Simulating Verilog RTL using Synopsys VCS

CS250 Tutorial 4 (Version 092509a)
September 25, 2009
Yunsup Lee

In this tutorial you will gain experience using Synopsys VCS to compile cycle-accurate executable simulators from Verilog RTL. You will also learn how to use the Synopsys Waveform viewer to trace the various signals in your design. Figure 1 illustrates the basic VCS toolflow and SMIPS toolchain. For more information about the SMIPS toolchain consult *Tutorial 3: Build, Run, and Write SMIPS Programs*.

VCS takes a set of Verilog files as input and produces a simulator. When you execute the simulator you need some way to observe your design so that you can measure its performance and verify that it is working correctly. There are two primary ways to observe your design: (1) you can use `$display` statements in your Verilog RTL to output textual trace information, or (2) you can instruct the simulator to automatically write transition information about each signal in your design to a file. There is standard text format for this type of signal transition trace information called the Value Change Dump format (VCD). Unfortunately, these textual trace files can become very large very quickly, so Synopsys uses a proprietary compressed binary trace format called VCD Plus (VPD). You can view VPD files using the Synopsys waveform viewer called Discovery Visual Environment (DVE).

You will be using a simple unpipelined SMIPSv1 processor as your design example for this tutorial, and thus you will also learn how to build and run test codes on the processor simulator. Figure 2 shows the block diagram for the example processor. Figure 1 shows the SMIPS toolchain which starts with an SMIPS assembly file and generates a Verilog Memory Hex (VMH) file suitable to run on the cycle-accurate simulator. This tutorial assumes you are familiar with the SMIPS ISA. For more information please consult the *SMIPS Processor Specification*.

The following documentation is located in the course locker `~cs250/docs/manuals` and provides additional information about VCS, DVE, and Verilog.

- `vcs-user-guide.pdf` - VCS User Guide
- `vcs-quick-reference.pdf` - VCS Quick Reference
- `vcs_dve-user-guide.pdf` - Discovery Visual Environment User Guide
- `vcs_ucli-user-guide.pdf` - Unified Command Line Interface User Guide
- `verilog-language-spec-1995.pdf` - Language specification for the original Verilog-1995
- `verilog-language-spec-2001.pdf` - Language specification for Verilog-2001

Getting started

Before using the CS250 toolflow and SMIPS toolchain you must run the course setup script with the following command.

```
% source ~cs250/tools/cs250.bashrc
```

