**PROJECT REPORT**

**High-Level Synthesis Design Flow using open source tools**

**Project by:-**

**Batch no.: B17**

**Chapala Harsha Vardhan (1604-18-735-076)**

**Syed Muzammil (1604-18-735-093)**

**Mohamani Kalilulla (1604-18-735-120)**

**UNDER THE GUIDANCE OF**

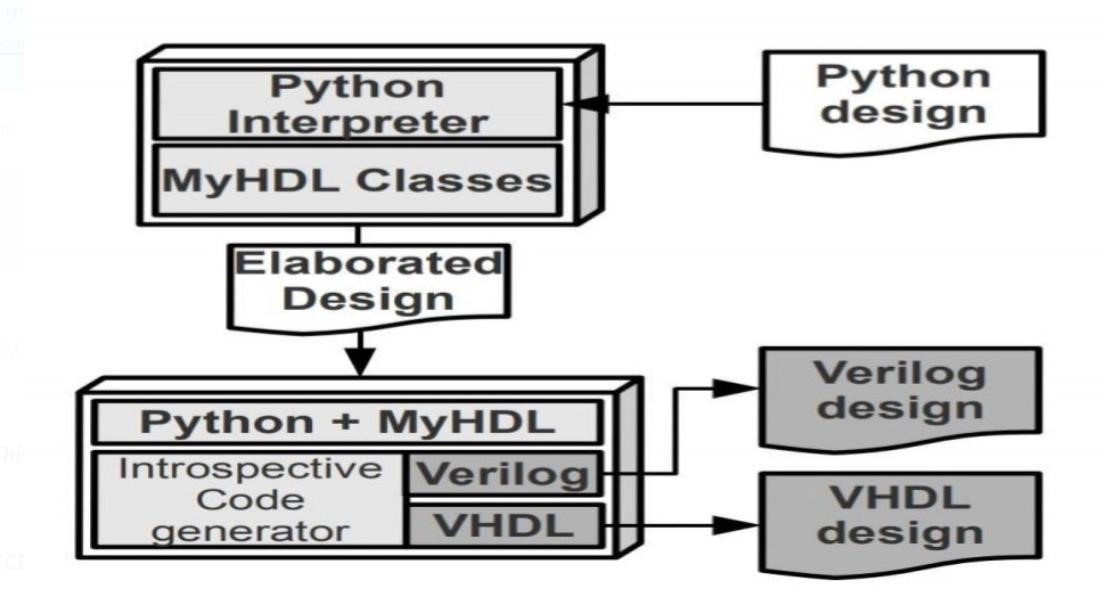
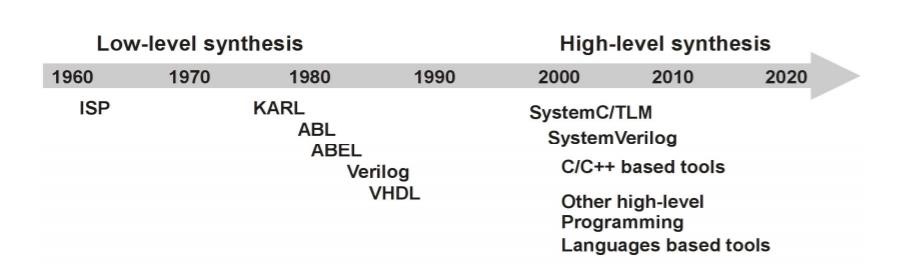
**Mr. MD. Zakir Hussain Asst .Prof., ECED MJCET.**

**Abstract**

High Level Synthesis (HLS) has many productivity advantages over traditional RTL design .The HLS Compiler is a high-level synthesis (HLS) tool that takes in high level language as input and generates register transfer level (RTL) code or Verilog HDL code that is applicable for ASIC,FPGAs. This tool accelerates verification time over RTL by raising the abstraction level for ASIC(application specific integrated circuit) ,FPGAs(field programable gate arrays) hardware design. The main focus of this presentation is how to migrate the High level language to Verilog HDL by using open source software algorithms

