MyHDL vs. PyMTL: Two Python to HDL Packages

COMS E6998 - FPGAs Then and Now - Spring 2018

Samuel Beaulieu
srb2208@columbia.edu

ABSTRACT

With the end of Dennard scaling and the slowing of Moores law, engineers must turn to methods other than reducing transistor sizes to realize performance gains. These resources come in the form of FPGAs and ASICs. Both utilize specialized hardware, either re-configurable or static, to accelerate certain applications. Designing and testing these accelerators, however, requires specialized knowledge of VHDL or Verilog and special software to simulate and synthesize. They are also time consuming to design, and challenging to integrate and test. Using higher level languages offers a way to alleviate some of these challenges. In this project, we look at the feasibility of designing hardware in Python. Specifically, we compare two Python to Verilog packages, MyHDL and PyMTL, and analyze their feasibility in terms of ease of use, cost, and performance. Using these packages we show that there are advantages to designing hardware in Python, however knowledge of Hardware Description Languages (HDLs) and hardware design principles is still necessary.

1. INTRODUCTION

With the end of Dennard scaling and the slowing of Moores law, engineers must turn to other resources to realize performance gains. These other resources come in the form of FPGAs and ASICs. FPGAs have become common in commercial data centers such as those owned by Microsoft [1] and Amazon [2]. Similarly, ASICs have made their impact in Google's data centers [3] and hardware accelerators [4]. Regardless of the hardware choice, both FPGAs and ASICs require hardware design typically done in VHDL or Verilog. Not only does the increased demand in specialized hardware require more, experienced VHDL and Verilog designers, but it also constrains the delivery pipeline. The time it takes to develop, integrate, and verify new hardware is an extremely important metric in the success of a new product.

Higher level languages such as Python are widely known, easy to write, debug, and test, and often free to use. In this project, we look at the feasibility of using Python to design hardware. Specifically, we look at two packages, MyHDL [5] and PyMTL [6], and analyze their feasibility in terms of ease of use, cost, and performance. By looking at the installation process, learning curve, support system, and design process of these two packages, we see that they do offer advantages over strictly using Hardware Design Languages (HDLs). Despite these advantages, we see that these two packages are not a complete substitute for a general knowledge of HDLs and standard hardware design concepts. All the code and results for this project can be found in the project Git repository here [7].

2. DESCRIPTION

In this project, we compare and contrast two hardware design packages for Python: MyHDL and PyMTL. We investigate their support base, their difficulty to set up and use, their benefits, and their performance.

This analysis is based on taking two Verilog designs and porting them to Python using each of the packages. One of the designs in primarily compute based while the other is memory based. We then use the packages to convert the designs back to Verilog and compare them against the native Verilog based on simulation and synthesis for an Altera Cyclone V FPGA. Section 2.1 describes each of the software packages while Section 2.2 discusses the installation process. Section 2.3 describes the test environment and Section 2.4 describes the test designs. Section 2.5 examines the development process using each of these packages while Section 2.6 looks at the conversion to Verilog and the simulation and synthesis process.

2.1 Package Overview

At a high level, both MyHDL and PyMTL preform the same function. Both allow designers to design hardware using Python syntax. Though the syntax is entirely Python and the packages follow typical Pythonic principles, the actual design process is the same as it would be if HDLs were used. Just like with Verilog or VHDL, the designer defines modules that interact with each other based on a sensitivity list. The designer still has to think spatially instead of sequentially and define what happens on clock edges.

Where the packages differ is in which Python features they leverage to define hardware. MyHDL is based on Python generators. A Python generator is a function that runs to the first `yield` keyword and returns an object. Unlike normal functions, a generator maintains its state and can have multiple yields in it. When `.next()` is called on the function, it continues to run until the next yield where it returns an object and saves state until called again. MyHDL uses functional programming concepts and generators to implement HDL modules and processes. The generators run until a yield and then return when they should be called again, whether it is after a certain time or when an event occurs. Even MyHDL modules are functions which return handles to each of their internal generators. This allows the MyHDL simulation object to call each of a module's generators as required. MyHDL also defines a handful of Python decorators which it uses to automatically create generators which is helpful when defining sensitivity lists.

