PHDL: A Python Hardware Design Framework

by

Ali Mashtizadeh

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2007

Author . . . .
Department of Electrical Engineering and Computer Science
May 25, 2007

Certified by . . . .
Steve Ward
Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses

PHDL: A Python Hardware Design Framework

by

Ali Mashtizadeh

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 2007, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents PHDL, a new hardware design language (HDL). In PHDL digital hardware components can vary in input/output widths, target platform, and have optional inputs and outputs. PHDL enables developers to write software to make intelligent compile time decisions far beyond the capabilities of current HDLs. The approach taken is to build PHDL on the Python scripting language and to build a component library sufficiently large to design a microprocessor. As an example a microprocessor is designed in PHDL to show the practicality of the language. The example shows that on average designs can be written with less complexity than a corresponding Verilog implementation, while achieving better portability and platform specific optimizations.

Thesis Supervisor: Steve Ward
Title: Professor

Acknowledgments

I would like to thank my parents for supporting me though all of this. I would also like to thank my adviser, Prof. Steve Ward for giving me a chance to work on this project. Also to Chris Terman who's two stage Beta I used to design my example. I have to also thank everyone who read my thesis and who helped me develop my ideas during the development of my research project. I would like to thank Hubert Pham and Justin Mazzola Paluska. I have to thank my friends who supported me through this thesis Lizzy Godoy and Gabriel Bacerra. Finally, I would like to thank Chris Lyon for his guidance in planning my MIT career.

Contents

1 Motivation **10**

1.1 Problems with Current HDLs 10

1.2 The PHDL Approach 11

1.3 Summary 12

2 Previous Work **13**

2.1 Alternative Languages and HDL Generators 14

2.1.1 JHDL 14

2.1.2 C/C++ Based Languages 14

2.1.3 PyHDL 14

2.1.4 PyGen 15

2.2 Conclusion 15

3 PHDL Framework **16**

3.1 Why Python? 17

3.2 Connectors 17

3.2.1 WireConnector 18

3.2.2 ConstantConnector 19

3.2.3 AnonymousBulkConnector 20

3.2.4 BulkConnector 20

3.3 Components 22

3.3.1 Meta-Components 22

3.3.2 Low Level Implementation Components 23

3.3.3 High Level Implementation Components 25
3.4 Connections 27
3.5 PHDL's Evolution 28

4 Example: The Beta Processor **31**

4.1 Beta Processor 31
4.2 Starting a Circuit 34
4.3 Program Counter and Control 35
4.4 Register File 38
4.5 Arithmetic and Logic Unit 40
4.6 Write Back Stage 41
4.7 Control Logic 42
4.8 Creating a Component 44

5 Conclusion **46**

5.1 Future Work 46
5.1.1 Improving Component Selection 46
5.1.2 Address Debugging Challenges 47
5.1.3 General Improvements 47
5.2 Concluding Remark 48

A PHDL Framework Source Code **49**

A.1 Connectors 66

B PHDL Two Stage Beta Component **76**

C The Verilog Two Stage Beta **82**

List of Figures

3-1 Diagram of the SimpleMAC. 26

4-1 Simplified diagram of the Beta microprocessor. 33
4-2 Both forms of the instruction encoding available on the Beta. 34
4-3 Program counter control logic. 37
4-4 Diagram of the register file and surrounding logic. 39
4-5 Diagram of the Arithmetic and Logic Unit 40

List of Tables

4.1 Beta Instruction Summary. 32
4.2 List of signals used in the two stage Beta. 36

Listings

3.1 WireConnector Instantiations 18
3.2 Example use of connect to generate a ConstantConnector 19
3.3 Example of the use of an AnonymousBulkConnector 20
3.4 A Simply Memory Connector 21
3.5 Not Meta-Component 22
3.6 Not Implementation of a Core Component 23
3.7 SimpleMAC Component Implementation 25
3.8 Standard Syntax 29
3.9 New Syntactic Sugar 29
4.1 Beta Processor Inputs and Outputs 34
4.2 PC Control Logic and the Instruction Register 35
4.3 Register File and Related Logic 38
4.4 Arithmetic and Logic Unit 40
4.5 Write Back Stage and Memory Read/Write 41
4.6 Control Logic (w/o ROM) 42
4.7 Control Logic ROM 44
4.8 Beta Meta-Component 45
A.1 PHDL/__init__py 49
A.2 PHDL/Component.py 51
A.3 PHDL/ComponentImpl.py 56
A.4 PHDL/Connection.py 57
A.5 PHDL/Connector.py 57
A.6 PHDL/Dev.py 60

A.7 PHDL/HDLIOType.py 61
A.8 PHDL/HDLNetType.py 62
A.9 PHDL/HDLWriter.py 62
A.10 PHDL/HDLNetType.py 62
A.11 PHDL/Util.py 64
A.12 PHDL/VerilogWriter.py 65
A.13 PHDL/Connectors/__init__.py 66
A.14 PHDL/Connectors/AnonymousBulkConnector.py 67
A.15 PHDL/Connectors/BulkConnector.py 68
A.16 PHDL/Connectors/ConstantConnector.py 70
A.17 PHDL/Connectors/SubWireConnector.py 71
A.18 PHDL/Connectors/WireConnector.py 73
B.1 Beta Processor Meta-Component 76
B.2 Beta2Impl Implementation 76
C.1 Beta Main: beta.v 82
C.2 Beta Decoder Logic: decode.v 84
C.3 Beta Right Shift: shift_right.v 86

Chapter 1

Motivation

Manufacturers of electronic devices invest large amounts of resources in their design. To speed up the development cycle, Hardware Design Languages (HDLs) have been created. These languages, while improving the development cycle, are still insufficient to meet developers' needs.

Verilog[6] and VHDL[5] are two major industry standards for HDLs that enable hardware developers to design digital logic. Currently, a large infrastructure of tools exists to aid HDL developers to go from testing to manufacturing. Developers can target different technologies on which to produce their final design, such as custom integrated circuit and Field Programmable Gate Arrays (FPGA).

1.1 Problems with Current HDLs

Just as in the case of programming languages, abstraction is the key to rapid hardware development. Developers build many basic components and combine them into more complex components to build whole systems. Verilog and VHDL do not provide high enough levels of abstraction, for example, to automatically resize the width of an adder. In these languages it is hard, if not impossible, to implement a variable sized component in one abstract form. Another source of problems is caused by pressure on developers to optimize their circuit for a particular hardware technology. Developers are forced to optimize their circuits by targetting the design to a particular technology's component library. Integrated circuits have standard cell libraries and FPGAs often have many custom components that are available to developers. A Verilog or VHDL developer often targets these particular components to maximize the performance and minimize the area of the design. This leads to portability issues between different target technologies. Thus, hardware developers are forced to implement many versions of components that vary both the size and the target technology of that component.

Verilog's parameterization ability allows developers to construct abstract components to a limited extent. A variable size adder can be built by defining a width parameter, and then using it to set the widths of various wires. Verilog provides *for* loops that allow the instantiation of a dynamic number of logic components. The problem occurs when the developer needs to do some calculation to arrive at the parameter width. A typical example of a component we cannot construct with parameterization is a VGA timing circuit. We would like a parameterized VGA timing circuit that will generate the proper timing for given a resolution. The calculations are easily done with Python[8], but complicated and not practical to implement with parameterization in Verilog.

CMOS standard cell libraries or specialized FPGA components are often explicitly used in hardware designs. This creates difficulty for designers who need to move from one target technology to another. The solution for Verilog would be an unmaintainable set of *if* statements that choose between different implementations, each targetting a particular FPGA or standard cell library. This means that a developer's code will have to grow linearly with the number of targets for which he hopes to optimize.

1.2 The PHDL Approach

This paper introduces PHDL, a Python framework for hardware design, as a solution to the problem of enabling hardware design with a significant level of abstraction. PHDL enables the developer to write software that can make intelligent compile time decisions far beyond the capabilities of Verilog and VHDL. The PHDL framework is entirely written in Python, and it consists of core framework classes and a component library. The library contains most of the low level components that a user may need, such as logic gates, shifters, and registers. In PHDL it is relatively easy to build components that vary in size and target platform, and have optional inputs and outputs. In order to be able to take advantage of a complete tool chain from testing to manufacturing, PHDL generates Verilog as its output.

It can be very useful for a developer to be able to have their code make compile time decisions. Hardware developers can optimize special cases throughout their circuits. They can solve for complex constants and generate ROM data at compile time. For example, the decoding logic for a microprocessor may need a ROM. PHDL, as a fully programmable environment, allows developers to create a routine that could generate that ROM from a human readable table.

Component selection is a powerful and important feature. Imagine a multiplexor component for which, depending on the number of inputs added to it, PHDL will generate the corresponding N input mux in Verilog. No longer will a developer have to make a plethora of versions of a component for different sizes and widths. PHDL allows users to develop and reuse one abstract implementation of a component.

The design choice of having PHDL output Verilog allows developers to use traditional testing and development tools. It is also possible for a developer to integrate PHDL with any existing component library. Only a simple wrapper component that will instantiate an external Verilog component must be created. This feature eases the adoption of PHDL by maintaining compatibility with existing designs.

1.3 Summary

PHDL builds on Verilog to provide a more powerful way to design components. In PHDL component selection and parameterization are fully programmable. PHDL helps developers to make intelligent component selection choices at compile time. Outputting Verilog code is a simple yet important design choice, because it allows the user to use standard existing development tools.

Chapter 2

Previous Work

Previous work in the field of modern hardware design languages includes creating HDLs out of compiled languages, such as C and Java. Some more recent attempts use Python as a scripting language for rapid hardware development. These toolkits have provided better ways to develop hardware. They provide a programmable environment for hardware design. Like Verilog and VHDL these languages have been designed to be very concrete, meaning that developers get the hardware they design; no behind the scenes optimization takes place.

PHDL is different from these languages in three respects. First, each component selects the best implementation of itself as a high level optimization. This enables easy porting from one target technology to another. It effectively eliminates the time a developer would have to go through to replace implementations of components, such as when a better algorithm is discovered. Secondly, PHDL attempts to satisfy a user's parameters and solve for any missing parameters. PHDL is good at helping developers build circuits at high level. Users do not need to go through the detail of specifying all parameters because if possible PHDL solves for any missing parameters. The last major difference is that PHDL generates Verilog, while all these languages generate Electronic Design Interchange Format (EDIF) as their final output. EDIF is the output most synthesis tools provide. It is a standardized format used to describe netlists. EDIF is more platform dependent than Verilog since it describes the particular logic of a technology. Most of the languages below are only targetted to a few FPGAs. PHDL on the other hand can generate non-platform specific Verilog if the user so desires.

2.1 Alternative Languages and HDL Generators

2.1.1 JHDL

JHDL[1] is a hardware toolkit written in Java, for reconfigurable hardware designs. The main goal is to provide designers with a way to build hardware that can change over time. This enables it to take an interesting approach to FPGA resources, such as built-in multipliers, by allocating and deallocating them during a hardware operation. JHDL offers a solution to the unique problem of building dynamically modifiable hardware. JHDL suffers from the compile time step slowing the development cycle. In JHDL a developer writes a piece of JHDL code that inturn must be compiled then it can be generated into an actual hardware design.

2.1.2 C/C++ Based Languages

Three of C toolkits exist for hardware design are System-C, Handle-C, and PAM-Blox. These are all compiled languages based on C that are turned into logic. System-C[2] and Handle-C[3] are both commercial products that are in use to day. All three systems can slow the development process because of the compilation step that is needed.[4] The compilation step is not needed in systems like PyHDL[4] and PHDL, which are based on interpreted languages. PAM-Blox was extended by PyHDL to create a python interface.

2.1.3 PyHDL

PyHDL is one of the early developments towards a scripting language based HDL. The compiled toolkit PAM-Blox is used as a backend for PyHDL. It gives PyHDL a component library through Python language bindings. PyHDL allows for scriptable creation of hardware design. PyHDL enables rapid hardware development. It eliminates the need for compilation time. PyHDL makes a great argument for why Python should be used over a compiled languages. PyHDL does not address the automatic configuration features that PHDL does.

2.1.4 PyGen

A python based hardware generator PyGen was created to offer power optimization. Using MATLAB to do the calculations as a backend, PyGen can help engineers build low power systems.[7] This research while interesting is trying to solve a more specific problem than PHDL. PHDL does not solve optimizations, instead it satisfies parameters and solves for any missing parameters. The user is allowed to specify a global optimization goal. This optimization goal could be one of power, speed, or area. PHDL is primarily interested in matching all the required parameters. Given two equally satisfactory components it may then select one based on the global optimization effort. PyGen is a very specific solution to the power problem.

2.2 Conclusion

The PHDL Framework offers a simple and powerful way to do hardware design. PHDL empowers the developer by making abstraction and component reuse easy. It offers a fast rapid development platform. It is a very general Framework that will not currently solve specific problems, such as power optimization. The current PHDL language is very capable of building intelligent and flexible components.

Chapter 3

PHDL Framework

The PHDL Framework allows the user to create intelligent digital hardware designs. The user builds components and systems using the following three major types of objects: connectors, components, and connections. Connectors implement various I/O interfaces and allow for special operations; for example, concatenating several connectors into a single wire of width that equals the sum of the individual connectors. Components come in two flavors, meta-components or vanilla components. Meta-components take care of selecting the optimal component to use in a design. Vanilla components, usually just referred to as components, implement the actual logic by instantiating subcomponents and/or generating Verilog. The Connection object is an internally used class to tie connectors together.

The user can specify global parameters to affect the component selection process. Currently, the system accepts a target and a goal parameter. The *target* parameter tells us for what target technology we are generating the code for. For example, a Virtex II FPGA, CMOS standard cell library, or a technology independent Verilog. The *goal* parameter is a general optimization goal, which can either be for size, speed, or power. In the future, some components may allow for other global parameters. By changing these parameters the design will optimize differently to try to match the user's needs. Currently, given a set of parameters, PHDL should deterministically build the same design every time. The majority of parameters and the user preferences are specified during component instantiation. The global parameters are implemented as a dictionary, to make it simple to add new global parameters in the future.

PHDL allows users to take advantage of code reuse by allowing components to be designed without knowledge of widths of buses and other parameters. Components designed by the user take advantage of PHDL's automatic component selection and parameter solving abilities. As hardware changes, or better algorithms become available, a user's design can take full advantage of them, since individual subcomponents will be substituted by PHDL. Only functionally identical components can be substituted transparently. To keep the implementation of PHDL simple, the design does not use individual scoring functions. Instead, the meta-component has a select function to choose the best implementation. This allows us to keep the selection scheme simpler. With the current limited number of components, this is a practical solution to get PHDL working and improve its features quickly. In the future, probably a decentralized scoring system could be developed for selecting components and making it easier to add new ones to PHDL.

3.1 Why Python?

Python seems to be a natural choice as a language to build PHDL on. Python has widespread popular use as a scripting language and it is mature. In python, it is possible to overload how instance variables of classes are set, read, and deleted. This enables PHDL to have a natural way for adding connectors and subcomponents to a particular component. The language was an important choice allows PHDL to have an attractive syntax without needing additional parsing/compiling times. A key point from the PyHDL[4] project is that compile times slow development. C and Java, cannot dynamically add/remove instance variables to a class which leads to an uglier syntax or requires a compiler.

3.2 Connectors

Connectors represent actual wires and special collections of wires. The core set of connectors built into PHDL take the brunt of the complexity in intelligent connectors. The basic connector that represents Verilog wire or reg types is the WireConnector.

3.2.1 WireConnector

The most often used connector is the WireConnector. It is used to make arbitrary width buses, and can be passed several parameters. An *HDLIOType* which specifies whether it is an Input, Output, Inout, or Wire port. The user may also specify a *HDLNetType*, which specifies if the bus is a wire, register, or several other standard Verilog types. In addition, a wire may be fixed with a specific width. Otherwise, the PHDL framework will attempt to solve for the bus width. After all components have fully propagated all their properties, any remaining unconfigured connectors and components will give error messages. Below is an example of four WireConnectors added to an empty component.

```
comp = Component("examplecomponent","exampleinstance")
comp.A = WireConnector(HDLIOType.Input)
comp.B = WireConnector(HDLIOType.Output)
comp.C = WireConnector(HDLIOType.Output,HDLNetType.Reg,width = 5)
comp.D = WireConnector(HDLIOType.Wire)
```

Listing 3.1: WireConnector Instantiations

In Listing 3.1 above, four connectors are instantiated named A, B, C and D, which are connectors inside the component comp. The first line instantiates a plain empty component, and in the following lines the connectors are added to the component. Connector A is a simple input connector with an unspecified width. Connector B, likewise, is a simple output connector. Connector C is an output connector that is a register, and has a width of 5 bits. Finally, connector D is not an input/output connector; it is an internal wire used to tie subcomponents together. This code will generate a verilog module with some inputs and outputs, but no logic. Later, during the discussion of components, it will become clear how to specify relationships between wire bus widths and other parameters.

The PHDL way to address components and connectors is to access them as attributes of a Python class. Originally, the design would use paths that were implemented by using strings that fully specify the path to a connector. This leads to messy and harder to read code. It also looked very non-pythonic. The better solution leads to an improvement in code readability. However, it is more difficult to determine where the connection is made, whether it is made at a component or its parent. This problems has mostly been solved, possibly corner cases still exist. The Beta Processor is a real world example that shows that most of the corner cases are correctly implemented. Keep in mind that implementing paths using strings does not avoid all of these problems. The downside of the current implementation is that it resulted in complicating PHDL's connection code. The cost is justifiable, because the end result is an easier to use language. In some places, strings are still in use. For example, there are methods that will allow the use of regular expressions to iterate over connectors with certain names. This particular method is useful in implementing components with a variable number of connectors.

Connectors are usually tricky to write and require many methods to be implemented. Existing connectors provide the user with almost every task that they would like to accomplish. In addition to the WireConnector, there are ConstantConnector, AnonymousBulkConnector, and BulkConnector. Users wishing to build connectors that are in groups, such as busses, should refer to Section 3.2.4 of this thesis.

3.2.2 ConstantConnector

In the special case that a literal constant is passed into a Connect statement, a ConstantConnector is instantiated. This connector should not be instantiated directly, as it is used as a bookkeeping hack. It helps PHDL implement sanity checks, and reduces the problem of connecting a constant to a wire. Listing 3.2, shows how to connect the wire *comp.addr* to a constant 0x12. Line 2 shows an adder that has a constant four attached to it. This example implements a PC + 4 function for the beta processor. An intelligent version of the adder may try to realize that connector *b* is a constant value and simplify its logic. Given that such a component exists, PHDL will take advantage of it.

```
1  Connect(comp.addr,0x12)
2  comp.adder = Adder(a = comp.pc,b = 4,o = comp.pcplusfour)
```

Listing 3.2: Example use of connect to generate a ConstantConnector

3.2.3 AnonymousBulkConnector

The AnonymousBulkConnector is the equivalent to Verilog's curly brackets. It allows the developer to concatenate a bunch of WireConnectors and constants into one single connector. The Beta Processor example uses AnonymousBulkConnectors in several places. The primary way to construct an AnonymousBulkConnector is by using the *Add* method to add all the necessary connectors and constants. When adding wires using the *Add* method, an optional second argument specifies the number of times to repeat that connector. This feature makes wire manipulation operations, such as like sign extending a fixed width wire, very easy to implement in PHDL. The alternate way of constructing an AnonymousBulkConnector is to pass a coma separated list of connectors as an argument to the constructor.

```
1  beta.romaddr = AnonymousBulkConnector()
2  beta.romaddr.Add(beta.op[31:26])
3  beta.romaddr.Add(beta.irq)
4  beta.romaddr.Add(beta.z)
5  beta.romdata = AnonymousBulkConnector(beta.aluop,beta.pcsel,...)
```

Listing 3.3: Example of the use of an AnonymousBulkConnector

In Listing 3.3, there are two examples of AnonymousBulkConnectors. Lines 1-4, contain a simple example concatenating the wire *beta.op[31:26]* to *beta.irq* and *beta.z*. Line 5, shows the shorthand way to create an AnonymousBulkConnectors.

