert vector to a signed integer and print as string

FUNCTION VS. PROCEDURE

Function	Procedure
No delay, timing, or event control (combinational behavior)	Can have delay, timing, or event control (sequential behavior)
There is at least one input argument, but NO output or inout arguments	The arguments are flexible (zero or many)
Cannot store values in global objects or their arguments, only maintain local copies	Can store values in global objects or arguments
All statements are executed sequentially, no non-blocking assignments	Can have non-blocking assignments (concurrent statement)
Signal and object declaration are not allowed	Signal and object declaration are not allowed

NEXT LECTURE

- Floating point arithmetic