A Modern Approach to System Verilog

Mark Oskin
Professor
School of Computer Science and Engineering
University of Washington

Rev Winter 2024.1

System Verilog coding advice

This guide is intended to serve as advice on how to write efficient, easy to maintain, easy(ier) to debug synthesizable Verilog. There are other high quality guides that I recommend reading, notably those by Scott Hauck and Michael Taylor. My approach is slightly different than these two guides, however, and so perhaps the advice in here will be useful for you. This guide uses the idea of writing an in-order RISC style processor as an example, but the recommendations carry forward to any digital design.

Adopt a naming convention and stick to it

Software developers use one of four ways to name their variables and functions: bsd_style, camelCaps, typeFunction and anythinggoes. The first two are the most common. Unless I am working within an existing project where camelCaps or typeFunctoin styles are mandated I prefer bsd_style, but camelCaps or typeFunction are fine too. I also use very wordy variable, function and module names. For example, in my code a statement like this can be found:

```
exec_result_comb = execute(
    cast_to_ext_operand(rd1),
    cast_to_ext_operand(rd2),
    cast_to_ext_operand(decode_out_imm),
    decode_out_pc,
    decode_out_op_q,
    decode_out_f3,
    decode_out_f7);
```

To me this statement is completely obvious what it means just by reading it. The result of execution (combinatorial logic) is from the execute function. Input operand register data 1 is cast to the input type needed. Various fields from the result of the decode stage such as PC and operand (q or clocked) are passed in as well. The odd names f3 and f7 are parts of the RISC-V instruction set and so have semantic meaning as well. In other words this block of code can be directly read and understood in isolation *because* the variable names are wordy.

Now there is a case *against* long variable names and that is synthesis tools will name items in the netlist by the full module depth / variable name. Long names lead to items in the netlist that do not display well on a monitor. I don't agree with this argument but it is made at times. My counter to this argument is good naming practices will lead to less debugging time both in the front end and back end of development.

Avoid random constants in your code

In software projects I outright ban the use of constants inserted in the made code. Anything worth having a number for is worth giving a semantic name to. For example I prefer this:

```
typedef logic bool;
localparam logic true = 1;
localparam logic false = 0;
```

Similarly for more complex application specific constants I prefer human-readable forms:

```
localparam f7_add = 7'b0000000;
localparam f7_sub = 7'b0100000;
```

I *never* want to look at code like:

```
if (inst[31:24] = 7'b0000000)
```

Instead, this code should be written as:

```
if (decode32_funct7(instruction, format) == f7_add)
```

When to use localparam vs define? For me I prefer to use localparam whenever possible because I believe it makes the code easier to read. I'll even go so far as to duplicate the definition of a localparam value (from the perspective of the compiler) by using it in includes. For example, here's a file I often include in my code:

```
`ifndef _base_
`define _base_

// Useful macros to make the code more readable
localparam true = 1'b1;
localparam false = 1'b0;
localparam one = 1'b1;
localparam zero = 1'b0;

`endif
```

Note the *required* use of `define around the include guard.

Make heavy use of functions

This is a controversial approach, but I believe modern System Verilog is best written by adopting what parts of good software engineering practices can apply. To this end functions reduce code duplication and hence lead to less bugs. Moreover I find the "result = function(…)" more intuitive to read than instantiating a module with inputs and outputs. In general, my rule is if a module is purely combinatorial logic it should be a function instead --- period. Moreover, I believe in many small functions. I've written several different processors and many hardware projects. I always start by writing a function library. For processors this often means a library of functions for decoding and executing the instruction set.

