HARDWARE C (HWC)

By

ROBERT JACKSON KUNIO LINDSAY

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 1 9

Approved by:

Russell Lewis
Department of Computer Science

SECTIONS

ABSTRACT

Hardware C (HWC) is an original hardware description language designed to imitate the syntax of the *C* family of languages. When HWC code is compiled, a parts list called the *wiring diagram* is created, which describes the wires and the relations between wires that comprise the user's code. In this project, the parser and the compiler for HWC were created, including a semantic analyzer to produce said *wiring diagram*. This paper discusses the grammar and syntax of HWC, provides example HWC code, and explains the various stages of compilation HWC undergoes. Having explained HWC in its entirety, the practical and pedagogical applications of HWC are discussed. It is believed that HWC's *C*-styled design will help students understand how the CPU and various other hardware components function.

INTRODUCTION

Hardware C (hereafter HWC) is a programming language designed for prototyping digital logic. HWC allows a programmer to represent the logical circuits inside a computer by offering a mapping from abstract versions of wires and components into feasible hardware specs. As such, HWC is a "Hardware Description Language" (hereafter HDL), a type of programming language used to describe hardware schematics. However, HWC differs from other HDLs in that its focus is on exploration and pedagogy rather than implementation. HWC's design allows coders to experiment with circuit designs and understand how said circuits communicate with each other, without bogging said coders down with electrical or timing details.

Hardware C has two main components: A compiler and a simulator. The compiler transforms HWC code into something called a *wiring diagram*. The *wiring diagram* could be thought of as the "hardware description" that HWC models. It enumerates all of the memory cells, logical operators, and connections between wires within a circuit. While the *wiring diagram* is human-readable, it is, by nature, tedious to examine in great detail. This is where the simulator comes in. By taking the *wiring diagram* and enumerating through its elements, the simulator explains what occurs in the hardware at each clock cycle. This allows a user to examine how their hardware runs, bridging the gap between HWC code and actual hardware.

When running, the simulator operates under the guiding paradigms of HWC. In HWC, all statements are modeled as executing simultaneously. As HWC is meant to model circuits, a user is not allowed to write to a wire more than once per clock cycle (in actual hardware, this would result in a short circuit). In the simulator, each execution of HWC code represents a single clock-tick. In this way, HWC differentiates itself from other hardware descriptions languages that model hardware through continuous motion. Instead, HWC models the operations of all circuits using combinatorial logic time steps. This means that the resulting state of each time step is directly correlated to the memory at the start of the time step, and no previous state. This is in contrast with other HDLs, which model the circuit as a continuous operation. For example, the HDL *VHDL* allows a user to "wait" some amount of time before sending a signal across a wire, meaning the user can wait until a certain clock cycle to send the signal. In contrast, HWC has no concept of time outside of a singular clock cycle and refuses to perform logic that crosses between clock cycles. In order for a user’s circuit to evolve over time, the memory created from the previous clock cycle feeds into the current cycle, which then writes memory to affect the immediate next clock cycle. All of this is done to simplify the complexities of hardware simulation for the user.

As the name Hardware C implies, the language was specifically designed to leverage the knowledge about the language *C* that a programmer has. This ties into HWC’s pedagogical purpose: Students with an understanding of the syntax of *C* should be able to carry over this understanding when reading and writing HWC code. For example, a directed connection in HWC transfers the value of one wire into another. This type of statement utilizes the equality symbol (=) and transfers the value on the right of the symbol to the value on the left, in order to take advantage of a programmer’s pre-existing knowledge of how the symbol is used in *C*. In another example, HWC’s *plugtypes* are structured similarly syntactically to *C*’s *structs*; both are a list of the various types that comprise the *plugtype* or *struct*. By creating similarities like these, HWC molds itself towards use as an intuitive tool of experimentation for burgeoning computer scientists.

