ECE 385

Modelsim .do file Quick Reference

Table of contents

Introduction

Tired of adding waves to Modelsim every time you start it? A .do file is a script which Modelsim can use to setup a wave window for you. They can be used to generate beautiful output waveforms and accelerate debugging. Two files were used to generate the waveforms in this tutorial and are included as appendices at the end of this document:

psudo_rand.sv – Implementation of a psudo-random number generator. This is the code that will be simulated

psudo_rand.do – Script used to generate simulation waveforms and create test cases for the hardware. This can also be used to debug the circuit.

1) Creating a new .do file

1) Start Modelsim from Quartus  .
2) Go to the Modelsim terminal and type `notepad filename.do` (Windows) or `gedit filename.do` (Linux). You can use other editors if you prefer.
3) Type your script in the window that appears. You can use the file in Appendix B as a template.

Pro Tip: Modelsim re-compiles all of your SystemVerilog code when you load it and generally has stricter requirements than Quartus. Scroll up in the Modelsim console after startup and look for red text if things aren't working.

2) Running a .do file

Once you've created and saved your .do file, type `do filename.do` in Modelsim's terminal.

Pro Tip: If you get an error saying "`could not find wave window`", simply display the wave in Modelsim: `view->wave`.

3) Clearing old data from your waveform

Modelsim's default behavior is to continue where you left off from a previous simulation and retain all the signals in the wave window. To clear data from your previous run, add the following to your .do file:

```
delete wave *
restart -f
```

4) Comments

Lines with a pound (#) as the first character are comments. Beware: the # must be the first character in the line.

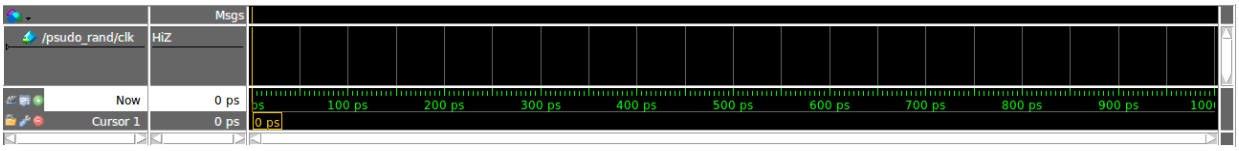
```
# This is a comment
<some cryptic command> # this is not a comment
```

5) Adding signals to your wave window

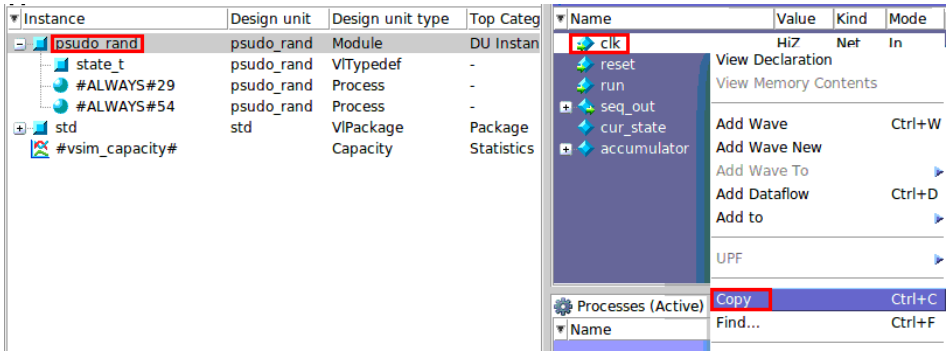
Getting a wave to show up on the screen typically requires at least 2 commands: one to add the wave to the window and another to place data on that line. This is the first of those two steps. Notice that the picture below only has "clk" in the left side of the screen but there's no green line indicating a value yet.

The following lines correspond to the code in Appendix A and Appendix B.

```
# simplest way to add a wave
add wave clk
```



Pro Tip: If your signals have long signal names, you can right click the signal inside Modelsim and use the `copy` option by right-clicking the signal. Then paste the name into your .do file.