In contrast, PyMTL takes a more object oriented approach. Modules are defined by classes which have member variables and functions. When the PyMTL simulator runs, it calls the class's functions based on a sensitivity list for the functions. Because PyMTL follows object oriented practices vs functional generators, it may be easier to understand to new designers than MyHDL.

On a non-technical level, both packages have some similarities and differences. The last release of both of them was in 2015 and both are open source. Both let designers simulate in Python and convert their designs to Verilog as well generate a Value Change Dump (VCD) file for waveform tracing. Aside from that, the two packages have very different backgrounds and user bases.

MyHDL was developed by a man named Jan Decaluwe and has been used for actual hardware. The support base seems significantly larger with multiple channels on different sites. The MyHDL website has multiple examples and the documentation is fairly thorough. Jan also provides commercial support on MyHDL based projects.

PyMTL, on the other hand, was designed at Cornell and is/was used in a few computer architecture courses. Though there are some great tutorials that were made for those courses, the documentation as a whole is much less comprehensive and the support base seems smaller than that of MyHDL. PyMTL uses Verilator for conversions to Verilog and integrating Verilog modules with Python designs.

2.2 Installation

To test each software package, we spun up an Amazon EC2 t2.large instance running Ubuntu 16.04 with Python 2.7 and `pip` installed. Installing the two packages was a very different experience. MyHDL was very easy to install and required running one command: `pip install myhdl`.

PyMTL is dependent on a whole host of other libraries and packages such as Verilator and was not as simple. Verilator is first built from source and then libraries need to be installed using a package manager like `apt-get`. Once all the prerequisites are installed, the PyMTL GitHub repository needs to be cloned and then `pip` can be used to finish the installation. Though the actual process took a while and was slightly confusing, the PyMTL GitHub page [8] was detailed enough to make it painless.

2.3 Test Enviroment

With MyHDL and PyMTL installed, we needed a way to simulate and synthesize the native Verilog code and the Verilog generated by the Python packages. To do this, we installed the lite edition of Intel Quartus Prime [9]. The lite edition requires no licenses, is completely free, and supports "Intel's low-cost FPGA device families" [9]. It also includes ModelSim which can be used for simulation.

The default installation bundles ModelSim with Quartus which made it cumbersome to simulate. After the install, we determined the directory ModelSim was located in and added it to the `PATH` environment variable so the ModelSim executables could be used directly. Next we set up a folder structure, Makefile, and TCL script to automate simulation and synthesis.

The Makefile itself contains a few design specific variables that have to be changed for each design. These variables indicate which module is the test bench and which one is the device under test. Once set, the design can be simulated by loading all the source Verilog files into a folder `./sim/src/` and running `make sim` from the `./sim/` folder. Similarly, running `make syn` synthesizes the design and generates fitting and timing reports which are placed in `./sim/syn/`.

2.4 Test Designs

The two designs we decided to port to MyHDL and PyMTL are a SHA1 hash accelerator and a RAM module. The SHA1 accelerator [10] was designed by a man named Joachim Strombergson who, according to his GitHub, "develops embedded and hardware implementations of cryptographic primitives." The SHA1 design has been used in actual hardware, is compute heavy, and is not incredibly complex which was perfect for learning the basics of MyHDL and PyMTL. The RAM module is one of Altera's example designs [11] and was chosen to examine whether the Verilog generated by MyHDL and PyMTL would use memory blocks when synthesized. The module is a dual port, byte accessible, 1 KB RAM module.

Both of these designs were written using MyHDL and PyMTL. After Python simulations showed the proper results (clock cycles and values), the designs were used to generate Verilog and simulated with the original Verilog test bench. Once simulation proved successful, the designs were synthesized. Section 3 details the results of synthesis.

