VHDL Simulation Environment

We will use the *modelsim* environment for both VHDL and Verilog simulation in this class.

I will provide a directory structure + Makefiles that will work with the *modelsim* software on the UNIX machines.

A free version of the *modelsim* software for Window-based PCs is available, you can copy the file:

http://www.ece.msstate.edu/~reese/tmp/webpack_mxe_simulator.exe

There are some code-size limitations with the free version, but all of the early labs seem to work ok with it.

Directory Structure

Grab the zip archive at:

http://www.ece.msstate.edu/~reese/EE8993/vhdl_course.zip

Unpack this with: ‘unzip vhdl_course.zip’.

This will setup a directory structure that looks like:

```
vhdl_course/src/
               /Makefiles  -- Makefiles for VHDL  libraries
               /exam1      -- source directory for VHDL example #1
              /utilities   -- various files that define useful
                           -- VHDL packages which we will use in
                           -- this course
 vhdl_course/obj/qhdl/
                /exam1 -- compiled VHDL object code for example #1
                /utilities -- compiled VHDL object code for utilities
```

Makefiles

Except for the 'Makefiles' directory, each directory under the 'src' directory represents a VHDL 'library'.

The VHDL files within the library contain VHDL entities, packages, and configurations that reside within the library.

Under the 'Makefiles' directory, there is a 'Makefile' for each VHDL library, i.e:

src/Makefiles/Makefile.exam1 - makefile for library 'exam1'
src/Makefiles/Makefile.utilities - makefile for library 'utilities'

Compiling Using a Makefile

To compile the contents of a library using one of the Makefiles, change directories to the 'src' directory and do:

```
swsetup modelsim
gmake -f Makefiles/Makefile.exam1 TOOLSET=qhdl
```

This will compile the contents of the 'exam1' library.
The Makefile has been written to be compatible with several VHDL simulators, hence the use of the 'TOOLSET' variable.

The 'swsetup' command only has to be issued once in order to put the Mentor QHDL tools on your path; you may want to add this to your .cshrc file.

Adding new VHDL entities/packages/configurations to a Makefile

If you want to add new VHDL entities/packages/configurations to an existing library then:

•Create the files in the target library directory (i.e., src/exam1/myfile.vhd).

•Create the update rules for the file in the Makefile (i.e., src/Makefiles/Makefile.exam1).

When editing the Makefile, look at how the update rules for the other VHDL files are done and simply follow the same pattern.

Adding a new library

If you want to add your own VHDL library, then follow these steps, all of which must be executed from within the 'src/' directory:

• Create the new library directory
```
mkdir mylib
```

•Create the QHDL library object directory via:
```
qhlib ../obj/qhdl/mylib
mkdir ../obj/qhdl/mylib/ts
```

The result of this command will be a new directory '../obj/qhdl/mylib' which will have some QHDL setup files in it. The 'ts' directory holds timestamp information required by our Makefile setup.

Adding a new library (cont)

•Create a logical to physical name mapping for the new library via:

```
qhmap mylib ../obj/qhdl/mylib
```

This command edits the 'src/modelsim.ini' file and adds a logical to physical name mapping entry. You could also simply edit the src/modelsim.ini file manually.

•Create a 'Makefiles/Makefile.mylib' makefile that will handle the compilation of any VHDL files in 'mylib'. You can copy one of the other library Makefiles and edit it

src/exam1 Files

The 'src/exam1' directory contains the following files:

•dlatch.vhd - VHDL entity and architecture of simple D latch

•tb.vhd - VHDL entity and architecture of a test bench which instantiates two latches, a stim component, and a trace component.

