RTL Simulation using Synopsys VCS

ECE5745 Tutorial 1 (Version 606ee8a)
January 26, 2017
Derek Lockhart

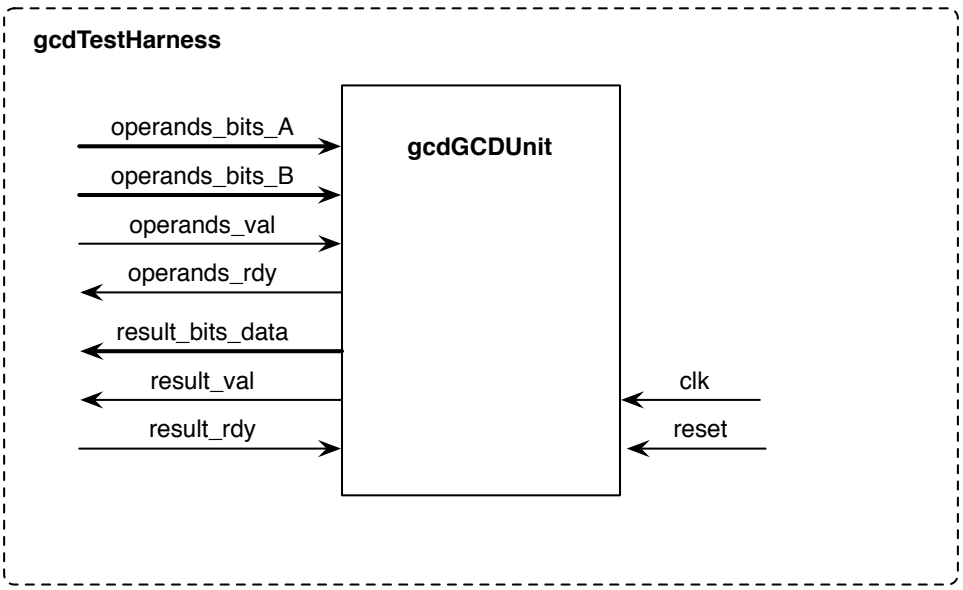
Contents

1 Introduction

In this tutorial you will gain experience compiling Verilog RTL into cycle-accurate executable simulators using Synopsys VCS. You will also learn how to use the GTKWave Waveform Viewer to visualize the various signals in your simulated RTL designs. Figure 1 illustrates the basic VCS toolflow and how it fits into the larger ECE5745 flow.

VCS takes a set of Verilog files as input and produces an executable simulator as an output. VCS is capable of compiling both behavioral Verilog models and RTL Verilog models. Behavioral models are often not synthesizable, so any hardware we intend to synthesize will need to be written at the register transfer level. However, behavioral verilog will still be useful when writing code we do not intend to synthesize, such as test harnesses. Please be conscious of this distinction, attempting to push behavioral code through the next step in the toolflow (Synthesis) will likely result in much pain and suffering.

You will be using an RTL model of a greatest common divisor (GCD) circuit as your design example for this tutorial. Figure 2 shows the block diagram for the GCD circuit you will be simulating.