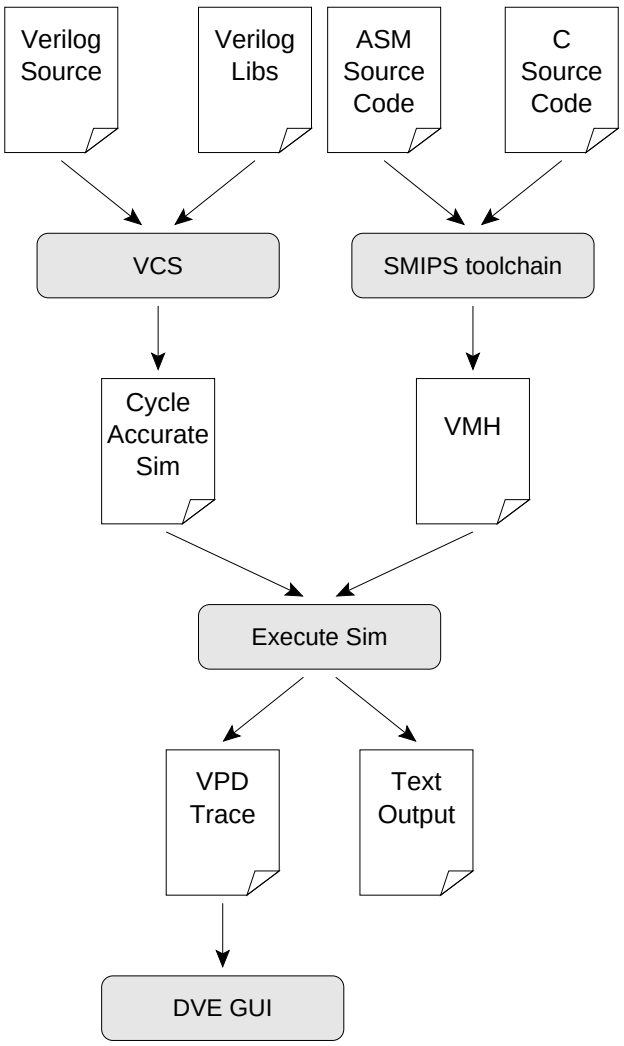

Figure 1: VCS Toolflow and SMIPS Assembler Toolchain

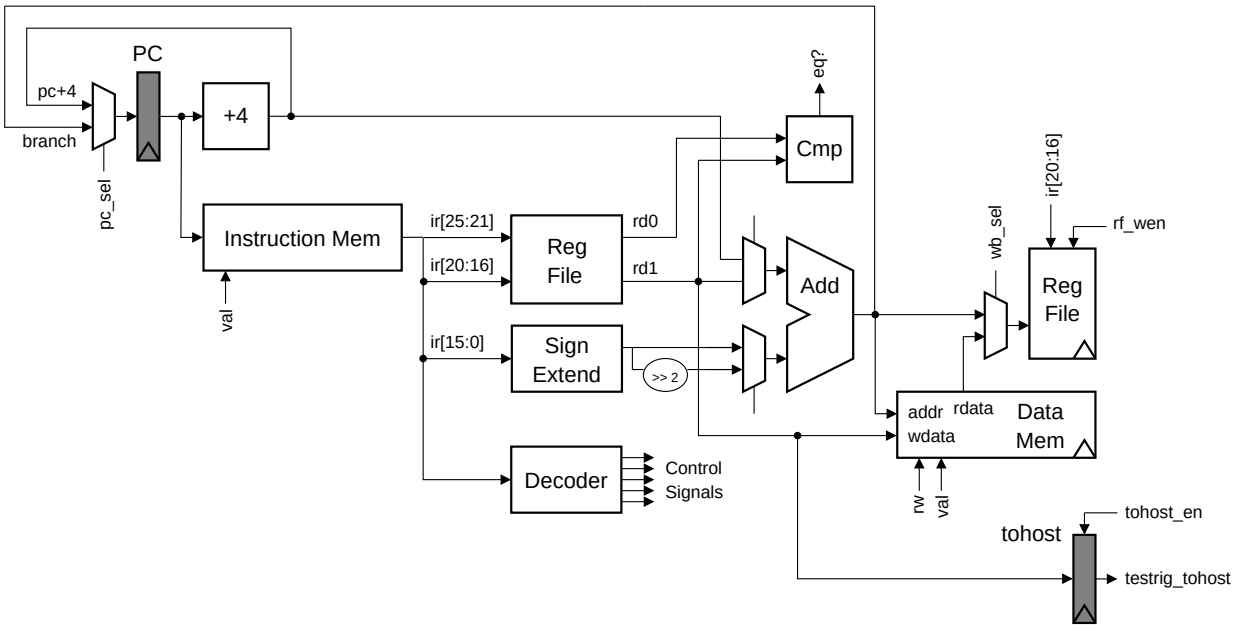

Figure 2: Block diagram for Unpipelined SMIPSv1 Processor

For this tutorial you will be using an unpipelined SMIPSv1 processor as your example RTL design. You should create a working directory and copy files from the course locker using the following commands.

```
% mkdir tut1
% cd tut1
% cp -R ~cs250/examples/v-smipsv1-1stage/* .
```

Before starting, take a look at the subdirectories in the project directory. All of your projects will have a similar structure. Source RTL should be placed in the `src` directory and test input files should be placed in the `smips-tests` directory. The `build` directory will contain 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. There are subdirectories in the `build` directory for each major step in the CS250 toolflow. These subdirectories will contain scripts and configuration files necessary for running the tools required for that step in the toolflow. For example, the `build/vcs-sim-rtl` directory contains a makefile which can build Verilog simulators and run tests on these simulators. You should browse the source code for the processor in `src` to become familiar with the design. The example code makes use of the simple Verilog component library (VCLIB) located in `~cs250/install/vclib`. VCLIB includes a variety of muxes, flip-flops, latches, RAMs, memories, and queues. You are welcome to either use the globally installed VCLIB or to create your own component library.