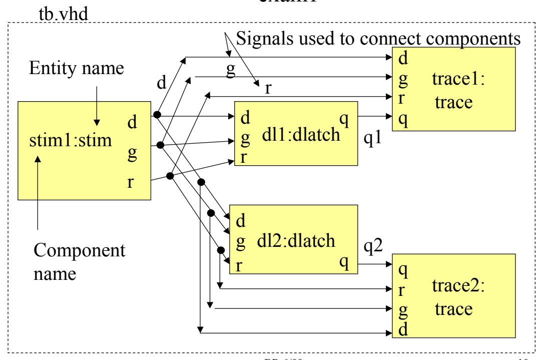
•stim.vhd - VHDL entity of a stimulus module for the testbench
- •stim_nofile.vhd - architecture for 'stim' which hardcodes testvectors
- •stim_readfile.vhd - architecture for 'stim' which reads testvectors from file

•trace.vhd - VHDL entity of a trace module for the testbench
•cfg_tb.vhd - VHDL configuration for the testbench which specifies uses the 'nofile' architecture for 'stim'.
•cfg_tb2.vhd - VHDL configuration for the testbench which specifies uses the 'readfile' architecture for 'stim', plus a different set of generics for the latch delays.

Some VHDL Vocabulary

- *entity* - specifies the interface (inputs/outputs/generics) to a module
- *architecture* – code that specifies the behavior of a module
  - There can be more than architecture for an entity
  - Each architecture would specify some different implementation (I.e., behavioral, RTL, gate level, etc).
- *configuration* - specifies the entities,generics, architectures to use for components within a particular model
- *package* – a collection of VHDL type definitions, procedures, functions, and component declarations.
- *library* – a collection of entities, architectures, configurations, packages

‘exam1’



dlatch.vhd – contains both entity and architecture for a simple dlatch model

```
-- simple dlatch
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

entity dlatch is
 generic (
  r_delay : time := 0 ns;
  d_delay : time := 0 ns;
  g_delay : time := 0 ns

  );

 port (
  d : in std_logic;
  g : in std_logic;
  r : in std_logic;
  q : out std_logic );

end dlatch;
```

For digital simulation will use IEEE 1164 types.

a *generic* block is method of passing parameters to a vhdl model. Values for generics shown are default values.

Port declaration for *dlatch* entity

```
architecture behv of dlatch is
 begin
   process (d,g,r)
     begin
       if (r = '0') then
        -- reset went low
        q <= transport '0' after r_delay;
       elsif (g = '1') then
        -- changes can only occur on output when g is '1'
        -- see if event occurred on either g or d
        -- ignore otherwise
        if (g'event) then
          -- just went to a one, schedule the event
          q <= transport d after g_delay;
        end if;
        if (d'event) then
          -- change on d, schedule change
          q <= transport d after d_delay;
        end if;
       end if;
     end process;
 end behv;
```

Sensitivity list – process triggered on any event on these signals

Delay specification in VHDL

tb.vhd – testbench that connects stimulus, dlatches, trace models

```
-- simple dlatch
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

LIBRARY exam1;
USE exam1.exam1_components.all;
```

VHDL *package* (exam1_components.vhd) that contains component declarations for all models used by this model (stim, trace, dlatch)

```
-- test bench for dlatch
ENTITY tb IS


end tb;
```

Entity declaration is empty because there are not external ports, everything is generated internally.

```
architecture A of tb is
  signal d,g,r: std_logic;
  signal q1,q2: std_logic;
begin
  stim1:stim
    port map ( d => d, g => g, r => r);
  dl1 : dlatch
    port map ( d => d, g => g, r => r, q => q1);
  trace1: trace
    port map ( d => d, g => g, r => r, q => q1);
  dl2 : dlatch
    port map ( d => d, g => g, r => r, q => q2);
  trace2: trace
    port map ( d => d, g => g, r => r, q => q2);
end;
```

Signals used to connect ports of components

Component instantiation
component name first, then entity name

Port map shows how ports connect to signals
port_name => *signal_name*
If two ports connect to same signal, ports are connected.

Component Declarations

A *component declaration* for an entity must be included inside of a VHDL architecture before that architecture can instantiate an instance of that component.

A component declaration is simply a copy of the entity declaration with a couple of minor syntax changes:

```
component dlatch
    generic (
            r_delay : time := 0 ns;
            d_delay : time := 0 ns;
            g_delay : time := 0 ns );
    port (
         d,g,r : in std_logic;
         q : out std_logic );
  end component;
```

'entity' replaced by 'component'

'end dlatch' replaced by 'end component'

Packages

Component declarations can be placed directly within the architecture that uses the corresponding entity.

