SIMULATION OF CLOCKED DIGITAL LOGIC IN HARDWARE C (HWC)

by

DILEN SANAT GOVIN

A Thesis Submitted to The Honors College

In Partial Fulfillment of the Bachelors degree
With Honors in

Computer Science

THE UNIVERSITY OF ARIZONA

M A Y 2 0 2 0

Approved by:

Russell Lewis
Department of Computer Science

SECTIONS

ABSTRACT

Hardware C (hereafter HWC) is an original hardware description language designed to imitate the syntax of *C* family of languages. HWC is composed of two parts, a compiler and a simulator. The HWC compiler parses HWC files (*.hwc) and compiles the code into a *wiring diagram*, which describe logic gates and the wires between them. The simulator parses the *wiring diagram* files (*.wire) and builds a runnable representation of that part list by connecting the wires and simulating a clock cycle with inputs. In this project, the simulator was redesigned from *C* into Python and the components of the wiring diagram were fully flushed out to support the simulator. This paper discusses the goal of HWC, recaps previous work on HWC, discusses the additions to the compiler, and explains the various stages of simulation HWC supports. After explaining HWC simulation in its entirety, the practical and pedagogical applications of HWC simulation are discussed. It is believed that HWC's *C*-styled design could help students understand the CPU and various other hardware component functions.

INTRODUCTION

Hardware C is an original hardware description language for prototyping digital logic. It is designed to imitate the syntax of *C* family of languages. It aims to be a useful pedagogical tool by allowing students to simulate digital logic at a variety of levels, from simple sum-of-products up to simple CPUs. HWC allows a programmer to represent the simple and more complex logical circuits inside a computer by allowing the programmer to use abstract versions of wires and components to create functional hardware specs. With this description, HWC can be classified as a “Hardware Description Language” (hereafter HDL). HDL’s are a category of programming language used to describe hardware schematics in depth. However, HWC differs from other HDLs in that its focus is on exploration and pedagogy rather than implementation, by offering coders the ability to experiment with circuit designs and understand how said circuits communicate with each other without bogging said coders down with electrical or timing details.

As mention earlier, Hardware C has two main components: a compiler and a simulator. The compiler transforms HWC code into a *wiring diagram* which can be thought of as the “hardware description” that HWC models. It assigns bit addresses to all the memory cells, logical operators, connections between wires, assertions, constants, inputs and outputs within a modeled circuit. The format of the *wiring diagram* is set up to be readable to a human who has knowledge of HWC, but it is tedious to examine in detail. The simulator bridges this gap by parsing the *wiring diagram* and enumerating through its elements to explain what occurs in the hardware at each clock cycle. This allows the user to examine how hardware runs, connecting HWC code to actual hardware.

The simulator for Hardware C operates under the premise that all statements execute simultaneously. That is, each execution of HWC code represents a single clock tick. Using this approach, HWC is able to offer a clear clock tick by clock tick representation of hardware. To perform cross clock tick logic the user will have to make use of memory, in which the memory of the previous clock cycle feeds into the current clock cycle which would be used to affect the immediate next clock cycle. This design approach is to simplify the complexities of hardware simulation for the user.

This paper will begin by examining the language of HWC in detail but focusing mainly on the changes made during this year of work. This will mainly focus on the syntax that composes the language and how it has evolved. The compiler side of HWC will be touched on briefly. Next, the paper will go into great detail regarding the simulator side of HWC, focusing its discussion on the changes to the wiring diagram and explanation for its operation. Lastly, potential applications of HWC will be discussed regarding both pedagogical and industry use cases.

HWC LANGUAGE

Similar to other programming languages, Hardware C has a specific collection of symbols, keywords, and patterns. These are tools that the programmer uses when constructing HWC code. From a high level, HWC organizes itself around single-bit wires and the various operations done with those or upon those wires. In HWC, a bit is a single primitive variable type.

```
bit a;
bit[32] bit_array_size_32;
```

Figure 1: declaration of a single bit and a bit array of size 32

Bits serve as the building blocks for two larger representations of hardware components in HWC, *parts* and *plugtypes*.

Parts can be roughly thought of as objects in an object orientated programming language. To draw a comparison to computer architecture, an example of *part* would include an arithmetic logic units (ALUs) or a multiplexer (MUX). *Parts* compose a majority of HWC code, as this serves as the container for logic that connects the various wire in a circuit and instructions for the circuit to write things to memory.

```
part example_part {
     private bit a;
}
```

Figure 2: example part declaration with single private bit

Parts are designed to handle input and output. This is done by declaring the desired input and output bits with the keyword *public*. These signify that these bits or bit arrays will be accessible to parts that use this part or for use in the simulator. There is no current limit to the amounts of inputs and outputs a part can have upon declarations. Inputs into parts are bits that are declared as public but never assigned a value. Outputs from *parts* are bits that are declared as public and are bits where the part itself writes a value.

```
public bit in;
public bit out;
```