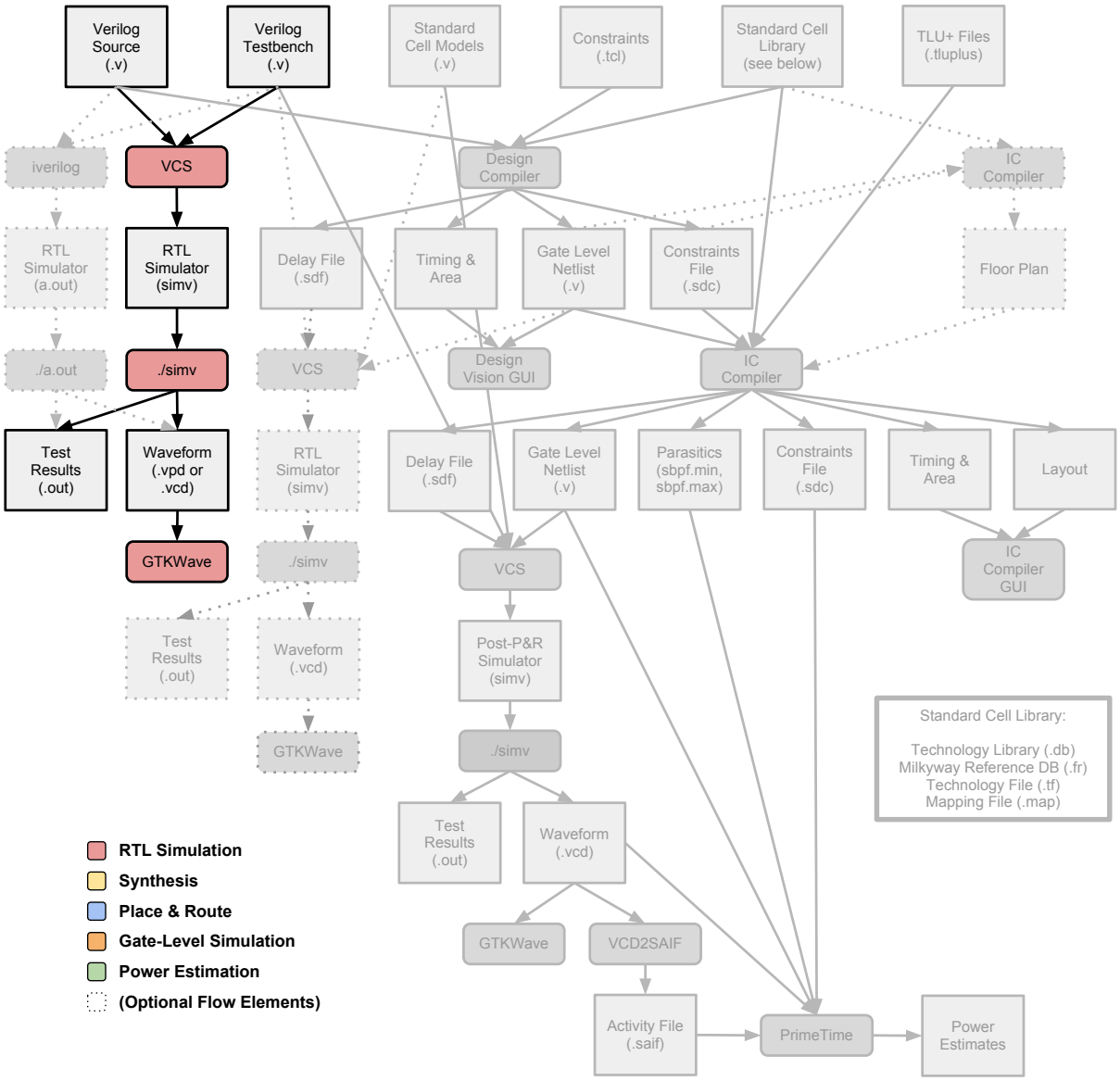


Figure 1: VCS Toolflow

Figure 2: Block diagram for the GCD circuit

```
module gcdGCDUnit#( parameter W = 16 )
(

  input clk, reset,

  input  [W-1:0] operands_bits_A,   // Operand A
  input  [W-1:0] operands_bits_B,   // Operand B
  input          operands_val,      // Are operands valid?
  output         operands_rdy,      // ready to take operands

  output [W-1:0] result_bits_data,  // GCD
  output         result_val,        // Is the result valid?
  input          result_rdy         // ready to take the result

);
```

Figure 3: Interface for the GCD module

2 Getting The Tutorial Code

All of the ECE5745 tutorials should be run on the `ecelinux` machines. Before proceeding further, please log into one of these machines.

You should follow along through the tutorial yourself by typing in the commands marked with a '%' symbol at the shell prompt. To cut and paste commands from this tutorial into your bash shell (and make sure bash ignores the '%' character) use an alias to "undefine" the '%' character:

```
% alias %=""
```

Once you have logged into a BRG machine you will need to setup the ECE5745 toolflow with the following commands:

```
% source setup-ece5745.sh
```

For this tutorial you will be using a GCD circuit as your example RTL design. Create an `ece5745` folder in your home directory and clone the tutorial files from the git repository:

```
% mkdir ${HOME}/ece5745
% cd ${HOME}/ece5745
% git clone git@github.com:cornell-ece5745/ece5745-tut-asic.git
% cd ece5745-tut-asic/tutorial
% TUTROOT=$PWD
```

Before starting, take a look at the subdirectories in the project directory. All of your projects will have a structure similar to the following:

```
/src     Source RTL and test harnesses should be placed in this directory.
         You should browse the source code for the GCD circuit in src to
         become familiar with the design.

/build   Contains all generated content including simulators, synthesized
         gate-level Verilog, and final layout. In this course you will always
         try to keep generated content separate from your source RTL. This
         keeps your project directories well organized, and helps prevent you
         from unintentionally modifying your source RTL.
```

```
/build/vcs-sim-rtl
/build/dc-syn
/build/icc-par
/build/vcs-sim-gl
/build/pt-pwr
```

> A subdirectory exists for each major step in the toolflow. These subdirectories will contain scripts and configuration files necessary for running the tool indicated by the subdirectory. For example, the build/vcs-sim-rtl directory contains a makefile which can build Verilog simulators and run tests on these simulators.