5.1) Changing the radix of your output

```
# this register will be displayed in hex instead of binary
add wave -radix hex accumulator
```

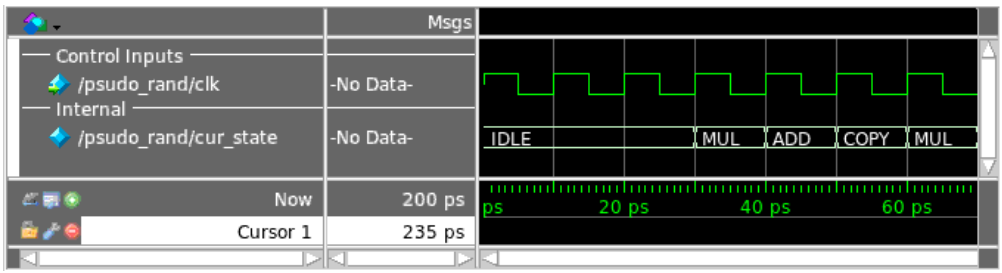
You can replace `hex` with any of the following values:

`bin`	`oct`	`dec`	`signed`
`unsigned`	`hex`	`ascii`	`symbolic`

Pro Tip:
`hex` is useful for data registers/busses of 8-bits or more.
`symbolic` is incredibly useful for state machines with enumerations. This is why we recommend using enums for your state machines whenever possible.

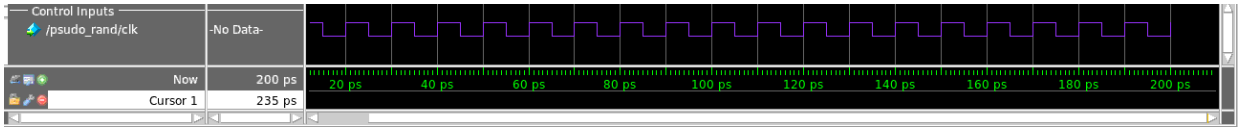
```
typedef enum
{
    IDLE,
    MUL,
    ADD,
    COPY
} state_t;

state_t cur_state;
```



5.2) Changing the color of your output

```
# this will make the clock signal appear purple
add wave -color purple clk
```



Here's a list of some valid colors, but there may be more

`red`	`orange`	`yellow`	`green`
`blue`	`purple`	`cyan`	`magenta`
`pink`	`white`	`grey`	`black`
`turquoise`			

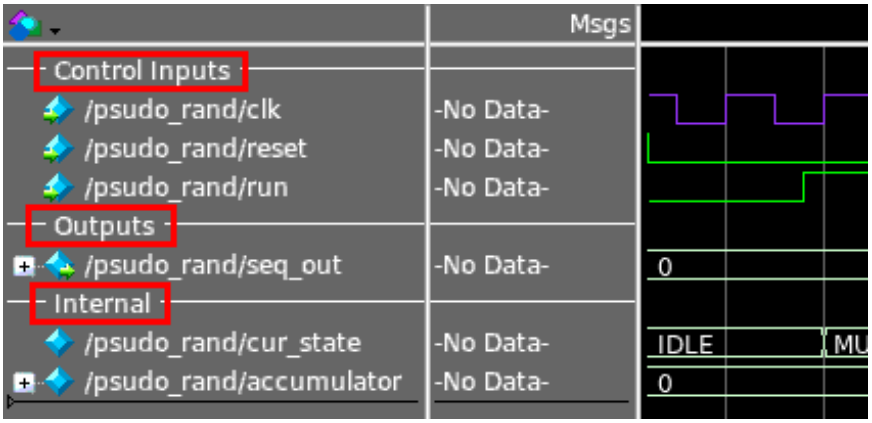
Wow... the guys at Modelsim gave us plenty of choices!

5.3) Adding dividers

Dividers add spaces between groups of signals and also provide some labeling. They are useful for projects with a lot of signals.

```
add wave -divider "Control Inputs"
```

The quotes are only necessary if your label contains spaces.