Function return types should be explicitly defined in your code. Don't use the implicit definition support Verilog says it has. This doesn't work for all tools.

```
function automatic bool take_branch(…
```

Make heavy use of typedef, enum and struct

Just as with software development languages, types help to make code more readable and maintainable. For example, consider this type:

```
typedef logic [`word_size:0]                ext_operand;
typedef logic [`word_size - 1:0]            operand;
// Must match instruction encoding
typedef enum logic [2:0] {
     f3_ext_m_mul = 0
    ,f3_ext_m_mulh = 1
    ,f3_ext_m_mulhsu = 2
    ,f3_ext_m_mulhu = 3
    ,f3_ext_m_div = 4
    ,f3_ext_m_divu = 5
    ,f3_ext_m_rem = 6
    ,f3_ext_m_remu = 7
}   f3_ext_m_op;

function automatic ext_operand alu(ext_operand in1, ext_operand in2,
funct3 f3, funct7 f7); begin
…
    else if (`enable_ext_m && f7 == f7_ext_mul) begin
        case (cast_to_ext_m(f3))
      …
```

The code above I trimmed (…) to illustrate the point. The ext_operand, which in this case is just one more bit than the word size of the machine helps to differentiate it from an ordinary word. Similarly the enum type helps to make the decode of the f3 field easier to read. While not required it is helpful for efficiency reasons to force the enum values to match the values in the instruction set. This makes the cast_to_ext_m(…) function a no-op in practice; i.e., it makes the code easier to read and debug but it reduces down to wires.

Structures are another super useful feature to use. For example:

```
typedef struct packed {
    logic [31:0]    addr;
    logic [31:0]    data;
    logic [3:0]     do_read;
    logic [3:0]     do_write;
    logic           valid;
    logic [2:0]     dummy;
}   memory_io_req;
```

Here I am packaging up all the control signals which go from the processor to a memory module into a structure. Besides collecting all of the variables into a single location, which gives them a combined

semantic meaning, structures allow you to easily extend a module input and output parameters without modifying a ton of code. Namely, consider the declaration of the memory module now:

```
module memory #(
    parameter size = 1024                          // in bytes
    ,parameter initialize_mem = 0
    ,parameter byte0 = "data/data0.hex"
    ,parameter byte1 = "data/data1.hex"
    ,parameter byte2 = "data/data2.hex"
    ,parameter byte3 = "data/data3.hex"
    ) (
    input   clk

    ,input memory_io_req  req
    ,output memory_io_rsp rsp
    );
```

Now if I want to add more inputs or outputs, for debugging say, instead of modifying the module and its use everywhere I just edit the structures and the rest works out automatically.

Note there is one unfortunate thing about structures and modules and that is the *entire* structure is either an input or an output. The inout semantics do not work well here as tools such as verilator will think you are writing the same variable more than once, even though you are not, you are just writing different portions of the struct. System Verilog has *interfaces* to try and solve this problem, but I stay away from interfaces in my code because while you can pass arrays of structures to modules and all modern tools will synthesis that code, arrays of interfaces is a limitation of current tools (2023).

Note the placement of "," above. I adopted this style after reading Michael Taylor's design guide. I like it and have even carried it over to most of my software development.

Further note the "dummy" variable in the structure. This is used to pad the structure to 4 bit alignment. I do this because my preferred waveform viewer (gtkwave) is rather simple and unless you go through the work of teaching it about types it just treats structures as a bit vector. This will make the display of the structure unreadable as it considers the last bit in the structure as the lowest bit of the bitvector. The dummy is there to pad things out and make hex display of the structure reasonably sane. This brings me to my next point.

Finally, I consider structs so important for readability that I refuse to let outside requirements mess up my code. For example, tools like Vivado want you to play ball with how AXI interfaces are declared. Your module will get a lot of independently named signals for an AXI interface. I immediately "pack" these signals into input and output structs:

```
axi_pack_64_32_in axi00_pack_in(`pack_in_interface(m00), …
```

I then pass around the structs within my codebase. See below on templatized types regarding the pack_in_interface syntax used in the example here.

Use templates

Modern Verilog supports templates (in a way). For example you can write a generic FIFO module like this:

```
module fifo #(
             parameter fifo_depth = 2
            ,parameter type fifo_type = logic) (
             input logic     clk
            ,input logic     reset

            ,input fifo_type in
            ,input logic     in_valid
            ,output logic    in_ready

            ,output fifo_type out
            ,output logic    out_valid
            ,input logic     out_ready
            );
```

The type here for the template is passed in as a parameter. Any time you are writing a base class you may reuse, make it a template.