**1) Introduction**

High-level synthesis (HLS) is a technology that assists with the transformation of a behavioral description of hardware into an RTL model. It is considered to be part of an electronic system level (ESL) design flow. The input description is an untimed description of functionality written in C, C++ or SystemC. HLS tools also exist that use Matlab, Bluespec or OpenCL as their input language. The tools assist with the selection of an architecture that will optimize the performance, area and power of the implementation. The untimed functionality may be augmented with a description of the interface that the block is to use for connectivity into the rest of the design. This interface may define communications protocols that can in turn affect the synthesis process. HLS tools extract the available parallelism in the input description, schedule the operations, allocate the necessary resources and optimize the sharing of those resources so as to minimize the area while maintaining the necessary performance. While HLS is applicable to all types of design, the tools provide the most benefit to algorithmic blocks, such as video decoders, wireless compression schemes and encryption/decryption applications. These algorithms generally have loops, involve memory accesses and are amenable to pipelining. Commercial tools vary in several ways, including the input languages that they accept, the amount of automation they provide, quality of results and integration with other aspects of an ESL flow. Standardization efforts are underway within Accellera to define a synthesis subset of C/C++/SystemC so that users are kept away from using constructs that may only work in one implementation.



# 2 Intoduction to MyHDL

The goal of the MyHDL project is to empower hardware designers with the elegance and simplicity of the Python language. MyHDL is a free, open-source package for using Python as a hardware description and verification language. Python is a very high level language, and hardware designers can use its full power to model and simulate their designs. Moreover, MyHDL can convert a design to Verilog or VHDL. This provides a path into a traditional design flow.

## 2.1 Modeling

Python’s power and clarity make MyHDL an ideal solution for high level modeling. Python is famous for enabling elegant solutions to complex modeling problems. Moreover, Python is outstanding for rapid application development and experimentation.

The key idea behind MyHDL is the use of Python generators to model hardware concurrency. Generators are best described as resumable functions. MyHDL generators are similar to always blocks in Verilog and processes in VHDL. A hardware module (called a block in MyHDL terminology) is modeled as a function that returns generators. This approach makes it straightforward to support features such as arbitrary hierarchy, named port association, arrays of instances, and conditional instantiation. Furthermore, MyHDL provides classes that implement traditional hardware description concepts. It provides a signal class to support communication between generators, a class to support bit oriented operations, and a class for enumeration types.

## 2.2 Simulation and Verification

The built-in simulator runs on top of the Python interpreter. It supports waveform viewing by tracing signal changes in a VCD file. With MyHDL, the Python unit test framework can be used on hardware designs. Although unit testing is a popular modern software verification technique, it is still uncommon in the hardware design world. MyHDL can also be used as hardware verification language for Verilog designs, by co-simulation with traditional HDL simulators.

## 2.3 Conversion to Verilog and VHDL

Subject to some limitations, MyHDL designs can be converted to Verilog or VHDL. This provides a path into a traditional design flow, including synthesis and implementation. The convertible subset is restricted, but much wider than the standard synthesis subset. It includes features that can be used for high level modeling and test benches. The converter works on an instantiated design that has been fully elaborated. Consequently, the original design structure can be arbitrarily complex.moreover, the conversion limitations apply only to code inside generators. Outside generators, Python’s full power can be used without compromising convertibility. Finally, the converter automates a number of tasks that are hard in Verilog or VHDL directly. A notable feature is the automated handling of signed arithmetic issues.

**3 Prerequisites**

You need a basic understanding of Python to use MyHDL.

# 4 Signals and concurrency

An actual hardware design is typically massively concurrent, which means that a large amount of functional units are running in parallel. MyHDL supports this behavior by allowing an arbitrary number of concurrently running generators. With concurrency comes the problem of deterministic communication. Hardware languages use special objects to support deterministic communication between concurrent code. In particular, MyHDL has a Signal object which is roughly modeled after VHDL signals. We will demonstrate signals and concurrency by extending and modifying our first example.We define a hardware block that contains two generators, one that drives a clock signal, and one that is sensitive to a positive edge on a clock signal:

|  |
| --- |
| from myhdl import block, Signal, delay, always, now  @block def HelloWorld() :    clk = Signal(0)    @always(delay(10)) def drive clk() :  clk.next = not clk    @always(clk.posedge) def say hello() :  print(”%s Hello World!” % now())    return drive clk, say hello inst = HelloWorld() inst.run sim(50) |