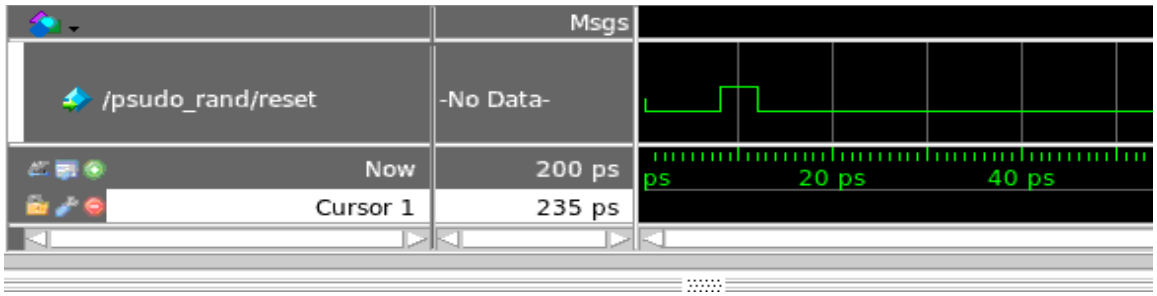
6) Setting values for your circuit's inputs

After you add waves to the wave window, you should stimulate your circuit with some test inputs. You can even overwrite registers that are normally driven by another part in the circuit.

```
# set a 1-bit input to a sequence of values
# value time, value time, value time...
force reset 0 0, 1 8, 0 12
```

Equivalently, you could use multiple force statements...

```
force reset 0 0
force reset 1 8
force reset 0 12
```



6.1) Periodic signals

You can generate periodic signals (like clocks) using the `-repeat` option

```
# periodic signal
# everything up to 10 time units will be repeated forever
force clk -repeat 10 1 0, 0 5
```

6.2) Force value radix

You can write constants in hex or decimal by preceding your constants with **16#** and **10#** respectively. The default radix is binary. This abuse of the # symbol is why comments cannot begin in the middle of a line. Note that this will NOT change the radix displayed in the wave window (see **5.1**).

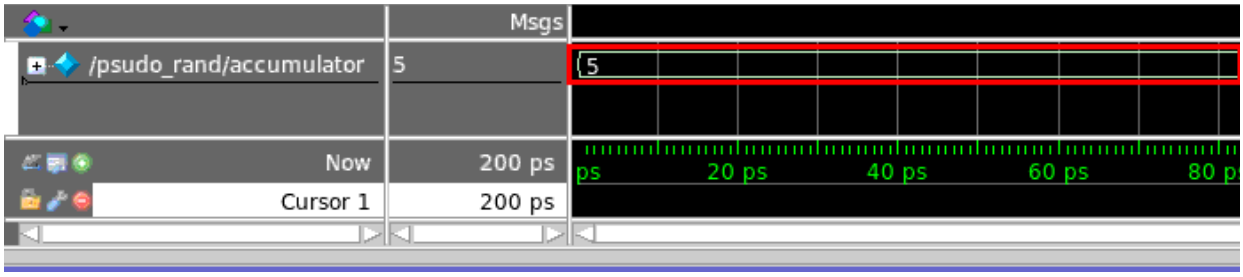
```
# force accumulator to 0xE at time 0
force accumulator 16#E, 0
```

```
# force accumulator to 14 (decimal) at time 0
force accumulator 10#14
```

