

QUICK START TUTORIAL

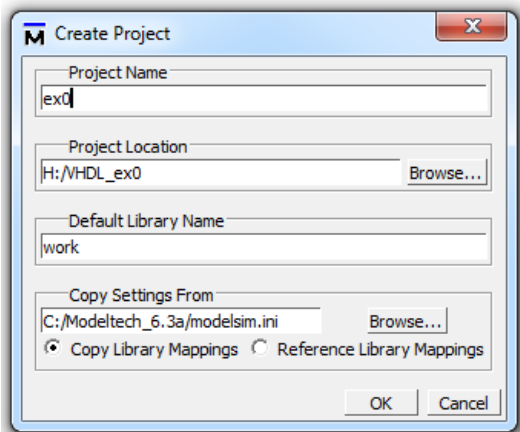
The course uses **Modelsim** (simulator) & **Synplify** (Synthesis tool) as installed on department computers. VHDL may be intelligently edited & beautified very nicely with **Sigasi**, which is free and can be downloaded to any system.

In this tutorial, you will first learn how to use Modelsim to compile files and simulate hardware. Then, you will learn how to use Synplify to synthesise your hardware and further debug your code. The last part of the tutorial shows an example on making a .do file which automatically compiles files and runs simulations in Modelsim.

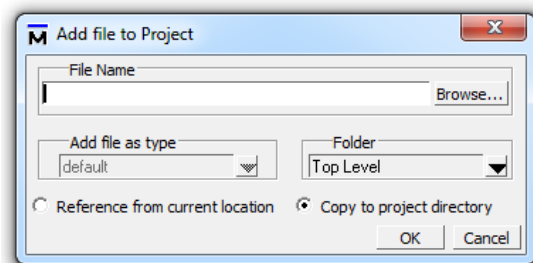
MODELSIM

NEW PROJECT

- Start ModelSim
- Create a new project by using **File -> New ->Project**



- In the pop-up window, enter your project name and location, leave the **Default Library Name** “work” as it is
- After the project has been created, clicking **Add Existing File** to add VHDL files to be simulated to the project. It is better to **copy VHDL files into the directory** if they are not already there, rather than to link to files to somewhere else.



- With the project open you can edit files inside ModelSim by clicking on file names in project window.

COMPILATION

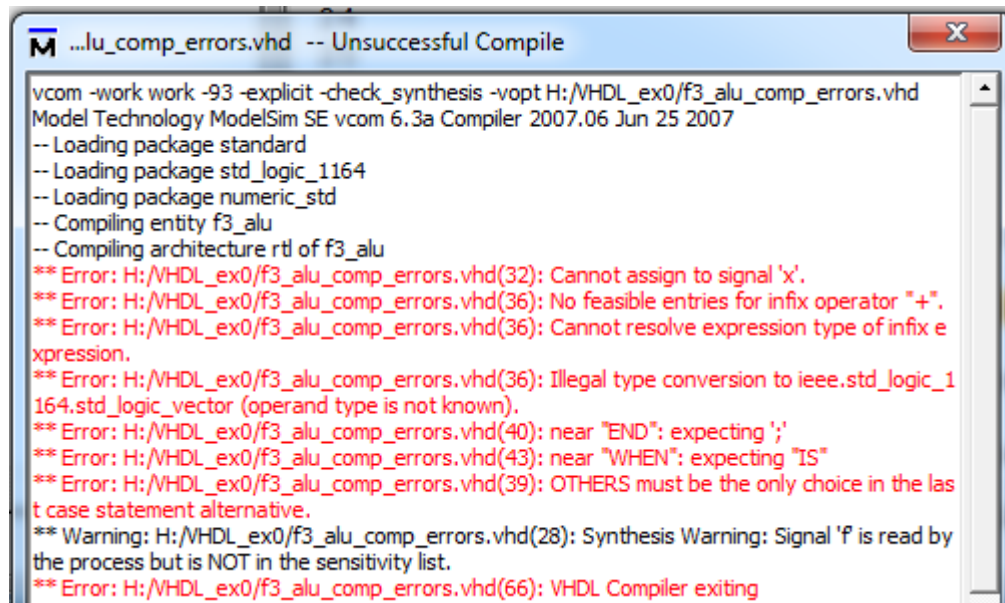
- Before you compile files, **right click -> compile -> compile properties -> VHDL**, make sure:
 - 1) Language Syntax – Use 1076-1993
 - 2) Check for – Synthesis