3.2.4 BulkConnector

Most connectors that users would like to create are just collections of other connectors. For example any bus, such as PCI or I^2C, in the real world is just a collection of wires. PHDL calls this the BulkConnector, it allows users to easily implement a collection of a few connectors. Simply add the subconnectors to instance variables, and the BulkConnector class takes care of the rest. An example of this is a memory connector, shown in Listing 3.4, with four subconnectors memory address (*ma*), read data (*rd*), write data (*wd*), and memory write enable (*we*).

```
 1 class MyMemoryConnector(BulkConnector):
 2   def __init__(self,master = True,addrwidth = None,buswidth = None):
 3     BulkConnector.__init__(self)
 4     if (master):
 5       m = HDLIOType.Output
 6       s = HDLIOType.Input
 7       mt = HDLNetType.Reg
 8       st = HDLNetType.Wire
 9     else:
10       m = HDLIOType.Input
11       s = HDLIOType.Output
12       mt = HDLNetType.Wire
13       st = HDLNetType.Reg
14     self.ma = WireConnector(m,mt,addrwidth)
15     self.rd = WireConnector(s,st,buswidth)
16     self.wd = WireConnector(m,mt,buswidth)
17     self.we = WireConnector(m,mt,width = 1)
18     self.Width = buswidth
19     self.AddrWidth = addrwidth
20
21   def ConnectorConstraints(self,prj):
22     ConfigureEqual(prj,self.rd,self.wd,"Width")
23     ConfigureEqual(prj,self,self.wd,"Width")
24     ConfigureEqual(prj,self,"AddrWidth",self.ma,"Width")
```

Listing 3.4: A Simply Memory Connector

The constructor at line 3 calls the super class' constructor constructor. Lines 4 through 13 just allow us to easily select if this bus connector is a master or a slave. In this example it may be a CPU (master) or a block of Memory (slave). Lines 14 through 17 instantiate the four connectors. When they are assigned to instance variables, BulkConnector takes care of the magic and ties them properly to itself. The *addrwidth* and *buswidth* parameters allow the user to set a default width to the connector. The write enable pin is fixed with a width of one. In the current implementation of PHDL, parameters already set cannot be changed during the configuration phase. Once we set it to one we no longer have to worry about PHDL changing it later. The last four lines implements the method *ConnectorConstraints*, which verifies that our data connectors have the same width. This is the same mechanism that is used by components to add configuration constraints. Though optional, Lines 21 and 22 exist to keep the *Width* and *AddrWidth* variables in sync with the width of the wires themselves. In this example, the *Width* variables are not used, but possibly a component developer may wish to access them.

3.3 Components

Components bundle up connectors, connections and sub-components into one neat abstraction. Meta-Components have the job of making intelligent choices on which implementation to select. Components can be implemented as core components that generate Verilog code directly. Most component implementations are composite components, which build upon other components. Composite components can take advantage of new components that are added in the future. For example, a user may implement a new multiplier for a specific technology, such as a FPGA. Existing components that need a multiplier may use the newly created one, if it better suites the component's needs. An example of a small core component is the Not gate shown below. Core components either generate verilog and/or are specific to a target technology.

3.3.1 Meta-Components

```
 1  class Not(Component.Component):
 2    def __init__(self,width = None,**cons):
 3      self.Init()
 4      self.Width = width
 5      self.i = WireConnector(HDLIOType.Input,HDLNetType.Wire,width)
 6      self.o = WireConnector(HDLIOType.Output,HDLNetType.Wire,width)
 7      self.AutoConnect(cons)
 8
 9    def ConfigureComponent(self,prj):
10      if self.Instance is None:
11        self.Instance = NotImpl()
12        self.InitInstance()
13      self.Instance.ConfigureComponent(prj)
14      self.Instance.GenerateName()
15
16    def ParameterizationCheck(self,prj):
17      pass
```

Listing 3.5: Not Meta-Component

Listing 3.5 is a meta-component. It is only implemented by a NotImpl component. In lines 5 and 6, the input and output connectors *i* and *o* respectively are created. In the constructor of the component, first a call to the *Init()* method is made. The *Init()* method is inherited from the Component class. After that the meta-component will initialize any needed variables and add all connectors. The call to *AutoConnect(cons)* will allow the user, at construction, to automatically connect any connector in the component to another connector. The user will pass in the connectors as parameters to the constructor of the meta-component.

ConfigureComponent is the function that should select a component implementation which can include another more specialized meta-component. The call to *InitInstance()* allows the component to automatically tie together variables, connectors, between the meta-component and the instance. Otherwise, *ConfigureComponent* should call the Instance's *ConfigureComponent* and ask it to generate a name. *GenerateName* is an optional function that generates a unique name that fully specifies the configuration of a component. For components such as a ROM, which may contain instance specific data, *GenerateName* should not be implemented in the instance so that a default implementation provides a guaranteed unique name.

ParameterizationCheck allows a component to check certain instance parameters for sanity checks just before generating the code. Usually, we should test that important variables have been computed, and that all necessary wires are connected. In most component implementations the *ConfigureComponent* method removes any optional unconnected connectors.

3.3.2 Low Level Implementation Components

```
class NotImpl(ComponentImpl.ComponentImpl):
  def __init__(self):
    pass

  def ConfigureComponent(self,prj):
    for e in self.Connectors:
      e.ConfigureConnector(prj)
    ConfigureEqual(prj,self,self.i,"Width")
    ConfigureEqual(prj,self,"Width",self.o,"Width")

  def ParameterizationCheck(self,prj):
    if self.Width is None:
      Dev.Debug(Dev.Error,"Error: NotImpl failed to configure...")
    if (self.Width != self.GetConnector("i").Width)
        or (self.Width != self.GetConnector("o").Width):
      Dev.Debug(Dev.Error,"Error: Consistency check in NotIm ...")

  def GenerateName(self):
    self.Name = "Not" + str(self.Width)

  def GenerateVerilogHDLBody(self,hdlwriter):
    hdlwriter.Write("   assign o = ~i;\n")
```

Listing 3.6: Not Implementation of a Core Component

Listing 3.6 shows the actual implementation of Not component. It implements four methods: *ConfigureComponent*, *ParameterizationCheck*, *GenerateName*, and *GenerateVerilogHDL-Body*. The minimum required to implement a vanilla component is to create a constructor. This example generates Verilog code directly and thus is required to implement the *ConfigureComponent* and *ParameterizationCheck* methods.

In the constructor of NotImpl, developers may do some operations such as adding internal sub-components and connectors. As the NotImpl is very simple, we do not have add any internal connectors or sub-components.

The *ConfigureComponent* method allows us to put constraints on variables in the wires and in the component itself. This method is called iteratively, and lasts until all components have settled to a final solution of their configuration. It starts by the *ConfigureComponent* method calling all its sub-connector's *ConfigureConnector* methods. The lines following the connector loop, two configure ConfigureEqual statements ensure that the Width parameters are equal. The second line shows how to specify two different parameter names. There are other Configure statements available, such as ConfigureLog2 and ConfigureExp2 that give a logarithmic or exponential relationship. In both of these cases the operation is rounded to the nearest integer.

Currently, all components only pass parameters around and never replace existing parameters with new values. If an inconsistency is reached, PHDL will throw an error, rather than trying to resolve this inconsistency. This eliminates the problem of PHDL not terminating and staying in an infinite loop attempting to solve a system of parameters. As an additional safety check, a variable holds the maximum number of iterations allowed (the default is 50). In the Beta Processor example only about 5 iterations are needed to completely solve all wire widths.

The *GenerateName* method allows users to use a unique name that fully specifies a particular component's configuration. This is not always possible or practical. For example, building multiple ROMs with different data. In these cases the *GenerateName* method is left out and the default implementation generates guaranteed unique name.

In the NotImpl component we want to output Verilog code directly. We must implement the *GenerateVerilogHDLBody* method that is called when we are allowed to output Verilog code directly. Most users will design components that do not generate Verilog or any other HDL directly, instead they will use subcomponents. As in the NotImpl component, the body only consists of code the declaration of local connectors and the header and footer of a Verilog module are generated by PHDL.

ParameterizationCheck is used as a final sanity check before generating the Verilog. *GenerateName* allows users to generate a unique name that completely specifies the type of component. This is not always possible or easy to do, but this method is optional and a randomly generated unique name will be used instead. *GenerateVerilogHDLBody* is used by the core components to output Verilog. In Listing 3.6 the *GenerateVerilogHDLBody* just outputs an assign statement to implement the not gate.

3.3.3 High Level Implementation Components

Multiply and Accumulate component is an example of a higher level component. The SimpleMAC component was created as an early demonstration of the PHDL Framework. Listing 3.7 shows the code for SimpleMAC that uses an old style syntax with many connect statements creating connections between various wires.

```
class SimpleMAC(Component.Component):
  def __init__(self,width = None):
    self.Init()

    # Inputs
    self.clk = WireConnector.WireConnector(HDLIOType.Input)
    self.reset = WireConnector.WireConnector(HDLIOType.Input)
    self.a = WireConnector.WireConnector(HDLIOType.Input)
    self.b = WireConnector.WireConnector(HDLIOType.Input)
    self.out = WireConnector.WireConnector(HDLIOType.Output)

    # Subcomponents
    self.macmult = Multiplier.Multiplier(width = width)
    self.macadder = Adder.Adder()
    self.macregister = DLatch.DLatch()

    Connect(self.macregister.q,self.out) # Final Output

    Connect(self.a,self.macmult.a) # Multiplier Inputs
    Connect(self.b,self.macmult.b)

    Connect(self.macmult.o,self.macadder.a) # Adder Inputs
    Connect(self.out,self.macadder.b)
    Connect(self.macadder.o,self.macregister.d) # Register Input

    Connect(self.macregister.clk,self.clk)
    Connect(self.macregister.reset,self.reset)

  def GenerateName(self):
    if (self.Name is None) and not(self.out.Width is None):
      return "SimpleMAC" + str(self.out.Width)
```

Listing 3.7: SimpleMAC Component Implementation

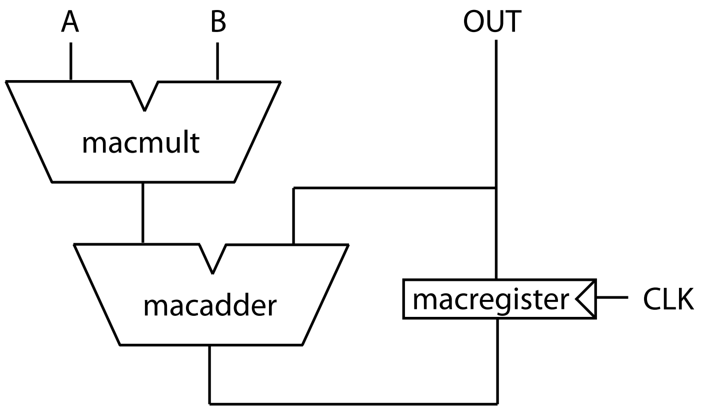

Figure 3-1: Diagram of the SimpleMAC.

SimpleMAC generates a multiply and accumulate component, show in Figure 3-1 is the corresponding schematic. In this simple component we just define our inputs and outputs then connect our components together. The constructor of SimpleMAC just wires together the subcomponents together. SimpleMAC does not output any Verilog on its own. Since its a very simple example and generates no Verilog code on its own we don't even need to implement a *GenerateVerilogHDLBody* Method. The SimpleMAC places no additional constraints between wires and subcomponents so it does need to overload the *Configure-Component* or *ParameterizationCheck* methods. All constraints come from SimpleMAC's subcomponents. Although, if a developer wishes he could apply extra constraints by overloading the *ConfigureComponent* and *ParameterizationCheck* methods. To implement *GenerateName* in this component just concatenates the width of the component to the name "SimpleMAC." Since *ConfigureComponent* is not implemented, we just get the bus width from the output connector out.

3.4 Connections

As previously shown in Listing 3.7, Connect statements tie various connectors together. Automatically a connection is instantiated that aids a connector to iterate over its siblings and to configure each other with various parameters. The Connection class is not something to implement its a single generic implementation used only to maintain lists of all connected connectors. The only things a developer needs to know about the connection class is two instance variables is *Connectors* and *LocalConnector*. *Connectors* is a dictionary of all connectors that are connected to the instance of this component through a *LocalConnector*. To get the local component of a Connection we simply can look at *LocalConnector.Component*.

Connections are simple classes that manage lists of connectors that are tied together. They are mostly for internal use and the user will not need to interact with them at all. Specialized connectors may need to use the methods and instance variables provided by the Connection class. Two important instance variables are provided: *Connectors* and *LocalConnector*. *Connectors* points to all connectors that are tied to the local connector. *LocalConnector* is the local connector itself. In the current PHDL implementation we don't generate assign statements; instead all connectors in the current component that connect to a parent components connector go in the *Connectors* dictionary. The *LocalConnector* the input or output connector itself. The choice not to make different kinds of connections was a deliberate one, as it would add unneeded complexity. The choice was either put all the complexity in the Connector class or spread it with a greater overhead into the various connection classes. Placing the complexity in the Connection class can be very difficult to maintain when compatible connectors of different types exist. For example, a WireConnection class would have to deal with special case when an AnonymousBulkConnector connects to a WireConnector.

3.5 PHDL's Evolution

As PHDL evolved, a handful of problems occurred with implementing the meta-component and component relationship. A user who wished to build a meta-component/component would be burdened with extra code or some inconsistencies may arise during configuration phase. The inconsistencies occur when the component has a copy of the meta-components variables or objects. This can lead to one component left with variables that are not configured. Originally, to fix this problem all access to certain variables were stored in the meta-component. This lead to slightly messier code that tended to be unreadable. Inheritance did make sense and was not possible, because we wanted to take an instance of a meta-component and essentially convert it into a component. This transformation is not easy to implement because of all the references that meta-component may have pointing to it. What was needed was to verify that all variables and parameters were centrally stored in one single class instance. Thankfully, Python offered a solution that may make some cringe at first. Python allows for *__setattr__* and *__getattr__* to intercept variable access to non-existent variables. Using these two methods, the Component class can provide a component with the ability to link itself with a meta-component. This also simplifies the way Connectors are added to a component. In the past, we used a method call to add a connector. Currently, users may just assign to an instance variable a connector and it is automagically linked to the component. The behavior becomes almost like inheritance that is dynamically modifiable. The term used is virtual inheritance.

PHDL does uses pythons ability to manipulate a class's namespace using *__setattr__* and *__getattr__* methods. The major example that affects the user is in Component design. Using this feature, PHDL can give the user a sort of run time dynamic inheritance. It works by chaining Components together in a linked list. Every time something accesses a variable only the bottom level meta-component to store the variables. In this design only single inheritance is permitted. It behaves like inheritance because a meta-component can import new functionality by linking itself with its component instance. All currently implemented components only have a meta-component and a component. Although, in the foreseeable future users may want more complicated hierarchies of meta-components that select more specialized meta-components, which in turn select an implementation component.

This implementation of virtual inheritance enables many simplifications to the framework to be made. Before implementing virtual inheritance problems occurred with properties not being propagated properly due to replacing instances of connectors in the components internal lists. Virtual inheritance eliminates this problem and enables more powerful reuse of code since we can also access functions.

During the development of several components it became clear that syntactic sugar was needed to help simplify commonly used code. The two main syntactic sugars were operator overloading for connectors and the short hand notation for connecting wires through the component's constructor.

Operator overloading for PHDL instantiate the corresponding PHDL component. Using a '+' operator, for example, will automatically instantiate a PHDL ADDER with two inputs. This design choice allows these operators to take full advantage of the PHDL framework. Implementing this feature is quite difficult; it requires improving the way connections are made in order to allow for automatically created connectors to get renamed and/or merged with a local connector that the user constructed.

An arguably more important syntactic improvement reducing the number of CONNECT statements that allows developers to reduce the code size greatly. To implement this the constructor of meta-components can take in arbitrary number of parameters, and a helper function is called that connects connectors together. The Beta processor makes extensive use of this feature. Listing 3.8 shows the general form and Listing 3.9 shows the new short hand form for instantiating an adder.

```
comp.adder = Adder()
Connect(comp.adder.a,inputone)
Connect(comp.adder.b,inputtwo)
Connect(comp.adder.o,output)
```

Listing 3.8: Standard Syntax

```
comp.adder = Adder(a = inputone,b = inputtwo,o = output)
```

Listing 3.9: New Syntactic Sugar

For components with many more inputs this feature can reduce code size easily in half. The only caveat is that the connector being passed in must already exist! Although, that should not matter because either component can tie to the output or input of another. Thus, a Connect statement should not be required.

During the design of the Beta it became clear that this syntax can reduce code complexity. The Beta was shortened by more than half of the number the lines of code and the file size. It seems that forcing the condition that the connector must already exist could be bothersome. However, during the design of any component, including the Beta, the logical design method was to follow a flow through a major data path. This means that a bunch of logic devices get chained together in sequence; only a few connectors are left as Connect statements.

Chapter 4

Example: The Beta Processor

A large project is needed to show off PHDL's features and prove that it is a practical tool to build hardware. The Beta processor that is used as a teaching tool for MIT 6.004, an introductory course in computer architecture, serves as a major example. A Verilog implementation of the Beta exists allowing for a good comparison. Some of PHDL's shortcomings that were discovered during the development of Beta were corrected and some will be addressed by future development. For example, the AnonymousBulkConnector was created to make it easy to concatenate multiple connectors together. Many improvements and bugfixes to the component library were made. Many of the corner case bugs in PHDL's core were also removed by developing and testing the Beta.

The choice was made to avoid using operator overloading, since this practice is questionable. Operator overloading can obscure the implementation. It is sometimes convenient to use the shorthand of overloading an operator, but it would have saved about ten lines of code in the Beta. In cases were lots of algebraic equations are used, operators can save large amounts of code. If one were building digital signal processing circuitry, then it may be useful to use operator overloading.

4.1 Beta Processor

The Two Stage Beta Processor was inspired by Chris Terman's two stage Beta implemention in Verilog. The initial Two Stage Beta Processor was implemented as a circuit. An outline

Opcode	Description
ADD[C]	Addition
SUB[C]	Subtraction
MUL[C]	Multiplication
DIV[C]	Division
CMPEQ[C]	Compare Equal
CMPLT[C]	Compare Less Than
CMPLE[C]	Compare Less Than or Equal
AND[C]	Bitwise And
OR[C]	Bitwise Or
XOR[C]	Bitwise Xor
SHL[C]	Shift Left
SHR[C]	Shift Right
SRA[C]	Arithmetic Shift Right
LD	Load Word
ST	Store Word
JMP	Jump Register
BEQ	Branch Equal to Zero
BNE	Branch Not Equal to Zero
LDR	Load Word PC-Relative

Table 4.1: Beta Instruction Summary.

of how to turn the Beta into a component is in Section 4.8. The full source code is available in Appendix B.

Figure 4-1 shows the high level diagram of the Beta Processor. The Beta is a simple 32 bit RISC architecture processor. All instructions are 32 bits wide. Table 4.1 contains a summary of all the instructions available in the Beta processor. The Multiply and Divide instructions are not implemented in the Beta discussed in this chapter. The "C" suffix that exists in some opcodes is for the constant forms of the opcode. Figure 4-2 shows the two encodings of instructions in the Beta. The register form operates on two registers and writes the result to a third one. The constant form operates on a register and a sign extended constant and writes the result to another register.

