Verilog Code Review

Scott Borisch

Jan 2021

Review Types

- Code Review
  - vs.
- Architecture Review
  - This will cover elements of both

High Level Issues to review

- Language choice – Verilog vs. System Verilog
- System Verilog is a superset of Verilog
  - System Verilog adds a lot of nice features
  - I see no reason not to use it (other than lack of tool support)

System Verilog Features

- Features that are useful in typical rtl designs
  - Packages
  - Parameters and Localparams
  - Typedefs
    - Enums
    - Structs
    - typecasting
  - Functions
  - Generate Statements
    - Condition Generates
    - For Generates

System Verilog Features

- Conditional Generates vs. IFDEF
- Typedef enums are very nice for state machines

System Verilog Features

- reg vs. wire vs. logic
- reg and wire are legacy verilog
  - always @*, always @(posedge clk) => reg
  - Assign statements, outputs drive by module substantiations => wire
- This distinction offers no value (IMO), and just complicates your code.
- Use logic instead – doesn't care how a signal is driven
- For port signals, it is possible to drop the reg/wire/logic entirely. This however makes it inconsistent with internal signal declarations. I recommend all signals are declared as logic, logic [x:0], or as typedefs.

System Verilog Features

- Packages: use packages aggressively for values (localparams)/data types (typedefs) that are shared between multiple modules.
- Example:
  - localparam int k_FIFO_DEPTH = 256;
  - localparam int k_FIFO_DEPTH_LOG2 = $clog2(k_FIFO_DEPTH);
  - typedef logic [k_FIFO_DEPTH_LOG2-1:0] fifo_wrd_cnt_t;
- For example, if you have a module that does not instantiate a given fifo, but it is using the fifo's depth to control when it writes/reads the fifo.

System Verilog Features

- Localparams and Parameters – implicit vs. explicit type declaration
  - localparam k_FIFO_DEPTH = 256; // vs.
  - localparam int k_FIFO_DEPTH = 256;
- Both are legal but the second makes it clear you expect an integer value.
- For parameters, this has the extra advantage that if a parameter override has the wrong datatype, it can be error checked (so long as the tool is smart enough to do this)

System Verilog Features

- Using this together with typecasting is useful:
  - localparam int k_CMD_WRD_SIZE = 8;
  - typedef logic [k_CMD_WRD_SIZE-1:0] cmd_word_size_t;
  - typedef enum logic [k_CMD_WRD_SIZE-1:0] {

    k_WRITE_CMD = cmd_word_size_t’(0),

    k_READ_CMD = cmd_word_size_t’(1)

    } cmd_wrd_t;

System Verilog Features

- Conditional Generate logic:
- generate // keyword
-   If (SOME_VALUE == 1’b1) begin : LabelOne
-     // code if true
-   end : LabelOne
-   else begin : LabelTwo
-     // code if false
-   end: LabelTwo
- endgenerate

System Verilog Features

- For Generate logic:
- genvar zdx; // The “generation loop index variable”
- generate // keyword
-   for (zdx=0;zdx<8;zdx=zdx+1) begin : LabelOne
-     // code for EACH iteration of the loop
-   end : LabelOne
- Endgenerate
- Useful to generate identical/nearly identical code N times
  - You can use conditionals based on the genvar to alter the logic as a function of the loop number if needed.

Design Conventions

- Minor issues
  - Assigning default values to all signals (where possible):
    - Assigning a default value to signals driven by always @(posedge clk) – that is, the output of a flip-flop => This can be implemented (as a power up value) by the PAR (place and route tool) such as Quartus.
      - If you don’t specify a default, tools will generally assume zero. However, the simulator will treat unspecified defaults as “X”. Therefore, always specifying the default is good to minimize differences between simulation and the actual FPGA design.

Design Conventions

- Naming Conventions
  - There is no such thing as universal RTL naming conventions. Having worked at four companies, esp. those where there has been a lot of acquisition of other companies, I treat the goal of having company wide naming conventions as mostly a pipe dream.
    - It is much more important to focus on naming conventions within a single team, where multiple authors are contributing code to the same FPGA design.
    - Only insist on conventions for the most important issues. If you have too many rules, the more likely they will be ignored.

Design Conventions

- Some Naming/Design Style Conventions I Find Worthwhile:
  - Port signals – for all port signals other than clocks and resets (which are almost always inputs), use _In and _Out suffixes.
    - This makes module instantiations much easier to understand.
  - Multiple clock domains
    - Minimize the number of unique clock domains (where possible)
    - Try to isolate multiple clock domains to as few modules as possible
    - For modules where multiple clock domains are required, add a _<clock_name> suffix for ALL signals (other than the clocks themselves).

Design Conventions

- Some Naming/Design Style Conventions I Find Worthwhile:
  - Debug signals – include _Dbg or similar as a suffix for debug signals.
  - Signals that are simply clock cycle delayed versions of other signals.
    - A common convention here is to add _dX to the signal name:
    - **always** @(posedge Clk) **begin**
    -   my_sig_d1 <= my_sig;
    -   my_sig_d2 <= my_sig_d1;
    - **end**

Design Conventions

- Some Naming/Design Style Conventions I Find Worthwhile:
  - Although verilog/system verilog allows for “one-liner” **if statements**, I strongly discourage these.
    - It is way to easy to overlook this and miss that you need to add a begin/end if you need to add more logic to a one liner if statement.
  - Use _n for active low signals:
    - cs_n, not cs for active low chip select.

Design Conventions

- Registering Outputs
  - It is a good idea to always register all outputs of a module – this helps with static timing.
  - For more demanding designs, this rule may extend to all (almost all) inputs.

System Verilog Features

- Personal preference – propagation of X/Z values
  - The following is more of a personal preference (there are pros/cons) and I would not debate if someone disagrees with these:

System Verilog Features

- Personal preference – propagation of X/Z values
  - Use of logical OR/AND without use of equality comparisons:
    - always @* begin
    -   error_next = (error1 || error2);
    - end
    - vs.
    - always @* begin
    -   error_next = ((error1 === 1’b1) || (error2 === 1’b1));
    - end
- Note the use of the “triple equals” – this approach means that even if error1/error2 take on the values ‘Z’ or ‘X’, this value will not propagate to error_next. I find this makes it easier to locate which signals are the “aggressor” (the one you actually need to fix) vs. ”victim” signals

System Verilog Features

- Use of sync vs. async resets.
- Too much to go into for this presentation – I can discuss with Andrew offline – can also cover this in another presentation.

System Verilog Features

- Module names vs. filenames
  - Although System Verilog (and most tools) don’t require this, I find it much easier to follow a design if
    - There is only one module per text file.
    - Module name and filename are equivalent
      - Module **my_module** is in file **my_module.sv**
      - Keep consistent capitalization.
      - This is one rule I have seen followed everywhere I have worked.
      - Admittedly, Xilinx/Altera have some autogenerated code that does not follow these rules.
- Same thing applies for package files, not just module files.