- You can now compile a single file by **right clicking on the file name in project window**.
NOTE that files have to be compiled in the correct order: packages or sub-entities before entities.
- Compile f3_alu_comp_errors.vhd**, compile status appears in transcript window.

```
Transcript
# Compile of f3_alu_comp_errors.vhd failed with 2 errors.

ModelSim>
```

- Double-clicking on the status gives more details in pop-up, double-clicking on individual error moves to correct source line in edit window. *If the pop-up for errors annoys you set project -> project settings -> display compiler output.*



COMPILER ERRORS

A large number of errors come from incorrect syntax: missing keywords, missing “;” at end of statements, etc. these are self-explanatory, and should be easy to understand. **Correct syntax errors first! Then get the following type errors:**

```
ALU : PROCESS(y, z_int, y1)
```

```
BEGIN
```

```
# ** Warning:/ex0/f3_alu_comp_errors.vhd(28): (vcom-1400) Synthesis
Warning: Signal "f" is read in the process but is not in the
sensitivity list.
```

```
# ** Warning:/ex0/f3_alu_comp_errors.vhd(42): (vcom-1400) Synthesis
Warning: Signal "f" is read in the process but is not in the
sensitivity list.
```

NOTE* The error here will only be seen if “check synthesis” box is ticked under:

right click f2_alu_comp_errors->compile->compile properties->VHDL

It shows you have not added f to the sensitivity list as is required. This must be done since anything else will not synthesise correctly.

```
x <= slv((y1) + usg(z_int));
```

```
# ** Error: /ex0/f3_alu_comp_errors.vhd(36): No feasible entries for  
infix operator "+".
```

```
# ** Error: /ex0/f3_alu_comp_errors.vhd(36): Cannot resolve  
expression type of infix expression.
```

```
# ** Error: /ex0/f3_alu_comp_errors.vhd(36): Illegal type conversion  
to ieee.std_logic_1164.std_logic_vector (operand type is not known).
```

The + operator expects its two operands (`y1` & `usg(z_int)`) to have compatible types. In this case `y1` is type `std_logic_vector` & `usg(z_int)` is type `unsigned`. Solution is to use `usg()` to convert `y1` to type `unsigned` too.

The next two messages are consequences of the first.

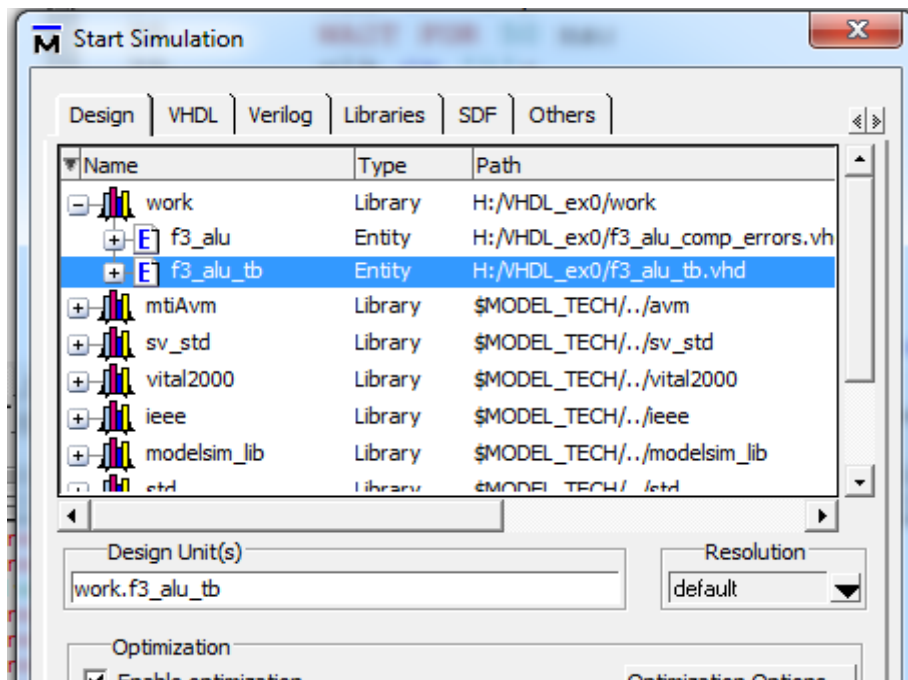
```
x <= (usg(y1) - usg(z_int));
```

```
# ** Error: /ex0/f3_alu_comp_errors.vhd(36): Type error resolving  
infix expression "-" as type ieee.std_logic_1164.std_logic_vector.
```