However, this clutters the architecture code unnecessarily, and often a particular entity is used by more than one architecture.

A better method is to collect commonly used component declarations into a VHDL *package*.

A *package* is a separately compiled VHDL object and is used to group related component declarations, functions, subroutines, type definitions. We will study packages in more detail later in the semester.

exam1_components.vhd

```
library ieee;use ieee.std_logic_1164.all;
package exam1_components is
  component stim
    generic (fname : string := "in.dat" -- trace file );
    port( signal  d,g,r: out  std_logic);
   end component;
  component dlatch
    generic (
            r_delay : time := 0 ns;
            d_delay : time := 0 ns;
            g_delay : time := 0 ns);
    port (d,g,r : in std_logic;
         q : out std_logic );
  end component;
   ............... (…etc..all code not shown)
package body exam1_components is
end exam1_components;
```

Package header

This package used to group component declarations for exam1. The package header contains externally accessible objects.

Package body is empty because in this package only contains component declarations which are all externally accessible.

stim.vhd – entity for stimulus which provides inputs for dlatches.

```
LIBRARY ieee;USE ieee.std_logic_1164.ALL;
-- stim for dlatch
ENTITY stim IS
  generic (
         fname : string := "in.dat" -- trace file
         );
  port(
      signal  d,g,r:  out  std_logic );
end stim;
```

Filename used by *readfile* architecture

stim_nofile.vhd contains architecture that has input values hardcoded in vhdl file.

stim_readfile.vhd contains architecture that reads input value from external file.

stim_nofile.vhd

```
LIBRARY ieee;USE ieee.std_logic_1164.ALL;


architecture nofile of stim is
begin
  main: process
  begin
    -- apply stimulus every 10 ns
---- FF reset                              ----            D  G  R
    d <= '0';    g <= '0';  r <= '0';  ---- 0 ns  ==>  0  0  0
    wait for 10 ns;
---- negate reset
    d <= '0';    g <= '0';  r <= '1';  ---- 10 ns ==>  0  0  1
    wait for 10 ns;
---- D had no effect, clock low
    d <= '1';    g <= '0';  r <= '1';  ---- 20 ns ==>  1  0  1
    wait for 10 ns;
  …………..
     wait;
  end process main;
end nofile;
```

No sensitivity list, process started at time 0.

Process suspended for 10 ns

Entire file not shown

Suspends process forever

Other Entities/Architectures

- *stim_readfile.vhd* - VHDL architecture that reads test vectors from a file – we will cover file I/O in a later lecture
- *trace.vhd* - VHDL entity/architecture that tracks changes on dlatch component inputs/outputs and write these to a file
  - Typically want to write vector outputs to a file so that you can compare against a 'golden' file to see if the results match expected results.

VHDL Configurations

A VHDL *configuration* can be used to select entities/architectures/generics for components within an architecture.

In this case, want to choose delay values for *dlatch* components and architecture for the stim entity (either architecture *nofile* or architecture *readfile* .

Configuration *cfg_tb.vhd* uses the nofile architecture for the *stim* entity and a particular set of delay values for the dlatch components.

Configuration *cfg_tb2.vhd* uses the *readfiles* architecture for the *stim* entity and the same delay values as used in *cfg_tb.vhd* .

Can have more than one configuration for an architecture. A configuration is not required for an architecture.

cfg_tb.vhd

```
LIBRARY ieee;USE ieee.std_logic_1164.ALL;
LIBRARY exam1; USE exam1.exam1_components.all;
CONFIGURATION cfg_tb OF tb IS
 FOR A
  FOR stim1: stim
   use entity work.stim(nofile);
  end for;
  FOR dl1 : dlatch
    use entity work.dlatch(behv)
     generic map(r_delay => 3 ns,d_delay => 4 ns,g_delay => 2 ns);
  end for;
  FOR trace1 : trace
    use entity work.trace(files)
      generic map(  TraceFileName => "dl1.out");
  end for;
  FOR dl2 : dlatch
    use entity work.dlatch(behv)
    generic map(r_delay => 3 ns,d_delay => 2 ns,g_delay => 2 ns);
  end for;
  FOR trace2 : trace
    use entity work.trace(files)
      generic map(  TraceFileName => "dl2.out");
  end for;
 END FOR;
END cfg_tb;
```

