Verilog Style Guidelines for EECS 470 and Final Project Tips

By Doug Li *et al.*
October 14, 2009

1. Whatever you do, agree on it **as a team**. Set and agree upon unified policies in the first two team meetings (as well as brainstorm which advanced features to tackle and have a general idea what the top-level diagram might look like).
2. Design your top-level module as a team. This should include all inputs and outputs of the major modules. If this fits on one 8.5x11 sheet (even with small print) you're not being detailed enough. See point 7.b, found below.
3. When you have a high-level design for your pipeline, try walking a few instructions through the pipeline on a whiteboard. This will likely introduce extra signals/data you might need, cases you didn't consider, etc… You will undoubtedly need to go back to your design later to add things, but going through this in the beginning will likely save you time and pain.
4. Code your first module as a group. It will help immensely with getting everyone on the same page and working out coding standards as a team. For larger groups this can feel like a waste of time—it is worth it in the end. Classrooms have been found to be great places to gather for initial meeting. They have projectors and many have whiteboards.
5. Roughly at the third or forth meeting, agree on all the **namings** for intermediate wires which interface between components owned by more than one member. Wires inside your own modules – you can call whatever you want, but you might want to find a consistent naming convention so others can edit/debug your code.
6. http://www.eecs.umich.edu/courses/eecs470/Verilog/verilog_guidelines.pdf Provides some really basic guidelines for coding.
7. Since it is inevitable to naturally happen, **appoint members for specific tasks** based on their strengths and weaknesses. If you do not do this early, every member expects that some other member is doing it, when in actuality it will never get done.
   a. For instance, determine who has the strongest shell scripting skills. Trust me; this will come in handy.
   b. Assign one member the duty of *maintaining* a top-level diagram of the design. This can be done in MS Project, MS Visio, and similar programs. Do this early on!
   c. It may NOT be wise to assign a "Manager" role too early. Leave this as an "open position" in order to foster a positive atmosphere where any member may later rise to the occasion. This happens near the time of "design integration" I find.
   d. Since it IS bound to happen, everybody should put their teammates on speed-dial at the first meeting.
   e. Most importantly, try to have each member contribute equally. Problems in team-dynamics at the end of the semester may have been preventable by being open and honest about disclosing one's weaknesses early on.

8. Monitor your **disk usage**. Be sure you are using the 10 GB, AFS space provided by ITD instead of the tiny AFS provided under Engin (talk to CAEN hotline to have it changed). (This may only apply to people with older accounts, such as 5th year seniors, or grad students who did their undergrad here.) Type "fs lq" to list the quota for the fileserver. The "du" command will print out more detailed disk usage stats for a given directory (don't run it on your home directory, that'll be very lengthy).

9. Use a version control system such as **SVN** on your final project. Do not debate this point. Set this up as soon as possible, so just have whoever is most experienced with it, do it and send out an e-mail to everybody with the step-by-step one how to use it. Speaking of which, setting up a group e-mail address on UMOD may not be a bad idea.

10. Map **tabs to spaces**. This can be set in almost every editor, and if not, I recommend switching editors. For smaller modules 4 spaces per tab/indent may suffice, but generally **2 space indentation** through out. Be sure to use an editor that colors keywords.

11. Use continuous assignments over "always @*" blocks whenever possible and reasonable as to increase readability. If you use nested ?: assignments, format them carefully to maximize readability.

12. It is recommended to use "**_in**" and "**_out**" to differentiate inputs and outputs inside a module. Consider "_reg" for the identifiers which hold state.

13. Use tabbing to align codes in a readable fashion, where the bit-widths line up. (see example on last page)

14. Use "// Inputs" and "// Outputs", to further increase readability. (see example on last page)

15. Use lots and lots of **tick-defines**! These go in the "sys_defs.vh" file. But each team member should have his or her own section, for which they are responsible for. This typically will correspond with stages of the pipeline. Use all-caps and underscores for them. It is common and often necessary to have tick-defines based off earlier tick-defines, especially when determining bit ranges for parts of an "instruction bundle".

16. Much more powerful than tick-defines, and very useful in certain circumstances are **parameters**. In some ways, these are analogous to C++ templates. Certain information can be passed at the point-of-instantiation (e.g. bit-widths). Thus, it is used in module definitions of very general-purpose modules such as muxes, decoders, priority decoders, memory cells, etc.. Moreover, place such general-purpose components in a special utility subdirectory, say "verilog/misc", as these do not pertain only to one particular pipeline stage but are used by many. Depending on their usage, either all-caps or not.