In this chapter, the problem of building the Beta, is broken down into four main chunks: program counter control, register file, arithmetic and logic unit (ALU), and control logic. In Section 4.3 the implementation of the program control circuit is explained in detail. Section 4.4 contains the register file implementation. It utilizes PHDL's built in register file

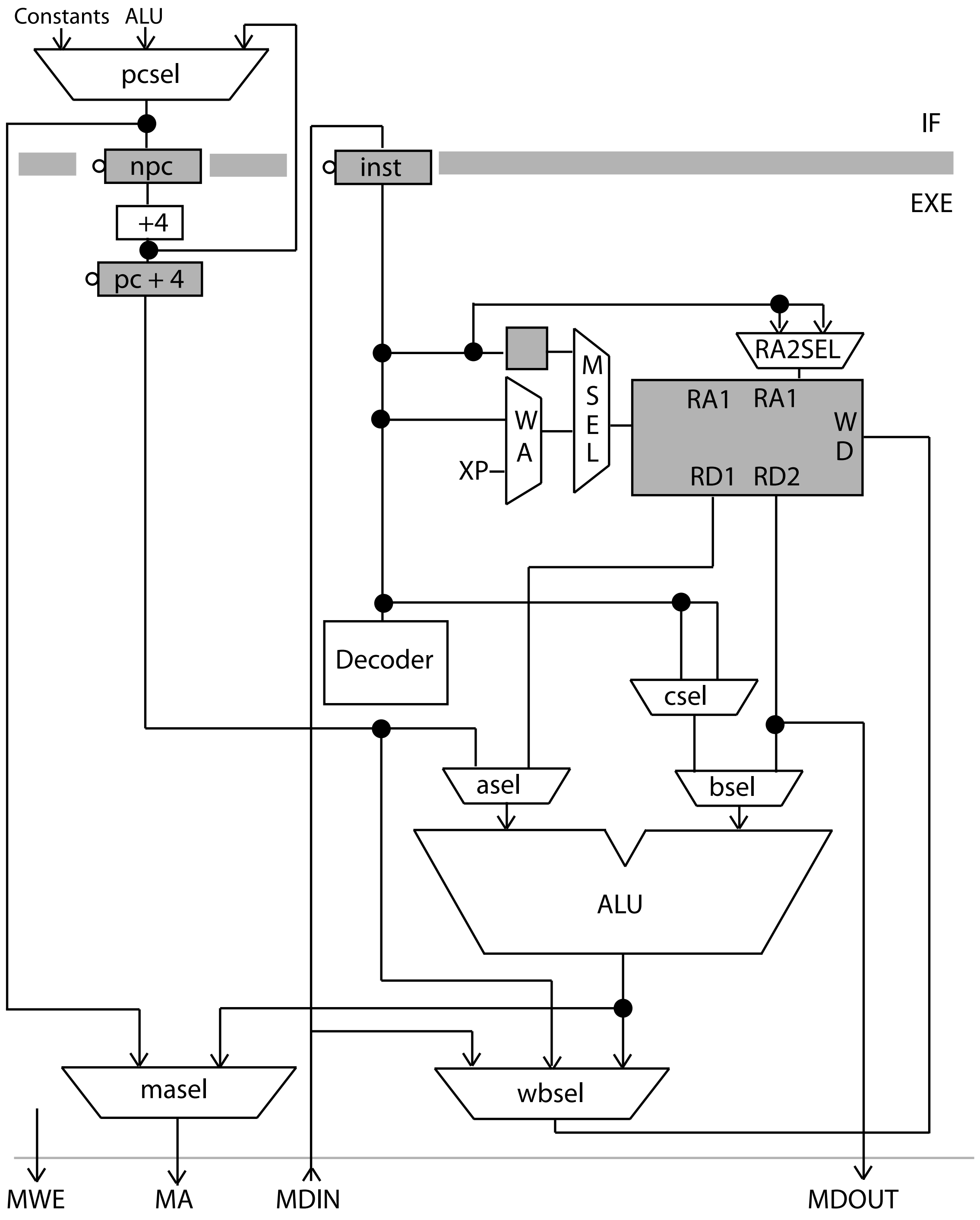

Figure 4-1: Simplified diagram of the Beta microprocessor.

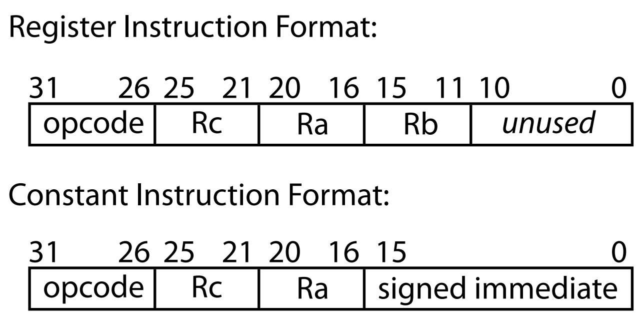

Figure 4-2: Both forms of the instruction encoding available on the Beta.

component. Thus, the black box of the register file is never implemented, as a very general implementation is built in to PHDL. Instead, the surrounding logic is what is developed in that section. The ALU is one of the more involved sections as no generic enough of an implementation is available for use. The ALU is implemented from scratch in Section 4.5. Finally, Section 4.7 explores the implementation of the control logic. It is implemented with a ROM and extra logic.

4.2 Starting a Circuit

A Circuit in PHDL is just an empty Component with connectors and components added to it. Then, it is passed to the Project class along with any global user parameters. PHDL will solve missing parameters and recursively construct the circuit. If all parameters are solved to valid values, PHDL can then generate the corresponding Verilog code.

Listing 4.1, shows the beginning of the code to instantiate the Beta. To avoid warning messages the inputs and outputs have been marked used. This is done by calling the *SetUsed()* method. The call can be compactly written in one line since the *SetUsed()* method conveniently returns *self*. This is completely optional and is only used to suppress some warning messages.

```
comp = Component.Component("betainstance","beta2")
comp.clk = WireConnector(HDLIOType.Input).SetUsed()
comp.reset = WireConnector(HDLIOType.Input).SetUsed()
comp.irq = WireConnector(HDLIOType.Input,width = 1).SetUsed()
comp.ma = WireConnector(HDLIOType.Output,width = 32).SetUsed()
```

```
6 comp.mdin = WireConnector(HDLIOType.Input).SetUsed()
7 comp.mdout = WireConnector(HDLIOType.Output).SetUsed()
8 comp.mwe = WireConnector(HDLIOType.Output).SetUsed()
```

Listing 4.1: Beta Processor Inputs and Outputs

Line 1 shows the creation of the component assigned to the variable comp. In the following lines the input and output connectors are attached to it. In the example above the minimum number of widths possible were specified. In this case only the memory address has a width of 32 and the *irq* wire has a width of one. All other wires receive their parameters from their relationship to these wires. Notice that some wires connect to components that already force a relationship. For example, the DLatch component forces *clk* and *reset* to have a width of one. Other constraints can come from using sub-wires as will be shown later on.

A handful of internally used wires were declared without any parameters. Table 4.2 summarizes all the defined wires and their purpose. Only two of the 29 signals have the width parameter defined. The remaining signals will have their width parameters solved by PHDL.

4.3 Program Counter and Control

The program counter control circuit is what enables processors to implement branches, interupts, and other control flow operations. The Beta requires a fairly simple program counter implementation. The three constant targets that are required are reset, illegal opcode, and irq. In addition, the PC + 4 and a jump target from the ALU are also possible. Figure 4-3 shows what must be implemented. The Beta uses the most significant bit(MSB) of the PC to disable interupts. The control logic has been modified to inspect the MSB.

```
comp.npc = DLatch(reset = comp.reset,clk = comp.clk,
    en = comp.not_msel.o)
comp.pcinc = Adder(a = comp.npc.q,b = 4)
comp.pcplusfour = DLatch(reset = comp.reset,clk = comp.clk,
    en = comp.not_msel.o,d = comp.pcinc.o)
comp.pcmux = Mux(sel = comp.pcsel,a = 0x80000000,b = 0x80000004,
    c = 0x80000008,d = comp.mamux.b,e = comp.pcinc.o,
    o = comp.npc.d)
```

Name	IO Type	Width(if specified)	Description
clk	Input	None	Processor Clock
reset	Input	None	Processor Reset
irq	Input	1	Interupt Request
ma	Output	32	Memory Address
mdin	Input	None	Memory Data (Input)
mdout	Output	None	Memory Data (Output)
mwe	Output	None	Memory Write Enable
inst	Wire	None	Current Instruction
z	Wire	None	Zero from RA1
co	Wire	None	Carry Out from ALU
aluz	Wire	None	ALU output is zero
v	Wire	None	Overflow in ALU
n	Wire	None	ALU output is negative
werf	Wire	None	Write Enable Register File
annul	Wire	None	Annul the IF stage
msel	Wire	None	Memory Select
msel_next	Wire	None	Memory Select (next cycle)
mwrite	Wire	None	Memory Write
pre_werf	Wire	None	Unmodified werf from ROM
wasel	Wire	None	wasel mux select
pcsel	Wire	None	pcsel mux select
wdsel	Wire	None	wdsel mux select
asel	Wire	None	asel mux select
bsel	Wire	None	bsel mux select
csel	Wire	None	csel mux select
shiftdir	Wire	None	Shift Direction
shiftsext	Wire	None	Shift with Sign Extend
addsub_op	Wire	None	Add/Subtract Operation
compare_op	Wire	None	Compare Operation

Table 4.2: List of signals used in the two stage Beta.

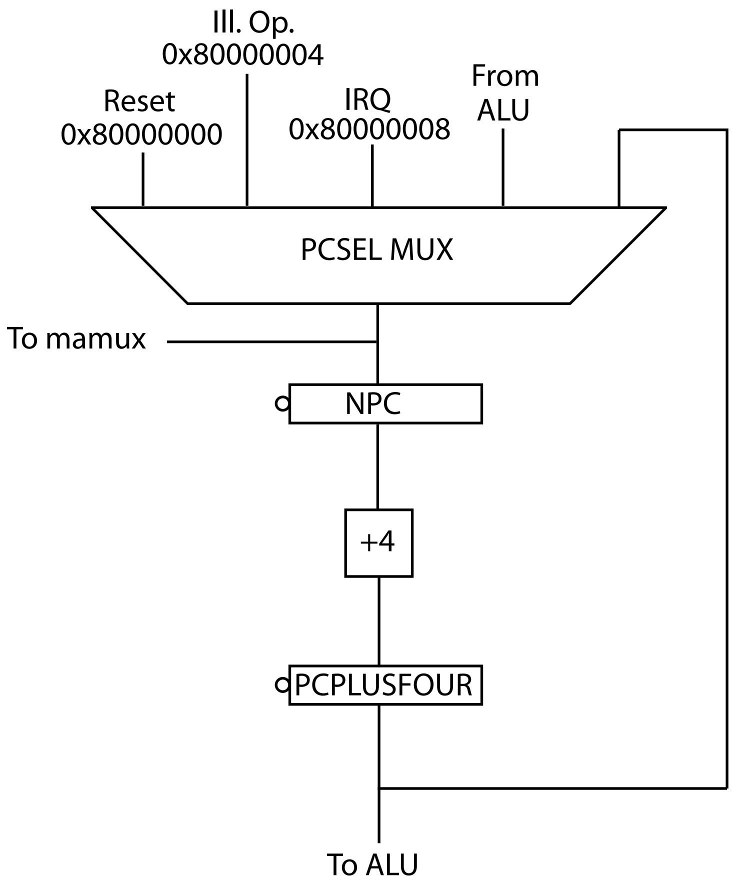

Figure 4-3: Program counter control logic.

```
 9 Connect(comp.pcmux.o,comp.mamux.a)
10 Connect(comp.mamux.o,comp.ma)
11 Connect(comp.mamux.sel,comp.msel_next)
12
13 comp.instreg = DLatch(reset = comp.reset,en = comp.not_msel.o,
14     clk = comp.clk,d = comp.mdin,q = comp.inst)
```

Listing 4.2: PC Control Logic and the Instruction Register

The program control logic implementation is fairly straightforward, using the short hand of passing in the wire names into a component's instantiation that can be accomplished with a few lines of code. Lines 1-8 implement the two D-Latches, the incrementer, and the mux. Lines 9-14 implement the instruction register and tie the memory address output to the program counter. With a fairly short piece of code we are able to tie together the entire circuit.

Lines 1-2 creates *npc*, the next pc register, and ties the clock and reset signal. The input comes from the *pcmux*, and the output goes to the pc incrementer. In line 3 the incrementer was implemented as an Adder with an input tied to a constant 4. A few cases like this, where constants are put into arithmetic or shifting units, lead to the idea that some units should be able to optimize themselves and implement constant versions. Currently this is not the case, but all the infrastructure is there and all it takes is making new implementations of components. Lines 4-5 contains the register of the incremented pc. Finally, there is the mux that forms the loop of these four components.

There are some Connect statements here that could have been avoided if we instantiated mamux inside the code. Connect statements are an ugly construct to use, but it is possible to write just about anything in PHDL without using them. Often, due to organization of the design it may feel necessary to use a few Connect statements. In this case, the organization is laid out and a few of these statements are required. The last two lines create the instruction register that loads instructions from memory.

This first part of the Beta is fairly easy to implement using PHDL. It should be no longer than any implementation in Verilog. If we take into account the implementations of the various component's themselves then PHDL would be longer in code size. Although, this is not a fair comparison, since PHDL adds the intelligence of good component selection. PHDL empowers component reuse, so PHDL will be able to address more complex problems easier as components are added to its library.

4.4 Register File

Register files are common enough that PHDL has a generic register file component. The RegisterFile component allows developers to specify several commonly implemented behaviors. Parameters such as making one register a constant zero are available in PHDL's RegisterFile component.

```
comp.mnextmux = Mux(sel = comp.msel_next,a = comp.inst[15:11],
    b = comp.inst[25:21])
comp.mmux = Mux(sel = comp.msel) # Write Back Register Address
```

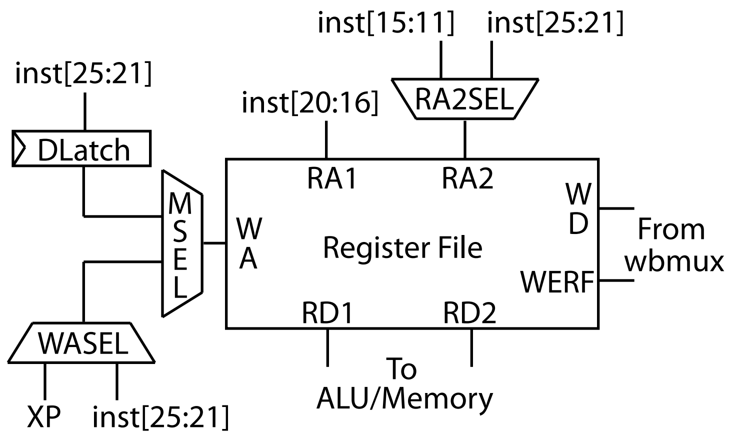

Figure 4-4: Diagram of the register file and surrounding logic.

```
 4  comp.msellatch = DLatch(d = comp.inst[25:21],q = comp.mmux.b,
 5      clk = comp.clk,reset = comp.reset)
 6  comp.regfile = RegisterFile(constantzero = 31,regs = 32,
 7      x = comp.wdmux.o,xsel = comp.mmux.o,xwe = comp.werf,
 8      clk = comp.clk,asel = comp.inst[20:16],bsel = comp.mnextmux.o)
 9  comp.wamux = Mux(sel = comp.wasel,a = comp.inst[25:21],b = 30,
10      o = comp.mmux.a)
11  comp.zbitand = BusNor(i = comp.regfile.a,o = comp.z) # Compute Z
```

Listing 4.3: Register File and Related Logic

The register file is fairly simple to implement in PHDL. In the register file code we modify as needed the selection inputs to the register file. Figure 4-4 shows what needs to be implemented. Lines 1-5 and 9-10 implement the modifications to the register file's select inputs. The store instruction requires that the register use bits 25 through 21 of the opcode for the source register. This functionality is implemented in lines 1-2. Line 3-5 and 9-10 in combination implement the write back address modification. Lines 4-5 is a DLatch to delay the write back address by one clock cycle in all cases, except when an exception occurs. In line 5 the *wamux* is used to switch the write back register for exceptions. Lines 6-8 instantiates the register file with most of the inputs included as parameters. The *constantzero* parameter ensures that reads from register 31 return zero and the *regs* parameter sets how many registers total to have. The *regs* parameter could been left out to be solved by PHDL from the fact that we have 5 bit wide inputs to our register select inputs. The RegisterFile component knows that there is an exponential relationship between the two. The final line computes the zero test using a BusNor component. BusNor and similar components allow us to take a bus and perfrom a particular logic operation on all the bits.

4.5 Arithmetic and Logic Unit

Arithmetic and Logic Unit (ALU) implements most of a processors integer computational abilities. The ALU in the two stage Beta processor implements all mathematical operations, including those for branching and memory access.

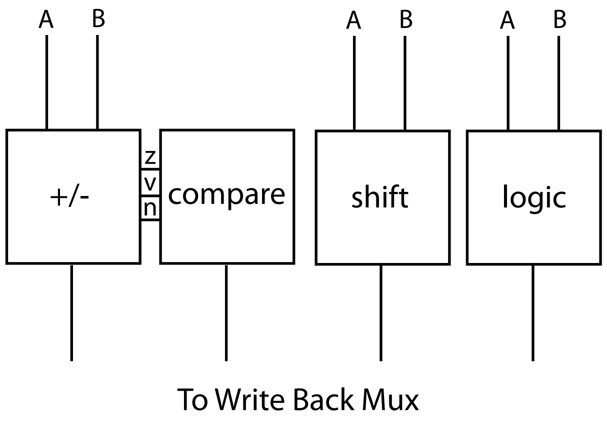

Figure 4-5: Diagram of the Arithmetic and Logic Unit

Figure 4-5 shows a diagram of the ALU internals. The inputs to all blocks, are A and B, discussed below. These two inputs are from the register file that are passed to three muxes. This enables the Beta to easily substitute operand B with a constant, and to take care of other special inputs to the ALU. The adder/subtractor connects its outputs to the compare circuit to implement the compare operations; in addition, shifting and logic operations are implemented. The Beta implements the or, and, xor logic operations. The not logic operation can be implemented by xor'ing a value with -1.

```
comp.amux = Mux(a = comp.regfile.a,b = comp.pcplusfour.q,
    sel = comp.asel)
comp.pcext = SignExt(i = comp.inst[15:0])
comp.pcextshift = AnonymousBulkConnector(comp.pcext.o[29:0],(0,2))
comp.cmux = Mux(a = comp.pcext.o,b = comp.pcextshift,
    sel = comp.csel)
comp.bmux = Mux(a = comp.regfile.b,b = comp.cmux.o,
    sel = comp.bsel)
comp.alu_shrl = Shift(i = comp.amux.o,s = comp.regfile.b[4:0],
10     o = comp.wdmux.b,dir = comp.shiftdir,sext = comp.shiftsext)
11 comp.alu_adder = AddSub(a = comp.amux.o,b = comp.bmux.o,
12     o = comp.wdmux.c,op = comp.addsub_op,co = comp.co,
13     z = comp.aluz,v = comp.v,n = comp.n)
14 Connect(comp.jt,comp.alu_adder.o)
15 comp.alu_compare = Compare(z = comp.aluz,v = comp.v,n = comp.n,
16     op = comp.compare_op,o = comp.wdmux.d) # Compare Circuit
17 comp.alu_or = Or(a = comp.amux.o,b = comp.bmux.o,o = comp.wdmux.e)
18 comp.alu_and = And(a = comp.amux.o,b = comp.bmux.o,
19     o = comp.wdmux.f)
20 comp.alu_xor = Xor(a = comp.amux.o,b = comp.bmux.o,
21     o = comp.wdmux.g)
```

Listing 4.4: Arithmetic and Logic Unit

The Arithmetic and Logic Unit in the PHDL Beta connects to the large write back mux that selects between all the ALU components, as well as the program counter and memory. The beginning of the ALU is the modifications to the input sources used by ALU components. Path A is modified by *amux*, while path B is a bit more complicated. Path B contains the *bmux* to select either the register, or a constant input. The constant input uses *cmux* to select between the sign extended program counter and the same number multiplied by four. Sign extension occurs with the use of an intelligent SignExt component. The multiply by four operation actually just uses an AnonymousBulkConnector to implement a two bit shift. In theory it should be as efficient as using a shifter with a constant input of two (at the time of the implementation the constant shifter was not implemented). Lines 9-13 instantiate a shifter and an adder/subtractor. In line 14 we tie the jump target to the output of the adder. In this implementation of the Beta the jump target wire is used as an output address for memory operations, as well as the address for jumps. Single cycle implementations of the Beta often split these two wires and take the jump target directly from the register file. Comparison opcodes use the comparator, in lines 15-16, that is tied to the state zero(z), overflow(v), and negative(n) outputs of the *alu_adder*. Lines 17 through 21 implement the logic operations needed by the Beta instruction set.

4.6 Write Back Stage

```
1 Connect(comp.wdmux.sel,comp.wdsel)
```

```
Connect(comp.mdin,comp.wdmux.a)
Connect(comp.wdmux.h,comp.pcplusfour.q)
Connect(comp.mdout,comp.regfile.b)
```

Listing 4.5: Write Back Stage and Memory Read/Write

The write back code only has a few connect statements. One ties the *wdmux's* select line to *wdsel* input from the control ROM. Connect *wdmux's* first input to the memory input and tying the last input to the PC + 4 input from the program counter control logic. The last statement also ties memory output to the second read output from the register file. This wraps up most of the beta processor; only the control logic remains.

