EECS 470 Verilog Tips

Contents

1 Introduction

This document is intended to provide a high-level overview of some helpful SystemVerilog tips, from basic constructs to more complex editing modes. You don't have to use everything here.

2 Basics

2.1 Sequential Vs Combinational

In short: sequential logic corresponds to flip flops / registers, and combinational logic translates to logic gates. Sequential logic only updates at the positive edge of the clock (or whatever edge you define it to trigger on) while combinational logic updates its output as soon as its inputs change.

	Combinational	**Sequential**
Updates…	as soon as inputs change	at positive clock edge
Procedural block	`always_comb`	`always_ff`
Assignment Operator	Blocking (`=`)	Nonblocking (`<=`)
Hardware cost	Small	Medium

Table 1: Differences between combinational and sequential logic

Sequential signals are of type "`reg`" while combinational signals are of type "`wire`". However, in SystemVerilog you should declare all signals of type "`logic`" and let the compiler figure out the type. Even though wires and registers are defined as the same type, it is very important to understand what type each signal behaves as.

2.2 Synthesizability

All hardware you write must be **synthesizable**. This means it must be possible to take the behavioral verilog you write and synthesize it into structural verilog composed of basic logic gates.

The process of synthesizing designs adds a hardware cost to different operators. By cost, we mean area, delay, and power of the resulting hardware gates. For example, multiplication (`*`) is much more expensive than a bitwise or (`|`). Additionally, modulus by a power of two is much cheaper than modulus by a non-power of 2. A good hardware designer understands the implicit cost of each character they type.

Here are some *non-synthesizable* constructs:

- Certain procedural blocks (`initial`, `repeat`, `forever`)
- Giving multiple assignments to the same signal (this creates a short between gate outputs!)
- Some system calls (which begin with $).
  - Some can only be used with compile-time constants (ex: `$clog2`)
  - Others are fully non-synthesizable (ex: `$time`)
  - And others still are synthesizable but may have an unknown hardware cost (ex: `$countones`)

And here are some things that causes issues with synthesis:

- Latches! Aka a wire that doesn't have an explicit assignment in all cases
  - Check the synthesis output for "Latch", such as by using the `grep` command on the output log
- Combinational loops
  - These can be tricky to track down as they may span multiple modules
  - The synthesis script will notify you of a "timing arc" in its output

3 Data Types

3.1 Enums

Enums are useful for defining a set of possible values for a signal. This is most commonly used for states in a finite state machine (FSM). Assigning bit encodings to specific states is allowed, but not necessary.

Here is a two-state enum with no bit encodings defined:

```
typedef enum logic { LOCKED, UNLOCKED } lock_state;
```

Here is a 2-bit enum with encodings explicitly defined:

```
typedef enum logic [1:0] {
    BYTE   = 2'h0,
    HALF   = 2'h1,
    WORD   = 2'h2,
    DOUBLE = 2'h3
} MEM_SIZE_T;
```

3.2 Structs

Structs are useful for grouping signals that often stay together. This could be module inputs and outputs, a description of the rows in a memory structure, or a description of a way to interpret data (such as a program instruction).

Structs can single bit signals, multi-dimensional arrays, and even other structs. Just be careful about growing the size of a struct too large.

Here is an example of a struct containing other custom-defined types:

```
typedef struct packed {
    REG_IDX_T      Tnew;
    REG_IDX_T      Told;
    ARCH_REG_IDX_T arch;
} ROB_ENTRY_T;
```

The number of bits in this struct can be given by the system function `$bits(ROB_ENTRY_T)`.

3.3 Unions

Unions provide a way to interpret the same set of bits in multiple ways. This allows the same set of bits to be interpreted in different ways based on the context.

Here is an example where a 64-bit line of memory line can be interpreted as either 8 bytes, 4 half-words, 2 words, or one double.

```
typedef union packed {
    logic [7:0][7:0]  byte_level;
    logic [3:0][15:0] half_level;
    logic [1:0][31:0] word_level;
    logic      [63:0] dbbl_level;
} MEM_BLOCK_T;
```

3.4 Packed and Unpacked Arrays

Packed arrays have all dimensions defined on the left hand side of the signal name, while unpacked arrays have all but one dimension defined on the right side of the signal name:

Packed Array	Unpacked Array
`logic [DIM1-1:0][DIM2-1:0] signal_name;`	`logic [DIM2-1:0] signal_name [DIM1-1:0];`

Explicitly, the difference between the two is that packed arrays are laid out contiguously in hardware, whereas unpacked arrays are distributed. What this means is that a packed a array can be set to all zeros (or ones) with one operation, while unpacked arrays require setting each unpacked dimension individually.

For the purposes of this project, we recommend just using packed arrays since unpacked arrays can often cause unintended issues.