Compiling the Simulator

In this section you will first see how to run VCS from the command line, and then you will see how to automate the process using a makefile. To build the simulator you need to run the `vcs` compiler with the appropriate command line arguments and a list of input Verilog files.

```
% cd build/vcs-sim-rtl
% vcs -PP +lint=all +v2k -timescale=1ns/10ps \
      -v ~cs250/install/vclib/vcMuxes.v \
      -v ~cs250/install/vclib/vcArith.v \
      -v ~cs250/install/vclib/vcStateElements.v \
      -v ~cs250/install/vclib/vcMemories.v \
      ../../src/smipsInst.v \
      ../../src/smipsProcCtrl.v \
      ../../src/smipsProcDpathRegfile.v \
      ../../src/smipsProcDpath_pstr.v \
      ../../src/smipsProc.v \
      ../../src/smipsCore_rtl.v \
      ../../src/smipsTestHarness_rtl.v
```

By default, VCS generates a simulator named `simv`. The `-PP` command line argument turns on support for using the `VPD` trace output format. The `+lint=all` argument turns on Verilog warnings. 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. Since you will be making use of various Verilog-2001 language features, you need to set the `+v2k` command line option so that VCS will correctly handle these new constructs. Verilog allows a designer to specify how the abstract delay units in their design map into real time units using the `‘timescale` compiler directive. To make it easy to change this parameter you will specify it on the command line instead of in the Verilog source. After these arguments you list the Verilog source files. You use the `-v` flag to indicate 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 this command, 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.

Typing in all the Verilog source files on the command line can be very tedious, so you will use makefiles to help automate the process of building your simulators. The following commands will first delete the simulator you previously built, and then regenerate it using the makefile.

```
% rm -f simv
% make
```

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.

```
% make clean
% make simv
```

Building SMIPS Test Assembly Programs

A test program called `smipsv1_example.S` is located locally in the `smips-tests` directory. If you want to add your own test programs, you would add them to this directory. There are additional globally installed SMIPS assembly test programs located in `~cs250/install/smips-tests` which you can use for your lab assignments and projects. The following command will build all of the local tests and run it on the SMIPSv2 ISA simulator.

```
% cd ../../smips-tests
% make
% make run
```

Please refer to *Tutorial 3: Build, Run, and Write SMIPS Programs* for more information about building, running, and writing assembly test programs.

Running the Simulator and Viewing Trace Output

Now that you have learned how to build the simulator and how to build SMIPS test assembly programs, you will learn how to execute SMIPS test assembly programs on the simulator. The following command runs the local `smipsv1_example.S` test program on the simulator.

```
% cd ../build/vcs-sim-rtl
% ./simv +exe=../../smips-tests/smipsv1_example.S.vmh
```

Try running a globally installed SMIPS test assembly program.

```
% cd ../build/vcs-sim-rtl
% ./simv +exe=$UCB_VLSI_HOME/install/smips-tests/smipsv1_addiu.S.vmh
```

You should see some textual trace output showing the state of the processor on each cycle. The trace output includes the cycle number, reset signal, pc, instruction bits, register file accesses, testrig_tohost signal, and the disassembled instruction. The test program does a series of loads and verifies that the loaded data is correct. After running all the tests, the program writes a one into the `tohost` coprocessor register to indicate that all tests have passed. If any test fails, the program will write a number greater than one into the `tohost` register. The test harness waits until the `testrig_tohost` signal is non-zero and displays either `PASSED` or `FAILED` as appropriate.