Figure 3: two public bit declarations showing in and out are declared the same

Parts can also have internal wires that are only accessible in the current part and inaccessible to the outside circuit. These are declared using the keyword *private*. *Parts* can also define sub-components, essentially other *parts,* that aid the *part* in its function. These can be declared by using the keyword *subpart*. This declaration is also implicitly *private*, so they are only accessible in the scope of the current part.

```
part example_part {
     private bit a;
     public bit out;
     subpart subpart_decl subpart_name;
}
```

Figure 4: example part declaration with private, public, and subpart components

Plugtypes are the second building block to hardware representation in HWC. In actual hardware *plugtypes* are more akin to a group of wires run together. They are a form of programming abstraction in HWC similar to *structs* in C. They are simply named collection of bits placed together for their related usage. They are simply for declaration purposes and similar to *structs* in C, they contain no internal logic. They are a way to simplify the internal logic of *parts*. *Plugtypes* can contain other *plugtypes* because plugtypes lack internal logic. *Parts* cannot be declared in any *plugtypes* because they are internal logic. Due to the lack of internal logic, *plugtypes* cannot make use of private bits and all components declared in a *plugtype* are *public*.

```
plugtype example_plugtype {
     bit a;
     plugtype plugtype_name;
}
```

Figure 5: example plugtype declaration

While *plugs* and *plugtypes* are the core building blocks of Hardware C, there are other fundamental elements to HWC. One of these is memory. Memory cells are essential to HWC because they allow the output of one clock cycle to carry over into the next cycle. When declaring a memory cell in HWC it can be bits or *plugtypes. Parts* cannot be a memory cell because they represent internal logic which is more than a collection of bits. Memory are a form of data storage between clock cycles similar to a latch in hardware. Memory cells have two specific value stores: a read and write. When a memory cell is read, the value is read from the read value store because this is the value written to the memory cell in the previous clock cycle.

This side of memory can never be changed during a clock cycle. The only time this is changed is at the end of a clock cycle based on the write value. When writing to memory, the value is written to the write side of the memory cell. This value will be read in the next clock cycle. On the first clock cycle, the read side of memory has a value of zero, while the write side is set to be a value of floating. The write side gets this floating value at the beginning of each clock cycle.

```
memory(bit[4]) mem_example;
```

Figure 6: example of 4 bit memory

A wire that has the floating value is a wire whose value is currently unknown. In HWC, wires can exist in three states: low voltage (0), high voltage (1), and floating. The floating state is reserved for wires that are currently not being driven to high or low voltage. All wires exist in a floating state at the start of a clock cycle. During the clock cycle, it is likely a value of 0 or 1 will be driven to the wire, but if not, the wire remains as floating. This allows users the ability to drive values to a wire without causing a short circuit due to the wire not having a proper value. This also allows HWC to implement combinatorial logic. This is something that is in fact modeled in hardware as wires that are floating are essentially not set. As such, this floating value exists in HWC but cannot be explicitly set.

Like memory, connections are another essential part of HWC. There are three types of connections in HWC: directed connections (=) , undirected connections (<−>), and conditional connections. Directed connections set the its left side with the values on the right side. The value from the right side of the equality sign passes to the left side. Undirected connections work differently to direct connections in that they allow either side of the operator to influence the other. Setting a value on either side of the connection will attempt to set the value on the other side of the connection. This can lead to possible short circuits at runtime as wire values can only be set once. The last connection type is a conditional connection. This functionality is given by using an if-statement. The if-statement expression is structured just like it is in the C-family of language. Conditional connections only allow connections if the certain condition is met. This condition, a boolean expression, is defined in the header of the if-statement block. The reason if-statements were used is to serve as a representation of gates in hardware that output an electrical charge based on a series of input charges. These connections apply to not only wires but *plugtypes* of the same type as well as they are abstract collections of wires.

```
bit a, b, c, d;
a = b;
c <-> d;
```

Figure 7: examples of directed connection a to b and undirected connection c to d

Since HWC has not sense of time during a clock cycle, all operations are running in parallel. This means that the ordering of expression or operations has no effect on the operation of said code. Take the following example:

```
bit a, b, c;
a = b;
b = c;
```

Figure 8: example of two directed connections to show parallel operations

In the C family of languages, the variable a would have no relation to the variable c because the order of the operations. This is because the variable a would be assigned a copy of variable b's value then the variable b would be assigned would be assigned variable c's value. However, in HWC the value of a is the value of c. To make up for this lack of temporal dependency, HWC uses a sort of logical dependency. HWC knows that the variable b is dependent on the variable c and the variable a is dependent on the variable b. HWC knows that the variable a is dependent on the variable c, so if the value of c gets set somewhere, the value will propagate through variable b to variable a.

```
Comparisons                        Array Slicing
     expr == expr                       expr[int..int]
     expr != expr                  Booleans
Logical Negation                        true
     !expr                              false
AND Operation                      Parenthetical
     expr & expr                        (expr)
OR Operation                       Bit-shifting
     expr | expr                        expr << expr
XOR Operation                           expr >> expr
     expr ^ expr
Array Indexing
     expr[int]
```

Figure 9: examples of expressions and operations in HWC

Connections are one form of expression in HWC that is supported. Other expressions are familiar to those familiar with C-family languages. In hardware, expressions can be represented by gates or separate components. Common expressions are AND, OR, NOT, or XOR operations.