Matches arch name of entity *tb*

Use *nofile* arch of entity stim

Delay values for component *dl1*

Output file name for trace1

Different delay values for *dl2*

Output file name for trace2

cfg_tb2.vhd

```
LIBRARY ieee;USE ieee.std_logic_1164.ALL;
LIBRARY exam1; USE exam1.exam1_components.all;
CONFIGURATION cfg_tb2 OF tb IS
 FOR A
  FOR stim1: stim
   use entity work.stim(readfile);
  end for;
  FOR dl1 : dlatch
    use entity work.dlatch(behv)
     generic map(r_delay => 3 ns,d_delay => 4 ns,g_delay => 2 ns);
  end for;
  FOR trace1 : trace
    use entity work.trace(files)
      generic map(  TraceFileName => "dl1.out");
  end for;
  FOR dl2 : dlatch
    use entity work.dlatch(behv)
    generic map(r_delay => 3 ns,d_delay => 2 ns,g_delay => 2 ns);
  end for;
  FOR trace2 : trace
    use entity work.trace(files)
      generic map(  TraceFileName => "dl2.out");
  end for;
 END FOR;
END cfg_tb2;
```

Matches arch name of entity *tb*

Use *readfile* arch of entity stim

Delay values for component *dl1*

Output file name for trace1

Different delay values for *dl2*

Output file name for trace2

Files versus Entities/Architecture/Packages/Configurations

Modelsim does not link file names to VHDL objects (entities/architecture/packages/configurations).

Could have everything in one file, or a separate file for each VHDL object.

The advantage of multiple files is that when an edit is made, can use Makefile dependencies to only recompile changed VHDL object and any VHDL objects that depend on that object.

Placing everything in one file makes recompilation rules simple – if you edit the file, you have to recompile the file. However, the file can get large and unwieldy.

Recompilation Rules for Modelsim

If an architecture is changed, must recompile that architecture.

If architecture and entity are in separate files, then only have to recompile the architecture file, not the entity file.

If architecture and entity are in the same file, then recompiling the architecture, also recompiles the entity. This means that Modelsim now thinks the entity has changed, so any architectures that use that entity (or packages with component declarations for that entity) must also be recompiled.

To have minimum recompilation and clear dependency rules should put each VHDL object in a separate file (entities/architectures/configurations in separate files, package headers/bodies in separate files).

How you arrange VHDL objects and files is up to you!

Makefiles

- Makefiles use to control recompilation of VHDL objects
  - Dependencies can be used in Makefiles to trigger recompilation of an object if another object changes
  - Our approach is to use a separate Makefile for each VHDL library
  - This does not handle dependencies between libraries, but should not encounter this problem very much
- Our makefile template assumes source files reside in ‘src/*libname*’, and object files in ‘obj/*toolset*/*libname*
  - The libname is used to select a particular VHDL compiler - for Modelsim use TOOLSET=qhdl
- When a VHDL object is recompiled, the makefile should update a timestamp for that object under ‘obj/*toolset*/*libname*/ts/*filename*’ where filename is the file that contained the VHDL object.
  - This timestamp can be used to trigger recompilation of a dependent object.

Makefile.exam1

```
##
ifeq (${TOOLSET}, qhdl)
  COMPILER = qvhcom ${DEBUG} ${LIBS} -93 -source $<
  DEBUG =
  LIBS = -work ${LIB_OBJ}
endif

# This is the dummy directory for the timestamps
TS = ts
TOUCH = touch $@
LIB_SRC    = ./exam1
LIB_OBJ    = ../obj/${TOOLSET}/exam1

all:   dlatch stim stim_nofile stim_readfile trace tb cfg_tb
cfg_tb2 exam1_components

dlatch: ${LIB_OBJ}/${TS}/dlatch.vhd
stim:   ${LIB_OBJ}/${TS}/stim.vhd

# ENTITY dlatch
${LIB_OBJ}/${TS}/dlatch.vhd: ${LIB_SRC}/dlatch.vhd
        ${COMPILER}
        ${TOUCH}
```

Compiler command for modelsim

Use VHDL 93 syntax

Update timestamp command

Library name

‘all’ is default target – will execute rules for all targets

Dependency – if dlatch.vhd changes, then do recompile, and update timestampe.