4.7 Control Logic

The control logic, the brains of any processor, are implemented as a ROM with a small amount of boolean logic. The boolean logic implements branching and signals dealing with annulling, conditional branches, reset, and other operations.

```
comp.annul_not = Not(i = comp.pcsel[2])
comp.annul_reg = DLatch(clk = comp.clk,reset = comp.reset,
    d = comp.annul_not.o,q = comp.annul)
comp.msel_next_not_annul = Not(i = comp.annul)
comp.msel_next_andA = And(a = comp.not_reset.o,
    b = comp.msel_next_not_annul.o)
comp.msel_next_andB = And(a = comp.mem_next,b = comp.not_msel.o)
comp.msel_next_andC = And(a = comp.msel_next_andA.o,
    b = comp.msel_next_andB.o,o = comp.msel_next)
comp.msel_reg = DLatch(clk = comp.clk,reset = comp.reset,
    d = comp.msel_next,q = comp.msel)
comp.werf_not = Not(i = comp.werf)
comp.mwrite_next_andA = And(a = comp.msel_next,b = comp.mem_next)
comp.mwrite_next_andB = And(a = comp.werf_not.o,
    b = comp.mwrite_next_andA.o,o = comp.mwe)
comp.mwrite_reg = DLatch(clk = comp.clk,reset = comp.reset,
    d = comp.mwe,q = comp.mwrite)
comp.werf_not_mwrite = Not(i = comp.mwrite)
comp.werf_not_mem_next = Not(i = comp.mem_next)
comp.werf_and = And(a = comp.msel_next_not_annul.o,
    b = comp.werf_not_mem_next.o)
comp.werf_mux = Mux(a = comp.werf_not_mwrite.o,
    b = comp.werf_and.o,o = comp.werf,sel = comp.msel)
comp.notreset_three = AnonymousBulkConnector((comp.not_reset.o,3))
comp.z_three = AnonymousBulkConnector((comp.z,3))
comp.branch_three = AnonymousBulkConnector((comp.branch,3))
```

```
27 comp.pcsel_andbz = And(a = comp.z_three,b = comp.branch_three)
28 comp.pcsel_xorbzpc = Xor(a = comp.pcsel_eff,
29     b = comp.pcsel_andbz.o)
30 comp.pcsel_andrbzpc = And(a = comp.pcsel_xorbzpc.o,
31     b = comp.notreset_three,o = comp.pcsel)
```

Listing 4.6: Control Logic (w/o ROM)

The implementation of the two stage Beta includes a ROM, which does most of the decoding operations. The ROM is not enough because many of the control signals have to be modified for special cases. Lines 1-2 are used for annulling the next instruction which occurs. In lines 3-7 the *msel_next* wire is calculated, which is used for deciding memory operation cycle versus a normal instruction fetch/execute. In the two stage pipeline, the next cycle must be annulled for memory operations because memory bus is shared for data and code. Line 7 is clocking *msel_next* through a DLatch to create msel since that must be delayed for the second stage for memory operations.

The memory write enable signal is calculated from lines 8-10 and again it is delayed one cycle in line 11 using another DLatch. The *mem_next* and *msel_next* wires say that a memory operation is occurring in the current instruction. Checking that the *werf* signal is not asserted shows that it is a memory write as apposed to a memory read.

Lines 12-15, in Listing 4.6, determine the *werf* signal value. Essentially, *werf* is always asserted except during memory stores and annulled cycles. This is easily implemented as a mux, which during memory operations checks that the *mwrite* signals is not asserted. During non-memory operations it ensures that neither *annul* and *mem_next* are asserted.

The last complex operation left is the *pcsel* modification. This is used only for conditional branches and for reset. IRQs and illegal opcodes are handled by the ROM itself. It functions because the branch *pcsel* value is the logic inverse of the *pcsel* increment value. During a branch instruction if the zero bit is asserted, then *pcsel* effective (*pcsel_eff*) is inverted. When a reset occurs, *pcsel* is set to zero this is implemented with the and statement in line 19. In lines 16 through 18 the AnonymousBulkConnectors are used to repeat concatenated wires three times. The parenthesis with a connector, a comma, and a number is the syntax used to allow single wires to be repeated in an AnonymousBulkConnectors. In the case of constants that are being passed in, it is used to specify the width of the constant that

AnonymousBulkConnectors should output. The ROM is the source of most signals, including ones used to compute the signals above.

```
comp.not_npc31 = Not(i = comp.nextpc[31])
comp.irqandnpc = And(a = comp.irq,b = comp.not_npc31.o,
    o = comp.interupt)
comp.rcin = AnonymousBulkConnector(comp.interupt,comp.inst[31:26])
comp.rcout = AnonymousBulkConnector(comp.branch,comp.mem_next,
    comp.wasel,comp.pcsel_eff,comp.wdsel,comp.shiftdir,
    comp.shiftsext,comp.asel,comp.bsel,comp.csel,comp.addsub_op,
    comp.compare_op)
comp.control = ROM(a = comp.rcin,d = comp.rcout) # ROM

for i in range(64): # Set Invalid Opcodes
  comp.control[i] = 0x04F00

for i in range(64): # IRQ
  comp.control[i+64] = 0x06700

comp.control[0x018] = 0x0A010 # LD
comp.control[0x019] = 0x0A010 # ST
...
```

Listing 4.7: Control Logic ROM

The ROM here needs as input the 6 bit opcode number and the interrupt wire which is irq when interrupts are enabled, and zero when interrupts are disabled. In the Beta architecture the interrupt enable is determined by the most significant bit of the program counter. By using AnonymousBulkConnectors we concatenate all the significant wires together for the input address to the ROM, and the output data from the ROM. The PHDL ROM component allows for indexed access to the ROM contents. This allows the user to write Python code to generate the ROM contents. In this case, two loops set the invalid opcodes and the IRQ's. Then a partial listing shows the writes to the ROM for the load and store instructions. The full contents of the ROM is shown in Appendix B.

4.8 Creating a Component

With a complete Beta implemented as a circuit it is easy to create a component. The first task is to create a simple meta-component. The constructor consists of a call to *Init()*, the initializing the input/output wires, and a call to *AutoConnect()*. The only constraints applied in this component are the 32 bit memory input and the one bit input to *irq*, which has been left as before. In Listing 4.8 is the Beta meta-component.

```python
class Beta(Component.Component):
  def __init__(self,**cons):
    self.Init()
    self.clk = WireConnector(HDLIOType.Input)
    self.reset = WireConnector(HDLIOType.Input)
    self.irq = WireConnector(HDLIOType.Input,width = 1)
    self.ma = WireConnector(HDLIOType.Output,width = 32)
    self.mdin = WireConnector(HDLIOType.Input)
    self.mdout = WireConnector(HDLIOType.Output)
    self.mwe = WireConnector(HDLIOType.Output)
    self.AutoConnect(cons)

  def ConfigureComponent(self,prj):
    if self.Instance is None:
      self.Instance = Beta2Impl()
      self.InitInstance()
    self.Instance.ConfigureComponent(prj)
    self.Instance.GenerateName()

  def ParameterizationCheck(self,prj):
    pass
```

Listing 4.8: Beta Meta-Component

The *ConfigureComponent* method only needs to instantiate the BETA2IMPL if the instance has not been initialized. Then the standard calls to configure the instance and generate a name.

The remainder of the code that exists can just be placed into a vanilla component inside the constructor. Where we had *comp* we will once again replace it with *self*. Also, the *GenerateName* function was implemented to always return "Beta2." This is a two stage pipelines beta and given any number of instances within a single session of using PHDL, they will be all identical in their configuration. The full code listing is available in Appendix B. One can see that it is a fairly straight foward task to create a component from an existing circuit.

Chapter 5

Conclusion

PHDL is a developing language. As with any language, it should continue to improve and serve as a solution to more complex user problems. From the development of PHDL a handful of major challenges have emerged that can be addressed in the future.

5.1 Future Work

Using PHDL to build the Beta processor showed that PHDL is a practical language for hardware development. Several major issues still remain. Component implementation can be made easier by creating a decentralized system for component selection. Debugging the Beta has proved it can be difficult to debug generated Verilog, and a PHDL simulator must be made available. A handful of syntactic improvements can be made to enable faster development. Tools to help developers with common problems such as state machines and truth tables need to be developed.

5.1.1 Improving Component Selection

The current component selection scheme has been satisfactory for now, but as more implementations are added, meta-components will become complex with long conditional statements to decide which component to use. An important future direction is the use of goal oriented programming. If we instead treat our components as goals and allow them to score how well they can implement a function it becomes easy to add components. Meta-components may not be need to be written by a developer, and the goal oriented planner will auto-generate skeleton meta-components.

Goal orientation is a very natural extension to the PHDL framework. During the late development of the PHDL framework, it became clear that components should score themselves, but this overlapped so much with goal oriented design that it ought to be left as part of the extension of goal orientation. The future view of PHDL is really part of a two part system using the goal oriented planner to achieve tasks.

5.1.2 Address Debugging Challenges

During the development and testing of the Beta it became abundantly clear that debugging the generated Verilog code was difficult. Thankfully, a decision was made to keep the generated code as clean as possible. Developers can read the Beta PHDL code and see the auto-generated code and quickly understand what it is doing. However, many IP Core generators used by commercial companies to license their IP's in a configurable form do not always generate human readable code. The real solution to the problem of debugging is to enable PHDL itself to debug and simulate components. Using python to write a test bench can be much more powerful that the current Verilog test benches.

All that is required is that any component that generates Verilog code directly must simulate that code, although this is a complicated addition. High level components that are built on other components would inherit the ability to simulate themselves.

5.1.3 General Improvements

The last major improvement is the continued addition of components to the PHDL Framework. This could be done by a community open source effort as people who find this useful can contribute new components and specialized implementations for various FPGA targets.

PHDL could benefit from specialized components that allow users to organize their code and thought processes better. For example enabling users to design components using truth tables or state diagrams can very useful.

5.2 Concluding Remark

As PHDL gains new features the Beta processor example can be reduced in complexity. Currently, the Beta example was about equal to the Verilog implementation in number of lines of code. The Verilog code available in Appendix C relies completely on the synthesis tool to decide the best way to implement a particular logic function. For example, it explicitly uses the add operator to implement an adder. This keeps the Verilog code easy to read but constrains the use of optimized hardware specific components. The synthesis tool may or may not implement a particular function in an optimized way. The PHDL version is able to replace every component including the adders with hardware specific versions. The Beta example shows that a design can be written with similar complexity to a Verilog implementation, while achieving better portability and platform specific optimizations.

The future work on PHDL will serve to make PHDL a useful language that can be adopted for rapid hardware development. PHDL has many benefits over other HDLs and can serve as a powerful front end to Verilog.

Appendix A

PHDL Framework Source Code

The core of the PHDL Framework comes first then the majority of the components that I have implemented are included. Some of these components may not use the short hand way of implementing a feature. Due to the constant evolution to improve the syntax and functionality some components have been left using deprecated functions or may not look as neat as they should.

```
#!/usr/bin/env python

__all__ = [ "Component", "ComponentImpl", "Connector", "Dev", "HDLIOType", "HDLNetType",
      "HDLWriter", "Project", "Util", "VerilogWriter" ]

# Here I define all special functions like Connect

import math

"""
This is probably the most complicated single function it takes
care of all the special cases of tieing two connectors together.
Hopefully I will continue to clean it up.
"""
def connect(a,b):
  if (a is None) or (b is None):
    Dev.Debug(Dev.Error,"Connect: Trying to connect None to something!")
    return
  if isinstance(a,int) or isinstance(a,long):
    a = Connectors.ConstantConnector.ConstantConnector("CONST" + str(a),b.Comp.Parent,a)
  if isinstance(b,int) or isinstance(b,long):
    b = Connectors.ConstantConnector.ConstantConnector("CONST" + str(b),a.Comp.Parent,b)
  if (a.Comp is b.Comp):
    if not(a.Conn is None) and not(b.Conn is None):
      # Check if a connection is anonymous then rename else we make an assign
      if (a.Conn.IsAnonymous()):
        a.Comp.Parent.__delattr__(a.Conn.LocalConnector.Name)
        a.Conn.ReconnectTo(b.Conn)
        # Fix the connector!
      elif (b.Conn.IsAnonymous()):
        b.Comp.Parent.__delattr__(b.Conn.LocalConnector.Name)
```

```python
        b.Conn.ReconnectTo(a.Conn)
      else:
        Dev.Debug(Dev.Stop,"Connect: Assign gen incomplete!!!")
    else:
      Dev.Debug(Dev.Stop,"Connect: Assign gen incomplete!!!!")
    return
  elif (a.Comp.Parent is b.Comp.Parent) and not(a.Comp.Parent is None):
    connectionA = a.Comp.InstanceName + ":" + a.Name
    connectionB = b.Comp.InstanceName + ":" + b.Name
    if not(a.Conn is None) and (b.Conn is None):
      conn = a.Conn
    elif not(b.Conn is None) and (a.Conn is None):
      conn = b.Conn
    elif not(a.Conn is None) and not(b.Conn is None):
      # This deals with the icky case of anonymous connector renaming!
      if (a.Conn.IsAnonymous() == 1):
        a.Comp.__delattr__(a.Name)
        a.Conn.ReconnectTo(b.Conn)
      elif (b.Conn.IsAnonymous() == 1):
        b.Comp.__delattr__(b.Name)
        b.Conn.ReconnectTo(a.Conn)
      else:
        Dev.Debug(Dev.Stop,"Connect: Assign gen incomplete!")
      return
    else:
      # Generate (unconfigured) duplicate connector
      localconnector = a.Duplicate()
      a.Comp.Parent.Anonymous = localconnector
      # Generate connection - Any assignment to Anonymous should SetAnonymous
      conn = Connection.Connection(localconnector)
      localconnector.SetLocalConnection(conn)
    a.Connect(conn)
    b.Connect(conn)
  elif (a.Comp.Parent is b.Comp) or (a.Comp is b.Comp.Parent):
    if (a.Comp is b.Comp.Parent):
      t = a
      a = b
      b = t
    connectionA = a.Comp.InstanceName + ":" + a.Name
    connectionB = b.Name
    if not(b.LocalConn is None):
      conn = b.LocalConn
    else:
      conn = Connection.Connection(b)
    if not(a.Conn is None):
      # Hard case connector renaming!
      if (a.Conn.IsAnonymous() == 1):
        a.Comp.Parent.__delattr__(a.Conn.LocalConnector.Name)
        a.Conn.ReconnectTo(conn)
      else:
        print a.Name + a.Comp.InstanceName
        Dev.Debug(Dev.Stop,"Connect: Assign gen incomplete!!")
    else:
      # Simple case a new connector!
      a.Connect(conn)
    b.SetLocalConnection(conn)
  else:
    print "Trying to connect: " + a.Comp.InstanceName + "." + a.Name + " " +
          b.Comp.InstanceName + "." + b.Name
    Dev.Debug(Dev.Stop,"Connect: Cannot connect connectors that are far apart!")
  conn.Attach(connectionA,a)
  conn.Attach(connectionB,b)

# Make the Connect function a builtin
__builtins__["Connect"] = connect

del connect

```

```python
# Configure Rules
def configureequal(prj,obja,parama,objb,paramb = None):
  if paramb is None:
    # Fix it to the behavior I want
    # ConfigureEqual(obja,objb,"Width") - Shorthand
    paramb = objb
    objb = parama
    parama = paramb
  if (obja.__getattr__(parama) is None) and not(objb.__getattr__(paramb) is None):
    obja.__setattr__(parama,objb.__getattr__(paramb))
    if isinstance(obja,Connector.Connector):
      prj.AddChangedConnector(obja)
  if (objb.__getattr__(paramb) is None) and not(obja.__getattr__(parama) is None):
    objb.__setattr__(paramb,obja.__getattr__(parama))
    if isinstance(objb,Connector.Connector):
      prj.AddChangedConnector(objb)

def configurelog2(prj,obja,parama,objb,paramb = None):
  if paramb is None:
    # Fix it to the behavior I want
    # ConfigureLog2(obja,objb,"Width") - Shorthand
    paramb = objb
    objb = parama
    parama = paramb
  if (obja.__getattr__(parama) is None) and not(objb.__getattr__(paramb) is None):
    obja.__setattr__(parama,int(math.ceil(math.log(objb.__getattr__(paramb),2))))
    if isinstance(obja,Connector.Connector):
      prj.AddChangedConnector(obja)
  if (objb.__getattr__(paramb) is None) and not(obja.__getattr__(parama) is None):
    objb.__setattr__(paramb,math.pow(2,obja.__getattr__(parama)))
    if isinstance(objb,Connector.Connector):
      prj.AddChangedConnector(objb)

def configureexp2(prj,obja,parama,objb,paramb = None):
  if paramb is None:
    # Fix it to the behavior I want
    # ConfigureExp2(obja,objb,"Width") - Shorthand
    paramb = objb
    objb = parama
    parama = paramb
  if (obja.__getattr__(parama) is None) and not(objb.__getattr__(paramb) is None):
    obja.__setattr__(parama,int(math.pow(2,objb.__getattr__(paramb))))
    if isinstance(obja,Connector.Connector):
      prj.AddChangedConnector(obja)
  if (objb.__getattr__(paramb) is None) and not(obja.__getattr__(parama) is None):
    objb.__setattr__(paramb,int(math.ceil(math.log(obja.__getattr__(parama),2))))
    if isinstance(objb,Connector.Connector):
      prj.AddChangedConnector(objb)

__builtins__["ConfigureEqual"] = configureequal
__builtins__["ConfigureLog2"] = configurelog2
__builtins__["ConfigureExp2"] = configureexp2
```