HWC allows the use of integer constants in expressions, such as comparison and directed connections. While normally, expressions would require arrays of bits, HWC automatically

```
bit a;
a = 1;
```

Figure 10: example of a bit having a constant value

converts the integer to its binary representation during compile time. The size of the binary representation is determined by its value. This allows HWC to access the information as 1-bit wires.

Another central component of HWC are loops. There is a singular loop in HWC, the for-loop. Unlike its counterpart in a C-family, the for-loop is not used as form of time keeping or changing over time because HWC has no concept of time. Instead for-loops are used for enumeration through bit or multiple comparisons. It is handy for building duplicates of the same basic structure. For example, it would be used if a user wanted to set all 32 bits to one value. In HWC, for-loops require an iterator variable and two constant variables that represent the bound of the for-loop. The lower bound is inclusive and the upper bound is exclusive similar to C-family languages.

```
for(i; 0..4)
     array_a[i] = array_b[i];
```

Figure 11: example of for loop in HWC that sets the bits of array_a to the bits of array_b

During the creation of the *wiring diagram*¸ for-loops are essentially unrolled into and enumerated collection of statements linked to the loop's bounds.

Similar to the C-family of languages, a function named *main* is required, but only in

```
The above loop is unrolled as:

array_a[0] = array_b[0];
array_a[1] = array_b[1];
array_a[2] = array_b[2];
array_a[3] = array_b[3];
```

Figure 12: example of figure 10 unrolled

HWC's case, a *part* called main is required. Much like in C with the main function being the actual program, *main* represents the actual component being built. Since *main* is declared as a part, the other *parts* in the code serve as *subparts* to be utilized in *main*. If a *plugtype* or *part* is not used directly in *main* or by another *part* that main uses, the compiler ignores it. The *wiring diagram* would reflect this by not having any of the wires or components from those unused parts. It would only contain components used directly in *main*.

While working on the simulator the core functionality and purpose of HWC has not changed. Its syntax has also remained the same as a result. As such, this is just quick explanation of HWC purpose and more so the current planned features.

HWC EXAMPLES

While short examples were given above when discussing the language syntax, lets dive into some full HWC programs. Consider the simple HWC program below:

```
part main {
     public bit a;
     public bit b;
     public bit out;

     out = a & b;
}
```

Figure 13: an example of a simple part that takes two inputs and returns the logical and result of them

The above program in HWC takes two user inputs in bits “a” and “b.” The code also has one output bit with the “out” labeled bit. The part simply passes the result of the and operation between the bits labeled “a” and “b” to the bit labeled “out.”

Now let’s consider a more complex HWC program. The following program shows how HWC handles a lot more of the language’s features:

```
part main {
     private memory(bit) state;
     public bit out;

     if (state == 1)
          out = state;

     state = !state;
}
```

Figure 14: an example of a more complex part that returns out if 1 is stored in memory and then stores the negation back in memory

The above code is an example of a more complex HWC component. This component checks the value stored in memory and returns an out if the stored value is true. The code then stores the negation of the stored value for the next clock cycle.

HWC COMPILER

The compiler for Hardware C follows the standard compiler structure (see figure 1). The first step to compilation is lexical analysis. In this step, a lexer using flex scans the “*.hwc” file and passes the tokens it reads to the parser. From there the parsers constructs a parse tree according to the tokens. The next step is semantic analysis, in which the semantic analyzer undergoes four passes through the parse tree. The overall goal of the semantic analyzer is to build the semantic tree based on the HWC code. The first pass of the semantic analyzer builds the semantic tree and creates *namescope* objects. The second pass of the semantic analyzer does name look-ups for declarations and expressions. The third pass of the semantic analyzer calculates the size of every part and plugtype using the declaration pointers from the previous phase. The last pass of the semantic analyzer verifies that all indices used in array expressions are valid. After these four phases of semantic analysis is complete, a wiring diagram is produced. For a more in-depth description of the compiler, refer to Robert Jackson Kunio Lindsay’s thesis regarding HWC compilation.

While the majority of work for this thesis was spent on the simulator, there were two features added to the compiler. These were equality comparison in the form of EQ, and NEQ. While not exactly the same as EQ and NEQ, NOT gates were also added to the compiler. In addition to these, asserts were also implemented and registered in the compiler.