2.5 Development Process

After getting a feel for each of the Python packages, rewriting the Verilog designs in Python was straightforward. For both packages, the most challenging part was understanding their unique way of describing hardware which is explained in more detail in the following two subsections. A third subsection describes how the two packages test and simulate the designs.

2.5.1 MyHDL

As explained in Section 2.1, MyHDL is based on functional programming and generators. Aside from that and the Python syntax, MyHDL has essentially the same structure as Verilog. This is done using decorated functions and special data types for bit vectors and signals. These custom types facilitate the communication between and within generators in the same way registers and wires facilitate communication in Verilog modules.

Figure 1 shows the three main types of MyHDL decorators. The first and third decorators indicate that the function beneath them should be called based on a sensitivity list while the middle one defines combinational logic. Like in Verilog, a MyHDL sensitivity list is typically composed of clock edges and resets or all the signals that are read from inside the function. Unlike in Verilog, MyHDL does not support wild cards inside sensitivity lists which can become problematic as the number of signals that are read from increases. It is very easy to miss a signal when writing the sensitivity list or forget to add it to the list when updating the function.

MyHDL defines two types of bit vectors: `intbv()` and `modbv()`. The number of bits and initial value of both can be set and they are both overloaded to support arithmetic and bit-wise operations and bit slicing. The two types differ in that the second is a modular bit vector that wraps around when it reaches its maximum value while the first is not. In Verilog, these types correlate to the actual values of wires and registers but not the registers themselves. When bit vectors of more than one bit are written to, it is crucial that they are bit sliced.

If the whole bit vector is to be written to, it can be sliced like: `vector[:] = 0x59`. In a very Pythonic way, if the bit vector is not sliced, the name is re-associated with the new value and no longer with a bit vector type. Slicing is done with a high and low bound, however unlike with

```
@always(clk.posedge, reset_n.negedge)
def reg_update():

    if not reset_n:
        init_reg.next = 0
```

```
@always_comb
def logic():
    core_block.next[512:480] = block_reg[0]
    core_block.next[480:448] = block_reg[1]
    core_block.next[448:416] = block_reg[2]
```

```
@always(state_init, first_block, H0_reg,
    H1_reg, H2_reg, H3_reg, H4_reg,
    round_ctr_reg, a_reg, b_reg, c_reg,
    d_reg, e_reg, w, state_update)
def state_logic():
    a5 = modbv(0)[32:]
    f  = modbv(0)[32:]
```

Figure 1: MyHDL Decorators

Verilog, the higher bound is exclusive while the lower bound is inclusive. If an index is not provided, it is assumed to go to the extreme (highest possible or zero). The last example in Figure 1, shows two 32 bit vectors being initialized to zero. The `modbv()` variable is initialized and then sliced to be exactly 32 bits. Slicing is not required for the variable to be read from.

Registers and wires are implemented in MyHDL using a special `Signal()` data type. The definition of a signal might look something like: `core_block = Signal(intbv()[512:])` which defines a 512 bit signal. In MyHDL, there is no distinction between registers and wires; they are both implemented using signals. Signals can be the argument of sensitivity lists and written to from inside combinational and sequential blocks using `.next`. For example, the core block signal from before would be written to like so: `core_block.next[:] = 0x59`. The `.next` operator gets the bit vector from the signal which is then sliced so it can be written to and not overwritten.

2.5.2 PyMTL

Like MyHDL, PyMTL is structurally very similar to Verilog as well. It uses object oriented concepts to define modules and processes which is a bit clearer than MyHDLs functional programming concepts at first, but quickly becomes irritating in practice. Because of the class based outline, PyMTL variables must always be prefixed by a reference to the class instance as shown in Figure 2 by each variable having an "s" before it.