In addition to the textual output, you should see a `vcdplus.vpd` in your build directory. Use the following command to start the Synopsys Discovery Visual Environment (DVE) waveform viewer and open the generated VPD file.

```
% dve -vpd vcdplus.vpd &
```

Figure 3 shows the DVE Hierarchy window. You can use this window to browse the design's module hierarchy. Choose *Window > New > Wave View* to open a waveform viewer (see Figure 4). To add signals to the waveform window you can select them in the Hierarchy window and then right click to choose *Add to Waves > Recent*.

Add the following signals to the waveform viewer.

- `smipsTestHarness.clk`
- `smipsTestHarness.core.proc.dpath.pc_mux_sel`
- `smipsTestHarness.core.proc.dpath.pc`
- `smipsTestHarness.dasm.minidasm`
- `smipsTestHarness.core.proc.dpath.rf_raddr0`
- `smipsTestHarness.core.proc.dpath.rf_rdata0`
- `smipsTestHarness.core.proc.dpath.rf_raddr1`
- `smipsTestHarness.core.proc.dpath.rf_rdata1`
- `smipsTestHarness.core.proc.dpath.rf_wen`
- `smipsTestHarness.core.proc.dpath.rf_waddr`
- `smipsTestHarness.core.proc.dpath.rf_wdata`
- `smipsTestHarness.testrig_tohost`

The `dasm` module is a special tracing module which includes Verilog behavioral code to disassemble instructions. The `minidasm` signal is a short text string which is useful for identifying which instruction is executing during each cycle. To display this signal as a string instead of a hex number, right click on the signal in the waveform viewer. Choose *Set Radix > ASCII* from the popup menu. You should now see the instruction type in the waveform window. Use *Zoom > Zoom Out* to zoom out so you can see more of the trace at once. Figure 5 shows the waveforms in more detail. You should be able to identify the `addiu` instructions correctly loading the register file with various constants and the `lw` instructions writing the correct load data into the register