Makefile.exam1 (cont.)

```
# Package exam1_components
## NOTE dependency  on 'stim', 'trace', and 'dlatch' entities
${LIB_OBJ}/${TS}/exam1_components.vhd:
${LIB_SRC}/exam1_components.vhd ${LIB_OBJ}/ts/stim.vhd
${LIB_OBJ}/ts/trace.vhd ${LIB_OBJ}/ts/dlatch.vhd
       ${COMPILER}
       ${TOUCH}
```

Note that the list of dependencies for the exam1_components.vhd timestamp says to recompile exam1_components if the source file (exam1_components.vhd) changes or if any of the timestamps for *stim*, *trace*, and *dlatch* are updated.

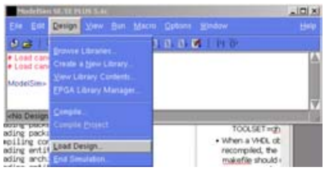
The latter is needed because the *exam1_components* package contains the component declarations for *stim*, *trace*, and *dlatch* and this must be recompiled if these entities are recompiled.

Last Word on Makefiles

- You can look at Makefile.exam1 and see how the other dependencies were done
- How you group VHDL objects (entities, architectures, package headers, package bodies, configurations) into files will determine how you set your dependency rules.
- These makefiles are only compatible with Gnu make program (‘gmake’). They are not compatible with the normal Unix make (‘make’).

Running Modelsim

After compiling your code, you can run modelsim via:
qhsim –lib *libname*   (I.e. qhsim –lib exam1)

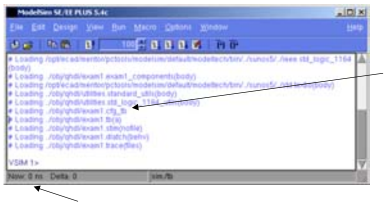


This will not load any VHDL executable objects (configurations or enties). Use Design→ Load Design to load a configuration or entity.

Can also specify the configuration or entity to load on the command line:

```
qhsim –lib exam1 cfg_tb
```

This design will be loaded when the modelsim command window opens.



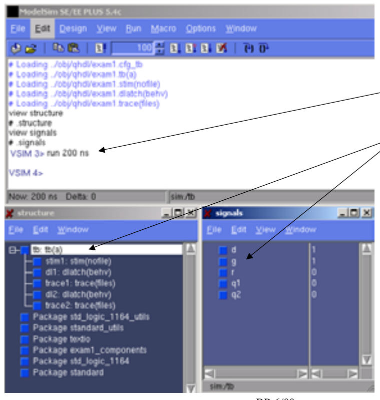
```
# Loading /opt/ecad/mentor/pctools/modelsim/default/modeltech/bin/../sunos5/../ieee.std_logic_1164(body)
# Loading ../obj/qhdl/exam1.exam1_components(body)
# Loading /opt/ecad/mentor/pctools/modelsim/default/modeltech/bin/../sunos5/../std.textio(body)
# Loading ../obj/qhdl/utilities.standard_utils(body)
# Loading ../obj/qhdl/utilities.std_logic_1164_utils(body)
# Loading ../obj/qhdl/exam1.cfg_tb
# Loading ../obj/qhdl/exam1.tb(a)
# Loading ../obj/qhdl/exam1.stim(nofile)
# Loading ../obj/qhdl/exam1.dlatch(behv)
# Loading ../obj/qhdl/exam1.trace(files)

VSIM 1>
Now: 0 ns   Delta: 0        sim:/tb
```

cfg_tb and all dependent objects loaded

Simulator started, time = 0 ns.