Figure 2 also shows how submodules can be instantiated in PyMTL. Unlike MyHDL which instantiates submodules by just calling the submodule's function with signals as arguments, PyMTL requires an instance of the class to first be created and then all the signals connected. One of the strangest things about PyMTL, which is partially seen in Figure 2, is that it inherently defines a clock and reset signal. Since the SHA1 design uses a negative edge triggered reset, it had to be defined separately.

Another similarity between MyHDL and PyMTL is that functions are decorated to indicate when the function should be called. Figure 3 shows the same three functions as Figure 1 decorated in PyMTL's style. PyMTL offers much less customization in defining when a function is called.

```
s.address    = InPort (Bits(8))
s.write_data = InPort (Bits(32))
s.read_data  = OutPort(Bits(32))
s.error      = OutPort(Bits(1))
s.init_reg   = Wire(Bits(1))
s.init_new   = Wire(Bits(1))
```

```
# Intantiate the core
s.core = sha1_core()
s.connect(s.reset_n, s.core.reset_n)
s.connect(s.init_reg, s.core.init)
s.connect(s.next_reg, s.core.next_in)
s.connect(s.core_block, s.core.block)
```

Figure 2: PyMTL Variables

```
@s.tick_rtl
def reg_update():
    if not s.reset_n:
        s.init_reg.next = 0
```

```
@s.combinational
def logic():
    s.core_block.value = concat(s.block_reg[0],
                                s.block_reg[4],
                                s.block_reg[8],
                                s.block_reg[12])
```

```
@s.combinational
def state_logic():
    a5 = Bits(32)
    f  = Bits(32)
```

Figure 3: PyMTL Decorators

`@s.tick_rtl` tells the PyMTL simulator to call the function at every clock tick while `@s.combinational` tells the simulator that the function is sensitive to all the signals that are read from within the function. There is no way to specifically make the function sensitive to a few signals.

A few differences that can be seen between Figures 3 and 1 are the ability to concatenate bit vectors. MyHDL does not have a way to concatenate bit vectors which means specifying the bit slice that each vector needs to be written to. PyMTL conveniently offers a `concat()` function similar to `{}` in Verilog.

PyMTL's bit vector data type is called `Bits()`, and it can be sliced and used in arithmetic and bit-wise operations. In PyMTL, all registers and wires are defined as either `InPort()`, `OutPort()`, or `Wire()` and are specified for a set number of bits as shown in Figure 2. Once these ports and wires are declared, they can be read from by their names or written to with a `.next` operator in sequential logic or a `.value` operator in combinational logic as shown in Figure 3.

2.5.3 Python Simulation

Both MyHDL and PyMTL offer a built-in simulation feature which allows designers to simulate everything in Python.

The MyHDL test bench is very similar to any other module in MyHDL. It is a function which has some member variables, sub-modules, and generator functions. It returns a handle to each of these functions and sub-modules which are used to simulate the design. MyHDL has a simulation object which is passed the tuple of generators and sub-module references returned by the test bench. Calling `.run()` on the object will simulate until completion. Optionally, a value can be passed to the run command to run the simulator for

```
tb = bench()
sim = Simulation(tb)
sim.run()
```
Figure 4: MyHDL Simulation Code

```
test_bench.sim = SimulationTool(test_bench.model)
test_bench.sim.reset()
```
Figure 5: PyMTL Simulation Code

a specified number of clock cycles. The code to initialize the test bench and simulate it can be seen in Figure 4.

PyMTL's simulation environment is a bit different. A PyMTL test bench is a function that first declares signals, an instance of the device under test, and a simulation object which is passed the device under test. The test bench then repeatedly sets its local signals and moves forward one clock cycle by calling `sim.cycle()` until the whole simulation is complete. The code to initialize the PyMTL simulation object can be seen in Figure 5.