HWC SIMULATOR

At the point of writing this thesis, the simulator currently supports un updated version of the *wiring diagram* and has been redesigned in Python. The original simulator was written in C and could take a *wiring diagram* as input and execute the diagram's code over a series of clock cycles. During the time this thesis took place, a majority of the time was spent reworking the simulator in Python as to have an easier time creating a graphical user interface to use. This section will go into depth about key concepts and changes to the simulator by discussing the old simulator, the wiring diagram, the new simulator, and the future plans for this simulator.

The Old Simulator

The original design for the simulator is as follows. The HWC simulator is designed to allow the execution, analysis, and debugging of HWC models. It reads the compiled .hwc files and any input files, and then executes the model as directed by the user.

The following description is copied from Jackson Lindsay's thesis, and summarizes the compiler as it stood at the beginning of this project:

> "As the simulator runs, it will log any errors that it encounters. These encountered errors fall into two broad categories: fatal and non-fatal. Fatal errors indicate serious problems which either make the simulation unable to be run, or which might result in damage to the chip in the real world. An easy example of this error would be an event that causes a short circuit. For example, if two elements drove different values into the same fan-in (a fan-in element is a component that takes inputs and decides the output based on the values driven into input) element.[1] Non-fatal errors indicate flawed logic,

[1] While this works for our example, it is a sort of gotcha in real hardware

which is not fatal to the simulation, but which must be resolved. Examples: two elements driving the same value to the same fan-in elements; storing an indeterminate value to a memory element.

The simulator will also produce warnings. These are issue which often indicate logic errors, but which sometimes are harmless and can be ignored. For example, connecting a floating line to the input of a logic gate, and producing an indeterminate value from the output.

The initial version of the simulator will be very simplistic. Eventually, the simulator might be expanded to include features such as running multiple chips, emulating external hardware elements, and perhaps other features. The original simulator aimed to solve the problems laid up in the above specification. The original simulator can be found under the hwcSim_raw directory in the main repository. This directory contains the main driver code and make files. Test code and scripts are found in this directory as well. There is another directory name hwcSim, in the repository, that holds a significant amount of the object and header files for use in the simulator.

The old simulator was written in *C* and can read the old version of the *wiring diagram.* A description of the old simulator "Taking a wiring diagram as input, the simulator executes the diagram's code over a series of clock cycles. To do this, the simulator uses a model of the wires called the bitspace, which can be thought of as all of the wiring connections in a breadboard. In application, the bitspace models the current state of all wires plus dependencies for wires that have not been written to. At time of writing, it is able to simulate very simple circuits. In the future, it will allow the user to write the initial values of all memory bits, changing the initial state of the simulated component."

Wiring Diagram

While a high-level description of the *wiring diagram* has been given above, it has changed significantly from the previous iteration. The *wiring diagram* is the output of the compiler and essentially describes how the wires connect in the circuit. It is an itemized list of all the base types that would are required to simulate the HWC code. It is essentially a list of parts. The wiring diagram separates the components of the circuit into its own separate sections with a total bit count at the top of the document. The previous iteration of the *wiring diagram* only has sections for connections, logic operators, memory, and assert. Currently, the *wiring diagram* supports connections, logic operators, memory, asserts, input, output, and constants. Each of these sections have specific translations to their respective component in the section.

Let's dive further into the wiring diagram, by examining its components closely. The first part of the wiring diagram is considered to be the header of the *wiring diagram*.

```
# HWC Wiring Diagram
version: 1.0

debug=""
```

Figure 15: header for a .wire file

The header has three parts which are purely for placeholding purposes. This header may be subject to change in the future, but it is purely debugging purposes. The next line is where the simulator actually looks on the *wiring diagram*.

```
bits 4
```

Figure 16: example of bits line with a wire file that has 4 bits

This line signifies the total space to allocate for bit space for the simulator. This line exists in both the old simulator's version and the updated simulator's version of the *wiring diagram*. The way bits are counted have changed as well. Previously, each wire connection counted as one bit, allowing logic gates and connections to exist on the same bit. Now, bit addresses are unique, so the progression and value propagation cane be seen more clearly. This does in turn make the *wiring diagram* more complex as these unique bit addresses require more connection objects to make a connection between components. Ultimately, this was done to aid in the creation of the bit dictionary that would be the internal data structure used to model bit spaces for the simulator, which will be described later in this section.

```
# HWC Wiring Diagram
version: 1.0

debug=""

bits 6

memory count 0

logic count 1
  logic AND size 1 a 3 b 4 out 5

connection count 3
  connection               size 1 to 3 from 0
  connection               size 1 to 4 from 1
  connection               size 1 to 2 from 5

assert count 0

constant count 0

i/o count 3
  input size 1 in 0
  input size 1 in 1
  output size 1 out 2
```

Figure 17: example wiring diagram for the program in figure 13

The first section in the diagram is the memory. This section contains all the information the simulator needs to create all the memory cells in the circuit. The number next to the label "memory count" gives the user the total count of memory cells in the circuit. The label is then followed by a list of descriptions equal to the count given in the label. An example memory line looks like this:

memory size # read # write #

The description is preceded by the keyword "memory" signifying this description is for a memory component. The next keyword is "size" which is followed by a number. This number describes how big the memory cell is. So, a size of one, means the memory cell can only store one bit of information. A size of eight, means the memory cell can store up to eight bits of information. The next section is the read and write addressing. These are the addresses that the other components use to interact with a memory cell. Components that are reading the memory cell's current value will connect to the read bits. Components that want to change the value stored in memory for the next clock cycle will connect to the write bits. This will be explained in further detail later.

