Assignment 4

Overview

In this assignment, you will try alternative approaches to hardware design and compare the results. You will reimplement the rasterizer in HLS as well as magma, and then write a short report comparing the results.

Part 1: HLS

This part of the assignment will introduce you to high-level synthesis using the Catapult HLS tool. HLS takes in behavior C++ code and generates RTL. This allows for higher productivity in design as well as verification. You will evaluate the advantages and drawbacks of HLS by reimplementing the rasterizer in HLS and verifying the functionality.

Goals

Your job is to complete the HLS implementations of the bounding box (`src/bounding_box_hls.h`), test iterator (`src/test_iterator_hls.h`) and the sample test (`src/sample_test_hls.h`) modules and verify them. For full credit your implementation must pass all end-to-end test vectors. A description of the files is given below in the code organization section. Please go through the entire assignment description before starting to work on the code.

Catapult HLS Documentation

The big advantage of HLS is that it's simply C++ with a few pragmas and different datatypes. Documentation for the bit-accurate datatypes is located at: /cad/mentor/2019.11/Catapult_Synthesis_10.4b-841621/Mgc_home/shared/pdfdocs/ac_datatypes_ref.pdf

The following is a brief summary that will help you:

```
ac_int<8, false> val1; // 8 bit unsigned integer
ac_int<16, true> val2; // 16 bit signed integer

ac_int<4, false> slc1 = val1.slc<4>(0); // lower 4 bits of val1, starting at index 0
val2.set_slc(7, slc1); // sets 4 bits of val2 starting at index 7 to slc1
```

In addition, ac_channel<> must be used to connect hierarchical blocks. Note that ac_channel is not required to communicate with CCOREs (such as SampleTest and JitterSample). Hierarhical blocks have different data rates (BoundingBox will produce outputs at a different rate than TestIterator will read them in), while a CCORE is really just a set of operations. ac_channel<> is used for point to point communication between blocks. A channel can only be read by 1 block and written to by 1 block.

Another advantage of HLS code is that debugging is much easier. You can attach to the compiled binary (located in) with gdb and debug like normal C++ code.

Finally, testbench creation is greatly simplified. C++ testbenches will test the HLS C++ code, and the same C++ testbench is then used to test the generated RTL.

Reporting Issues

Please use piazza to report issues or if you need support. Please post a private question to the instructors if you would like to share code using which we can reproduce the issue at our end.

Instructions

For each of the modules in `src/bounding_box_hls.h`, `src/test_iterator_hls.h` and `src/sample_test_hls.h`, the implementation to be completed is marked in the files by `// Start code here`.

There are two tests you need to pass:

1. C Test- checks that the HLS C++ code matches the gold reference model from part 1. In the hls folder, run: `make c_test`. This will create an executable binary from the HLS C++ code in test/Rasterizer.v1/scverify/orig_cxx_osci/scverify_top. The testbench will run, and report any mismatches. If everything is correct, you will see "No errors found!" To help you debug, unit tests for the bbox module can be run with `make bbox_c_test`.
2. RTL Test- generates RTL and checks that the produced RTL matches the HLS C++ code. In the hls folder, run: `make rtl_test`. This will generate the RTL code from the HLS C++ code in build/Rasterizer.v1/concat_rtl.v. NCSim will open up. Hit the blue play button in the waveform window. The testbench will first run, and input and output values will be captured. Then, the RTL will be stimulated with these inputs and the outputs will be observed. Look for the first two red _error_cnt_sig signal, which tells you the number of mismatches between the C++ code and the RTL. If everything is correct, you should see that the signal stays at 0.

Change the test by passing in TEST as a variable to the make command, like:
`make c_test TEST=$EE271_VECT/vec_271_01_sv_short.dat`

Once both the C test and RTL test pass, you are ready to move on to synthesis.

Run `make run_dc`. Reports will be located in hls/synth/reports. The clock period can be adjusted by changing the value in the Makefile or by passing CLK_PERIOD as a variable to the make command, like: `make run_dc CLK_PERIOD=1.5`. However, make sure that if you change it, you run `make clean` first.

Increasing throughput of your design can be accomplished by using pragmas to adjust the pipelining and loop unrolling. Without any pragmas, you will likely have a much lower throughput than the Verilog implementation (although the area/power will also be much lower).