4 GUI Debuggers

Creating a GUI debugger can be very helpful for visualizing the state of a design and quickly finding issues. These are a large undertaking and are only recommended if you have a member of your group with significant software experience. Moreover, it is most helpful to your project if you finish it very early so that it can be used as soon as you begin integrating your processor.

The development can be logically split into two sections: the backend and the frontend. The backend is responsible for taking the state of your processor and making it accessible to the frontend for display to the user. The frontend is responsible for cleanly displaying the data to the user.

4.1 Backend

Here are some previously used options for the GUI debugger backend:

- Dumping state to a `.vcd` or `.vpd` file. These are open-source and easy to parse.
- Dumping state to a `.fsdb` file. This format is more compressed than vcd files, but is harder to parse.
- Running simulation live through the `ucli` interface provided by VCS.
- Creating your own custom output through dumping desire variables to a manually formatted output (ex: `.json file`)

To break structs into their individual fields, you need to use `.vpd` instead of `.vcd` files. VCS generates `.vcd` files by default, and you can take the following steps to instead generate a `.vcd` file:

1. Add `+memcbk` to the `$VCS` variable in the Makefile
2. Add the following two lines to set up the vpd file in your testbench. The first one specifies the dumpfile, and the second one specifies to dump all signals. This is in place of using the `$dumpvars` command

```
$vcdplusfile("<filename>.vpd");
$vcdpluson();
```

3. Run your simulation like normal (for example, `make <program>.out`)
4. Convert the vpd file to a vcd file using the Synopsys converter.
   - The executable is located at `/opt/caen/synopsys/vcs-2023.12-SP2-1/bin/vpd2vcd` and must be run with the `+splitpacked` argument.
   - To make this executable work, you need to export the `VCS_HOME` environment variable with the following line. Run it manually or add it to your `/.bashrc` file.

     ```
     export VCS_HOME=/opt/caen/synopsys/vcs-2023.12-SP2-1
     ```

   - Then the first argument is the vpd file, and the second is the filename of the output vcd file.
   - Alternatively, add the following as an alias, either manually or in your `/.bashrc` file:

     ```
     alias vpd2vcd="/opt/caen/synopsys/vcs-2023.12-SP2-1/bin/vpd2vcd +splitpacked"
     ```

     This allows you to do the conversion with the following: `vpd2vcd <filename>.vpd <filename>.vcd`

4.2 Frontend

Here are some previously used options for the GUI debugger frontend:

- Python (using a library such as PyQT)
- React
- Terminal-based programs

5 Assertions

The SystemVerilog Assertion (SVA) language is a powerful tool that allows you to precisely describe the correct functionality of a module. Although the syntax can be cumbersome to learn, it can make your testing run much more smoothly.

Assertions come in two flavors:

- **Immediate** assertions are called once in a testbench to verify correct functionality at one instant
- **Concurrent** assertions run on a clock and evaluate every clock cycle to verify continually correct functionality

Immediate assertions are fairly straightforward, so we will focus on concurrent ones here.

We'll first cover basic syntax and then provide some examples at the end of this section.

5.1 Syntax

Assertions can be described directly within an `assert` statement, or as a property that is later given as the argument to an assert statement. Describing them as properties first makes for cleaner code.

It is good practice to describe properties within a `clocking` block that specifies a default clock to use for timing the assertions. This generally looks like:

```
default clocking cb @(posedge clk)
    property property_name;
        // describe property here
    endproperty
endclocking
```

Then, once those properties are defined, you can assert them as follows:

```
assert property(cb.property_name);
```

`cb.property_name` describes the property with name `property_name` within the block named `cb`.

You can optionally give this assertion a name which will show up when it is triggered to make it easier to track down an assertion when it fails:

```
AssertionName:  assert property(cb.property_name);
```

Lastly, if you want to trigger some other action when an assertion fails (such as exiting simulation), you can add an `else` statement that is executed after the assertion is triggered:

```
AssertionName: assert property(cb.property_name)  else some_other_task;
```

5.2 Implication

The backbone of assertions are the implication operator. Assertions work by first waiting for the antecedent sequence (the logical statement or sequence of statements prior to the implication operator) to be true, and then verifying that the consequent sequence that follows the implication operator. Both sides of the implication can be a description of a combination of signals on one cycle, or a sequence of statements spanning multiple cycles.