2.6 Conversion to Verilog

After re-writing the SHA1 and RAM module in Python using both MyHDL and PyMTL and verifying their implementation with native Python simulations, we converted each of the modules to Verilog using the packages' built-in conversion features. In both cases, we were pleasantly surprised with how easy the conversion process was. Figures 6 and 7 show the code required to make the conversion. In both cases, this code has to simply be added to the test bench and on the next run, Verilog versions of the modules are generated.

MyHDL's conversion code is seen in Figure 6. Though there are two lines of code in the figure, only one is necessary for simulation. The first instantiation of the device under test is just for simulation while the second will generate Verilog during simulation. Changing the first line to the second is all that needs to be done to generate Verilog using MyHDL.

PyMTL's conversion code is similarly simple. The code shown in Figure 7 just needs to be added to the test bench. Since PyMTL's simulation tool, seen in Figure 5, and translation tool, seen in Figure 7, both elaborate the device, they need to be called on separate models to simulate at the same time as generating Verilog. In the figures, you can see that the simulation tool is being called on an object called `model` while the translation tool is called on `vModel` which is the same thing as `model` just re-declared with a different name.

3. RESULTS

After generating Verilog using both packages for both designs, all four design combinations were simulated using the original Verilog test benches and then synthesized. This section details the process of simulating and synthesizing the generated designs and the results of synthesis compared to the original Verilog modules.

3.1 Simulating and Synthesizing Generated Verilog

When we first generated the Verilog designs using MyHDL, we ran into a few errors during simulation and synthesis. They turned out to be issues with the Python code

```
# Pure python simulation
dut = sha1(tb_clk, tb_reset_n, tb_cs, tb_write_rea

# Generate verilog during simulation
dut = toVerilog(sha1, tb_clk, tb_reset_n, tb_cs, t
```
Figure 6: MyHDL Verilog Conversion Code

```
vModel = sha1()
translated = TranslationTool(vModel)
```
Figure 7: PyMTL Verilog Conversion Code

that were fine during native Python simulation but not for RTL simulation. After fixing them in Python, the generated Verilog code simulated and synthesized fine without changes.

The process was similar for PyMTL. Aside from having to rename the top Verilog module to match the test bench, the only other change required to simulate was to add the negative edge reset signal to the sequential sensitivity lists (a fallout of the reset being implicitly included by PyMTL). Synthesis spewed out a bunch of warnings but was successful overall. Looking closer at the warnings showed them to be from two main sources. The first was that PyMTL's converter generated a wire for each element in an array of bit vectors which was never accessed and the second was that it converted hex constants to decimals which, when written to variables, had undefined bit widths and could potentially overflow. Since the values were constant and less than the max value the variable's bit width could hold, the warnings could be ignored.

3.2 Synthesis Results

The Synthesis results for the SHA1 hash and RAM designs are shown in the tables below. We do not include the number of clock cycles as it was the same for all versions of both designs. Since the hash design doesn't use huge bit structures, it is not surprising to see that neither version utilizes the M10K RAM blocks on the board. Looking at the Arithmetic Logic Modules (ALMs) and Frequency maximum (Fmax), PyMTL seems to have generated an ever so slightly faster and smaller design than the native Verilog and Verilog generated from MyHDL. MyHDL on the other hand looks to be slightly worse in terms of frequency and area though the differences are minimal.

SHA1			
Metric	Native Verilog	MyHDL Generated	PyMTL Generated
Fmax* (MHz)	102.71	101.06	108.61
ALMs	960	965	958
M10K Blocks	0	0	0

RAM			
Metric	Native Verilog	MyHDL Generated	PyMTL Generated
Fmax* (MHz)	-	107.53	115.74
ALMs	1	13,807	19,729
M10K Blocks	1	0	0

*Fmax is for running at 1100mV and 85C

Though the synthesized SHA1 hash designs have similar metrics across the board, the RAM designs have wildly

different results. Synthesizing the native Verilog used one M10K RAM block and only one ALM which likely controls access to the memory block. The native Verilog design has no maximum frequency because there is no path between registers in the design.