Listing A.1: PHDL/__init__py

```python
#!/usr/bin/env python
# PHDL Component

import re

import Dev
import Connection

import HDLIOType

import Connector

class Component(object):
  """Component Class

  All PHDL components are built upon this class. This takes
  care of a lot of the tedious work of building a component.
  It also manages I/O and code generation. It provides a generic API for that.
  """
  def __init__(self,name = None,instancename = None):
    Dev.Debug(Dev.Info,"Component.__init__(self)")
    self.Init(name,instancename)

  # *PUBLIC* Initializes the component
  def Init(self,name = None,instancename = None):
    self.Instance = None
    self.MetaInstance = self
    self.Parent = None
    self.Name = name
    if not(self.__dict__.has_key("InstanceName")):
      self.InstanceName = instancename
    self.Connectors = [ ]
    self.Components = { }
    self.NetNameNumber = 0
    self.CompNameNumber = 0
    self.DelayedAutoConnect = { }

  # *PUBLIC* Initializes the instance of this component
  def InitInstance(self):
    # These two variables form the linked list that provide our view
    # of namespaces.
    self.Instance.Instance = self.Instance
    self.Instance.MetaInstance = self
    self.Instance.InitLogic()

  def InitLogic(self):
    Dev.Debug(Dev.Info,"Component.InitLogic(self)")

  # *PUBLIC* Overload how namespaces work so we can clean syntax
  def __setattr__(self,attr,val):
    Dev.Debug(Dev.Info,"Component.__setattr__(self,attr,val)")
    # Attempts to set anonymous turn into an auto-generated name!
    if attr == "Anonymous":
      if isinstance(val,Component):
        val.Instance = None
        val.MetaInstance = val
        val.InstanceName = self.GenerateComponentName()
        val.Parent = self.GetMetaInstance()
        self.AddSubcomponent(val)
      elif isinstance(val,Connector.Connector):
        val.Name = self.GenerateNetName()
        val.Comp = self.GetMetaInstance()
        val.SetAnonymous()
        val.LateInit()
        self.AddConnector(val)
      else:
        Dev.Debug(Dev.Stop,"Error self.Anonymous is a reserved variable name!")
      return
    if isinstance(val,Component) and (attr != "Instance") and
         (attr != "MetaInstance") and (attr != "Parent"):
      #val.Init(instancename = attr)
      val.Instance = None
      val.MetaInstance = val
      val.InstanceName = attr
      val.Parent = self.GetMetaInstance()
      self.AddSubcomponent(val)
      return
    if isinstance(val,Connector.Connector):
      # If the connector exists then we run Connect!
      # If you want to overwrite a connector you must delete it first
      # so as to unbind it properly from the current Connector!
      if not(self.GetConnector(attr) is None):
        # Maybe I should check for autogen names but I think were ok!
        # At least throw an error
        # TODO: Make sure this throughs an error when its not possible(currently)
        Connect(self.GetConnector(attr),val)
      else:
        val.Name = attr
        val.Comp = self.GetMetaInstance()
        val.LateInit()
        self.AddConnector(val)
      return
    # Default case for non-special components/connectors
    # Modify all special variables to by tied between shell/instance
    if not(self.MetaInstance is self) and (attr != "Instance")
         and (attr != "MetaInstance"):
      self.MetaInstance.__setattr__(attr,val)
    else:
      self.__dict__[attr] = val

  # *PUBLIC* Overload how namespaces work so we can clean syntax
  def __getattr__(self,attr):
    Dev.Debug(Dev.Info,"Component.__getattr__(self,attr)")
    # Avoid Infinite Recursion on Initialization
    if (attr == "MetaInstance"):
      return None
    # Modify all special variables to by tied between meta/instance
    if not(self.MetaInstance is self) and not(self.MetaInstance is None):
      return self.MetaInstance.__getattr__(attr)
    elif self.MetaInstance is self:
      if self.__dict__.has_key(attr):
        return self.__dict__[attr]
    if not(self.GetSubcomponent(attr) is None):
      return self.GetSubcomponent(attr)
    if not(self.GetConnector(attr) is None):
      return self.GetConnector(attr)
    print "Trying to get " + attr + " in component " + self.InstanceName
    Dev.Debug(Dev.Stop,"Component.__getattr__(self,attr) does not exist!!!")

  # *PUBLIC* Overload how namespaces work so we can clean syntax
  def __delattr__(self,attr):
    Dev.Debug(Dev.Info,"Component.__delattr__(self,attr)")
    if not(self.GetConnector(attr) is None):
      # This has to be cleaned up to allow connectors a chance to cleanup logic!
      self.Connectors.remove(self.GetConnector(attr))
      return
    Dev.Debug(Dev.Stop,"Component.__delattr__: UH OH CANT DELETE!")
    # We can not delete components yet!

  # *INTERNAL* Used to add all subcomponents to the project for generation phase
  def AddBindings(self,project):
    """Adds Bindings to a project
    test
    """
    Dev.Debug(Dev.Info,"Component.AddBindings(self,project)")
    project.AddComponent(self)
    for k, v in self.Components.iteritems():
      v.AddBindings(project)

  # *INTERNAL* Adds a connector to self
  def AddConnector(self,con):
    Dev.Debug(Dev.Info,"Component.AddConnector(self,con)")
    # Check for duplicate connectors
    self.Connectors.append(con)

  # *INTERNAL* Gets a connector from self
  def GetConnector(self,conname):
    Dev.Debug(Dev.Info,"Component.GetConnector(self,conname)")
    for e in self.Connectors: # Normal connectors Including Subconnectors
      if e.Name == conname:
        return e;
    return None

  # *INTERNAL* Adds a subcomponent to self
  def AddSubcomponent(self,comp):
    Dev.Debug(Dev.Info,"Component.AddSubcomponent(self,comp)")
    self.Components[comp.InstanceName] = comp
    comp.AutoConnectInternal()

  # *INTERNAL* Gets a subcomponent from self
  def GetSubcomponent(self,compname):
    Dev.Debug(Dev.Info,"Component.AddSubcomponent(self,compname)")
    splitcomponentname = re.split("[\.]",compname,1)
    if self.Components.has_key(splitcomponentname[0]):
      if len(splitcomponentname) == 1:
        retvalue = self.Components[splitcomponentname[0]]
        if not(retvalue.Instance is None) and (retvalue.Instance != retvalue):
          return retvalue.Instance
        else:
          return retvalue
      else:
        return
            self.Components[splitcomponentname[0]].GetSubcomponent(splitcomponentname[1])
    else:
      return None

  # *PUBLIC* Allows you to iterate over a pattern of connectors
  def ConnectorIterator(self,pattern = "."):
    cons = []
    for e in self.Connectors:
      if re.match(pattern,e.Name,1):
        cons.append(e)
    return cons

  # *PUBLIC* Allows you to iterate over a pattern of components
  def ComponentIterator(self,pattern = "."):
    cons = []
    for k,v in self.Components.iteritems():
      if re.match(pattern,k,1):
        cons.append(v)
    return cons

  # *PUBLIC* Saves dictionary of connections
  def AutoConnect(self,wiredictionary):
    self.DelayedAutoConnect = wiredictionary

  # *INTERNAL* Used to autoconnect wires after component is attached
  def AutoConnectInternal(self):
    for k,v in self.DelayedAutoConnect.iteritems():
      for e in self.Connectors:
        if (k == e.Name):
          Connect(v,e)
    return

  """ConfigureComponent(self) - Configures the local component and its subcomponents
  This method configures all connectors and then components. The implementation of this
  method must select our optimimum instance (Allow it to add new components), configure
  connectors, and configure any subcomponents.
  """
  def ConfigureComponent(self,prj):
    Dev.Debug(Dev.Info,"Component.ConfigureComponent(self)")
    self.Instance = self
    for e in self.Connectors:
      e.ConfigureConnector(prj)
    for k,v in self.Components.iteritems():
      v.ConfigureComponent(prj)
    if self.Name is None:
      self.GenerateName(prj);

  # Last minute configuration check
  # This reports errors and warnings
  def ParameterizationCheck(self,prj):
    Dev.Debug(Dev.Info,"Component.ParameterizationCheck(self,prj)")
    for e in self.Connectors:
      e.ParameterizationCheck(prj)
    for k,v in self.Components.iteritems():
      v.ParameterizationCheck(prj)

  def GetInstance(self):
    Dev.Debug(Dev.Info,"Component.GetInstance(self)")
    if (self.Instance is None):
      Dev.Debug(Dev.Error,"Error: Unconfigured component " + self.InstanceName)
    else:
      return self.Instance

  def GetMetaInstance(self):
    Dev.Debug(Dev.Info,"Component.GetMetaInstance(self)")
    if (self.MetaInstance is self):
      return self
    else:
      return self.MetaInstance.GetMetaInstance()

  # For Input/Output Lists I need to build the list first
  # then write the list out in order to properly solve the comma problem
  def GenerateVerilogHDLHeader(self,hdlwriter):
    Dev.Debug(Dev.Info,"Component.GenerateVerilogHDLHeader(self,hdlwriter)")
    if self.Name is None:
      Dev.Debug(Dev.Stop,"PANIC: self.Name is not set in instance " + self.InstanceName)
    hdlwriter.Write("module " + self.Name + "(")
    i = 0
    for e in self.Connectors:
      if e.IOType != HDLIOType.Wire:
        i = i + 1
    for e in self.Connectors:
      if e.IOType != HDLIOType.Wire:
        e.WriteIOPortNames(hdlwriter)
        i = i - 1
        if i != 0:
          hdlwriter.Write(",")
    hdlwriter.Write(");\n");
    for e in self.Connectors:
      e.WriteIOPorts(hdlwriter)
    hdlwriter.Write("\n");
    for e in self.Connectors:
      e.WriteIOPortLogic(hdlwriter)
    for k,v in self.Components.iteritems():
      hdlwriter.Write("\n" + v.Name + " " + v.InstanceName + "(\n")
      i = 0
      for c in v.Connectors:
        if c.IOType != HDLIOType.Wire:
          i = i + 1
      for c in v.Connectors:
        if c.IOType != HDLIOType.Wire:
          i = i - 1
          c.WriteIOPortBindings(hdlwriter)
          if i != 0:
            hdlwriter.Write(",\n")
      hdlwriter.Write("\n);\n")

  def GenerateVerilogHDLBody(self,hdlwriter):
    Dev.Debug(Dev.Info,"Component.GenerateVerilogHDLBody(self,hdlwriter)")
    if not(self.Parent is None):
      Dev.Debug(Dev.Warning,"Warning: " + self.InstanceName + " has no local code.")
    hdlwriter.Write("\n\t// No Body\n");

  def GenerateVerilogHDLFooter(self,hdlwriter):
    Dev.Debug(Dev.Info,"Component.GenerateVerilogHDLFooter(self,hdlwriter)")
    hdlwriter.Write("endmodule\n\n\n")

  def GenerateVerilogHDL(self,hdlwriter):
    Dev.Debug(Dev.Info,"Component.GenerateVerilogHDL(self,hdlwriter)")
    self.GenerateVerilogHDLHeader(hdlwriter)
    self.GenerateVerilogHDLBody(hdlwriter)
    self.GenerateVerilogHDLFooter(hdlwriter)

  def GenerateVHDLHeader(self,hdlwriter):
    Dev.Debug(Dev.Stop,"Component.GenerateVHDLHeader(self,hdlwriter) NOT IMPLEMENTED")

  def GenerateVHDLBody(self,hdlwriter):
    Dev.Debug(Dev.Stop,"Component.GenerateVHDLBody(self,hdlwriter) NOT IMPLEMENTED")

  def GenerateVHDLFooter(self,hdlwriter):
    Dev.Debug(Dev.Stop,"Component.GenerateVHDLFooter(self,hdlwriter) NOT IMPLEMENTED")

  def GenerateVHDL(self,hdlwriter):
    Dev.Debug(Dev.Info,"Component.GenerateVerilogHDL(self,hdlwriter)")
    self.GenerateVHDLHeader(hdlwriter)
    self.GenerateVHDLBody(hdlwriter)
    self.GenerateVHDLFooter(hdlwriter)

  def GenerateNetName(self):
    Dev.Debug(Dev.Info,"HDLComponent.GenerateNetName(self)")
    tmpstr = "net" + str(self.NetNameNumber)
    self.NetNameNumber += 1
    return tmpstr

  def GenerateComponentName(self):
    Dev.Debug(Dev.Info,"HDLComponent.GenerateComponentName(self)")
    tmpstr = "comp" + str(self.CompNameNumber)
    self.CompNameNumber += 1
    return tmpstr

  def GenerateName(self,prj = None):
    Dev.Debug(Dev.Info,"HDLComponent.GenerateName(self)")
    if (self.Parent is None):
      # We are an unnamed main component just set the name to maincomponent
      self.Name = "maincomponent"
    elif not(prj is None):
      # Get a global name unique from the project
      if self.Name is None:
        self.Name = prj.GenerateComponentTypeName()
    else:
      Dev.Debug(Dev.Stop,"Error GenerateName must be implemented or something.")
    return self.Name
```

Listing A.2: PHDL/Component.py

```
#!/usr/bin/env python
# PHDL ComponentImpl

import Dev

import Component

class ComponentImpl(Component.Component):
  def ParameterizationCheck(self):
    Dev.Debug(Dev.Info,"ComponentImpl.ParameterizationCheck(self)")
```

Listing A.3: PHDL/ComponentImpl.py

```python
#!/usr/bin/env python
# PHDL Connection

import Dev

class Connection:
  def __init__(self,locconn):
    Dev.Debug(Dev.Info,"Connection.__init__(self)")
    self.Connectors = { }
    self.LocalConnector = locconn

  # *INTERNAL* Attach a wire to a connector
  def Attach(self,connectionpath,connector):
    Dev.Debug(Dev.Info,"Connection.Attach(self,comp,conname)")
    self.Connectors[connectionpath + " " + connector.Name] = connector

  # *INTERNAL* Sets a connection’s anonymous flag
  def SetAnonymous(self):
    self.LocalConnector.SetAnonymous()
    return self

  # *INTERNAL* Returns the connection’s anonymous flag
  def IsAnonymous(self):
    return self.LocalConnector.IsAnonymous()

  # *INTERNAL* Reconnects all wires to a target connector
  def ReconnectTo(self,targetconnector):
    if (self.IsAnonymous() == 0):
      Dev.Debug(Dev.Stop,"Connection: Ack! I’m not an anonymous connection!")
    for k,v in self.Connectors.iteritems():
      v.Conn = targetconnector
      targetconnector.Connectors[k] = v
```

Listing A.4: PHDL/Connection.py

```python
#!/usr/bin/env python
# PHDL Connector

import Dev
import Components
import HDLIOType

class Connector(object):
  def __init__(self):
    Dev.Debug(Dev.Info,"Connector.__init__(self)")
    self.Name = ""
    self.Conn = None
    self.LocalConn = None
    self.Comp = None
    self.IsUsed = 0
    self.Anonymous = 0

  # *PUBLIC* Optional method called after a connector is attached to a component
  def LateInit(self):
    return

  # *PUBLIC* Overload how namespaces work so we can clean syntax
  def __setattr__(self,attr,val):
    Dev.Debug(Dev.Info,"Connector.__setattr__(self,attr,val)")
    self.__dict__[attr] = val

  # *PUBLIC* Overload how namespaces work so we can clean syntax
  def __getattr__(self,attr):
    Dev.Debug(Dev.Info,"Connector.__getattr__(self,attr)")
    if self.__dict__.has_key(attr):
      return self.__dict__[attr]
    else:
      Dev.Debug(Dev.Stop,"Unknown attribute: " + attr)
      return None

  # *INTERNAL* Choose a common parent!
  def ChooseCommonParent(self,b):
    if isinstance(b,int):
      # I dont actually have to create a component
      # Connect does that automatically for me!
      return a.Comp
    if (self.Comp is b.Comp) and not(b.Comp is None):
      return self.Comp
    elif (self.Comp.Parent is b.Comp) and not(b.Comp is None):
      return b.Comp
    elif (b.Comp.Parent is self.Comp) and not(self.Comp is None):
      return self.Comp
    elif (self.Comp.Parent is b.Comp.Parent) and not(self.Comp.Parent is None):
      return self.Comp.Parent
    else:
      Dev.Debug(Dev.Stop,"No common parent between wires!")

  # *PUBLIC* Add operator
  def __add__(self,b):
    adder = Components.Adder.Adder()
    self.ChooseCommonParent(b).Anonymous = adder # Bind it to an autogenerated name
    Connect(self,adder.a)
    Connect(b,adder.b)
    mywire = self.Duplicate()
    self.ChooseCommonParent(b).Anonymous = mywire
    Connect(mywire,adder.o)
    return mywire# the return value is an anonymous wire

  # *PUBLIC* Sub operator
  def __sub__(self,b):
    subtractor = Components.Sub.Sub()
    self.ChooseCommonParent(b).Anonymous = subtractor # Bind it to an autogenerated name
    Connect(self,subtractor.a)
    Connect(b,subtractor.b)
    mywire = self.Duplicate()
    self.ChooseCommonParent(b).Anonymous = mywire
    Connect(mywire,subtractor.o).SetAnonymous()
    return mywire# the return value is an anonymous wire

  # *PUBLIC Or operator
  def __or__(self,b):
    orgate = Components.Or.Or()
    self.ChooseCommonParent(b).Anonymous = orgate # Bind it to an autogenerated name
    Connect(self,orgate.a)
    Connect(b,orgate.b)
    mywire = self.Duplicate()
    self.ChooseCommonParent(b).Anonymous = mywire
    Connect(mywire,orgate.o).SetAnonymous()
    return mywire# the return value is an anonymous wire

  # *PUBLIC And operator
  def __and__(self,b):
    andgate = Components.And.And()
    self.ChooseCommonParent(b).Anonymous = andgate # Bind it to an autogenerated name
    Connect(self,andgate.a)
    Connect(b,andgate.b)
    mywire = self.Duplicate()
    self.ChooseCommonParent(b).Anonymous = mywire
    Connect(mywire,andgate.o).SetAnonymous()
    return mywire# the return value is an anonymous wire

  # *PUBLIC Xor operator
  def __xor__(self,b):
    xorgate = Components.Xor.Xor()
    self.ChooseCommonParent(b).Anonymous = xorgate # Bind it to an autogenerated name
    Connect(self,xorgate.a)
    Connect(b,xorgate.b)
    mywire = self.Duplicate()
    self.ChooseCommonParent(b).Anonymous = mywire
    Connect(mywire,xorgate.o).SetAnonymous()
    return mywire# the return value is an anonymous wire

  # Do something intelligent with shift/rotate constant -> subwire
  # otherwise instantiate a component

  # *PUBLIC* Set name of a connector
  def SetName(self,str):
    Dev.Debug(Dev.Info,"Connector.SetName(self,str)")
    self.Name = str;

  # *PUBLIC* Configure an IO Port
  def ConfigureConnector(self,prj):
    Dev.Debug(Dev.Info,"Connector.ConfigureConnector(self,prj)")
    if not(self.Conn is None):
      for k,v in self.Conn.Connectors.iteritems(): # Check type?
        self.ConfigureEachOther(prj,v)
    elif (self.IOType != HDLIOType.Wire) and (self.IsUsed == 0):
      Dev.Debug(Dev.Warning,"Warning: Connector " + self.Name +
                " is not connected to anything.")
    # Check locally connected components
    if not(self.LocalConn is None):
      for k,v in self.LocalConn.Connectors.iteritems(): # Check type?
        self.ConfigureEachOther(prj,v)

  # *PUBLIC* Exchange parameters between two connectors
  def ConfigureEachOther(self,prj,v):
    Dev.Debug(Dev.Info,"Connector.ConfigureEachOther(self,prj,v)")

  # *PUBLIC* Check the configuration just before code generation
  def ParameterizationCheck(self,prj):
    Dev.Debug(Dev.Info,"Connector.ParameterizationCheck(self,prj)")

  # *INTERNAL* Write IO Port input/output/wire/reg definitions
  def WriteIOPorts(self,hdlwriter):
    Dev.Debug(Dev.Info,"Connector.WriteIOPorts(self,hdlwriter)")

  # *INTERNAL* Write IO Port names in a comma seperated list
  def WriteIOPortNames(self,hdlwriter):
    Dev.Debug(Dev.Info,"Connector.WriteIOPortNames(self,hdlwriter)")

  # *INTERNAL* Write IO Port bindings to another module
  def WriteIOPortBindings(self,hdlwriter):
    Dev.Debug(Dev.Info,"Connector.WriteIOPortBindings(self,hdlwriter)")

  # *INTERNAL* Writes local IO Port logic to the current module
  def WriteIOPortLogic(self,hdlwriter):
    Dev.Debug(Dev.Info,"Connector.WriteIOPortLogic(self,hdlwriter)")

  # *INTERNAL* Returns binding name string
  def WriteIOPortBindingName(self,relparent = None):
    Dev.Debug(Dev.Info,"Connector.WriteIOPortBindingName(self,hdlwriter)")
    if (relparent is None) or (relparent is self.Comp):
      return self.Name
    elif (relparent is self.Comp.Parent):
```

```python
    if (self.Conn is None) or (self.Conn.LocalConnector is None):
      print self.Name
      Dev.Debug(Dev.Stop,
                "PANIC: Trying to write a binding for an unconnected connector!")
    return self.Conn.LocalConnector.Name


  # *INTERNAL* Sets the parent connector
  def Connect(self,conn):
    Dev.Debug(Dev.Info,"Connector.Connect(self,conn)")
    self.Conn = conn

  # *INTERNAL* Sets the local connector
  def SetLocalConnection(self,conn):
    Dev.Debug(Dev.Info,"Connector.SetLocalConnection(self,conn)")
    self.LocalConn = conn

  # *PUBLIC* Mark a connector as used to avoid warning messages
  # Fools us into thinking its connected
  def SetUsed(self):
    Dev.Debug(Dev.Info,"Connector.SetUsed(self)")
    self.IsUsed = 1
    return self

  # *PUBLIC* Checks if a connector is connected
  def IsConnected(self):
    Dev.Debug(Dev.Info,"Connector.IsConnected(self)")
    if (self.Conn is None) and (self.IsUsed is 0):
      return 0
    else:
      return 1

  # *PUBLIC* Removes a connector from a module
  def Remove(self):
    self.Comp.__delattr__(self.Name)

  # *PUBLIC* Sets a connector’s anonymous flag
  def SetAnonymous(self):
    self.Anonymous = 1
    return self

  # *PUBLIC* Returns the connector’s anonymous flag
  def IsAnonymous(self):
    return self.Anonymous

  # *INTERNAL* Duplicates a connector to create a local wire of the same type
  def Duplicate(self):
    Dev.Debug(Dev.Error,"Error: Connector does not implement the
                         Duplicate method you cannot use anonymous connections.")
```

Listing A.5: PHDL/Connector.py

```python
#!/usr/bin/env python
# PHDL Debugging Utility Functions

# Enumeration
Debug = 4
Info = 3
Warning = 2
Error = 1
Stop = 0

# Change Me
DebugLevel = 2