There are two flavors of implication operators:

- `|->` The antecedent and consequence must be true the *same* cycle
- `|=>` The consequence must be true the cycle *after* the antecedent is true

5.3 Timing

Since we work with sequential logic, assertions also can also capture sequential logic. This is done with delays that can be a set number or range of cycles. Delays are described with `##` and ranges are encapsulated by brackets `[ ]`. For example:

- `##2` indicates a delay of 2 cycles
- `##[3:5]` indicates a delay of 3-5 cycles

There are also system functions that can describe certain timing behaviors:

- `$rose()`: A signal was low the previous cycle and high this cycle
- `$fell()`: A signal was high the previous cycle and low this cycle
- `$past()`: get a signal's value in the previous cycle
- `s_eventually`: The sequence following this keyword must eventually be true after the prior sequence occurs

There are also symbols to indicate consecutive repeated sequences, non-consecutive repeated sequences, and combinations (intersections and unions) of sequences. However, we will not cover that here.

5.4 Assumptions

Sometimes, a module has certain assumptions on its inputs that are required to be true in order for the assertions to be true. These assumptions can be encoded as properties in the same way as assertions, but use the `assume` keyword instead of the `assert` keyword when instantiating them.

5.5 Covers

Coverage is a useful verification metric that describes how many relevant corner cases a given simulation tested. If a module has a certain edge cases it wants to ensure the testing team verifies (ex: wrapping around a circular buffer), that can be encoded by a cover property. Covers can also be captured through properties, but are instantiated with the keyword `cover`.

5.6 Example

```
// Define properties
default clocking cb @(posedge clk);
    // ----------- Assertions ----------- //
    // For a given reservation station entry, it will eventually issue after being
    // dispatched into, unless it was on a mispredicted path
    property dispatch_will_issue (logic [$clog2(RS_SZ)-1:0] i);
        dispatch_en[i] && dispatch_req[i] |-> s_eventually (iss_req[i] && iss_gnt[i])
                                                        || mispredict;
    endproperty

    // ----------- Covers ----------- //
    // Multiple instructions woken up simultaneously
    // One cycle after no entries are issued, there are at least two entries in the
    // reservation station that request issue and didn't request it the previous cycle
    property multiple_wakeup;
        !iss_gnt ##1 ($countones(iss_req ^ $past(iss_req))) >= 2;
    endproperty

    // ----------- Assumptions ----------- //
    property sq_tail_in_range;
```

```
        sq_tail >= 0 && sq_tail < `SQ_SZ;
    endproperty
endclocking

// Now specify properties as assertions, assumptions, and covers
// Naming convention is as follows:
//   P_* denotes an assertion
//   A_* denotes an assumption
//   C_* denotes a cover
generate
    for (genvar i = 0; i < RS_SZ; i++) begin
        P_EventuallyIssue:  assert property(cb.dispatch_will_issue(i));
    end
endgenerate
A_SQTailRange:        assume property(cb.sq_tail_in_range);
C_MultipleWakeup:     cover property(cb.multiple_wakeup);
```

6 Verilog-Mode

Wiring up modules is one of the more tedious parts of writing Verilog. Wouldn't it be great if there was a tool that could automatically do this? Well, it turns out that there is! Verilog Mode is an emacs extension that automates all the tedious parts of writing Verilog. It matches names and can use regular expressions to connect your modules together easily.

6.1 Running Verilog-Mode

Verilog-Mode can be called from the command line with the following command:

```
emacs <file_name> -f verilog-auto
```

If there are no errors, you can type `Ctrl-x` then `Ctrl-c` to exit emacs. If the file was modified, you need to type `y` to save the changes.

If you are a vim user, you can add a keyboard shortcut to call this command from directly within vim. To do this, add the following line to your `~\.vimrc` file

```
map <C-\> :!emacs % -f verilog-auto<CR>
```

This command maps the command `Ctrl-\` to the command that calls `verilog-auto` on the currently open file. Emacs will open (within vim), and then exiting emacs as described above will return you to the vim editor. You may be prompted to reload the buffer in vim which you can do by typing `l` after emacs closes.

6.2 Instantiating Modules

At the very simplest, you can use the `AUTOINST` command to automatically wire an instantiation of a module.

For example, you can write the following lines:

```
psel_gen
psel_gen (/*AUTOINST*/);
```

And the `AUTOINST` command expands the module instantion:

```
psel_gen
psel_gen (/*AUTOINST*/
          // Outputs
          .gnt                          (gnt[WIDTH-1:0]),
          .gnt_bus                      (gnt_bus[WIDTH*REQS-1:0]),
          .empty                        (empty),
          // Inputs
          .req                          (req[WIDTH-1:0]));
```

Here, a wire with the same name (and size) of each port was automatically connected to its corresponding port in the module definition. The widths of each wire are explicitly stated to match the width of the port.

6.3 Using Templates

However, you won't always want the wire names to exactly match the names of the ports. In this case, you can use templates to connect specific port names to desired wire names. Templates are described with the module name, then the `AUTO_TEMPLATE` keyword, optionally a regular expression capturing the name of the module instance, and then a wiring description of the module's ports.