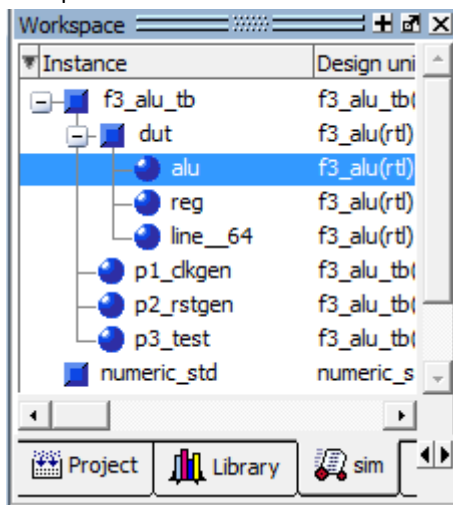
This is another type error. Here the two operands are both `unsigned`, and - operator can be applied in one of its many versions. However the - result type must be `unsigned`. The assignment is to an `std_logic_vector`, so the types do not match. Solution use `slv()` on the whole LHS to convert `unsigned` result into `std_logic_vector`.

SIMULATION

- Click Simulate -> Start Simulation



- From “work” library, choose top-level entity you want to simulate (normally the VHDL **testbench**, testbench f3_alu_tb in this case), then click ok. The simulator then starts.
- Before running a simulation, add signals to the **wave**(waveform), there are many ways to do this, for example:



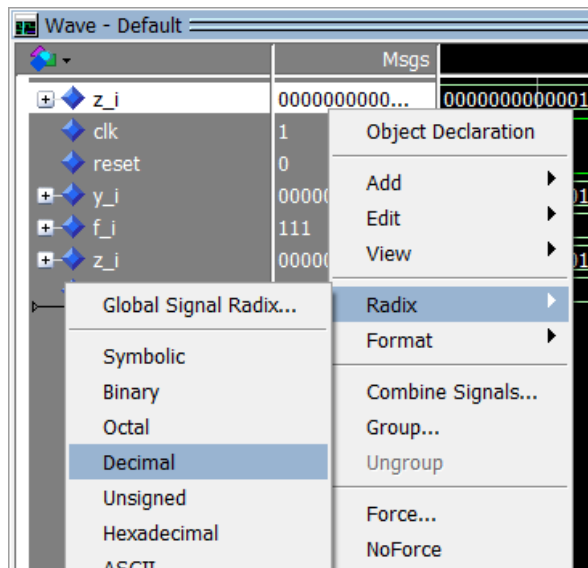
In **sim window**, instance -> right click -> add -> add all signals to wave.

- **Run simulation** (simulate -> run -> run all).
- Check waveforms in wave window (**view** -> **wave** if you can't see it). Unlock window if needed. Note that you can click on the window to display waveform values at given point.

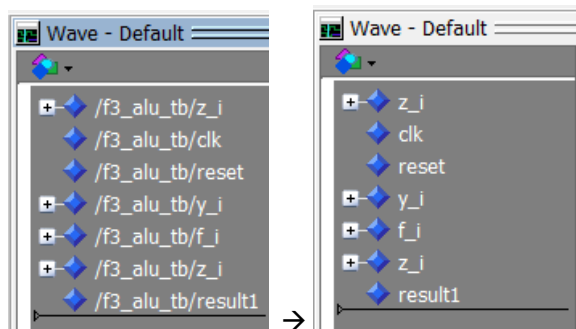
Tips using wave:

- Note that you can set the radix of every signal – it is often easier to view vectors in decimal (signed or unsigned).

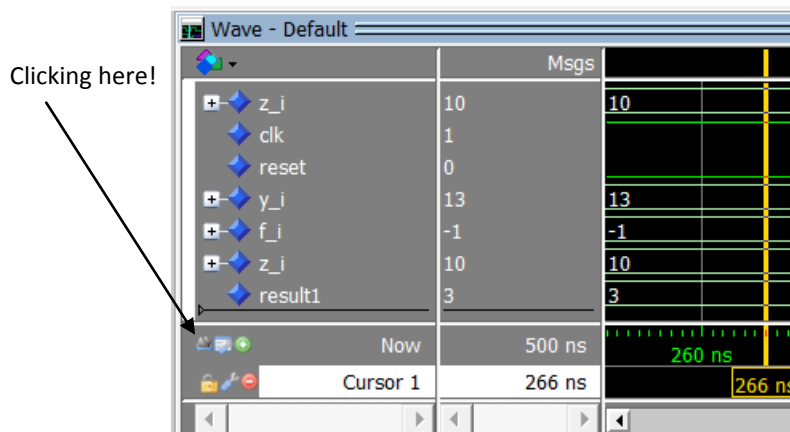
To do this, select signal or signals you want to change its radix, and then right click -> radix ->