```

```python
# State Variables
global totalwarnings
totalwarnings = 0
global totalerrors
totalerrors = 0
global represswarning
represswarning = 0

import os


def ResetErrorCount():
  global totalwarnings
  global totalerrors
  global represswarning
  totalwarnings = 0
  totalerrors = 0
  represswarning = 0

def ShowResults(msg):
  if totalerrors == 0:
    print msg + " Completed Successfully"
  else:
    print msg + " Failed: " + str(totalerrors) + " Errors"
  if totalwarnings != 0:
    print str(totalwarnings) + " Warnings encountered"
  print ""
  if totalerrors != 0:
    os._exit(-1)

def Debug(level,msg):
  global totalwarnings
  global totalerrors
  global represswarning
  if (level == Warning) and (represswarning == 1):
    return
  if DebugLevel >= level:
    print msg
  if level == Warning:
    totalwarnings += 1
  if level == Error:
    totalerrors += 1
  if totalwarnings >= 40:
    print "Error: Too many warnings"
    os._exit(-1)
  if totalerrors >= 20:
    print "Error: Too many errors"
    os._exit(-1)
  if level == Stop:
    print "FATAL ERROR EXITTING"
    os._exit(-1)

def DisableWarnings():
  global represswarning
  represswarning = 1

def EnableWarnings():
  global represswarning
  represswarning = 0
```

Listing A.6: PHDL/Dev.py

```python
#!/usr/bin/env python
# PHDL HDLIOType

```

```
Input = 1
Output = 2
InOut = 3
Wire = 4
```

Listing A.7: PHDL/HDLIOType.py

```
#!/usr/bin/env python
# PHDL HDLNetType

Wire = 1
Reg = 2
Tri = 3
Wand = 4
Wor = 5
Triand = 6
Trior = 7
Trireg = 8
```

Listing A.8: PHDL/HDLNetType.py

```
#!/usr/bin/env python
# PHDL HDLWriter

import Dev
import HDLNetType

class HDLWriter:
  def __init__(self):
    Dev.Debug(Dev.Info,"HDLWriter.__init__(self)")

  def Open(self,str):
    Dev.Debug(Dev.Info,"HDLWriter.Open(self,str)")
    self.outfile = open(str,'w+')
    self.Write("// Autogenerated by PHDL\n\n");

  def Close(self):
    Dev.Debug(Dev.Info,"HDLWriter.Close(self)")
    self.outfile.close()

  def Write(self,str):
    Dev.Debug(Dev.Info,"HDLWriter.Write(self,str)")
    self.outfile.write(str)

  def WriteModule(self,module):
    Dev.Debug(Dev.Error,"HDLWriter.WriteModule(self,module) ERROR NOT IMPLEMENTED")
```

Listing A.9: PHDL/HDLWriter.py

```
#!/usr/bin/env python
# PHDL Project

import Dev
import Component

"""
Project Parameters:
TARGET:
```

```
* XILINX_SPARTAN
* XILINX_SPARTAN2
* XILINX_SPARTAN3
* XILINX_VIRTEX
* XILINX_VIRTEX2
* XILINX_VIRTEX2PRO
* XILINX_VIRTEX3
* XILINX_VIRTEX4
* RTL
* SIM
GOAL:
* SIZE
* SPEED
* POWER
"""

class Project:
  def __init__(self,cirname = ""):
    Dev.Debug(Dev.Info,"Project.__init__(self)")
    self.Name = cirname
    self.UnconfiguredComponents = { }
    self.Components = { }
    self.MainComponentInstance = None
    self.MainComponentName = ""
    self.MultipleFiles = 0
    self.Parameters = { }
    self.ChangedComponents = { }
    self.CompNameNumber = 0

  # Component Interface
  def AddComponent(self,comp):
    Dev.Debug(Dev.Info,"Project.AddComponent(self,typename,comp)")
    self.UnconfiguredComponents[comp] = comp

  def RemoveComponent(self,comp):
    Dev.Debug(Dev.Info,"Project.RemoveComponent(self,typename,comp)")
    #del self.UnconfiguredComponents = comp

  def SetMainComponent(self,instance):
    Dev.Debug(Dev.Info,"Project.SetMainComponent(self,typename)")
    iterations = 0
    self.MainComponentInstance = instance
    TmpComponents = { }
    # New Auto-Configuration Method
    self.MainComponentInstance.AddBindings(self)
    TmpComponents = self.UnconfiguredComponents
    while len(TmpComponents) != 0:
      print "Iteration " + str(iterations) + ": " + str(len(TmpComponents)) +
            " Changed Components"
      self.ChangedComponents.clear()
      # - Add all components to the change list
      for c in TmpComponents:
        c.ConfigureComponent(self)
      if (iterations == 0):
        Dev.DisableWarnings()
      TmpComponents = self.ChangedComponents.values()
      iterations += 1
      if iterations > 50:
        Dev.Debug(Dev.Stop,"Project needs more than 50 iterations to solve
                  please check that there are no bugs.")
    Dev.EnableWarnings()
    self.MainComponentInstance.AddBindings(self)
    self.MainComponentInstance.ParameterizationCheck(self)

  # Generation Control
  def SetMultifileGeneration(self,tf):
    Dev.Debug(Dev.Info,"Project.SetMultifileGeneration(self,tf)")
    self.MultipleFiles = tf

  def GenerateHDL(self,hdlwriter):
    Dev.Debug(Dev.Info,"Project.GenerateHDL(self,hdlwriter)")
    for k,v in self.UnconfiguredComponents.iteritems():
      tmp = v.GetInstance()
      self.Components[tmp.Name] = tmp
    Dev.Debug(Dev.Info,"Project.GenerateHDL(self)")
    if self.MultipleFiles == 0:
      hdlwriter.Write("// "+self.Name+" Project\n\n")
    # Generate components
    for k, v in self.Components.iteritems():
      if not(v is self.MainComponentInstance):
        if self.MultipleFiles == 1:
          hdlwriter.Open(k + ".v") # language dep
          if self.Name != "":
            hdlwriter.Write("// Subcomponent Component: "+self.Name+" Project\n\n")
        hdlwriter.WriteModule(v)
        if self.MultipleFiles == 1:
          hdlwriter.Close()
    # Generate top component
    if self.MultipleFiles == 1:
      hdlwriter.Open(self.MainComponentInstance.Name + ".v") # language dep
      if self.Name != "":
        hdlwriter.Write("// Main Component: "+self.Name+" Project\n\n")
    hdlwriter.WriteModule(self.MainComponentInstance)
    if self.MultipleFiles == 1:
      hdlwriter.Close()

  # Global Parameter Control
  def SetParameter(self,param,value):
    Dev.Debug(Dev.Info,"Project.SetParameter(self,param,value)")
    self.Parameters[param] = value

  def GetParameter(self,param):
    Dev.Debug(Dev.Info,"Project.GetParameter(self,param)")
    return self.Parameters[param]

  # Auto-Configuration Control
  def AddChangedComponent(self,comp):
    Dev.Debug(Dev.Info,"Project.AddChangedComponent(self,comp)")
    self.ChangedComponents[id(comp)] = comp;

  def AddChangedConnector(self,conn):
    Dev.Debug(Dev.Info,"Project.AddChangedConnector(self,conn)")
    self.AddChangedComponent(conn.Comp)
    if not(conn.Conn is None):
      for k, v in conn.Conn.Connectors.iteritems():
        self.AddChangedComponent(v.Comp);
    if not(conn.LocalConn is None):
      for k, v in conn.LocalConn.Connectors.iteritems():
        self.AddChangedComponent(v.Comp);

  def GenerateComponentTypeName(self):
    Dev.Debug(Dev.Info,"Project.GenerateComponentTypeName(self)")
    tmpstr = "CompType" + str(self.CompNameNumber)
    self.CompNameNumber += 1
    return tmpstr
```

Listing A.10: PHDL/HDLNetType.py

```
#!/usr/bin/env python
# PHDL Utility Functions

import math

```

```python
# Warning unchecked width this may truncate numbers
def VerilogBinary(number,width = None):
  if width is None:
    printwidth = int(math.ceil(math.log(number+1,2)))
  else:
    printwidth = width
  if (printwidth == 0):
    printwidth = printwidth + 1
  retstr = str(printwidth) + "'b"
  for x in range(printwidth):
    retstr += str(number >> (printwidth - 1 - x) & 1)
  return retstr

# Warning unchecked width this may truncate numbers
def VerilogDecimal(number,width = None):
  return str(number)

# Warning unchecked width this may truncate numbers
def VerilogHex(number,width = None):
  return ""
```

Listing A.11: PHDL/Util.py

```python
#!/usr/bin/env python
# PHDL VerilogWriter

import Dev
import HDLIOType
import HDLNetType
import HDLWriter

class VerilogWriter(HDLWriter.HDLWriter):
  def __init__(self,str = ""):
    Dev.Debug(Dev.Info,"VerilogWriter.__init__(self,str)")
    if str != "":
      self.Open(str)

  def WriteModule(self,module):
    module.GenerateVerilogHDL(self)

  def WriteNet(self,Name,IOType,Type,Start,End):
    Dev.Debug(Dev.Info,"VerilogWriter.WriteNet(self,Name,IOType,Type,Start,End)")
    if IOType == HDLIOType.Input:
      self.outfile.write("input")
      self.writenetcommon(Name,Start,End)
    elif IOType == HDLIOType.Output:
      self.outfile.write("output")
      self.writenetcommon(Name,Start,End)
    elif IOType == HDLIOType.InOut:
      self.outfile.write("inout")
      self.writenetcommon(Name,Start,End)
    elif IOType == HDLIOType.Wire:
      Dev.Debug(Dev.Info,"VerilogWriter.WriteNet(self,Name,IOType,Type,Start,End) -
                Skipping I/O Definition (Wire)")
    else:
      Dev.Debug(Dev.Stop,"Error: Unknown HDLIOType")
    if Type == HDLNetType.Wire:
      self.outfile.write("wire");
    elif Type == HDLNetType.Reg:
      self.outfile.write("reg\t");
    elif Type == HDLNetType.Tri:
      self.outfile.write("tri");
    elif Type == HDLNetType.Wand:
      self.outfile.write("wand");
    elif Type == HDLNetType.Wor:
      self.outfile.write("wor");
    elif Type == HDLNetType.Triand:
      self.outfile.write("triand");
    elif Type == HDLNetType.Trior:
      self.outfile.write("trior");
    elif Type == HDLNetType.Trireg:
      self.outfile.write("trireg");
    else:
      Dev.Debug(Dev.Stop,"Error: Unknown HDLNetType")
    self.writenetcommon(Name,Start,End)

  def writenetcommon(self,Name,StartIndex,EndIndex):
    Dev.Debug(Dev.Info,"VerilogWriter.writenetcommon(self,Name,StartIndex,EndIndex)")
    self.outfile.write("\t")
    if (StartIndex != EndIndex):
      self.outfile.write("[" + str(StartIndex) + ":" + str(EndIndex) + "]")
    else:
      self.outfile.write("\t")
    self.outfile.write("\t" + Name + ";\n")
```

Listing A.12: PHDL/VerilogWriter.py

A.1 Connectors

```
#!/usr/bin/env python

import sys
import os
import re

# Fix the path to be able to include PHDL core components
# For now the worst this does it include the path twice
# it shouldn’t hurt anything

sys.path.append(sys.modules.get(__name__).__path__[0] + "/../../")

# Construct the __all__ variable

__all__ = os.listdir(sys.modules.get(__name__).__path__[0])

__all__.remove('__init__.py')
__all__.remove('__init__.pyc')

for e in __all__:
  if (re.compile("[a-zA-Z0-9]*\.py$").match(e,1) is None):
    __all__.remove(e)

tmplist = [ ]

for e in __all__:
  tmp = re.split("[\.]",e)
  tmplist.append(tmp[0])

__all__ = tmplist

print "PHDL Framework: " + str(len(__all__)) + " Connectors Loaded"
```

Listing A.13: PHDL/Connectors/__init__.py

```
#!/usr/bin/env python
# PHDL AnonymousBulkConnector

from PHDL import *

import re

"""
- Consider Makeing Subwires or allowing us to get a subwire from a connector
- Warn when wires output dont explicitly state their unconnected, Error for inputs
"""

def getportbindingname(x,relparent):
  if (isinstance(x[1],int) or isinstance(x[1],long)):
    return Util.VerilogBinary(x[1],x[0])
  else:
    if x[0] == 1:
      return x[1].WriteIOPortBindingName(relparent)
    else:
      return "{" + str(x[0]) + "{" + x[1].WriteIOPortBindingName(relparent) + "}}"

class AnonymousBulkConnector(Connector.Connector):
  def __init__(self,*cons):
    Dev.Debug(Dev.Info,"AnonymousBulkConnector.__init__(self)")
    self.Name = ""
    self.Comp = None
    self.Conn = None
    self.LocalConn = None
    self.IOType = HDLIOType.Wire
    self.Type = HDLNetType.Wire
    self.Width = None
    self.IsUsed = 0
    self.Anonymous = 0
    self.BundledWires = [ ]
    self.SubWires = { } # I need to support subwires!
    for e in cons:
      if isinstance(e,tuple):
        self.Add(e[0],e[1])
      else:
        self.Add(e)

  # Overload the methods to allow adding connectors

  """
  This needs work it has to be improved to support the situations better
  also it may have buggness
  """
  def ConfigureConnector(self,prj):
    Dev.Debug(Dev.Info,"AnonymousBulkConnector.ConfigureConnector(self,prj)")
    # Calculate the total Width
    # We only care about the inputs that were added because
    # there is no way to solve the 1 to N mapping
    width = 0
    for i in self.BundledWires:
      if (isinstance(i[1],Connector.Connector) and (i[1].Width is None)):
        width = None
      elif not(width is None):
        if (isinstance(i[1],int) or isinstance(i[1],long)):
          width = width + i[0]
        else:
          width = width + (i[0] * i[1].Width)
    self.Width = width

  def ParameterizationCheck(self,prj):
    Dev.Debug(Dev.Info,"AnonymousBulkConnector.ParameterizationCheck(self)")

  def WriteIOPorts(self,hdlwriter):
```

```python
    Dev.Debug(Dev.Info,"AnonymousBulkConnector.WriteIOPorts(self,hdlwriter)")

  def WriteIOPortNames(self,hdlwriter):
    Dev.Debug(Dev.Info,"AnonymousBulkConnector.WriteIOPortNames(self,hdlwriter)")

  def WriteIOPortBindings(self,hdlwriter):
    Dev.Debug(Dev.Info,"AnonymousBulkConnector.WriteIOPortBindings(self,hdlwriter)")

  def WriteIOPortBindingName(self):
    Dev.Debug(Dev.Info,"Connector.WriteIOPortBindingName(self,hdlwriter)")
    comps = [ ]
    for i in range(len(self.BundledWires)):
      comps.append(self.Comp)
    str = "{" + ",".join(map(getportbindingname,self.BundledWires,comps)) + "}"
    return str

  def Duplicate(self,name):
    Dev.Debug(Dev.Stop,"AnonymousBulkConnector.Duplicate(self) Not Implemented")

  def Add(self,conn,count = 1):
    Dev.Debug(Dev.Info,"AnonymousBulkConnector.Add(self)")
    self.BundledWires.append((count,conn))
```

Listing A.14: PHDL/Connectors/AnonymousBulkConnector.py

```python
#!/usr/bin/env python
# PHDL BulkConnector

from PHDL import *

import re

"""
- Consider Makeing Subwires or allowing us to get a subwire from a connector
- Warn when wires output dont explicitly state their unconnected, Error for inputs
"""


class BulkConnector(Connector.Connector):
  def __init__(self):
    Dev.Debug(Dev.Info,"WireConnector.__init__(self,type,start,end)")
    self.Name = ""
    self.Comp = None
    self.Conn = None
    self.LocalConn = None
    self.IsUsed = 0
    self.Anonymous = 0
    self.SubWires = { }
    self.GlobalToLocal = { }
    self.LocalToGlobal = { }
    self.IOType = 0 # Bogus

  def LateInit(self):
    newsubwires = { }
    for k,v in self.SubWires.iteritems():
      newsubwires[self.Name + "_" + k] = v
      v.Name = self.Name + "_" + k
      v.Comp = self.Comp
      self.GlobalToLocal[k] = self.Name + "_" + k
      self.LocalToGlobal[self.Name + "_" + k] = k
      v.LateInit()
    self.SubWires = newsubwires
    return

  # *PUBLIC* Overload how namespaces work so we can clean syntax
  def __setattr__(self,attr,val):
    Dev.Debug(Dev.Info,"WireConnector.__setattr__(self,attr,val)")
    if isinstance(val,Connector.Connector):
      self.SubWires[attr] = val
      return
    # Default case for non-connectors
    self.__dict__[attr] = val

  # *PUBLIC* Overload how namespaces work so we can clean syntax
  def __getattr__(self,attr):
    Dev.Debug(Dev.Info,"WireConnector.__getattr__(self,attr)")
    if self.SubWires.has_key(attr):
      return self.SubWires[attr]
    elif self.SubWires.has_key(self.Name + "_" + attr):
      return self.SubWires[self.Name + "_" + attr]
    elif self.__dict__.has_key(attr):
      return self.__dict__[attr]
    else:
      print "Trying to get " + attr + " in bulkconnector " + self.Name
      Dev.Debug(Dev.Stop,"WireConnector.__getattr__(self,attr) does not exist!!!")

  # *PUBLIC* Overload how namespaces work so we can clean syntax
  def __delattr__(self,attr):
    Dev.Debug(Dev.Info,"WireConnector.__delattr__(self,attr)")
    if self.SubWires.has_key(attr):
      # This has to be cleaned up to allow connectors a chance to cleanup logic!
      del self.SubWires[attr]
      return
    Dev.Debug(Dev.Stop,"WireConnector.__delattr__: UH OH CANT DELETE!")

  # Usually this is overloaded with constraints between connectors and parameters
  def ConnectorConstraints(self,prj):
    Dev.Debug(Dev.Info,"BulkConnector.ConnectorConstraints(self,prj)")
    return

  def ConfigureEachOther(self,prj,v):
    self.ConnectorConstraints(prj) # Calling it more often than needed
    for kwire,vwire in self.SubWires.iteritems():
      # Configure connectors
      name = v.GlobalToLocal[self.LocalToGlobal[kwire]]
      vwire.ConfigureEachOther(prj,v.SubWires[name])

  # Usually this is overloaded with constraints between connectors and parameters
  def ConstraintCheck(self,prj):
    Dev.Debug(Dev.Info,"BulkConnector.ConstraintCheck(self,prj)")
    return

  # *PUBLIC* Parameterization check
  def ParameterizationCheck(self,prj):
    Dev.Debug(Dev.Info,"BulkConnector.ParameterizationCheck(self)")
    self.ConstraintCheck(prj)
    for k,v in self.SubWires.iteritems():
      # Configure connectors
      v.ParameterizationCheck(prj)

  def WriteIOPorts(self,hdlwriter):
    Dev.Debug(Dev.Info,"BulkConnector.WriteIOPorts(self,hdlwriter)")
    for k,v in self.SubWires.iteritems():
      v.WriteIOPorts(hdlwriter)

  def WriteIOPortLogic(self,hdlwriter):
    Dev.Debug(Dev.Info,"BulkConnector.WriteIOPortLogic(self,hdlwriter)")
    for k,v in self.SubWires.iteritems():
      v.WriteIOPortLogic(hdlwriter)

  def WriteIOPortNames(self,hdlwriter):
    Dev.Debug(Dev.Info,"BulkConnector.WriteIOPortNames(self,hdlwriter)")
    # Comma seperated list
    i = len(self.SubWires)
    for k,v in self.SubWires.iteritems():
      # Hack to make hiarchical BulkConnectors work
      if isinstance(v,BulkConnector):
        hdlwriter.WriteIOPortNames(hdlwriter)
      else:
        hdlwriter.Write(k)
      i = i - 1
      if (i > 0):
        hdlwriter.Write(",")

  def WriteIOPortBindings(self,hdlwriter):
    Dev.Debug(Dev.Info,"BulkConnector.WriteIOPortBindings(self,hdlwriter)")
    if self.Conn is None:
      Dev.Debug(Dev.Stop,"BulkConnector " + self.Name + " in " + self.Comp.InstanceName +
                " is not connected to anything.")
    i = len(self.SubWires)
    for kwire,vwire in self.SubWires.iteritems():
      if isinstance(vwire,BulkConnector):
        vwire.WriteIOPortBindings(hdlwriter)
      else:
        name = self.Conn.LocalConnector.GlobalToLocal[self.LocalToGlobal[kwire]]
        hdlwriter.Write("\t." + kwire + "(" +
                   self.Conn.LocalConnector.SubWires[name].WriteIOPortBindingName() + ")")
        i = i - 1
        if (i > 0):
          hdlwriter.Write(",\n")
        else:
          hdlwriter.Write("\n")


  def Duplicate(self,name):
    Dev.Debug(Dev.Info,"BulkConnector.Duplicate(self)")
    dup = WireConnector(name,None)
    # Name already taken care of
    self.LocalConn = None
    # IsUsed already taken care of
    # Must map all connectors through the duplicate method!
    return dup
```