The memory block usage contrasts sharply with the synthesis results of the Verilog generated by the two Python packages. Both generated Verilog that did not synthesize to use memory blocks and instead implemented the 1 KB RAM design with logic modules. Because the logic modules likely use a large multiplexer and registers to store the bits, these designs do have a maximum frequency and use thousands of ALMs. For MyHDL, the 13,807 ALMs correllate to 24% of the logic on the specific device while the 19,729 ALMs used by the PyMTL design equates to 35%.

From the results, it is quite clear that the Verilog generated from MyHDL and PyMTL does not utilize block memory properly. The difference in results between the compute based SHA1 hash and memory based RAM module indicate that the two Python tools may be a fine choice for compute based designs but should not be used in ones that use lots of memory. Interestingly, unlike with the SHA1 hash, PyMTL's RAM module uses significantly more ALMs than MyHDL's RAM module. This may have to do with how PyMTL creates arrays of bits in Verilog.

4. NEXT STEPS

Though the values reported for the SHA1 hash and RAM module give some indication as to the value of the Python packages, they can not be used to definitively determine the advantages and disadvantages of MyHDL or PyMTL. To do so, a larger design that uses memories would have to be written in both languages, converted to Verilog, and synthesized.

If we were to go forward with this idea, we would choose to develop a small microcontroller with an attached memory bank. Through the lens of the controller, we would hopefully see a wider variation in the two Python packages and native Verilog if there is a considerable difference between them. Depending on the design, it might also give an idea of how the use of M10K RAM blocks might affect performance if they are used in one design and not the others.

5. CONCLUSION

The original objective of this project was to determine whether MyHDL or PyMTL could act as a replacement to learning HDLs like Verilog or VHDL. Through the course of the project, it is clear that though MyHDL and PyMTL have their benefits, they are no replacement for at least a general knowledge of HDLs and hardware design. Since MyHDL and PyMTL both closely follow the structure of Verilog, new hardware designers might as well learn Verilog instead. MyHDL and PyMTL are less of packages that allow hardware designers to use Python as a high level language and more of packages that wrap HDLs with Pythonic syntax.

The need to know HDLs is further motivated by MyHDL and PyMTL not always generating Verilog code that synthesizes in the same way as native Verilog. For instance, if a designer wants to use memories, they would likely have to modify the generated Verilog to use them.

Looking specifically at MyHDL and PyMTL, MyHDL seems to be the better choice. Though the results for the SHA1 hash were slightly in PyMTL's favor and its class structure may at first be easier to understand, it is much harder to install, harder to design in, has a smaller user base, and has far less support. Our experience using MyHDL was, in comparison, much smoother. Once we determined how the generators and decorators worked, the design process was very straightforward and the conversion to synthesizeable Verilog was quick and easy.

Though MyHDL and PyMTL are not perfect substitutes for HDLs, they can be a valuable tool in a designers arsenal. Because they can both be installed easily and for free with a Python distribution, they allow users to write and simulate hardware without large commercial software. Because of the dependencies, PyMTL's benefits are limited. MyHDL, on the other hand, is much more versatile and can be installed anywhere Python packages can be installed. Instead of designers needing to log into a machine with all the required software, they could simply install MyHDL on their personal computer and design, simulate, and test extensively all in Python.

6. REFERENCES

[1] A. Caulfield, E. Chung, A. Putnam, H. Angepat, J. Fowers, S. Heil, J. Y. Kim, D. Lo, M. Papamichael, T. Massengill, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. *IEEE Micro*, pages 1–1, 2017.

[2] Amazon ec2 f1 instances. `https://aws.amazon.com/ec2/instance-types/f1/`.

[3] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM.

[4] Moein Khazraee, Lu Zhang, Luis Vega, and Michael Bedford Taylor. Moonwalk: Nre optimization in asic clouds. *SIGPLAN Not.*, 52(4):511–526, April 2017.