3 Manual VCS Build Process

We will first manually go through the commands for VCS using the commandline so that you can see the steps required to make VCS work. Since this process is tedious we will only do it once, later we will use scripts to automate the steps in this portion of the flow for us. For now, you should be able to cut and paste the commands below into your terminal:

```
% cd $TUTROOT/build/vcs-sim-rtl
% vcs -full64 -PP +lint=all,noVCDE +v2k -timescale=1ns/10ps \
   +define+CLOCK_PERIOD=0.5 +incdir+../../../vclib/src \
   -v ../../../vclib/src/vcQueues.v \
   -v ../../../vclib/src/vcStateElements.v \
   -v ../../../vclib/src/vcTest.v \
   -v ../../../vclib/src/vcTestSource.v \
   -v ../../../vclib/src/vcTestSink.v \
   ../../src/gcdGCDUnitCtrl.v \
   ../../src/gcdGCDUnitDpath.v \
   ../../src/gcdGCDUnit_rtl.v \
   ../../src/gcdTestHarness_rtl.v
% ./simv +verbose=1
```

Important runtime flags:

- `-full64` executes the 64-bit version of VCS.

  NOTE: if you forget to include this flag, VCS will fail!
- `+lint=all,noVCDE` turns on Verilog warnings except the VCDE warning. Since it is relatively easy to write legal Verilog code which is probably functionally incorrect, you will always want to use this argument. For example, VCS will warn you if you connect nets with different bitwidths or forget to wire up a port. Always try to eliminate all VCS compilation errors *and* warnings.
- `+v2k` enables support for various Verilog-2001 language features.
- `+timescale` specifies how the abstract delay units in a design map into real time units. This can also be provided in verilog source files as the `` `timescale `` compiler directive.
- `-v` indicates which Verilog files are part of a library (and thus should only be compiled if needed) and which files are part of the actual design (and thus should always be compiled).

After running the above commands, you should see text output indicating that VCS is parsing the Verilog files and compiling the modules. Notice that VCS actually generates ANSI C code which is then compiled using `gcc`. When VCS is finished you should see a `simv` executable in the build directory. Running the simv executable with the verbose flag should show you that the simulation executes and successfully passes all tests.

4 Automated VCS Build Process

Typing each command via the commandline is a tedious and error-prone process, and should typically be avoided. Instead, we make use of scripts to automate the process of building our tools for us. The following commands will first delete the simulator you previously built, and then regenerate it using the makefile.

```
% cd $TUTROOT/build/vcs-sim-rtl
% rm simv
% make
% ./simv +verbose=1
```

The make program uses the `Makefile` located in the current working directory to generate the file given on the command line. Take a look at the `Makefile` located in `build/vcs-sim-rtl`. Makefiles are made up of variable assignments and a list of rules in the following form.

```
target : dependency1 dependency2 ... dependencyN
    command1
    command2
    ...
    commandN
```

Each rule has three parts: a target, a list of dependencies, and a list of commands. When a desired target file is "out of date" or does not exist, then the make program will run the list of commands to generate the target file. To determine if a file is "out of date", the make program compares the modification times of the target file to the modification times of the files in the dependency list. If any dependency is newer than the target file, make will regenerate the target file. Locate in the makefile where the Verilog source files are defined. Find the rule which builds `simv`. More information about makefiles is online at `http://www.gnu.org/software/make/manual`.

Not all make targets need to be actual files. For example, the `clean` target will remove all generated content from the current working directory. So the following commands will first delete the generated simulator and then rebuild it.

```
% cd $TUTROOT/build/vcs-sim-rtl
% make clean
% make simv
```

5 Viewing Trace Output With GTKWave

When executing the simulation with the verbose mode enabled, you see the simulator printing information about the current execution to the commandline. When executing a test harness the output will be the passed/failed status of unit tests, whereas when executing a simulation harness the output should be the result of the computation.

In addition to the textual output, you should see a `check.vcd` file in your build directory. This file contains the signal trace information from the most recent execution of the simulator. To view this file as a waveform, open it with GTKwave:

```
% gtkwave check.vcd &
```

Figure 4 shows the GTKWave Waveform Viewer. You can use this window to browse the design's module hierarchy using *Signal Search Tree (SST)* pane in the top left. If you select a specific module in the hierarchy, you should see the wires and registers contained within that module in the lower left pane. To add signals to the waveform pane, you can drag and drop them from the SST into the *Signals* pane.

Add the following signals to the waveform viewer.