Part 2: Magma

This part of the assignment will introduce you to a new python-based hardware description language called magma that has been developed at Stanford. You will learn magma by reimplementing your genesis-based RTL in magma and verifying the functionality of your design.

Goals

We provide starter code for this part at `/afs/ir/class/ee271/project/assignment4`. Your job is to complete the magma implementations of the bounding box (`bbox.py`), iterator FSM (`iterator.py`) and the sample test (`sampletest.py`) modules and verify them. For full credit your implementation must pass all provided end-to-end test vectors and unit tests. A description of the files is given below in the code organization section. Please go through the entire assignment description before starting to work on the code.

Magma Documentation

To learn magma and complete this assignment, magma documentation is your best friend. Start by going through the magma tutorial from the review session (code available here: https://github.com/leonardt/magma_tutorial), and then go through the documents at:

- The magmathon repository is the place to go to learn about magma (tutorials). Read through the notebooks https://github.com/phanrahan/magmathon/tree/master/notebooks in the order of tutorial/coreir-tutorial -> intermediate -> signal generator -> advanced. Post any question as an issue on the GitHub repository.
- magma/docs is the place to go for documentation about the system (e.g. supported operators): https://github.com/phanrahan/magma/tree/master/docs. There's also an html version at https://magma.readthedocs.io/en/latest/operators/.

- There's also mantle/docs at https://github.com/phanrahan/mantle/tree/master/docs and https://magma-mantle.readthedocs.io/en/latest/?badge=latest for learning more about the standard library.

Reporting Issues

Please use piazza to report issues or if you need support. Please post a private question to the instructors if you would like to share code using which we can reproduce the issue at our end.

Installation

If you are working on the project on the `caddy` machines, then all of the tools below are pre-installed for you. Just remember to do `source setup_ee271.cshrc` so that your environment is initialized properly.

To work on the project on your personal computer, you need to install the following. The installation instructions are in each repository. * CoreIR - https://github.com/rdaly525/coreir. Usually you add a directory called `lib` to your `LD_LIBRARY_PATH` and `bin` directories go in your `PATH`. * Magma - https://github.com/phanrahan/magma * Mantle - https://github.com/phanrahan/mantle * Fault - https://github.com/leonardt/fault. Install using pip install fault==0.35 * Verilator - https://www.veripool.org/projects/verilator/wiki/Installing * GTKWave - http://gtkwave.sourceforge.net/ (for debugging)

Code Organization

We only provide the new files that you need. Please add them to your already existing project folder. The hierarchy is:

```
rasterizer
---- magma
--------- rast_types.py : types used for the signals in all the modules
--------- dff.py : 1D, 2D and 3D pipelined D flip flops built on top of synopsys designware 1D pip
--------- bbox.py : implementation of bounding box module
--------- iterator.py : implementation of iterator FSM module
--------- tree_hash.py : hashing function called in hash_jtree.py
--------- hash_jtree.py : implementation of hash module
--------- sampletest.py : implementation of sample test module
--------- rast.py : top level rasterizer module that instantiates all of the above modules
--------- tester.py : reusable testing functions for unit testing all your modules; tester is in
--------- verilog
-------------- dff.v : verilog wrapper around pipelined D flip flop from synopsys' designware li
--------- build : when you run test_dut() function in each of the above modules, magma compiler c
```

```
--------- ComputeBoundingBox_vector.json : test vector for unit testing bbox.py
--------- Iterator_vector.json : test vector for unit testing iterator.py
--------- HashJTree_vector.json : test vector for unit testing hash_jtree.py
--------- SampleTest_vector.json : test vector for unit testing sampletest.py
--------- wrapper
-------------- rast_magma.sv : we are using the existing genesis infrastructure for end-to-end t
---- Makefile.magma : updated Makefile that points to magma generated verilog
---- rtl
--------- rast.vp : same file as earlier but has extra display statements for dumping out test ve
---- verif
--------- testbench_magma.sv : original testbench modified to work with the magma/wrapper/rast_
--------- top_rast_magma.sv : top level module for verification that instantiates the testbench
```

Instructions

For each of the modules in `bbox.py`, `iterator.py` and `sampletest.py`, the implementation to be completed is marked in the files by `Your code goes here` in the definition method.