The second section of the *wiring diagram* gives the descriptions for the logic operators. This section starts with the label "logic count" with the total number of logic operands following the label. This is similar to the memory count section and will be a part of every following section. The following descriptions for the logic gates take the following format:

logic TYPE size # a # b # out #

The description is preceded by the keyword "logic" signifying this describes a logic gate of some kind. This is followed by the type of logic. The simulator currently supports the following logic types: AND, OR, NOT, XOR, EQ, NEQ, GT, GE, LT, and LE. The next part of the description is the size of the logic gate, which encodes the size of the inputs as the output's size is determined by the operator type. The "a" and "b" refer to the addresses of the inputs into the logic gate. While most logic gates have two inputs, the logic type NOT only has one input. For this case we leave the "b" keyword out of the line. This is special case is handled by the simulator once is parses the NOT keyword. The last part of the description contains the address for the output. This is where the output goes once all inputs are given to the logic gate.

The next section of the *wiring diagram* gives the descriptions for the connections. These are essentially the wiring between most components. Connections serve as the roadways that connect the various components of the circuitry. Similar to every section, it starts with its label of "connection count" followed number of connection descriptions. The following descriptions take the following format:

connection size # to # from #

The description is preceded by the keyword "connection" signifying this describes a connection object. This is followed by the size keyword which tells the simulator the size of the to and from parts of the connection. The to and from keywords are followed by their bit addresses in the system. This is the section that will probably contain the most descriptions as connections are the main way values are moved in the hardware.

The next section of the *wiring diagram* gives the descriptions for the assertions in the code. These work identically to assertions in a normal programming context. Similar to previous sections, assert descriptions are preceded by the label "assert count" followed by number of assert descriptions. The following descriptions take the following format:

assert val #

The description starts with the keyword “assert” signifying this describes an assert object. This leads to the next keyword “val” which is then followed by a number. That number is the address of the bit the assert will look at when evaluating.

The next section of the *wiring diagram* gives the descriptions for input and output channels of the circuit. These allow for outputs and inputs the user can interact with. Similar to previous sections, input-output descriptions are sectioned off under the label “i/o count” and the number of following descriptions. Those descriptions have the following format:

(input | output) size # (in | out) #

The descriptions for these parts begin with either “input” or “output.” Depending on this word, they simulator knows which bits to treat like an input and bits to treat as an output. The next part of this description is the size. The keyword “size” is then followed by a number signifying how big inputs and outputs can be. Then either they keyword “in” or “out” which is linked to the beginning of the description with “input” or “output.” This is then followed by a number which refers to the starting point of the bit address.

The next section of the *wiring diagram* gives the descriptions for the constants. Constants are essentially bit addresses that map to a certain value. These values are always passed are the beginning of a clock cycle. The constant descriptions can be found in the *wiring diagram* under the label “constants count” followed by the number of succeeding descriptions. The descriptions take the following format:

constant size # from # val #

The descriptions for constants begin with the keyword “constant.” This signifies that the description is one for a constant. The next keyword that follows is “size” which is then followed by a number describing the total size the constant occupies in the bit space. The next keyword is “from” which a then followed by a number. This number is the starting bit of where the constant is stored in the bit space. The next keyword is “val” and then followed by a number. This number is the actual value the constant stores and passes when used.

These sections outline the specific addresses and parts that will make up the bit dictionary. The bit dictionary is how the simulator for HWC stores all the data about bit addresses which will be discussed later. After the constant section and its following descriptions are parsed, the simulator has reached the end of its need for the *wiring diagram*.

Simulator Design

As stated earlier, the majority of time working on Hardware C was spent on the simulator. The redesigned version of the simulator can be found in the directory hwcSim_py. This version of the simulator is written in Python and aims to expand on the abilities of the previous simulator. This simulator when given a *wiring diagram*, parses the wiring diagram into its correct components, allocates space for each bit in a bit dictionary, checks all connections are valid, and then drives the program from input.

The first step of the simulator is parsing the *wiring diagram*. The components of the wiring diagram have been outlined in detail above. The simulator parses the wiring diagram by its sections: memory, logic gates, connections, asserts, input-outputs, and constants. The simulator uses the bit information found in the descriptions. Depending on the category of the description, the part created changes. These parts are modeled by python classes.