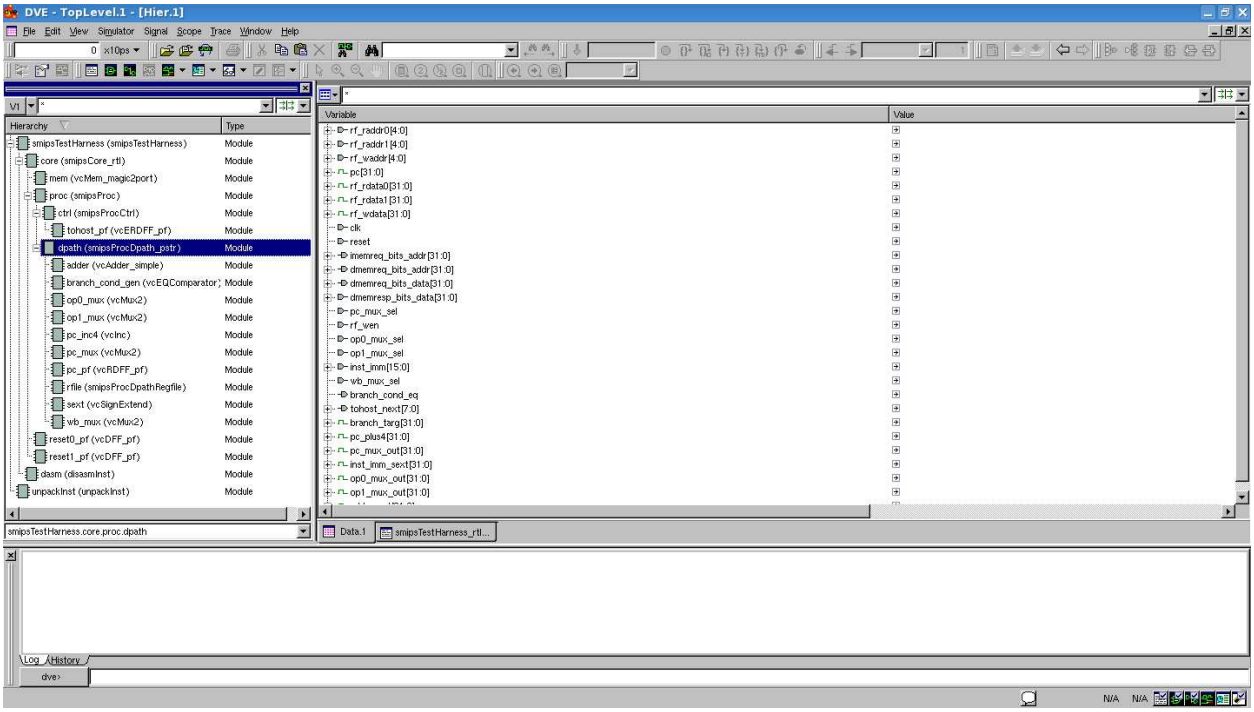

Figure 3: DVE Module Hierarchy Window

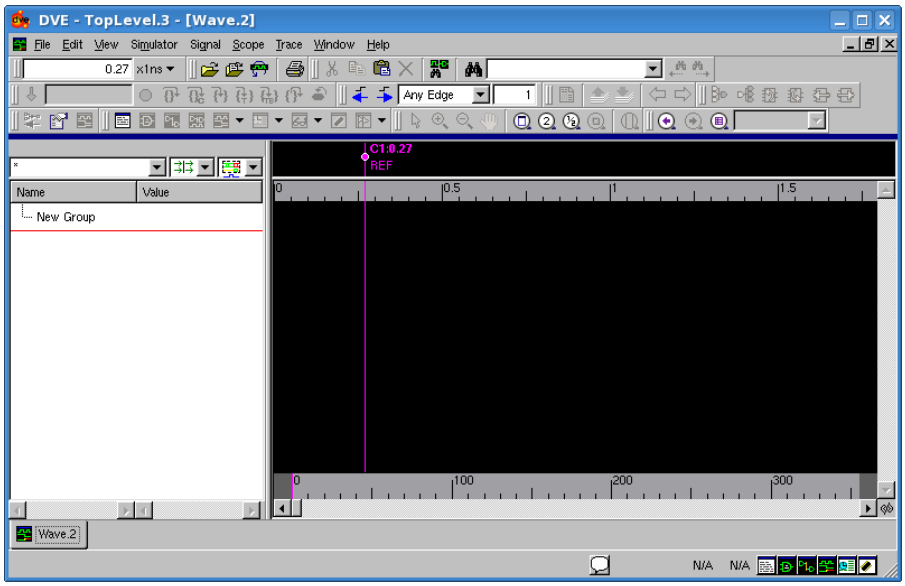

Figure 4: DVE Waveform Window

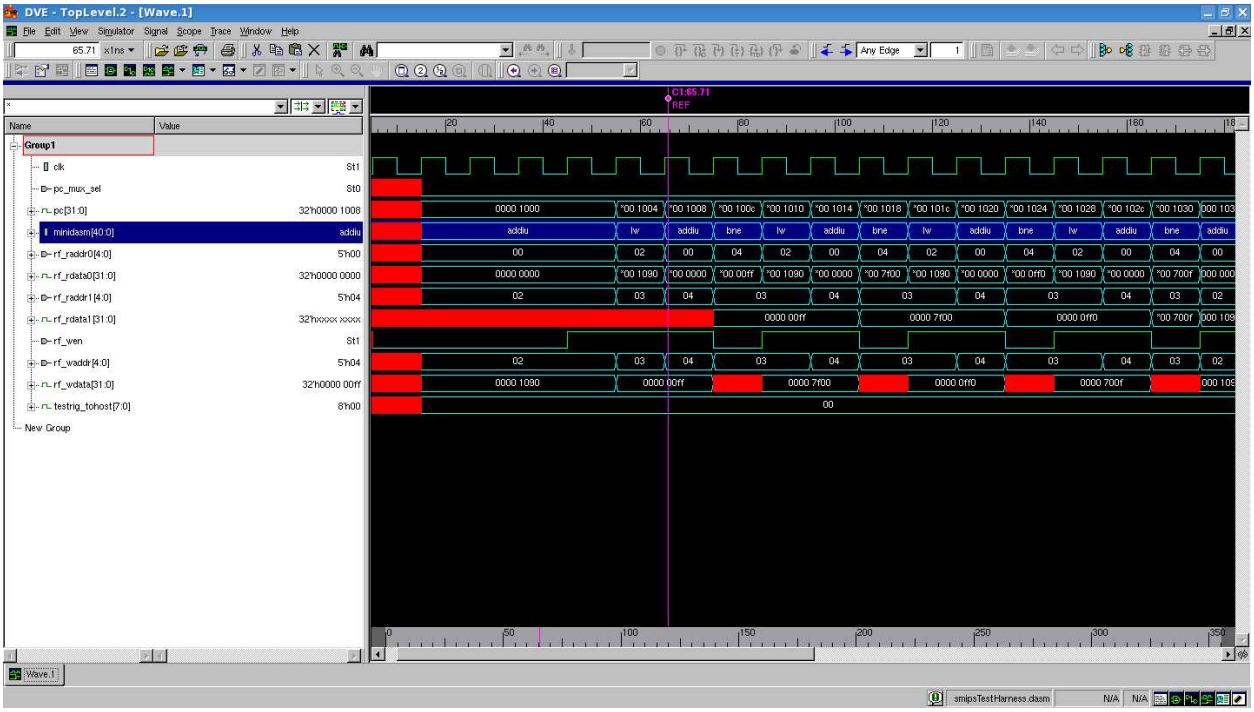

Figure 5: Waveforms for unpipelined SMIPSv1 processor executing `smipsv1_lw.S`

file. The `pc_mux_sel` control signal should remain low until the very end of the program when the code starts into an infinite loop after setting the `tohost` register to one. After reset, why is the `rf_rdata1` signal undefined for so many more cycles than `rf_rdata0`?

The Verilog test harness provides two optional command line arguments in addition to the required `+exe` argument as shown below:

```
simv +exe=<vmh-filename>
     +max-cycles=<integer>
     +verbose=<0|1>
```

By default, the harness will run for 2,000 cycles. This limit helps prevent bugs in test programs or the RTL from causing the simulator to run forever. When there is a timeout, the harness will display `*** FAILED *** timeout`. The `+max-cycles` argument allows you to increase this limit and is required for longer running programs. If the `+verify` argument is set to one (the default), then the harness will execute in "verification mode". This means that the harness waits until `testrig_tohost` is non-zero and then outputs either `PASSED` or `FAILED` as appropriate. If the `+verify` argument is set to zero, then the harness will execute in "performance mode". This means that the harness waits until `testrig_tohost` is non-zero and then it outputs a collection of statistics. You should use "verification