- Default signal names (full names) are not very neat, you can toggle leaf names and full names when it is necessary



You can do this by:



- If you need to wave different signals, or even edit and recompile code, do this without exiting the simulator and then **run -> restart**. This will re-initialize the simulation preserving wave window and other things you have set up.
- **.do files**
After you add signals to wave and set their format, you can save your settings to a .do file by saving file. Next time you run simulation, you can run this .do file instead of adding signals again by typing in transcript "**do** your_file_name.do".

```

Transcript
# Loading work.f3_alu(rtl)#1
VSIM 23> do simulation1.do

VSIM 24> |

```

You can modify your .do file by open them in Modelsim edit window or a text editor. Check this in details in the end of the tutorial.

- A key debugging aid is the **dataflow** window (**view -> dataflow**). Similar to **wave**, you can add all or part of a simulation using **dataflow -> add**. As simulation progresses it will show the structure of the hardware – processes and dataflow statement blocks. It will show the flow of data through each block. With this you can check that parts of your design communicate as you expect. At the end of simulation it will show driven lists for each process. You can check that each signal is correctly and uniquely driven (a common source of baffling bugs).

SYNPLIFY

You will find in previous tutorial that it is difficult to diagnose run-time error under Modelsim simulation. In this case, synthesis with Synplify should detect the error painlessly.

- Start **Synplify Premier**.
- **File -> New Project**
- Add all VHDL files you wish to synthesis (**add file**). In this case, **f3_alu_sim_errors.vhd**. Note that, testbenches, and packages used by testbenches but not hardware, must NOT be synthesized. User-defined Packages used by hardware must be synthesized.
- Set **implementation options -> VHDL -> top entity** to be the top-level hardware entity you want synthesized (in this case, **f3_alu**). Note that there are some options for optimizing synthesis output (implementation options ->).
- Run the synthesis.
- Check Project Status:

Project Status

Implementation Directory

Process View

Report

Project Settings

| | | | |
|-----------------------|--------|---------------------|-------|
| Project Name | proj_1 | Implementation Name | rev_1 |
| Top Module | f3_alu | Retiming | 0 |
| Resource Sharing | 1 | Fanout Guide | 12 |
| Disable I/O Insertion | 0 | FSM Compiler | 1 |

Run Status

| Job Name | Status | | | | CPU Time | Real Time | Memory | Date/Time |
|---|-------------|---|----|----|----------|-----------|--------|------------------------|
| Compile Input Detailed report | Error | 4 | 16 | 32 | - | 0m:00s | - | 11/06/2013 11:15:22 |
| Map & Optimize Detailed report | out-of-date | | | | 0m:00s | | | 11/06/2013 11:15:22 |

- Click underlined numbers to view notes, warnings and errors. From the report you can check the headline maximum clock frequency (performance summary), so that you can see how fast is your

design. Speed is determined by the critical path – the slowest propagation delay from any flip-flop output to input. The performance data will give a breakdown of what this is.


- Errors:

f3_alu_sim_errors.vhd(11) | Multiple non-tristate drivers for net z (15 downto 0) in f3_alu

f3_alu_sim_errors.vhd(11) | Unresolved tristate drivers for net z(15 downto 0) in f3_alu

- The output **z** is driven from two places, the dataflow statement and the registered process. This would only be ok if they represented tri-state logic, so that one could be switched off. The code written is not tri-state (and in fact tri-state logic is not normally used). If you don't catch this synthesis error you can run the simulation and the output connection will lead to very surprising results. It can be corrected by changing **z** to **z_int** in the if loop.
 - Go back to Modelsim and run the (incorrect) code in simulation. Use the dataflow window to see which signals each processes drives. See whether you can understand the (incorrect) structure of the faulty code from this – these should be two different outputs connected together to drive **z**.
- For post synthesis verification, see notes elsewhere to select appropriate target and extract and use VHDL output files.

.DO FILE IN MODELSIM

In previous Modelsim tutorial, we have introduced .do file. After you start simulation, you can add signals to wave. Then you can save a .do file by clicking "file -> save formats" (ctrl+s) or .