The ultimate goal of parsing is to create *reader* and *writer keys* that are used in the bit dictionary. These keys serve as the unique key that map the information and the connections in the bit dictionary. They can be thought of as addresses in the bit space model. In python, these keys are represented in a tuple that contains two pieces of information. The first index of the tuple gives the starting bit. The second index of the tuple gives the size of the respective component. These two keys are essentially what the outcome of the parsing step is.

The first of the sections to be parsed are the memory components. The memory class represents a singular memory unit in the circuit. This unit has two places to store information, the read and write internal stores. The information from the read gets read during the current clock cycle. The information stored in the write will overwrite the information in the read store at the end of the clock cycle.

The next of these classes is the logic operations. From the description, the type of logic operand is parsed first. This determines what logic Python object will be built. Depending on the logic type, this unit has up to three-bit addresses associated with it. These addresses are pulled from the sections of the description labeled *a*, *b,* and *out* in the *wiring diagram*. The following number corresponds to the starting bit of the address. This is where wires will connect to either take values out or put values into the logic gate.

The next of these classes is the connections. The simulator parses these descriptions into three parts: *size, to,* and *from*. The simulator parses these three labels and their respective values and stores them in local variables. The simulator then uses the *from* and *size* values to create the *reader key* tuple. The simulator then uses the *to* and *size* values to create the *writer key* tuple.

The next class that is parse are the inputs and outputs. The simulator parses these descriptions based on the keyword at the beginning of the description. If the keyword reads *input*, then the simulator handles this as a description as an input and builds an input object from it. Due to the way inputs work in the HWC simulator, inputs only require a *writer key*. The *writer key* is only required because the inputs can only write to other bits. The *writer key* tuple consists of the *in* bit and the *size* value. If the keyword reads *output*, then the simulator handles this as a description as an output and builds a corresponding output object from it. Due to the way output work in the HWC simulator, outputs only require a *reader key*. The *reader key* is only required because the outputs won't write to other bits at this point in time. The *reader key* tuple consists of the *out* bit and the *size* value.

The next class to be parsed are asserts. Asserts don't contain much information description as it simply checks if the value at a bit address is true or one. The simulator parses the assert description to get the number following the *val* keyword. This number is the bit that the assert checks to see if it is on. As such it does not need a *reader* or *writer key* as it won't be entered into the bit dictionary.

The last class to be parsed from the *wiring diagram* are the constants. Constants work similar to inputs but instead of requiring user input it contains a set value. The constant description is parsed into three parts: *size, from,* and *val.* Similar to inputs the constants only have *writer keys* because it will only write to other bits and never read. The *writer key* for constants is made using the *from* bit and the *size* value. To store the value, the simulator creates a constant object and keeps the value in that constant object's value field.

After the constants have been parsed, the parsing stage has come to an end. At this point the simulator has all the information needed to build the bit dictionary and eventually evaluate the lambda propagation. The bit dictionary is how the simulator is able to run and where all the important value connections are stored. It is essentially telling how the components connect to each other. As such, this step is the most important for the simulator to get right, because a misaligned bit dictionary will lead to inaccurate circuit simulations and possible short circuits.

The code for the bit dictionary can be found in the *wiring* directory in the *hwcSim_py* project folder. The bit dictionary is a custom class that is an extension of the built in Python dictionary object. The bit dictionary maps bit address keys, the *reader* and *writer* keys, created in the parsing stage to objects called *BD_Value*. The *BD_Value* stands for bit dictionary value and is an object with two fields, readers and writers. The readers field is a Python list object that contains the components that will receive the value from the current bit address. The writers field is different as it just holds a boolean. In HWC, a bit address can only be written to once per clock cycle. Due to this, it is only necessary to store a boolean value that signifies if another object writes to the current address. This is one of the ways HWC detects short circuits on the writer side.

