ECEn 620 Advanced Digital Design

The TL-Verilog Unit

A number of proposals have been made over the years for improving hardware design methodologies through the introduction of new languages and approaches. TL-Verilog represents one philosophy which espouses keeping an HDL but modifying its features to simplify many tasks. In the case of TL-Verilog it provides a *timing-abstract* way of coding, the promise being that once you have a large pipelined system done it is trivial to retime the system by changing how the pipelining is done. A typical figure of merit used in the TL-Verilog publications is number lines of code required (a) to express a pipelined system and (b) to make a change by retiming a pipelined system.

Where to start? There are so many resources and some are overlapping. Combined with this, TL-Verilog is new and so the tools and language are still being refined. Immersing yourself in the language and working through a ton of examples is the only approach that seems to work. So, do it!

Goals for the Unit

You will not become an expert on all the details of TL-Verilog (or TLV or short). Rather the goal is to learn enough of the language to:

1. Be able to understand and truly appreciate the philosophy behind it and contrast it with conventional HDL-based design and the newer HLS-based design approaches.
2. Be able to do some basic design in TLV and be able to test out what you have learned.

Step 1: Do Some Reading

Start with the paper Timing-Abstract Circuit Design in Transaction-Level Verilog.
Read https://www.linkedin.com/pulse/transaction-level-verilog-worlds-first-hardware-language-hoover/ [https://www.linkedin.com/pulse/transaction-level-verilog-worlds-first-hardware-language-hoover/] - in it the author argues that this is the world's first hardware ***description*** language (instead of a simulation language). Interesting food for thought.
Read the main body of the page at https://www.redwoodeda.com/tl-verilog [https://www.redwoodeda.com/tl-verilog] - it gives some selling points regarding this language.
Go to http://www.redwoodeda.com [http://www.redwoodeda.com] and then go to the Tech→TL-Verilog (you already went there above). There you will find a bunch of tabs. Click "For Me" and give the whole thing a read down to where it starts with "How Does Sandpiper relate to…". By the way, SandPiper is the compiler for TLV created by Redwood EDA.
At the same location click the "TLV vs. …" tab and read that carefully. It gives the philosophy behind what what the author believes the shortcomings are with HDL's, HLS, and embedded languages. Interesting thoughts that lay the groundwork for looking at this language.
Glance quickly through the FAQ tab for the same page.

Step 2: Work through the TLV Tutorials

You may work in pairs on the remainder of this project.

The tutorials are found at http://makerchip.com [http://makerchip.com]. Once you are on the http://makerchip.com [http://makerchip.com] website you will see a menu item for Tutorials. Work through the tutorials up through the one titled: “Alignment Tutorial”.

As you work through the tutorials, collect your code for each exercise and include it as Section 1 of your lab writeup for this assignment.

Step 3: Look Through Some Examples

On makerchip.com under the Tutorials tab there is a set of examples. Work your way through those mentioned below, the goal being to pick up more and more of the language through reading working tutorials.

The bit-level incrementer is interesting in that it shows how to use a scope that is indexed (“/slice[M4_WIDTH-1:0]”) as a way of expressing an array of computational elements. Figure out how that describes an incrementer circuit (a +1 circuit). Not really needed for doing a +1 since you can do it with an “assign” statement, but shows how to replicate logic easily.

The multiplier example is interesting in that it shows how to easily break up a big multiply into chunks and then schedule them across a pipeline. You will want to follow the link to the SURF drawing and understand how it works first and then figure out the correspondence with the code.

The long division example shows a classical pipelined approach to an iterative divide algorithm.

The Finite State Machine design shows a FSM design for an elevator controller in SV (version 1). You can skip version 2. In version 3 the design is all TLV. You will note that these designs look nothing like any FSM you learned about, where there is a current state variable (cs) and from that and the inputs you compute a next state value (ns). Rather, the decision making is distributed and the state is contained in a number of variables. Also, note that the scope called “floor[*]” creates circuitry in parallel for all 3 floors, and each operates independently (a distributed FSM).

The SystemVerilog Testbench example shows how to do real Verilog hierarchy. Although Makerchip can only handle one file, this shows how you can used modules and instantiate them. In this case the testbench is the main module and it instantiates the DUT.

In contrast, if you go to https://github.com/stevehoover/makerchip_examples [https://github.com/stevehoover/makerchip_examples] there is a Ripple Carry Adder example. It defines a macro for an adder stage (the first TLV section) and then instantiates it using the m4+ macro. These use the m4 macro preprocessor to do text expansion/replacement before the actual SandPiper compiler actually gets the code. It makes the code look hierarchical but results in flat code once the m4 proprocessor is done with it. It is an alternative to true hierarchy.