mode" for running test programs which verify the correctness of your processor, and you should use "performance mode" for running benchmarks to evaluate the performance of your processor. Try running the the `smipsv1_addiu.S` program in "performance mode". You should observe that the Instructions per Cycle (IPC) is one. This is to be expected since the processor you are evaluating is an unpipelined processor with no stalls.

The following makefile target will build all of the test programs, run them on the processor simulator, and output a summary of the results.

```
% make run
```

Review

The following sequence of commands will setup the CS250 toolflow and the SMIPS toolchain, checkout the SMIPSv1 processor example, build local SMIPS test assembly programs, build the simulator, run all assembly tests, and report the results.

```
% source ~cs250/tools/cs250.bashrc
% mkdir tut4
% cd tut4
% cp -R ~cs250/examples/v-smipsv1-1stage/* .
% cd smips-tests
% make
% make run
% cd ../build/vcs-sim-rtl
% make
% make run
```

Acknowledgements

Many people have contributed to versions of this tutorial over the years. The tutorial was originally developed for 6.375 Complex Digital Systems course at Massachusetts Institute of Technology by Christopher Batten. Contributors include: Krste Asanović, John Lazzaro, Yunsup Lee, and John Wawrzynek. Versions of this tutorial have been used in the following courses:

- 6.375 Complex Digital Systems (2005-2009) - Massachusetts Institute of Technology
- CS250 VLSI Systems Design (2009) - University of California at Berkeley