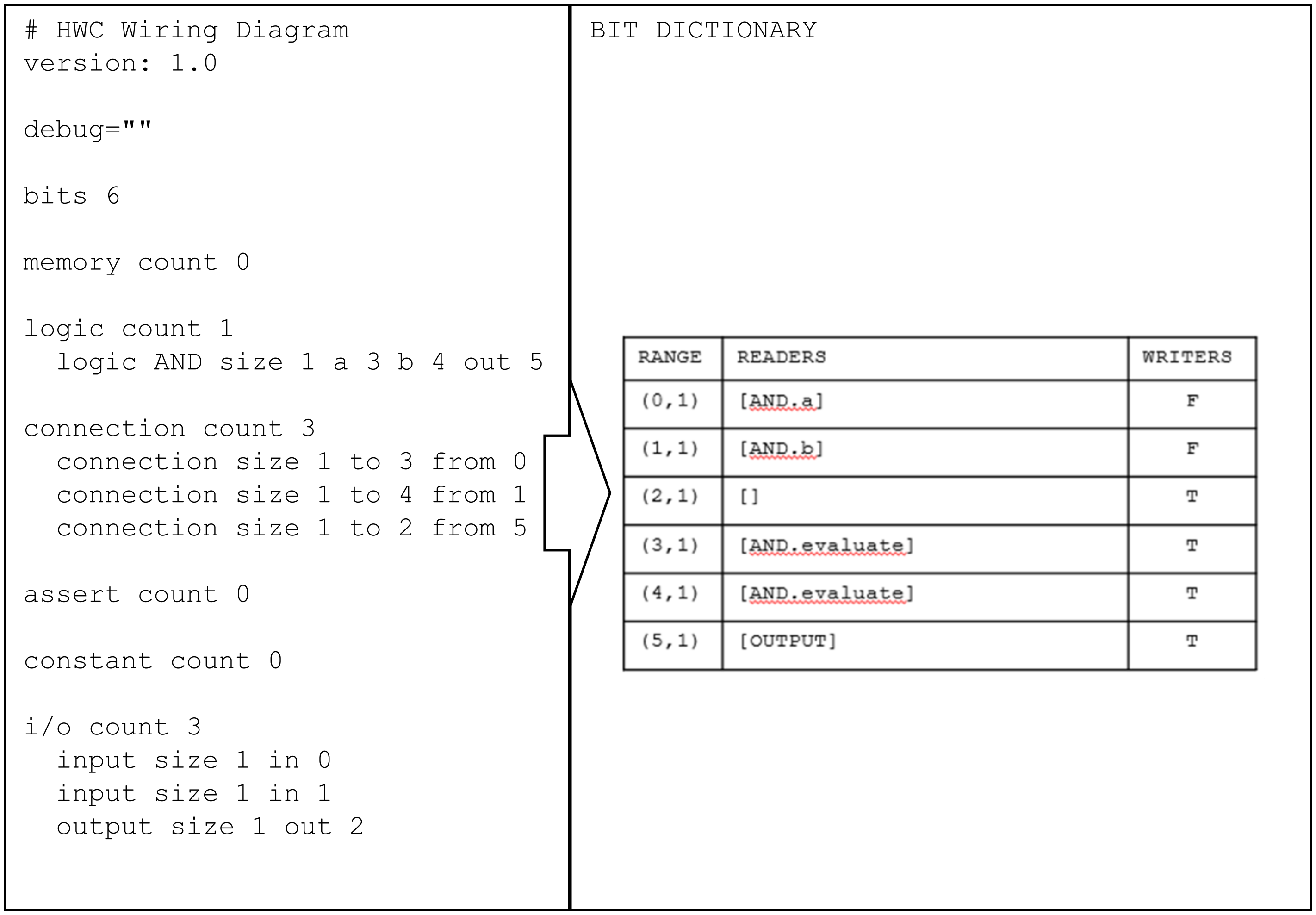
Once a component's key is found out after parsing, it is added to the bit dictionary. Due to the way components works in HWC simulation, it is not necessary to add both the reader and writer keys. As such there are three main ways: a component gets added to the bit dictionary: adding a new key, adding by reader key, and adding by writer key. It is safe to say that almost all components are added and tracked in the bit dictionary by a combination of these. The following explanation will be in order of easiest to most complicated when it comes to adding to the bit dictionary.

The easiest components to track in the bit dictionary are the inputs, outputs, and constants. As explained earlier inputs and constants work fairly similar to one another, and as such they are added to the bit dictionary identically. Both descriptions in the *wiring diagram* only account for a writer key, as inputs and constants are writers to other bits. As such their writers will always remain false, since they won't be wrote to or passed a value. Their list of readers will populate as the other components get added to the bit dictionary. Outputs work essentially the same except they act as readers. When the output component is parsed, its description only gives the reader key. As such, the output component will show up on the reader list of another component. Outputs can be thought of as connections directly to the simulator and as such it is instantiated with a reader in its reader list. This reader lambda is actually a print statement at the moment to show the value passed to this output. In addition to the bit dictionary, the simulator also tracks the inputs, outputs, and constants in separate list structures that are used when running the created circuit. This concludes all the work the simulator does when adding these components to the bit dictionary.

The next component that is fundamental to understand are connections and their relation to the bit dictionary. Connections are essentially the linking component in the bit dictionary. As such, when they are parsed, they have two keys, one for the reader and one for the writer. This is because a connection links to separate bit ranges so both of these ranges need to be documented in the dictionary. Thus, the simulator adds the writer key first into the dictionary, and then the reader key. It is created this way because the writer key will be in the reader key's list of readers and a lambda will have to be created for it. This is all that is required when adding a new connection into the bit dictionary.

The next component that is added to the bit dictionary are all the logic gates and operands. This works similar to connections except they have two reader keys for the respective input channels and one writer key for the output channel. The only logic operand that is an exception to this and works differently is the not gate as it only has one input channel. The writer key, the output channel, is the first thing added to the bit dictionary because this is how HWC recognizes the logic gate. Subsequently the simulator adds the writer key to a secondary dictionary used to track all the logic gate objects in the code. The reader keys are then added into the bit dictionary and the respective readers list is filled out. When a logic gate is added to a reader list it is actually a lambda checking if the logic gate can be evaluated. Once the readers and writer addresses have been added, the bit dictionary has all the information regarding the logic gates and can move on to the next component.