Modelsim Debugging Windows

- You can open different debugging windows via the 'View' menu
- The most useful ones are:
  - Structure – displays object structure of design. Selecting an object will change contents of source, signals windows
  - Source - displays source code of currently selected object
  - Signals – displays signals of currently selected object
  - Wave – display waveforms of selected signals
  - Variables – display variable values of currently executing process
- When debugging, should always at least have structure, source, signals and wave windows open



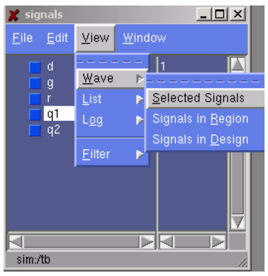
```
# Loading ../obj/qhdl/exam1.cfg_tb
# Loading ../obj/qhdl/exam1.tb(a)
# Loading ../obj/qhdl/exam1.stim(nofile)
# Loading ../obj/qhdl/exam1.dlatch(behv)
# Loading ../obj/qhdl/exam1.trace(files)
view structure
# .structure
view signals
# .signals
VSIM 3> run 200 ns

VSIM 4>
Now: 200 ns   Delta: 0        sim:/tb
```

```
tb: tb(a)
  stim1: stim(nofile)
  dl1: dlatch(behv)
  trace1: trace(files)
  dl2: dlatch(behv)
  trace2: trace(files)
Package std_logic_1164_utils
Package standard_utils
Package textio
Package exam1_components
Package std_logic_1164
Package standard
```

```
d    1
g    1
r    0
q1   0
q2   0
sim:/tb
```

Ran 200 ns

Current values of signals in component 'tb'



Can add signals to waveform window clicking on a signal in the *signals* window, then use:

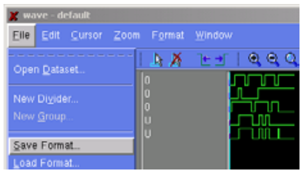
View →Wave → Selected Signal

Click on objects in the structure window to change the currently displayed *signals* in the signals window.

After adding a signal to the wave window, you will not see a waveform until you either run the simulation for more time, or restart the simulation and run it again:

```
VSIM >  restart –f
VSIM >  run 200 ns
```

-f option keeps currently displayed signals



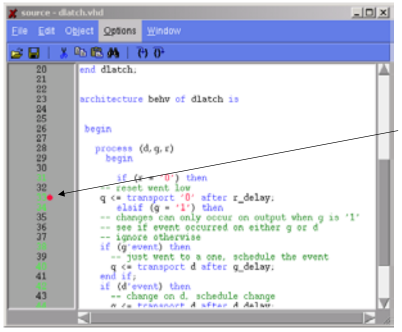
Once you have a desired set of signals displayed, use the File → Save Format

to an external command file (such as 'exam1.do')

If you have your wave format saved to an external file (such as exam1.do) , you can easily display these signals again when you execute the simulator via:

```
VSIM>  do exam1.do
```

This will execute the commands in the file 'exam1.do'.



```
end dlatch;


architecture behv of dlatch is


 begin

   process (d, g, r)
    begin

       if (r = '0') then
    -- reset went low
    q <= transport '0' after r_delay;
       elsif (g = '1') then
    -- changes can only occur on output when g is '1'
    -- see if event occurred on either g or d
    -- ignore otherwise
    if (g'event) then
      -- just went to a one, schedule the event
      q <= transport d after g_delay;
    end if;
    if (d'event) then
      -- change on d, schedule change
      q <= transport d after d_delay;
```

To set a breakpoint, just click on the source line in the source window (a red dot will appear)

To clear a breakpoint, just click on the line again. Use the structure window to select a component in order to view its source code.

Modelsim Misc. Comments

- You can single step code via the Run → Step menu
- These debugging facilities are very powerful – you should be able to determine exactly what your code is doing.
- If you want picosecond resolution instead of nanosecond resolution, you have to edit the modelsim.ini and change 'Resolution = ns' to 'Resolution = ps'.
- If you want to run a simulation in a batch mode, do:

```
qhsim –c –lib exam1 cfg_tb –do "run 200 ns;quit"
```

Inhibts GUI window from appearing (command line interface only)