The clock driver function clk driver drives the clock signal. If defines a generator that continuously toggles a clock signal after a certain delay. A new value of a signal is specified by assigning to its next attribute. This is the MyHDL equivalent of the VHDL signal assignment and the Verilog non-blocking assignment. The say hello function is modified from the first example. It is made sensitive to a rising edge of the clock signal, specified by the posedge attribute of a signal. The edge specifier is the argument of the always decorator. As a result, the decorated function will be executed on every rising clock edge. The clk signal is constructed with an initial value 0. One generator drives it, the other is sensitive to it. The result of this communication is that the generators run in parallel, but that their actions are coordinated by the clock signal.

# 5 Parameters, ports and hierarchy

We have seen that MyHDL uses functions to model hardware blocks. So far these functions did not have parameters. However, to create general, reusable blocks we will need parameters. For example, we can create a clock driver block as follows:

|  |
| --- |
| from myhdl import block, delay, instance    @block  def ClkDriver(clk, period=20) :    lowTime = int(period / 2)  highTime = period – lowTime    @instance def drive clk() : while True : yield delay(lowTime) clk.next = 1  yield delay(highTime)  clk.next = 0 |

return drive clk

The block encapsulates a clock driver generator. It has two parameters. 1. The first parameter is clk is the clock signal. A asignal parameter is MyHDL’s way to model a dfn:port:. 2. The second parameter is the clock period, with a default value of 20. As the low time of the clock may differ from the high time in case of an odd period, we cannot use the always decorator with a single delay value anymore. Instead, the drive clk function is now a generator function with an explicit definition of the desired behavior. It is decorated with the instance decorator. You can see that drive clk is a generator function because it contains yield statements. When a generator function is called, it returns a generator object. This is basically what the instance decorator does. It is less sophisticated than the

always decorator, but it can be used to create a generator from any local generator function. The yield statement is a general Python construct, but MyHDL uses it in a specific way. It has a similar meaning as the wait statement in VHDL: the statement suspends execution of a generator, and its clauses specify the conditions on which the generator should wait before resuming. In this case, the generator waits for a certain delay. Note that to make sure that the generator runs ”forever”, we wrap its behavior in a while True loop.

# REFERENCES

[1] Cieszewski, R., Linczuk, M., Pozniak, K., and Romaniuk, R.,

"Review of Parallel Computing Methods and Tools for FPGA

Technology" in [Photonics Applications in Astronomy,

Communications, Industry, and HighEnergy Physics Experiments

2013 ], vol. 8903, 2013, ISBN 9780819485823, 71 s.,

International Society for Optics and Photonics (2013)

1. Coussy, P. and Morawiec, A., [High-level synthesis: from algorithm to digital circuit ], Springer (2008).
2. Gajski, D. D., Dutt, N. D., and CH, A., [High-level synthesis], vol. 34, Kluwer Boston (1992).
3. Cong, J., Liu, B., Neuendorffer, S., Noguera, J., Vissers, K., and Zhang, Z., “High-level synthesis for fpgas: From prototyping to deployment,” Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 30(4), 473–491 (2011).
4. Babb, J., Rinard, M., Moritz, C. A., Lee, W., Frank, M., Barua, R., and Amarasinghe, S., “Parallelizing applications into silicon,” in [Field-Programmable Custom Computing Machines, 1999. FCCM’99. Proceedings. Seventh Annual IEEE Symposium on ], 70–80, IEEE (1999).
5. Liang, Y., Rupnow, K., Li, Y., Min, D., Do, M. N., and Chen, D., “High-level synthesis: Productivity, performance, and software constraints,” Journal of Electrical and Computer Engineering 2012, 1 (2012).
6. Philippe, C., Ghizlane, L.-L., and Dominique, H., “Multiple word-length high-level synthesis,” EURASIP Journal on Embedded Systems 2008 (2008).
7. Gokhale, M. and Gomersall, D., “High level compilation for fine grained fpgas,” in [FPGAs for Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on ], 165– 173, IEEE (1997).