The last component to be added to the bit dictionary is memory. Memory works a little differently from the other components because it essentially has two places to store bits. This is due to the memory having a latch mechanic that switches the value from the write to the read side. The bit dictionary models this by having a reader and writer key. While the memory object has one address externally, it has two specific addresses that map directly to the read and write stores. When adding the memory to the bit dictionary, it is added like the previous components with its writer key being added and then its reader key being added. That concludes the creation of the bit dictionary.



```
# HWC Wiring Diagram
version: 1.0

debug=""

bits 6

memory count 0

logic count 1
  logic AND size 1 a 3 b 4 out 5

connection count 3
  connection size 1 to 3 from 0
  connection size 1 to 4 from 1
  connection size 1 to 2 from 5

assert count 0

constant count 0

i/o count 3
  input size 1 in 0
  input size 1 in 1
  output size 1 out 2
```

BIT DICTIONARY

RANGE	READERS	WRITERS
(0,1)	[AND.a]	F
(1,1)	[AND.b]	F
(2,1)	[]	T
(3,1)	[AND.evaluate]	T
(4,1)	[AND.evaluate]	T
(5,1)	[OUTPUT]	T

Figure 18: shows the wiring diagram from figure 13 after its bit dictionary has been created

Once the bit dictionary has been created the simulator can move on to the next step of simulation: checking connections are valid. This step is mainly for checking overlapping bit addresses and making sure there is no overlaps or writes that would cause a short circuit based on how the bit dictionary is built. Once this step is completed the simulator can move on to actually running the simulation.

The last step of the simulator is actually running the circuit outlined in the bit dictionary. HWC is driven off inputs so logically this is where the simulator starts. The simulator looks to see if the system has inputs and subsequently receives user inputs if required. Once the inputs have been received the simulator stores the values and drives the constants. The constants are inputs except they always have the same value and always drive their value. Once the constants have been delivered the simulator moves on to memory. If there is a value stored in memory, it will be driven out to its respective bit. The last step is to drive the values from these three components to create the circuit.

HWC uses a system of lambdas and driving values as soon as they are known. This effectively creates a depth first algorithm in which a value goes as far as it can in a system before it reaches a point where another value is needed. Once this point is reached, the simulator moves on to the next reader on the queue and evaluates that. Once all branches and reader lists have been looped through and output will be produced and written. This effectively ends the explanation of the simulator and how it works.

HWC CONCLUSION

Hardware C is a hardware description language that was designed to prototype digital logic by modeling each clock cycle as a function of combinatorial logic. Due the familiar C-styled syntax and direct statement to implementation relationships, HWC can be used to explain hardware to students who are familiar with C. The language has two main abstractions that model hardware. The first is parts which are the components found in a circuit and plugtypes which act as connections between parts. HWC also makes use of memory and allows the user the ability to evolve circuits over multiple clock cycles. The HWC is compiled into a *wiring diagram*, which is a detailed description of the components of the circuit. These descriptions are what allow the simulator to create the main part. This wiring diagram is then fed into the simulator and the simulator reports back a given output based on the form of input specified in the wiring diagram. For this thesis, the simulator to run the compiled HWC code was rewritten in Python and the structure of the wiring diagram was changed to accommodate this. The simulator has four steps when it comes to simulating a HWC circuit: parsing the wiring diagram, building the bit dictionary, checking overlapping addresses, and driving the values from input to produce an output. As development continues for HWC, the features the simulator will be able to handle, and features of the language will only increase and become more comprehensive.

HWC still has more features that hope to be implemented and current features that hope to be expanded on. The main goal of HWC is to serve as a pedagogical tool for students learning hardware. HWC has always been envisioned with a GUI that would better display what is happening in the HWC code. This version would show the text of the user’s code and would highlight different characters and lines as values propagated through the circuit. An additional implantation of this includes building the circuit model using parts and wires similar to two dimensional representations of circuits. Essentially, it would be a form of a circuit diagram. In addition to this, more work on the compiler side needs to happen to accommodate the updates to the wiring diagram. With these updates, it could further the pedagogical abilities of HWC by allowing the user to debug and develop in an environment easy to understand. It also offers a better representation on how the values propagate within HWC like they do in real life. With these ideas and additions in mind, HWC will be able to growth further and become a more complete tool for students and potentially industry in the future.

CODE

The complete source of the HWC compiler and simulator can be found at the links below. This is the repository that holds a majority of the progress completed during the duration of the thesis. The code can be found in the directory labeled “hwcSim_py” and the link to this repository can be found here:

https://github.com/dilengovin/HWC

The repository reserved for future development of HWC can be found in this link:

https://github.com/russ-lewis/HWC