[5] Jan Decaluwe. Myhdl. `http://www.myhdl.org/`, 2015.

[6] Derek Lockhart, Gary Zibrat, and Christopher Batten. Pymtl: A unified framework for vertically integrated computer architecture research. In *47th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 280–292, Dec 2014.

[7] Coms-e6998-comparing-myhdl-and-pymtl. `https://github.com/saribe0/COMS-E6998-Comparing-MyHDL-and-PyMTL`.

[8] Pymtl. `https://github.com/cornell-brg/pymtl`.

[9] Quartus prime lite edition. `http://dl.altera.com/?edition=lite`.

[10] Joachim StrÃűmbergson. sha1. `https://github.com/secworks/sha1`.

[11] Peter Eston. *Intel Quartus Prime Standard Edition Handbook Volume 1 Design and Synthesis*, chapter 12, pages 826–827. Intel, 2017. `https://www.altera.com/en_US/pdfs/literature/hb/qts/qts-qps-5v1.pdf`.

Appendix: Git Folder Structure

The project Git repository [7] is organized like so:

```
./README.md
./src/
./sim/
./out/
./docs/
```

The `./src/` folder contains all the source files for the project split up by design and language. In the python folder, each of the two packages with their native and generated Verilog are included. The source folder also includes three example programs from the MyHDL and PyMTL examples. The python and Verilog files used in this project can be found at the following paths:

```
./src/ram/python/myhdl/native/*.py
./src/ram/python/myhdl/converted_verilog/*.v
./src/ram/python/pymtl/native/*.py
./src/ram/python/pymtl/converted_verilog/*.v
./src/ram/verilog/*.v
./src/sha1/python/myhdl/native/*.py
./src/sha1/python/myhdl/converted_verilog/*.v
./src/sha1/python/pymtl/native/*.py
./src/sha1/python/pymtl/converted_verilog/*.v
./src/sha1/verilog/*.v
```

The `./sim/` folder contains a Makefile, a template.sdc, and a scripts folder with a Quartus TCL script in it. Simulation and synthesis can be preformed by copying all Verilog files into a source folder at the path `./sim/src/` and running commands from the Makefile. Command line simulation using ModelSim can be run using `make sim` and synthesis using Quartus can be run using `make syn`. The Makefile needs to have the name of the test bench module and top level module defined for these commands to work. For clarity, the folder structure looks like:

```
./sim/Makefile
./sim/template.sdc
./sim/scripts/q_syn.tcl
```

The `./out/` folder contains all the output files from simulation and synthesis for the native Verilog designs, the Python versions in Python, and the generated Verilog. From the folder structure, it should be fairly straightforward which design the output files are for. The output file paths are:

```
./src/ram/python/myhdl/native/*
./src/ram/python/myhdl/converted_verilog/sim/*
./src/ram/python/myhdl/converted_verilog/syn/*
./src/ram/python/pymtl/native/*
./src/ram/python/pymtl/converted_verilog/sim/*
./src/ram/python/pymtl/converted_verilog/syn/*
./src/ram/verilog/sim/*
./src/ram/verilog/syn/*
./src/sha1/python/myhdl/native/*
./src/sha1/python/myhdl/converted_verilog/sim/*
./src/sha1/python/myhdl/converted_verilog/syn/*
./src/sha1/python/pymtl/native/*
./src/sha1/python/pymtl/converted_verilog/sim/*
./src/sha1/python/pymtl/converted_verilog/syn/*
./src/sha1/verilog/sim/*
./src/sha1/verilog/syn/*
```

Finally, the `./docs/` folder contains pdfs of the project proposal, midterm report, this report, and the final presentation. They can be found at the following paths:

```
./docs/Proposal/proposal-srb2208.pdf
./docs/Midterm/midterm-srb2208.pdf
./docs/Final/srb2208-final_presentation.pdf
./docs/Final/final-srb2208.pdf
```