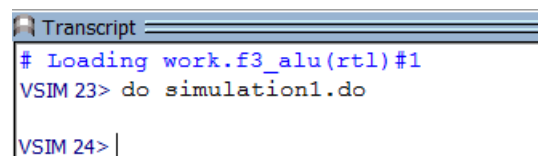
The .do file you saved will probably look like this:

```
1 onerror {resume}
2 quietly WaveActivateNextPane {} 0
3 add wave -noupdate -radix decimal /f3_alu_tb/clock
4 add wave -noupdate -radix decimal /f3_alu_tb/reset
5 add wave -noupdate -radix decimal /f3_alu_tb/y_i
6 add wave -noupdate -radix decimal /f3_alu_tb/f_i
7 add wave -noupdate -radix decimal /f3_alu_tb/z_i
8 add wave -noupdate -radix decimal /f3_alu_tb/result1
9 TreeUpdate [SetDefaultTree]
10 WaveRestoreCursors {{Cursor 1} {0 ns} 0}
11 quietly wave cursor active 0
12 configure wave -namecolwidth 150
13 configure wave -valuecolwidth 100
14 configure wave -justifyvalue left
15 configure wave -signalnamewidth 1
16 configure wave -snapdistance 10
17 configure wave -datasetprefix 0
18 configure wave -rowmargin 4
19 configure wave -childrowmargin 2
20 configure wave -gridoffset 0
21 configure wave -gridperiod 1
22 configure wave -griddelta 40
23 configure wave -timeline 0
24 configure wave -timelineunits ns
25 update
26 WaveRestoreZoom {0 ns} {1 us}
```

They are adding signals to wave window. You need to “start simulation” manually first then execute this file by calling it in transcript window. You can modify the .do file to avoid starting the simulation manually. Open your .do file in a text editor or Modelsim editor then write commands following below:

- set the library:
vlib work
- compile files needed:
vcom -93 -explicit -work work f3_alu_sim_errors.vhd
vcom -93 -explicit -work work f3_alu_tb.vhd
- load files for simulation
vsim f3_alu_tb
- open wave window (or other windows)
view wave
- add signals to wave window (if you don’t want to check wave, you can ignore this step)
{... lines already in your .do file which are for add signals to wave..}
If you don’t have any you can write:
add wave -noupdate your_signal_full_name
- run simulation for 100ns
run 100

Save your changes to the .do file. Now “do your_file_name.do” command in transcript window will compile your files automatically and run simulation for 100ns.



```
Transcript
# Loading work.f3_alu(rtl)#1
VSIM 23> do simulation1.do
VSIM 24> |
```

If you have any problems doing it, copy example commands from appendix to your .do file.

Note that you can also create your new .do file by saving a text file to format of .do.

SUMMARY

From this tutorial you should now understand:

- VHDL testbenches can be written to simplify hardware testing by presenting a sequence of stimulus values, derived from a file or table, and automatically checking outputs are correct.
- Modelsim has very sophisticated simulation which can be used to debug VHDL hardware.
- VHDL is a strongly typed language – with many errors flagged by the compiler. Type conversion functions are sometimes used to make sure that types match.
- Synthesis can catch errors not seen by the compiler.
- The Modelsim Dataflow window is helpful diagnosing errors due to incorrect signals being driven, and also a good way to visualize what your code represents.

APPENDIX - .DO FILE EXAMPLE

```
vlib work

vcom -93 -explicit -work work f3_alu_sim_errors.vhd
vcom -93 -explicit -work work f3_alu_tb.vhd

vsim f3_alu_tb

view wave

onerror {resume}
quietly WaveActivateNextPane {} 0
add wave -noupdate -radix decimal /f3_alu_tb/clk
add wave -noupdate -radix decimal /f3_alu_tb/reset
add wave -noupdate -radix decimal /f3_alu_tb/y_i
add wave -noupdate -radix decimal /f3_alu_tb/f_i
add wave -noupdate -radix decimal /f3_alu_tb/z_i
add wave -noupdate -radix decimal /f3_alu_tb/result1
TreeUpdate [SetDefaultTree]
WaveRestoreCursors {{Cursor 1} {0 ns} 0}
quietly wave cursor active 0
configure wave -namecolwidth 150
configure wave -valuecolwidth 100
configure wave -justifyvalue left
configure wave -signalnamewidth 1
configure wave -snapdistance 10
configure wave -datasetprefix 0
configure wave -rowmargin 4
configure wave -childrowmargin 2
configure wave -gridoffset 0
configure wave -gridperiod 1
configure wave -griddelta 40
configure wave -timeline 0
configure wave -timelineunits ns
update
WaveRestoreZoom {0 ns} {1 us}

run 100
```