Listing A.15: PHDL/Connectors/BulkConnector.py

```
#!/usr/bin/env python
# PHDL ConstantConnector

from PHDL import Connector
from PHDL import Dev
from PHDL import HDLIOType
from PHDL import HDLNetType
from PHDL import Util

import re
import math

"""
- Consider Makeing Subwires or allowing us to get a subwire from a connector
- Warn when wires output dont explicitly state their unconnected, Error for inputs
"""

# THIS IS VERY SKETCHY BUT IT CURRENTLY WORKS IT NEEDS A LOT OF CLEANUP

class ConstantConnector(Connector.Connector):
  def __init__(self,name,comp,value):
    Dev.Debug(Dev.Info,"ConstantConnector.__init__(self,supername,name,start,end)")
    self.Comp = comp
    self.Conn = None
    self.LocalConn = None
    self.IOType = HDLIOType.Wire
    self.Type = HDLNetType.Wire
    self.Value = value
    self.Width = None
    self.Name = name
    self.IsUsed = 1
    self.Anonymous = 0
    if not(comp is None):
      comp.AddConnector(self)

  def ConfigureEachOther(self,prj,v):
    if not(v.Width is None):
      if self.Width is None:
        self.Width = v.Width
        prj.AddChangedConnector(self)
      elif (self.Width != v.Width):
        Dev.Debug(Dev.Error,"Error: ConstantConnector " + self.Name +
                  " found an inconsistancy with Local Connector named " +
                  v.Name + " in the configuration.")

  def ParameterizationCheck(self,prj):
    Dev.Debug(Dev.Info,"ConstantConnector.ParameterizationCheck(self)")
    if self.Width is None:
      Dev.Debug(Dev.Error,"Error: ConstantConnector " + self.Name +
                " failed to configure the width parameter.");

  def WriteIOPorts(self,hdlwriter):
    Dev.Debug(Dev.Info,"ConstantConnector.WriteIOPorts(self,hdlwriter)")
    # I may want to support a direction flag for these connectors
    if self.Width is None:
      Dev.Debug(Dev.Stop,"Fatal Error: ConstantConnector " + self.Name +
                " was never configured with a Width.")
    #hdlwriter.WriteNet(self.Name,self.IOType,self.Type,self.Width - 1,0)
    # Add Assign Statement

  def WriteIOPortNames(self,hdlwriter):
    Dev.Debug(Dev.Info,"ConstantConnector.WriteIOPortNames(self,hdlwriter)")
    Dev.Debug(Dev.Warning,"Error: ConstantConnector cannot be used as an input/output.")

  def WriteIOPortBindings(self,hdlwriter):
    Dev.Debug(Dev.Info,"ConstantConnector.WriteIOPortBindings(self,hdlwriter)")
    hdlwriter.Write("\t." + self.Name + "(" + self.Conn.LocalName + ")")

  def WriteIOPortLogic(self,hdlwriter):
    Dev.Debug(Dev.Info,"ConstantConnector.WriteIOPortLogic(self,hdlwriter)")

  def WriteIOPortBindingName(self):
    Dev.Debug(Dev.Info,"ConstantConnector.WriteIOPortBindingName(self,hdlwriter)")
    return Util.VerilogBinary(self.Value,self.Width)

  def Duplicate(self):
    Dev.Debug(Dev.Info,"ConstantConnector.Duplicate(self)")
    Dev.Debug(Dev.Warning,
              "Error: ConstantConnector does not know how to duplicate itself.")
    return dup
```

Listing A.16: PHDL/Connectors/ConstantConnector.py

```
#!/usr/bin/env python
# PHDL SubWireConnector

from PHDL import *

```

```
#from PHDL.Connectors.WireConnector import WireConnector

import re
import math

"""
- Consider Makeing Subwires or allowing us to get a subwire from a connector
- Warn when wires output dont explicitly state their unconnected, Error for inputs
"""

class SubWireConnector(Connector.Connector):
  def __init__(self,parentcomp,supername,name,comp,start,end = None):
    Dev.Debug(Dev.Info,"SubWireConnector.__init__(self,supername,name,start,end)")
    self.Parent = parentcomp
    self.SuperName = supername
    self.Comp = comp
    self.Conn = None
    self.LocalConn = None
    self.IOType = HDLIOType.Wire
    self.Type = HDLNetType.Wire
    self.Start = start
    if (end is None):
      self.End = start
    else:
      self.End = end
    self.Width = int(math.fabs(self.Start - self.End)) + 1
    self.Name = name
    self.IsUsed = 1
    self.Anonymous = 0
    if not(comp is None):
      comp.AddConnector(self)

  def ConfigureConnector(self,prj):
    Dev.Debug(Dev.Info,"SubWireConnector.ConfigureConnector(self,prj)")
    #print self.Comp.InstanceName + ":" + self.Name
    # Check connected wires from parent component
    # Later check IO connections
    if not(self.Conn is None):
      for k,v in self.Conn.Connectors.iteritems(): # Check type?
        if not(v.Width is None):
          if self.Width is None:
            self.Width = v.Width
            prj.AddChangedConnector(self)
          elif (self.Width != v.Width):
            Dev.Debug(Dev.Error,"Error: SubWireConnector " + self.Name +
                      " found an inconsistancy with Connector named " +
                      v.Name + " in the configuration.")
    elif (self.IOType != HDLIOType.Wire) and (self.IsUsed == 0):
      Dev.Debug(Dev.Warning,"Warning: SubWireConnector " + self.Name +
                " is not connected to anything.")
    # Check locally connected components
    if not(self.LocalConn is None):
      for k,v in self.LocalConn.Connectors.iteritems(): # Check type?
        if not(v.Width is None):
          if self.Width is None:
            self.Width = v.Width
            prj.AddChangedConnector(self)
          elif (self.Width != v.Width):
            Dev.Debug(Dev.Error,"Error: SubWireConnector " + self.Name +
                      " found an inconsistancy with Local Connector named " +
                      v.Name + " in the configuration.")

  def ParameterizationCheck(self,prj):
    Dev.Debug(Dev.Info,"SubWireConnector.ParameterizationCheck(self)")
    if self.Width is None:
      Dev.Debug(Dev.Error,"Error: SubWireConnector " + self.Name +
                " failed to configure the width parameter.");

  def WriteIOPorts(self,hdlwriter):
    Dev.Debug(Dev.Info,"SubWireConnector.WriteIOPorts(self,hdlwriter)")
    # I may want to support a direction flag for these connectors
    if self.Width is None:
      Dev.Debug(Dev.Stop,"Fatal Error: SubWireConnector " + self.Name +
                " was never configured with a Width.")
    #hdlwriter.WriteNet(self.Name,self.IOType,self.Type,self.Width - 1,0)
    # Add Assign Statement

  def WriteIOPortNames(self,hdlwriter):
    Dev.Debug(Dev.Info,"SubWireConnector.WriteIOPortNames(self,hdlwriter)")
    hdlwriter.Write(self.Name)

  def WriteIOPortBindings(self,hdlwriter):
    Dev.Debug(Dev.Info,"SubWireConnector.WriteIOPortBindings(self,hdlwriter)")
    if self.Conn is None:
      Dev.Debug(Dev.Stop,"SubWireConnector " + self.Name + " in " +
                self.Comp.InstanceName + " is not connected to anything.")
    hdlwriter.Write("\t." + self.Name + "(" +
                    self.Conn.LocalConnector.WriteIOPortBindingName() + ")")

  def WriteIOPortLogic(self,hdlwriter):
    Dev.Debug(Dev.Info,"SubWireConnector.WriteIOPortLogic(self,hdlwriter)")

  def WriteIOPortBindingName(self,relparent = None):
    Dev.Debug(Dev.Info,"Connector.WriteIOPortBindingName(self,hdlwriter)")
    if (relparent is None) or (relparent is self.Parent.Comp):
      return self.Name
    elif (relparent is self.Comp.Parent):
      if (self.Parent.Conn is None) or (self.Parent.Conn.LocalConnector is None):
        Dev.Debug(Dev.Stop,
                  "PANIC: Trying to write a binding for an unconnected connector!")
      if (self.Width == 1):
        return self.Parent.Conn.LocalConnector.Name + "[" + str(self.Start) + "]"
      else:
        return self.Parent.Conn.LocalConnector.Name + "[" + str(self.Start) + ":" +
               str(self.End) + "]"

  def Duplicate(self):
    Dev.Debug(Dev.Info,"WireConnector.Duplicate(self)")
    from PHDL.Connectors.WireConnector import WireConnector
    dup = WireConnector()
    # IsUsed already taken care of
    dup.IOType = HDLIOType.Wire
    dup.Type = HDLNetType.Wire
    dup.Width = self.Width
    return dup
```

Listing A.17: PHDL/Connectors/SubWireConnector.py

```
#!/usr/bin/env python
# PHDL WireConnector

from PHDL import *

import re

"""
- Consider Makeing Subwires or allowing us to get a subwire from a connector
- Warn when wires output dont explicitly state their unconnected, Error for inputs
"""

class WireConnector(Connector.Connector):
  def __init__(self,iotype = HDLIOType.Wire,type = HDLNetType.Wire,width = None):
    Dev.Debug(Dev.Info,"WireConnector.__init__(self,type,start,end)")
    self.Name = ""
    self.Comp = None
    self.Conn = None
    self.LocalConn = None
    self.IOType = iotype
    self.Type = type
    self.Width = width
    self.IsUsed = 0
    self.Anonymous = 0
    self.SubWires = { }

  def ConfigureEachOther(self,prj,v):
    if not(v.Width is None):
      if self.Width is None:
        self.Width = v.Width
        prj.AddChangedConnector(self)
      elif (self.Width != v.Width):
        Dev.Debug(Dev.Error,"Error: WireConnector " + self.Name + " in component " +
               self.Comp.InstanceName + " found an inconsistancy with Connector named " +
               v.Name + " in the configuration.")
    elif not(self.Width is None):
      v.Width = self.Width

  def ParameterizationCheck(self,prj):
    Dev.Debug(Dev.Info,"WireConnector.ParameterizationCheck(self)")
    if self.Width is None:
      Dev.Debug(Dev.Error,"Error: WireConnector " + self.Name + " in component " +
                self.Comp.Name + " failed to configure the width parameter.")

  def WriteIOPorts(self,hdlwriter):
    Dev.Debug(Dev.Info,"WireConnector.WriteIOPorts(self,hdlwriter)")
    # I may want to support a direction flag for these connectors
    if self.Width is None:
      Dev.Debug(Dev.Stop,"Error: WireConnector " + self.Name + " in component " +
                self.Comp.Name + " failed to configure the width parameter.")
    hdlwriter.WriteNet(self.Name,self.IOType,self.Type,self.Width - 1,0)

  def WriteIOPortNames(self,hdlwriter):
    Dev.Debug(Dev.Info,"WireConnector.WriteIOPortNames(self,hdlwriter)")
    hdlwriter.Write(self.Name)

  def WriteIOPortBindings(self,hdlwriter):
    Dev.Debug(Dev.Info,"WireConnector.WriteIOPortBindings(self,hdlwriter)")
    if self.Conn is None:
      Dev.Debug(Dev.Stop,"WireConnector " + self.Name + " in " + self.Comp.InstanceName +
                " is not connected to anything.")
    hdlwriter.Write("\t." + self.Name + "(" +
                    self.Conn.LocalConnector.WriteIOPortBindingName() + ")")

  def Duplicate(self):
    Dev.Debug(Dev.Info,"WireConnector.Duplicate(self)")
    dup = WireConnector()
    dup.IOType = HDLIOType.Wire
    dup.Type = HDLNetType.Wire
    dup.Width = self.Width
    return dup

  def RenameConnector(self,newname):
    # Rename all subconnectors!
    Dev.Debug(Dev.Info,"WireConnector.RenameConnector(self,newname)")
    self.Name = newname
    newsubwires = { }
    for k,v in self.SubWires:
      # TODO: Construct new name and rename internal structure!
      v.RenameConnector(self,newname)
    # TODO: Fix Connection registration!

  def CreateSubconnector(self,start,end = None):
    Dev.Debug(Dev.Info,"WireConnector.CreateSubconnector(self,start,end)")
    from PHDL.Connectors.SubWireConnector import SubWireConnector
    name = ""
    if end is None:
      name = self.Name + "[" + str(start) + "]"
    else:
      name = self.Name + "[" + str(start) + ":" + str(end) + "]"
    if self.SubWires.has_key(name):
      return self.SubWires[name]
    else:
      subconn = SubWireConnector(self,self.Name,name,self.Comp,start,end)
      self.SubWires[name] = subconn
      return subconn

  def __getitem__(self, index):
    if isinstance(index,slice):
      return self.CreateSubconnector(index.start,index.stop)
    elif isinstance(index,int):
      return self.CreateSubconnector(index)
    else:
      Dev.Debug(Dev.Stop,"WireConnector: Unsupported subconnector type!")
```

Listing A.18: PHDL/Connectors/WireConnector.py

Appendix B

PHDL Two Stage Beta Component

```
#!/usr/bin/env python
# PHDL Beta

from PHDL import *
from PHDL.Connectors import WireConnector
from PHDL.IPLibrary.Beta2Impl import Beta2Impl

class Beta(Component.Component):
  def __init__(self,**cons):
    self.Init()
    self.clk = WireConnector(HDLIOType.Input)
    self.reset = WireConnector(HDLIOType.Input)
    self.irq = WireConnector(HDLIOType.Input,width = 1)
    self.ma = WireConnector(HDLIOType.Output,width = 32)
    self.mdin = WireConnector(HDLIOType.Input)
    self.mdout = WireConnector(HDLIOType.Output)
    self.mwe = WireConnector(HDLIOType.Output)
    self.AutoConnect(cons)

  def ConfigureComponent(self,prj):
    if self.Instance is None:
      self.Instance = Beta2Impl()
      self.InitInstance()
    self.Instance.ConfigureComponent(prj)
    self.Instance.GenerateName()

  def ParameterizationCheck(self,prj):
    pass
```