NOTE: the examples all use macros extensively. Steve says this was his way of experimenting with some features without making them a part of the language (which would require support long term). From that I assume the idea is that some of the macro features will turn into actual language features long-term, while some may go away or just stay as macros. If you are unfamiliar with m4, look it up on the web - it is similar to what a C compiler's preprocessor does with all the #define and #include things in a C program before it hands the resulting text off to the real compiler.

Include a description of your reading through these examples in your writeup.

Step 4: Create a Real Pipelined RISC-V CPU

There is a tutorial workshop at https://github.com/stevehoover/RISC-V_MYTH_Workshop [https://github.com/stevehoover/RISC-V_MYTH_Workshop]. It is a 5 day (30 hour) workshop that creates a fully pipelined RISC-V CPU. But, it is targeted at non-HW designers, meaning you could likely do it all in a handful of hours (I took 6 but with some guidance based on where I got confused, you could likely cut a couple of hours off it).

We will focus on the Day 4 and Day 5 content, which deals with creating a RISC-V processor and then pipelining it (taking care of hazards in the process). There are videos associated with the slides at ***. Watching them as you work through the design is highly recommended. As indicated via email, DO NOT share these videos with anyone - they have been provided solely for our class's use.

Step 5a: Get a Baseline Design Completed

To get started, go to the link above and grab the slides for days 4 and 5. Then, open the starter code for the RISC-V lab. As before, save it as a new project (upper right) and bookmark the resulting web page link (or you will lose your work when you close your browser).

Look at the starter code. It is pretty empty but has some of the blocks you need to instantiate commented out. Work through the slides starting at slide 1 to complete the first design checkpoint by slide 26. At this point you will have a functional design but it will not be pipelined.

It says on many of the slides to “Check behavior in simulation”. When it says this it is not suggesting you will get Simulation PASSED in the Log tab or that the design will work until it is truly complete. Rather, check the new signals you have added in the simulation waveforms to be sure they do what you think they should (example: does $is_i_instr go high properly based on the value of the $instr variable?). That sort of thing. You don't need to religiously check everything but anything you can check early on when things are less complex will greatly simplify your later work.

Once you reach Side 26, call this your “Non-Pipelined CPU”. Include your description of it as Section 2 of your final writeup that deals with this design, include the final code, and demonstrate how and why it works in simulation.

Step 5b: Pipeline Your Design

Save away the previous design so you don't lose it. Then, using it as a starting point complete up through slide 42. At this point you will have a functional RISC-V pipelined processor. Dedicate Section 3 of your project report to this design. Describe how it works and show it working via simulation. Be sure to include your code.

Finishing Up

Include the things you were asked above to include in your final project report.

Writeup a summary of your work on this final assignment as part of that.

In additon to the above-requested items, please include some feedback on TL-Verilog that we can provide back to Redwood EDA. Please address at least the following:

1. What is your impression of TLV after having come at it as a beginner?
2. What are its 3 best features?
3. What were the 3 hardest things about learning to use it? How could these be addressed?
4. Under what conditions might you see yourself using such a language and tool?

Turn your report in on LearningSuite. If you worked with a partner, say so in the writeup and both of you

attach it. Congrats! You are done.

Notes on the Slides From Prof Nelson's Experience Working Through Them

Slide 6

Note that when they say increment PC as in +1 they also say in parens it should be 4 for bytes. Don't miss that since +1 is wrong, it should be +4.

Slide 8

This slide is explicit on what PC bits to wire up to imem - don't miss that detail.
The slide gives the names to use and what to hook them to - specifically $imem_rd_en and $imem_rd_addr - the slide tells what to hook them to.
The slide does give range for $imem_rd_addr - use it!
Regarding the imem itself, it is word addressable, NOT byte addressable so you have to toss some PC bits. You can see how things are expanded in the Nav-TLV pane, if you have a question look at the imem code The imem has a \SV_plus section which allows you to mix SV code with TLV (the plus has to do with whether the SV code can access TLV variables (or the other way around).
The imem has a /imem[7:0] scope which basically builds out 8 copies (like a generate), each of which is indexed by #imem.
In the end, is just a 8:1 MUX, with each of the 8 entries an instruction.
In the picture, the imem straddles line between @0 and @1. Does that mean $instr assignment is in @1? Yes it does. You can put enable and address assignments in @0 and assignment to $instr in @1 to match picture - but it also works if everything is in @1.

Slide 10

If you copy code snippets from the slides, be aware that the single quotes you copy from the sides are different from the single quotes on your keyboard - the ones from the slides give compile errors

Slide 16

This slide took by far the most time. But my wrestling with it and examining the expanded m4 macro design pieces taught me a TON about the language and filled in a number of gaps in my knowledge. I was not sure what to hook up since it had the statement “Default values are provided for inputs”.