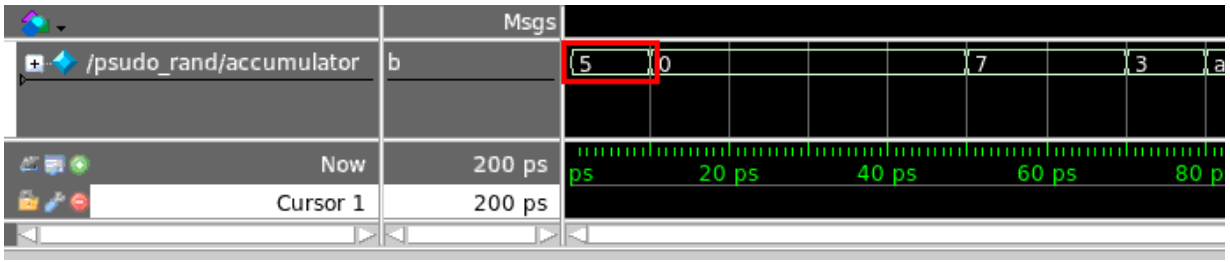
6.3) Overriding outputs with -deposit

If you use a basic `force` statement on a wire or register, that wire or register will maintain that value forever even if the circuit writes a new value. You can use the `-deposit` option to allow the circuit to write new values later.

```
# circuit can no longer update accumulator
force accumulator 101 0
```




```
# accumulator will only hold 0x5 until overwritten by the circuit
force -deposit accumulator 101 0
```



7) Running the simulation

At the end of your .do file you must execute the `run` command or else the window will be blank. You can specify the time units after the `run` command.

```
# run the simulation for 200 ps
run 200
```

```
# run the simulation for 200 ns instead of the default
run 200ns
```

Appendix A – psudo_rand.sv

```
/* psudo_rand.sv
 * File created Feb 3 2016
 *
 * Generates a sequence of psudo-random numbers 0, 7, 10, 9, 4, 11...
 * outputting a new number every 3 clock cycles.
 * Each number in the sequence is the 5 * (previous number) + 7 truncated
 * to fit within a 4-bit register */
module psudo_rand
(
    input   logic       clk,
    input   logic       reset,  // reset the state machine and zero all registers
    input   logic       run,    // sampled on the rising edge of clock in the idle and copy states
    output  logic[3:0]  seq_out // the random sequence appears here
);

    typedef enum
    {
        IDLE,
        MUL,
        ADD,
        COPY
    } state_t;

    /* Internal registers */
    state_t     cur_state;   // the operation being performed right now in the current cycle
    logic[3:0]  accumulator; // intermediate results

    /* Next state calculation */
    always_ff @(posedge clk) begin

        if (reset) begin
            cur_state <= IDLE;
        end else begin
            case (cur_state)
                IDLE:
                    if (run)
                        cur_state <= MUL;
                MUL:
                    cur_state <= ADD;
                ADD:
                    cur_state <= COPY;
                COPY:
                    if (run)
                        cur_state <= MUL;
                    else
                        cur_state <= IDLE;
            endcase
        end
    end

    /* Random number generation */
    /* Every 3 cycles, accumulator (and seq_out) become
     * (5 * (old accumulator) + 7) % 16 */
    always_ff @(posedge clk) begin

        if (reset) begin
            accumulator <= 4'h0;
            seq_out <= 4'h0;
        end else begin
            case (cur_state)
                MUL:
                    // fast way to multiply by 5 (shift by 2 then add)
                    accumulator <= (accumulator << 2'h2) + accumulator;
                ADD:
                    accumulator <= accumulator + 4'h7;
                COPY:
                    seq_out <= accumulator;
            endcase
        end

    end

endmodule
```

Appendx B – psudo_rand.do

```
# psudo_rand.do
# file created Feb 3 2016

# ECE 385 Model Sim .do file tutorial
# Comments have a '#' as the first character of a line

# clear your old waveform, and start it at time t=0
# if you get an error that says "could not find the wave window" then just open a wave
# in modelsim: view->wave
delete wave *
restart -f

# the lines below add signals to the wave window, but do not give them any values
add wave -divider "Control Inputs"
add wave -color purple clk
add wave reset
add wave run

add wave -divider "Outputs"
add wave -radix hex seq_out

add wave -divider "Internal"
add wave cur_state
add wave -radix hex accumulator

# the following lines actually assign values to the signals declared above
force clk -repeat 10 1 0, 0 5
force reset 0 0, 1 8, 0 12
force run 0 0, 1 28, 0 128, 1 172

# run the simulation for 200 ps (if ps is the default time unit)
run 200
```