For each module, we have provided you the list of signals that you need to assign values to. You may define any additional combinational functions outside of the definition method to compute these signals. For an example of how to go about doing this, you can look at `hash_jtree.py` file in which we provide a complete implementation of the hashing module.

These signals that you are assigning are then fed into pipeline registers. While not required to complete this assignment, to get a better understanding of how the registers are defined and connected, you can look at `magma/verilog/dff.v` and `magma/dff.py` that defines 1, 2 and 3D flip flops, which are then instantiated in each of the rasterizer sub-modules. This will also give you an understanding of how you can wrap verilog code in magma, and how to use magma's advanced metaprogramming functions such as `braid`. You can use such functions in the code that you write to make it more succinct.

Once you have written the definition in a module you must verify it independently of other modules. This is called unit testing. For example, if you look at bbox.py, the `define_dut()` function instantiates the bounding box module with some specific parameters. This function is called in the `test_dut()` function, which then compiles the magma bounding box module into verilog. To execute `test_dut()` uncomment the last line

```
test_dut()
```

and run the python file using

```
python bbox.py
```

The generated bounding box verilog shall be put in the magma/build directory. You can inspect this verilog for debugging.

Note: Please copy the test vectors from here: https://www.dropbox.com/sh/1axp5ww3od2nqlm/AABiY7ccRW 9urDrCxea?dl=0. You can also generate them yourself by running the simulation with the genesis files, and using the rtl/rast.vp provided with assignment 4. **The vectors are dumped by the `$fdisplay` statements in the files. There is an extra comma dumped in the second last line of the json file. Be sure to delete it before using the vectors, else the file won't be a valid json file.**

Once the module compiles, it is time to feed some test vectors in it to unit test it. You can do this by uncommenting the following lines in `test_dut()`:

```
testbench = tester.Tester(dut, tester.pack_vectors(dut, dut.name + '_vector.json', 10000))

testbench.compile_and_run(directory="build", target="verilator", flags=["-Wno-fatal", '--trac
```

The first line instantiates a testbench and passes it the module under test, a test vector json file and how many lines of the vector file to run on. If you do not specify the last argument, all the vectors in the file shall be run. The second line compiles and runs the testbench, using the verilog simulator - verilator. To understand how the testbench is written you can look at `magma/tester.py`. When you run the testbench, if it runs to completion without errors your implementation is correct, otherwise an assertion will be violated, and it will print out where the error happened.

To run the testbench, verilator will look for a file `DW_pl_reg.v`. This exists here on the class server: `/afs/ir.stanford.edu/class/ee/synopsys/syn/M-2016.12-SP2/dw/sim_ver/`

For looking at internal signals for debugging you can open the .vcd file which is dumped in `magma/build/logs` when you run the testbench. You can view this file using any waveform viewer such as GTKWave (http://gtkwave.sourceforge.net/).

There is a known bug related to calling `m.compile` multiple times through coreir (https://github.com/phanrahan/magma/issues/319). If you have multiple files calling `m.compile`, you are likely running into the same issue (the first compilation works, but the subsequent ones fail). We are investigating the issue and will push an update when it's fixed. For now, you'll want to separate calls to `m.compile` (e.g. only call `m.compile` on the top circuit for the current file, comment out the rest).

Once you have implemented and unit tested `bbox.py`, `iterator.py` and `sampletest.py`, run the end-to-end test as before but using `Makefile.magma`. Make sure your test passes for all provided test vectors.

In `magma` folder:

```
python rast.py
```

This will generate all design verilog files in `magma/build`.

Then, in top project folder:

```
make comp_gold
make -f Makefile.magma run RUN="+testname=$EE271_VECT/vec_271_00_sv.dat"

diff verif_out.ppm $EE271_VECT/vec_271_00_sv_ref.ppm
```

Write Up

Compare the results from your Verilog, HLS, and magma implementations. Report the area, power, and throughput for each and briefly talk about the advantages/disadvantages for each approach, as well as any thoughts on the ease of each approach.

Submission

For this assignment, you should submit the finished HLS code, as well as the finished magma code. The due date of this assignment is 12/13. This deadline is final and there shall be no extensions. The submission requirement is the same as assignment 2, except to name your folder and tar ball as assignment4.