In the end, you have to hook everything up, nothing is hooked up except the feedback path.
For this slide, hook up $rf_read_en1, $rf_read_en2, $rf_rd_index1, $rf_rd_index2 (the write signals come later). The register values are in the simulation pane as the /xreg's - if you open them up you will see initial values and then how they change. This is a nice way to monitor them in simulation.

Slide 20-2

Calling it Slide 20-2 means the second un-numbered slide after 20.
I assumed that we would not be hooking up the »1$value link, that it is already automagically done for us? If you check the Nav-TLV pane you can see the expanded definition of the reg file macro - and that shows that it is already hooked up. So, you just need to connect the other inputs and outputs.
When I ran the program the first instruction did not write to R10 because it looks like the reset had not ended. So, I got the wrong answer.

> So, I first added a NOP as the first instruction: m4_asm(ADD, r0, r0, r0) and it all worked.
> At about t=35 a 0x2d got written to /xreg[10] (address 0xa). That is a sign it worked.
> Then I decided that this was a dumb solution. I then changed the PC to reset when »1$reset is true - this delayed the PC coming out of reset until the rest of the pipeline was done being reset as well. Then, I could get rid of the NOP instruction and it all worked. This bug took quite some time for me to ultimately track down and fix.

Slide 25

Early in the slides, the Log tab message about “Simulation SUCCESS” simply means it reached cycle 40 (look in your code at the definition of success.
Only at slide 25 do you finally add code to make the success signal really mean success by looking for the correct value being written to the correct register.

Slide 27

Do figure out the waterfall diagram he uses - it is a useful depiction.

Slide 39

I was not sure whether I had to do the »2 between RF Wr and RF Rd or if it came for free. It is included in the RF macro and so you don't have to worry about it - just the things in red.

Slide 42

Now you don't need the $valid_taken_br signal, the PC can now update just on $taken_br.
But, note where the $taken_br signals are coming from - they are from the ALU inputs, NOT the RF outputs - they take advantage of bypassing too. Don't miss this detail (I initially missed this detail and spent a bunch of time debugging until it occurred to me).

Debugging Ideas

You can see the registers down below in the waveform pane as /xreg[10], etc.

Pay attention to pipe stage numbers! The PC is @0, but all the execute stuff is @1. So, if PC=5, then that instruction will get executed the next cycle.
Then, when the processor is pipeined, debugging is a bit more complex because a PC value in cycle *n* will fetch an instruction in cycle *n+1*, which will generate RF outputs in cycle *n+2* and so on. So, keep the pipe stages in mind as you trace through the behavior in the waveforms.
Also, note that signal names in the waveforms are prefixed with their pipe stage number and so you may have to time shift them in your mind to see what they are in a particular pipe stage at a particular clock cycle. A good quick syntax check is to hit CtlEnter and look at Nav-TLB
The register values are in the simulation pane as the /xreg's - if you open them up you will see initial values and then how they change. Nice way to monitor them and debug.
Get comfortable with the waveform viewer, you will need it.
Biggest problems I had debugging were: 1st instruction getting ignored due to reset timing, getting the reg file hooked up correctly, incrementing PC correct amount (+4 instead of +1), and missing where the $taken_br signals were tapping off in the pipeline on the very last step.

Coding Ideas

NEVER do an assignment to a signal as in $imem_rd_addr = $pc (you will get a 1-bit signal) - common error. So, remember on every assignment to include signal width for the left hand side unless you know it is a 1-bit signal! As Verilog or SV coders this is an easy thing to forget.
Pay careful attention to the picture regarding what goes in each pipe stage. For example,there is very little in @0, a lot is in @1, nothing is in @2 up to slide 26. And carefully check where each signal taps off - that is a source of errors (I speak from experience).
Look in Nav-TLV tab to see how macros were expanded. For the things like reg file it is useful to be able to read and understand. They are pretty straightforward. Example: during reset, the register file will not write. So, looking at the address and write signal in the waveforms can be deceiving. Only when I looked inside the register file expansion in Nav-TLV did I see that detail and that explained what I was seeing.
Copy your code to a local file regularly. I lost some stuff due to network issues. I also lost some stuff when I got confused about which version I was copying.
The color coding of text in makerchip will help you determine if something is a pipe signal name (has a $ in front of it).
Comment the code associated with each module (PC, ALU, DECODER). It will really help when you are scrolling around looking for things.
Don't forget indentation - the tool is very picky about 3-space indentation.

tl-verilog-unit.txt · Last modified: 2020/12/15 21:15 by nelson