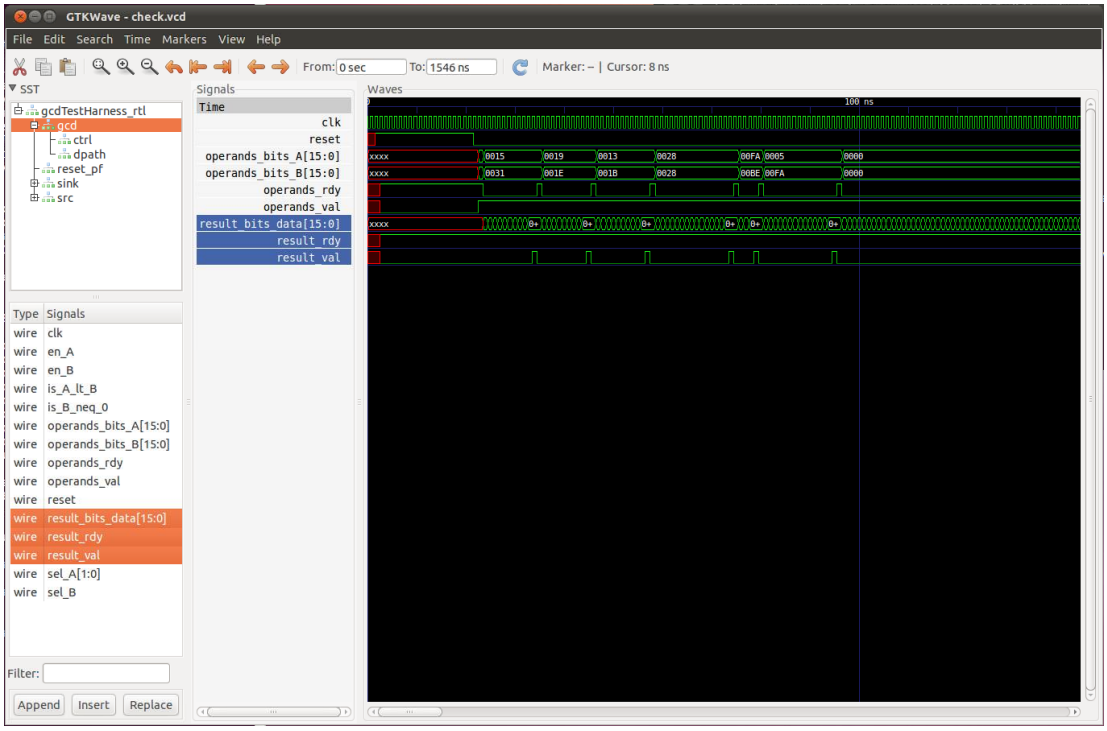


Figure 4: GTKWave visualizing waveforms from the GCD module

- `clk`
- `reset`
- `operands_bits_A`
- `operands_bits_B`
- `operands_rdy`
- `operands_val`
- `result_bits_data`
- `result_rdy`
- `result_val`

You should spend some time playing around with the GTKWave software, as you will most likely be using it extensively over the course of this semester. GTKWave will be the primary tool we use for debugging our RTL designs, so the more comfortable you are with this tool, the better off you will be.

One particularly useful feature of GTKWave you will want to be aware of is the ability to create signal save files. Since it may take a bit of time finding and pulling out exactly all the signals you want to view when debugging larger designs, it would be nice to be able to quickly recall these signal configurations (ie. when opening GTKWave at a later date, or when comparing waveforms of two different design alternatives side by side). Signal save files provide exactly this functionality.

To create a new signal save file, go to *File* → *Wire Save File As* in the menu bar of GTKWave. This will create a new *.sav* file with the name and location you provide. If you open up a new waveform, you can recall this signal save file by navigating to *File* → *Read Save File*. If you are already viewing several signals in GTKWave and then decide to load a signal save file, the list of signals in the save file will be appended to your current list of displayed signals.

Later in the semester we may explore using Synopsys Discovery Visual Environment (DVE) as an alternative to GTKWave. Although DVE is arguably more powerful than GTKWave, we have found it to be far less user friendly.

Notice that the GCD module is using the ValRdy signal protocol used extensively in ECE4750. All modules designed in this course will use the ValRdy interface. If you are not familiar with the ValRdy protocol it is highly recommended that you review the ECE4750 material.

6 Acknowledgements

Many people have contributed to versions of this tutorial over the years. The tutorial was originally developed for CS250 VLSI Systems Design course at University of California at Berkeley by Yunsup Lee. Contributors include: Krste Asanović, Christopher Batten, John Lazzaro, and John Wawrzynek. Versions of this tutorial have been used in the following courses:

- CS250 VLSI Systems Design (2009-2011) - University of California at Berkeley
- 6.375 Complex Digital Systems (2005-2009) - Massachusetts Institute of Technology
- CSE291 Manycore System Design (2009) - University of California at San Diego