17. Since memory arrays occur everywhere (especially in large ambitious projects). It may be advantageous to use the same core file. This would help with debugging and lowering of synthesis times. In each case, you instantiate one, and override the parameters accordingly, and write control logic encapsulating it. Make one with 1 read/write port, one with 2 read/write ports, one with 4 read/write ports, one with 4 read ports and 2 write ports, etc. to your heart's content. This way you only have to have ONE team member get it right ONCE.

18. Have a consistent scheme for each of the following: directory naming, pipeline stage naming, file naming, and variable naming. Use 2-letter stage acronyms: IF stage,

ID stage, RF stage, RS stage, RN stage, WB stage, CM stage, EX stage, etc.. For intermediate wires, follow the scheme such as "if_id_xxxx", "if_id_xxxx_in", of "if_id_xxxx_out". For things which hold state: "rs_xxxx_reg". Use a supporting subdirectory structure, since each stage will likely have multiple files. Instead of being in a subdirectory, leave files such as "pipeline.v", "icache.v", and "dcache.v" outside.

19. Use the `**SD** in synchronous non-blocking assignments.

20. Leave about TWO weeks prior to the deadline, for an all-hands-on-deck debugging effort. If you have tons of faith in your team's debugging skills, one week *may* suffice if you want to live life as dangerously as I had.

21. Note that you can modify a testcase (which you are stumped on) by commenting out lines or inserting a HALT. This can help to pinpoint the bug (classic "divide and conquer" strategy).

22. Code things up using "generics" (such as tick-defines, parameters, for-loops, generate-blocks), so that in the end you can tweak these values to optimize performance, which is very important in the final grading.

23. Perform full "integration" shortly after the second milestone. Typically one member will be doing a majority of this effort, since having too many people handle the code at this juncture could prove disastrous. Before this, consider using dummy stages as placeholders.

24. Since the load/store unit is usually one of the last modules that groups get working, it is not a bad idea to temporarily design your pipeline to just launch memory accesses when they hit the head of the ROB. This is simpler and will allow you to debug your pipeline with test cases that access memory while the load/store unit is still being designed and/or implemented.

25. When it comes to team work, hope for the best and expect the worst. It is not uncommon for teams to encounter internal disputes, especially when everybody is sleep deprived. DO identify and arbitrate disagreements among your team members. *If team work issues **seem** to be a problem, talk to the course instructor ASAP.* Each semester there are problems. Those groups that let them fester usually struggle the most.

26. Remember, the GSI/IAs have been though all of this, feel free to ask us for advice on anything from Verilog coding, using SVN, shell scripting, or repairing team dynamics. Chances are, one of us has been in a very similar situation or heard about it in previous semesters. Oh, and ask us about using "generate-blocks", they're nifty.

27. If you are up to it, try something experimental; come up with something refreshing that wasn't necessarily covered in lecture. "Take chances, make mistakes, and get messy." Have FUN doing it all, don't kill each other, and remember – there is no dishonor in having to SLEEP!

28. Meet BOB. He is looking stylish today. Here is a snippet of him:

```
module BOB (// Inputs
            clock, reset, flush, read_in, write_in, br_return_in, pred_conf_in,
            PC_in, choice_write_in, choice_update_in, pred_dir_in,
            local_hist_in, prev_pred_hist,
            RAS_inv_idx_in, RAS_idx_valid_in, RAS_rec_idx_in,
            ID_in,

            // Outputs
            output_valid, br_return_out, PC_out,
            choice_write_out, choice_update_out,
            pred_dir_out, local_hist_out, update_pred_hist,
            RAS_inv_idx_out, RAS_idx_valid_out, RAS_rec_idx_out,
            ID_out, //ID_invalid_out,
            mustStall
            );

  parameter IndexLen = 4;
  parameter NumLines = 16;

  input                 clock, reset, flush;
  input                 read_in, write_in;
  input                 br_return_in;
  input                 pred_conf_in;
  input          [15:2] PC_in;
  input                 choice_write_in;
  input                 choice_update_in;
  input                 pred_dir_in;
  input           [4:0] local_hist_in;
  input           [4:0] prev_pred_hist;
  input           [3:0] RAS_inv_idx_in;
  input                 RAS_idx_valid_in;
  input           [3:0] RAS_rec_idx_in;
  input  [IndexLen-1:0] ID_in;

  output                output_valid;
  output                br_return_out;
  output         [15:2] PC_out;
  output                choice_write_out;
  output                choice_update_out;
  output                pred_dir_out;
  output          [4:0] local_hist_out;
  output          [4:0] update_pred_hist;
  output          [3:0] RAS_inv_idx_out;
  output                RAS_idx_valid_out;
  output          [3:0] RAS_rec_idx_out;
  output [IndexLen-1:0] ID_out;
//output                ID_invalid_out;
  output                mustStall;
```

(This is real EECS 470 code, from a previous semester. It is NOT a ROB. I wonder what it does…)