Patterns are captured using regular expressions. You open a pattern with `\(` and close it with `\)`. Anything within the pattern is accessible later in the template, with `@` matching the pattern captured in the module instance name and `\1` through `\9` matching the 1st through 9th patterns described in a port name.

There are a couple of ways to specify connections within a template:

1. Connect exact port names to exact wire names
2. Use regular expressions to rename a group of wires in the same pattern
3. Use regular expressions to use the module name in individual wire names

Here is a simple example template:

```
/* psel_gen AUTO_TEMPLATE "psel_\(.*\)" (
    .req    (@_req),
    .gnt    (@_gnt),
    .gnt_bus(),
    .empty  (),
   );
*/
```

This template is doing a couple of thing:

1. `"psel_\(.*\)"` matches the name of the module instance and captures the pattern described within the parentheses. Here, the pattern is `.*` which matches any character repeated 0 or more times. We can later access this matching text with the `@` character. For example, if the module is named `psel_alu`, then the captured pattern is "alu".
2. `.req (@_req)` matches the port name `req` and connects it to a wire that pairs the pattern captured in the module instance name with the suffix `_req`
3. `empty ()` takes the port named `empty` and connects it to nothing.

Now, with the above template defined in the Verilog file,we can instantiate multiple priority selectors with different names and get the following:

```
psel_gen #(.REQS(1), .WIDTH(RS_SZ))
psel_alu (/*AUTOINST*/
          // Outputs
          .gnt                          (alu_gnt),              // Templated
          .gnt_bus                      (),                     // Templated
          .empty                        (),                     // Templated
          // Inputs
          .req                          (alu_req));              // Templated

psel_gen #(.REQS(1), .WIDTH(RS_SZ))
psel_mul (/*AUTOINST*/
          // Outputs
          .gnt                          (mul_gnt),              // Templated
          .gnt_bus                      (),                     // Templated
          .empty                        (),                     // Templated
          // Inputs
          .req                          (mul_req));              // Templated
```

Lastly, here is a more complicated example with a different module named `vldRdySlice` that adds a pipestage between two sections of combinational logic while maintaining backpressure between the two.

```
    /* vldRdySlice AUTO_TEMPLATE "\(.*\)_vldRdySlice" (
        .src\(.*\)  (@\1_pre),
        .dst\(.*\)  (@\1),
        .clr        (mispredict),
        );

    */
```

```
vldRdySlice (.WIDTH($bits(ALU_DATA)))
alu_vldRdySlice (/*AUTOINST*/
                 // Outputs
                 .src_rdy           (alu_rdy_pre),          // Templated
                 .dst_vld           (alu_vld),              // Templated
                 .dst_data          (alu_data),             // Templated
                 // Inputs
                 .clk               (clk),
                 .rst               (rst),
                 .clr               (mispredict),           // Templated
                 .src_vld           (alu_vld_pre),          // Templated
                 .src_data          (alu_data_pre),         // Templated
                 .dst_rdy           (alu_rdy));              // Templated

vldRdySlice (.WIDTH($bits(MUL_DATA)))
mul_vldRdySlice (/*AUTOINST*/
                 // Outputs
                 .src_rdy           (mul_rdy_pre),          // Templated
                 .dst_vld           (mul_vld),              // Templated
                 .dst_data          (mul_data),             // Templated
                 // Inputs
                 .clk               (clk),
                 .rst               (rst),
                 .clr               (mispredict),           // Templated
                 .src_vld           (mul_vld_pre),          // Templated
                 .src_data          (mul_data_pre),         // Templated
                 .dst_rdy           (mul_rdy));              // Templated
```

6.4 Auto Wiring

The last feature of verilog-mode that we'll cover is one that can automatically create internal wires, inputs, and outputs for a given module. This is very helpful when using verilog-mode to build module hierarchies.

The keywords relevant are:

- `AUTOWIRE`: define wires for all automatically wired signals contained within this module.
- `AUTOINPUT`: Define all automatically created inputs. This includes all inputs to automatically instantiated modules that do not come from another module's output port.
- `AUTOOUTPUT`: Identical to `AUTOINPUT` except for output ports.
- `AUTOARG`: Read all declared inputs and outputs (created either manually or automatically) and add them to the module port list.

Using the `AUTOINPUT` and `AUTOOUTPUT` commands are very helpful for checking that you've wired all internal signals that you've intended to. If any signals unexpectedly appear on the input or output list, that means you forgot to wire them to the correct place.

To make the wires default to type `logic`, add the following three lines after the `endmodule` keyword at the end of your Verilog file.

```
// Local Variables:
// verilog-auto-wire-type: "logic"
// End:
```

You can also tell verilog mode to recognize every type name that ends in `_T` as a custom type:

```
// verilog-typedef-regexp: "_T$"
```