Listing B.1: Beta Processor Meta-Component

```
#!/usr/bin/env python
# PHDL Beta

from PHDL import *

from PHDL.Components.Mux import Mux
from PHDL.Components.DLatch import DLatch
from PHDL.Components.Not import Not
from PHDL.Components.And import And
from PHDL.Components.Or import Or
from PHDL.Components.Xor import Xor
from PHDL.Components.BusNor import BusNor
from PHDL.Components.Adder import Adder
from PHDL.Components.AddSub import AddSub
from PHDL.Components.Compare import Compare
from PHDL.Components.Shift import Shift
from PHDL.Components.SignExt import SignExt
from PHDL.Components.RegisterFile import RegisterFile
from PHDL.Components.ROM import ROM

from PHDL.Connectors.WireConnector import WireConnector
from PHDL.Connectors.SubWireConnector import SubWireConnector
from PHDL.Connectors.AnonymousBulkConnector import AnonymousBulkConnector

class Beta2Impl(Component.Component):
  def __init__(self,**cons):
    # Subcomponents
    self.mamux = Mux() # Memory Address Mux
    self.wdmux = Mux() # Write Back Mux

    # Internal Wires
    self.inst = WireConnector(HDLIOType.Wire)
    self.co = WireConnector(HDLIOType.Wire)
    self.z = WireConnector(HDLIOType.Wire)
    self.aluz = WireConnector(HDLIOType.Wire)
    self.v = WireConnector(HDLIOType.Wire)
    self.n = WireConnector(HDLIOType.Wire)
    self.interupt = WireConnector(HDLIOType.Wire)

    # ROM Connectors
    self.branch = WireConnector(HDLIOType.Wire,width = 1)
    self.mem_next = WireConnector(HDLIOType.Wire)
    self.wasel = WireConnector(HDLIOType.Wire)
    self.pcsel_eff = WireConnector(HDLIOType.Wire)
    self.wdsel = WireConnector(HDLIOType.Wire)
    self.shiftdir = WireConnector(HDLIOType.Wire)
    self.shiftsext = WireConnector(HDLIOType.Wire)
    self.asel = WireConnector(HDLIOType.Wire)
    self.bsel = WireConnector(HDLIOType.Wire)
    self.csel = WireConnector(HDLIOType.Wire)
    self.addsub_op = WireConnector(HDLIOType.Wire)
    self.selfare_op = WireConnector(HDLIOType.Wire)

    # Control Signal's Computed From ROM
    self.werf = WireConnector(HDLIOType.Wire)
    self.annul = WireConnector(HDLIOType.Wire)
    self.msel = WireConnector(HDLIOType.Wire)
    self.msel_next = WireConnector(HDLIOType.Wire)
    self.mwrite = WireConnector(HDLIOType.Wire)
    self.pcsel = WireConnector(HDLIOType.Wire)

    self.jt = WireConnector(HDLIOType.Wire)
    self.jtchecked31 = WireConnector(HDLIOType.Wire)
    self.pcincunsafe = WireConnector(HDLIOType.Wire)
    self.nextpc = WireConnector(HDLIOType.Wire)

    # Commonly used wires that pass through logic
    self.not_msel = Not(i = self.msel) # !msel
    self.not_reset = Not(i = self.reset) # !reset

    # ******************************
    # *** Program Counter Control ***
    # ******************************

    self.pcplusfour = DLatch()
    self.instreg = DLatch()

    # Check jt Security Bit
    self.jtcheck = And(a = self.jt[31],b = self.nextpc[31],o = self.jtchecked31)
    self.jtsafe = AnonymousBulkConnector(self.jtchecked31,self.jt[30:0])

    # Tie the Security Bit for PC+4
    self.pcincsafe = AnonymousBulkConnector(self.nextpc[31],self.pcincunsafe[30:0])

    self.npc = DLatch(reset = self.reset,clk = self.clk,en = self.not_msel.o,q =
    self.nextpc)
    self.pcinc = Adder(a = self.nextpc,b = 4,o = self.pcincunsafe)
    self.pcplusfour = DLatch(reset = self.reset,clk = self.clk,
      en = self.not_msel.o,d = self.pcinc.o)
    self.pcmux = Mux(sel = self.pcsel,a = 0x80000000,b = 0x80000004,
      c = 0x80000008,d = self.jtsafe,e = self.pcincsafe,o = self.npc.d)

    Connect(self.mamux.b,self.jt)
    Connect(self.pcmux.o,self.mamux.a)
    Connect(self.mamux.o,self.ma)
    Connect(self.mamux.sel,self.msel_next)

    # if (posedge clk & !msel) inst <= mdin
    self.instreg = DLatch(reset = self.reset,en = self.not_msel.o,
      clk = self.clk,d = self.mdin,q = self.inst)

    # *********************
    # *** Register File ***
    # *********************

    # Read Port 2 Address
    self.mnextmux = Mux(sel = self.msel_next,a = self.inst[15:11],
      b = self.inst[25:21])

    self.mmux = Mux(sel = self.msel) # Write Back Register Address
    self.msellatch = DLatch(d = self.inst[25:21],q = self.mmux.b,
      clk = self.clk,reset = self.reset)
    self.regfile = RegisterFile(constantzero = 31,regs = 32,
      x = self.wdmux.o,xsel = self.mmux.o,xwe = self.werf,clk = self.clk,
      asel = self.inst[20:16],bsel = self.mnextmux.o)
    self.wamux = Mux(sel = self.wasel,a = self.inst[25:21],b = 30,o = self.mmux.a)

    self.zbitand = BusNor(i = self.regfile.a,o = self.z) # Compute Z bit

    # ********************************
    # *** Arithmetic and Logic Unit ***
    # ********************************

    # "A" Mux
    self.amux = Mux(a = self.regfile.a,b = self.pcplusfour.q,sel = self.asel)
    self.pcext = SignExt(i = self.inst[15:0]) # PC Sign Ext inputwidth=16
    self.pcextshift = AnonymousBulkConnector(self.pcext.o[29:0],(0,2))
    self.cmux = Mux(a = self.pcext.o,b = self.pcextshift,sel = self.csel)

    # "B" Mux
    self.bmux = Mux(a = self.regfile.b,b = self.cmux.o,sel = self.bsel)

    self.alu_shrl = Shift(i = self.amux.o,s = self.regfile.b[4:0],
      o = self.wdmux.b,dir = self.shiftdir,sext = self.shiftsext) # Shift
    self.alu_adder = AddSub(a = self.amux.o,b = self.bmux.o,
      o = self.wdmux.c,op = self.addsub_op,co = self.co,
      z = self.aluz,v = self.v,n = self.n) # Adder/Subtractor!
    Connect(self.jt,self.alu_adder.o)
    self.alu_selfare = Compare(z = self.aluz,v = self.v,n = self.n,
      op = self.selfare_op,o = self.wdmux.d) # Compare Circuit
    self.alu_or = Or(a = self.amux.o,b = self.bmux.o,o = self.wdmux.e) # Or
    self.alu_and = And(a = self.amux.o,b = self.bmux.o,o = self.wdmux.f) # And
    self.alu_xor = Xor(a = self.amux.o,b = self.bmux.o,o = self.wdmux.g) # Xor

    # *******************************************
    # *** Write Back Mux and Left Over Memory ***
    # *******************************************

    Connect(self.wdmux.sel,self.wdsel)
    Connect(self.mdin,self.wdmux.a)
    Connect(self.wdmux.h,self.pcplusfour.q)
    Connect(self.mdout,self.regfile.b)

    # *********************
    # *** Control Logic ***
    # *********************

    # A lot of this can be simplified with the use of operators
    # since Prof. Ward and I agree operators can be dangerous I’m avoiding them
    # For my thesis I should show both versions!

    # annul Logic
    self.annul_not = Not(i = self.pcsel[2])
    self.annul_reg = DLatch(clk = self.clk,reset = self.reset,
      d = self.annul_not.o,q = self.annul)

    # msel_next = (!reset && !annul) && (mem_next && !msel)
    self.msel_next_not_annul = Not(i = self.annul)
    self.msel_next_andA = And(a = self.not_reset.o,
      b = self.msel_next_not_annul.o)
    self.msel_next_andB = And(a = self.mem_next,b = self.not_msel.o)
    self.msel_next_andC = And(a = self.msel_next_andA.o,
      b = self.msel_next_andB.o,o = self.msel_next)

    # msel <= msel_next @ posedge clk
    self.msel_reg = DLatch(clk = self.clk,reset = self.reset,
      d = self.msel_next,q = self.msel)

    # mwrite_next = (msel_next && mem_next) && (!werf)
    # mwe == mwrite_next
    self.werf_not = Not(i = self.werf)
    self.mwrite_next_andA = And(a = self.msel_next,b = self.mem_next)
    self.mwrite_next_andB = And(a = self.werf_not.o,
      b = self.mwrite_next_andA.o,o = self.mwe)

    # mwrite Logic
    self.mwrite_reg = DLatch(clk = self.clk,reset = self.reset,
      d = self.mwe,q = self.mwrite)

    # werf = msel ? !mwrite : (!annul & !mem_next)
    self.werf_not_mwrite = Not(i = self.mwrite)
    self.werf_not_mem_next = Not(i = self.mem_next)
    self.werf_and = And(a = self.msel_next_not_annul.o,
      b = self.werf_not_mem_next.o)
    self.werf_mux = Mux(a = self.werf_not_mwrite.o,b = self.werf_and.o,
      o = self.werf,sel = self.msel)

    # Modify PC Select
    # pcsel = !reset && ((z && branch) ^^ pcsel_eff);
    self.notreset_three = AnonymousBulkConnector((self.not_reset.o,3)) # Repeat reset
    self.z_three = AnonymousBulkConnector((self.z,3)) # Repeat z
    self.branch_three = AnonymousBulkConnector((self.branch,3)) # Repeat branch
    self.pcsel_andbz = And(a = self.z_three,b = self.branch_three)
    self.pcsel_xorbzpc = Xor(a = self.pcsel_eff,b = self.pcsel_andbz.o)
    self.pcsel_andrbzpc = And(a = self.pcsel_xorbzpc.o,b = self.notreset_three,
                             o = self.pcsel)

    # *******************
    # *** Control ROM ***
    # *******************

    self.not_npc31 = Not(i = self.nextpc[31])
    self.irqandnpc = And(a = self.irq,b = self.not_npc31.o,o = self.interupt)

    # ROM Input
    self.rcin = AnonymousBulkConnector(self.interupt,self.inst[31:26])
    # ROM Output
    self.rcout = AnonymousBulkConnector(self.branch,
      self.mem_next,self.wasel,self.pcsel_eff,self.wdsel,self.shiftdir,
      self.shiftsext,self.asel,self.bsel,self.csel,self.addsub_op,self.selfare_op)
    # Add Extra Wires from this end so it's easy to patch up!

    self.control = ROM(a = self.rcin,d = self.rcout) # ROM

    for i in range(64): # Set Invalid Opcodes
      self.control[i] = 0x04F00

    for i in range(64): # IRQ
      self.control[i+64] = 0x06700

    self.control[0x018] = 0x0A010 # LD
    self.control[0x019] = 0x0A010 # ST
    #omp.control[0x01A] = 0x00000 # * INVALID *
    self.control[0x01B] = 0x01F10 # JMP
    #omp.control[0x01C] = 0x00000 # * INVALID *
    self.control[0x01D] = 0x11F38 # BEQ * JMP IN z=0 CASE *
    self.control[0x01E] = 0x12738 # BNE * NOP IN z=0 CASE *
    self.control[0x01F] = 0x0A038 # LDR

    self.control[0x020] = 0x02200 # ADD
    self.control[0x021] = 0x02204 # SUB
    #omp.control[0x022] = 0x00000 # * INVALID * MUL
    #omp.control[0x023] = 0x00000 # * INVALID * DIV
    self.control[0x024] = 0x02305 # CMPEQ
    self.control[0x025] = 0x02306 # CMPLT
    self.control[0x026] = 0x02307 # CMPLE
    #omp.control[0x027] = 0x00000 # * INVALID *

    self.control[0x028] = 0x02500 # AND
    self.control[0x029] = 0x02400 # OR
    self.control[0x02A] = 0x02600 # XOR
    #omp.control[0x02B] = 0x00000 # * INVALID *
    self.control[0x02C] = 0x02100 # SHL
    self.control[0x02D] = 0x02180 # SHR
    self.control[0x02E] = 0x021C0 # SRA
    #omp.control[0x02F] = 0x00000 # * INVALID *

    self.control[0x030] = 0x02210 # ADDC
    self.control[0x031] = 0x02214 # SUBC
    #omp.control[0x032] = 0x00000 # * INVALID * MULC
    #omp.control[0x033] = 0x00000 # * INVALID * DIVC
    self.control[0x034] = 0x02315 # CMPEQC
    self.control[0x035] = 0x02316 # CMPLTC
    self.control[0x036] = 0x02317 # CMPLEC
    #omp.control[0x037] = 0x00000 # * INVALID *

    self.control[0x038] = 0x02510 # ANDC
    self.control[0x039] = 0x02410 # ORC
    self.control[0x03A] = 0x02610 # XORC
    #omp.control[0x03B] = 0x00000 # * INVALID *
    self.control[0x03C] = 0x02110 # SHLC
    self.control[0x03D] = 0x02190 # SHRC
    self.control[0x03E] = 0x021D0 # SRAC
    #omp.control[0x03F] = 0x00000 # * INVALID *

  def GenerateName(self):
    if (self.Name is None):
      return "Beta2"
```

Listing B.2: Beta2Impl Implementation

Appendix C

The Verilog Two Stage Beta

This section contains Chris Terman's implementation of the Beta Processor in Verilog.

```
module beta2(clk,reset,irq,xadr,ma,mdin,mdout,mwe);
  input clk,reset,irq;
  input [30:0] xadr;
  output [31:0] ma,mdout;
  input [31:0] mdin;
  output mwe;

  // beta2 registers
  reg [31:0] regfile[31:0];
  reg [31:0] npc,pc_inc;
  reg [31:0] inst;
  reg [4:0] rc_save;  // needed for second cycle on LD,LDR

  // internal buses
  wire [31:0] rd1,rd2,wd;
  wire [31:0] a,b,xb,c,addsub,cmp,shift,boole,mult;

  // control signals
  wire wasel,werf,z,asel,bsel,csel;
  wire addsub_op,cmp_lt,cmp_eq,shift_op,shift_sxt,boole_and,boole_or;
  wire wd_addsub,wd_cmp,wd_shift,wd_boole,wd_mult;
  wire msel,msel_next,branch,trap,interrupt;

  // pc
  wire [31:0] npc_inc,npc_next;
  assign npc_inc = npc + 4;
  assign npc_next = reset ? 32'h80000000 :
                    msel ? npc :
                    branch ? {npc[31] & addsub[31],addsub[30:2],2'b00} :
                    trap ? 32'h80000004 :
                    interrupt ? {1'b1,xadr} :
                    {npc[31],npc_inc[30:0]};
  always @ (posedge clk) begin
    npc <= npc_next;   // Logic for msel handled above
    if (!msel) pc_inc <= {npc[31],npc_inc[30:0]};
  end

  // instruction reg
  always @ (posedge clk) if (!msel) inst <= mdin;

  // control logic
  decode ctl(.clk(clk),.reset(reset),.irq(irq & !npc[31]),.z(z),
             .opcode(inst[31:26]),
             .asel(asel),.bsel(bsel),.csel(csel),.wasel(wasel),
             .werf(werf),.msel(msel),.msel_next(msel_next),.mwe(mwe),
             .addsub_op(addsub_op),.cmp_lt(cmp_lt),.cmp_eq(cmp_eq),
             .shift_op(shift_op),.shift_sxt(shift_sxt),
             .boole_and(boole_and),.boole_or(boole_or),
             .wd_addsub(wd_addsub),.wd_cmp(wd_cmp),
             .wd_shift(wd_shift),.wd_boole(wd_boole),.wd_mult(wd_mult),
             .branch(branch),.trap(trap),.interrupt(interrupt));

  // register file
  wire [4:0] ra1,ra2,wa;
  always @ (posedge clk) if (!msel) rc_save <= inst[25:21];
  assign ra1 = inst[20:16];
  assign ra2 = inst[15:11];
  assign wa = msel ? rc_save : wasel ? 5'd30 : inst[25:21];
  assign rd1 = regfile[ra1];         // read port 1
  assign rd2 = regfile[ra2];         // read port 2
  assign mdout = regfile[inst[25:21]];  // read port 3
  always @ (posedge clk)
    if (werf && wa != 31) regfile[wa] <= wd;  // write port

  assign z = ~| rd1;   // used in BEQ/BNE instructions

  // alu
  assign a = asel ? pc_inc : rd1;
  assign b = bsel ? c : rd2;
  assign c = csel ? {{14{inst[15]}},inst[15:0],2'b00} :
                    {{16{inst[15]}},inst[15:0]};

  wire addsub_n,addsub_v,addsub_z;
  assign xb = {32{addsub_op}} ^ b;
  assign addsub = a + xb + addsub_op;
  assign addsub_n = addsub[31];
  assign addsub_v = (addsub[31] & ~a[31] & ~xb[31]) |
                    (~addsub[31] & a[31] & xb[31]);
  assign addsub_z = ~| addsub;

  assign cmp[31:1] = 0;
  assign cmp[0] = (cmp_lt & (addsub_n ^ addsub_v)) | (cmp_eq & addsub_z);

  //mul32 mpy(a,b,mult);

  wire [31:0] shift_right;
  // Verilog >>> operator not synthesized correctly, so do it by hand
  shift_right sr(shift_sxt,a,b[4:0],shift_right);
  assign shift = shift_op ? shift_right : a << b[4:0];

  assign boole = boole_and ? (a & b) : boole_or ? (a | b) : a ^ b;

  // result mux, listed in order of speed (slowest first)
  assign wd = msel ? mdin :
              wd_cmp ? cmp :
              wd_addsub ? addsub :
              //wd_mult ? mult :
              wd_shift ? shift :
              wd_boole ? boole :
              pc_inc;

  // assume synchronous external memory
  assign ma = msel_next ? {npc[31],addsub[30:0]} : npc_next;
endmodule
```

Listing C.1: Beta Main: beta.v

```
module decode(clk,reset,irq,z,opcode,
              asel,bsel,csel,wasel,werf,msel,msel_next,mwe,
              addsub_op,cmp_lt,cmp_eq,
              shift_op,shift_sxt,boole_and,boole_or,
              wd_addsub,wd_cmp,wd_shift,wd_boole,wd_mult,
              branch,trap,interrupt);
  input clk,reset,irq,z;
  input [5:0] opcode;
  output asel,bsel,csel,wasel,werf,msel,msel_next,mwe;
  output addsub_op,shift_op,shift_sxt,cmp_lt,cmp_eq,boole_and,boole_or;
  output wd_addsub,wd_cmp,wd_shift,wd_boole,wd_mult;
  output branch,trap,interrupt;

  reg asel,bsel,csel,wasel,mem_next;
  reg addsub_op,shift_op,shift_sxt,cmp_lt,cmp_eq,boole_and,boole_or;
  reg wd_addsub,wd_cmp,wd_shift,wd_boole,wd_mult;
  reg branch,trap,interrupt;

  // a little bit of state...
  reg annul,msel,mwrite;

  always @ (opcode or z or annul or msel or irq or reset)
  begin
    // initial assignments for all control signals
    asel = 1'hx;
    bsel = 1'hx;
    csel = 1'hx;
    addsub_op = 1'hx;
    shift_op = 1'hx;
    shift_sxt = 1'hx;
    cmp_lt = 1'hx;
    cmp_eq = 1'hx;
    boole_and = 1'hx;
    boole_or = 1'hx;

    wasel = 0;
    mem_next = 0;

    wd_addsub = 0;
    wd_cmp = 0;
    wd_shift = 0;
    wd_boole = 0;
    wd_mult = 0;

    branch = 0;
    trap = 0;
    interrupt = 0;

    if (irq && !reset && !annul && !msel) begin
      interrupt = 1;
      wasel = 1;
    end else casez (opcode)
      6'b011000: begin   // LD
                   asel = 0; bsel = 1; csel = 0;
                   addsub_op = 0;
                   mem_next = 1;
                 end
      6'b011001: begin   // ST
                   asel = 0; bsel = 1; csel = 0;
                   addsub_op = 0;
                   mem_next = 1;
                 end
      6'b011011: begin   // JMP
                   asel = 0; bsel = 1; csel = 0;
                   addsub_op = 0;
                   branch = !annul && !msel;
                 end
      6’b011101: begin   // BEQ
                   asel = 1; bsel = 1; csel = 1;
                   addsub_op = 0;
                   branch = !annul && !msel && z;
                 end
      6’b011110: begin   // BNE
                   asel = 1; bsel = 1; csel = 1;
                   addsub_op = 0;
                   branch = !annul && !msel && ~z;
                 end
      6’b011111: begin   // LDR
                   asel = 1; bsel = 1; csel = 1;
                   addsub_op = 0;
                   mem_next = 1;
                 end
      6’b1?0000: begin   // ADD, ADDC
                   asel = 0; bsel = opcode[4]; csel = 0;
                   addsub_op = 0;
                   wd_addsub = 1;
                 end
      6’b1?0001: begin   // SUB, SUBC
                   asel = 0; bsel = opcode[4]; csel = 0;
                   addsub_op = 1;
                   wd_addsub = 1;
                 end
      //6’b1?0010: begin   // MUL, MULC
      //             asel = 0; bsel = opcode[4]; csel = 0;
      //             wd_mult = 1;
      //           end
      6’b1?0100: begin   // CMPEQ, CMPEQC
                   asel = 0; bsel = opcode[4]; csel = 0;
                   addsub_op = 1;
                   cmp_eq = 1; cmp_lt = 0;
                   wd_cmp = 1;
                 end
      6’b1?0101: begin   // CMPLT, CMPLTC
                   asel = 0; bsel = opcode[4]; csel = 0;
                   addsub_op = 1;
                   cmp_eq = 0; cmp_lt = 1;
                   wd_cmp = 1;
                 end
      6’b1?0110: begin   // CMPLE, CMPLEC
                   asel = 0; bsel = opcode[4]; csel = 0;
                   addsub_op = 1;
                   cmp_eq = 1; cmp_lt = 1;
                   wd_cmp = 1;
                 end
      6’b1?1000: begin   // AND, ANDC
                   asel = 0; bsel = opcode[4]; csel = 0;
                   boole_and = 1; boole_or = 0;
                   wd_boole = 1;
                 end
      6’b1?1001: begin   // OR, ORC
                   asel = 0; bsel = opcode[4]; csel = 0;
                   boole_and = 0; boole_or = 1;
                   wd_boole = 1;
                 end
      6’b1?1010: begin   // XOR, XORC
                   asel = 0; bsel = opcode[4]; csel = 0;
                   boole_and = 0; boole_or = 0;
                   wd_boole = 1;
                 end
      6’b1?1100: begin   // SHL, SHLC
                   asel = 0; bsel = opcode[4]; csel = 0;
                   shift_op = 0;
                   wd_shift = 1;
                 end
      6’b1?1101: begin   // SHR, SHRC
```

```
                   asel = 0; bsel = opcode[4]; csel = 0;
                   shift_op = 1; shift_sxt = 0;
                   wd_shift = 1;
                 end
      6’b1?1110: begin   // SRA, SRAC
                   asel = 0; bsel = opcode[4]; csel = 0;
                   shift_op = 1; shift_sxt = 1;
                   wd_shift = 1;
                 end
      default:   begin   // illegal opcode
                   trap = !annul && !msel; wasel = 1;
                 end
    endcase
  end

  // state
  wire msel_next = !reset && !annul && mem_next && !msel;
  wire mwrite_next = msel_next && opcode==6’b011001;

  always @ (posedge clk)
  begin
    annul <= !reset && (trap || branch || interrupt);
    msel <= msel_next;
    mwrite <= mwrite_next;
  end

  assign mwe = mwrite_next;   // assume synchronous memory
  assign werf = msel ? !mwrite : (!annul & !mem_next);
endmodule
```

Listing C.2: Beta Decoder Logic: decode.v

```
module shift_right(sxt,a,b,shift_right);
  input sxt;
  input [31:0] a;
  input [4:0] b;
  output [31:0] shift_right;

  wire [31:0] w,x,y,z;
  wire sin;

  assign sin = sxt & a[31];
  assign w = b[0] ? {sin,a[31:1]} : a;
  assign x = b[1] ? {{2{sin}},w[31:2]} : w;
  assign y = b[2] ? {{4{sin}},x[31:4]} : x;
  assign z = b[3] ? {{8{sin}},y[31:8]} : y;
  assign shift_right = b[4] ? {{16{sin}},z[31:16]} : z;
endmodule
```

Listing C.3: Beta Right Shift: shift_right.v

Bibliography

[1] Peter Bellows and Brad Hutchings. JHDL - an HDL for reconfigurable systems. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 175–184, Los Alamitos, CA, 1998. IEEE Computer Society Press.

[2] System C. Systemc:welcome. http://www.systemc.org.

[3] Celoxica. Celoxica - the technology leader in c based electronic design and synthesis. http://www.celoxica.com.

[4] Per Haglund, Oskar Mencer, Wayne Luk, and Benjamin Tai. Hardware design with a scripting language. In *Field-Programmable Logic and Applications*, pages 1040–1043, 2003.

[5] IEEE. Eda industry working groups. http://www.eda.org/.

[6] IEEE. Ieee verilog standardization group. http://www.verilog.com/ieeeverilog.html.

[7] Jingzhao Ou and Viktor K. Prasanna. Pygen: A matlab/simulink based tool for synthesizing parameterized and energy efficient designs using fpgas. *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*, pages 47–56, 2004.

[8] Python. Python programming language – official website. http://www.python.org.