Note that I also take this a step further and use templatized type generation as well. Here's an example:

```
`define def_axi_write_bus_in(addr_width, data_width) \
typedef struct packed {  \
    logic aready;\
    logic ready;\
    logic bvalid;\
} axi_write_``addr_width``_``data_width``_in
```

This is a lot messier looking code than the built in support for template modules, but as yet I don't know another way to do it.

Synthesis tools are really good these days and lean on them for where they are strong

All of this use of functions, structures, types, etc could lead to inefficient logic, *if it was 1990*. But it is 2023 and tools are very good at applying common optimizations such as dead code elimination and common subexpression elimination. The means the advice I wrote up above doesn't lead to inefficient logic. Go for readability and maintainability in your code.

This also means that I do not write structural code for common idioms like adders and even multipliers. I use + and *. If the synthesize tool isn't giving me what I want I first try constraints. If that doesn't work I'll break down and look into adding a structural component. To me, readability of the source trumps everything.

Use one combinatorial and one clocked always block per module, except when it is best to break this rule

This is perhaps the most controversial advice I'll write. Most people will say have no more than one combinatorial and one clocked logic always block per module. These people are generally correct. For most service type modules (fifos, memories, routers, etc) I stick to this rule.

Violating this rule can lead to some difficult to debug Verilog code. The issue is most Verilog simulators when they see two combinatorial always blocks in a module will execute them in some order and changes to logic created in one always block may not cause the other always block to be "re-run" to update the logic output. What this means is simulation will not match reality where the logic is truly being executed concurrently.

That being said there is one place I violate this rule, routinely and I rely on the fact that simulators will execute always blocks in the order they are written in the Verilog source file. This is in "top" level modules (not necessarily the true top of a design). An example of this is a five stage pipelined CPU. Here I may have five combinatorial always blocks and likely five or more clocked ones. I do this because for me, I find it easier to develop and debug the CPU module, which has numerous wires within it, if I keep the logic for the stages together.

Alternatively one can build a module per stage and then using structures wire these modules together. Here however you really are depending on the order the modules are written in your source file for it to work correctly. So whether you write multiple always blocks or use the module approach, your code is making use of the subtle ordering relationship for it to correctly simulate.

Build in debug support

I rarely use a waveform viewer. How do I get away with this? I build a debugger into almost every module I write. Almost every module will have an added always_ff block that uses $display syntax to print useful information, or always_comb block that uses explicit validation checks to check for failed assertion conditions. For code maintainability you want to logically turn these debug sections off when not in use rather than comment them out. A simple way to do this is to surround them with an if statement.

When do I use a waveform viewer? Primarily when I need to match the specification for an external interface. Specifications just have to be met and sometimes the mix of synchronous and asynchronous signaling involved can be tricky to get correct. A waveform viewer is very handy here. Another area where a waveform viewer is handy is when there's interaction between modules (or large blocks) that is suspect. For example the forwarding networks in a processor and how they interact with stalls.

Other good practices that are elaborated elsewhere

There are many other very good practices that Taylor's and Hauck's advice guides do a good job describing. For brevity I'm not going to elaborate in detail on these matters. I will have list a bunch of them here as reminders.

Do use:

- Pass by name. E.g. `,.rsp(rsps[control_done])`
- Use modern syntax: `always_comb, always_ff, logic,`
- Use `#ifdef` guards around header files

- Remember vectors are declared [high:low] and arrays [low:high]

Tools I use

I make extensive use of Verilator for debugging my design. Usually I either have no output except simulated I/O or if I am really stuck with my design I output all signals and view execution with gtkwave.

I am a huge believer in verilator lint. All my source code must pass verilator lint before I even compile it. Rarely, and I mean perhaps one in ten thousand lines of code will I use the escape mechanisms to turn off lint around a portion of code and even then I return to that section of my code repeatedly throughout the development process to see if I can refactor it to make it pass lint. In general, I consider not passing lint a design flaw.

For synthesis I end up using whatever open source (yosys) or vendor tools an FPGA requires (vivado, etc). I try and write code that will successfully be synthesized by at least these two tools with no tool specific escape hatches.