This paper will begin by examining the language of HWC in greater detail. This includes the grammar and syntax that composes the language and the quirks and oddities associated with the language. Examples and explanations of HWC code will then be provided, in hopes of illustrating HWC’s intuitive design as well as its practical applications. Next, the way HWC operates as a compiler will be elaborated upon, by illustrating each step of the conversion from user code to *wiring diagram* and explaining the syntax of said *wiring diagram*. Lastly, the applications of HWC will be discussed, by examining both potential use cases and how HWC has already been utilized in a classroom environment.

HWC LANGUAGE

As a programming language, Hardware C has its own collection of symbols, keywords, and paradigms that must be used when constructing HWC code. The most relevant of these concepts will be discussed below, along with a short snippet of the related syntax. In its most general form, everything in HWC is based around single-bit wires and various operations upon those wires. As such, a bit is the single **primitive** variable type of HWC. However, HWC has two important, overriding metaphors for how it represents hardware components: *parts* and *plugtypes*.

```
bit name_of_bit;
bit[4] array_of_four_bits;
```

Figure 1: Two simple declarations of bits, corresponding to a single bit and an array of four bits, respectively.

parts form the bulk of HWC code, as they contain the logic that connects the various wires of a circuit and instructions for the circuit to write things to memory. If one is familiar with computer architecture, an ALU or a MUX would be an example of a *part*. While *parts* will likely be designed with an input and output in mind, HWC does not require an explicit declaration of what a *part*'s input and output is. This is done both to avoid burdening the programmer and to reinforce HWC's philosophy that its code is simply wires and operations upon those wires. *parts* can have internal wires that are inaccessible to the outside circuit. These are declared using the keyword *private*. Other wires that can be connected to and from outside the *part* must be specified as *public*. *parts* can also contain sub-components (ie, other *parts*) that help said *part* function. These are declared with the keyword *subpart* and they are implicitly *private*. This is done to enforce good encapsulation within the user's code.

```
part example_part_declaration
{
      public bit a;
      private bit b;
      subpart ex_subpart_decl example_name;
}
```

Figure 2: An example part declaration that contains a subpart declaration. Note that "ex_subpart_decl" would be declared as a part somewhere else in the file.

plugtypes, on the other hand, are simply named collections of bits that are grouped together because they have related usage. They are a programming abstraction, where each bit is treated as an individual wire, and are akin to a *struct* in *C* or an equivalent data structure. Like *structs* in *C*, *plugtypes* have no internal logic of their own; rather, they are used to simplify and clarify the internal logic of *parts*. Following from this lack of internal logic, all bits in a *plugtype* are implicitly *public* because *plugtypes* cannot make use of *private* bits. Additionally, *plugtypes* can contain other *plugtypes* but not any *parts*, because *plugtypes* lack internal logic. In hardware, a *plugtype* represents an organization of wires that run in parallel.

```
plugtype example_plugtype_declaration
{
      bit a;
      other_plugtype name;
}
```

Figure 3: An example plugtype declaration. Note that "other_plugtype" would be declared as a plugtype somewhere else in the file. Also note the lack of public and private as keywords within the plugtype.

In HWC, wires exist in one of three states. The first two, 0 and 1, represent low and high voltage respectively. The third state, *floating*, is reserved for wires that are currently not being driven to either of the other two states. *floating* is the initial state of all wires at the beginning of a clock cycle; if a wire is still *floating* at the end of a step of the simulation, the wire was not driven to either 0 or 1 by any other wire during that clock cycle. However, this means that one can safely drive the wire to a value without causing the component to short-circuit, since the wire currently does not have a "legitimate" value. In a functioning clock cycle, all *floating* wires are either ignored or are driven to a value of 0 or 1 by memory cells or other wires. As such, *floating* is both a transient and a final state. In a hardware implementation, it would not be true to say that a wire is ever *floating*, because a wire can either have an electrical charge or not. However, HWC uses *floating* as part of its implementation of combinatorial logic. Note that it is illegal to write a value of *floating* to a memory cell.

Memory cells are another fundamental element of HWC, because they allow the results of one clock cycle to carry over into the next. In hardware, they represent data storage between clock cycles, similar to a latch (also called flip-flop). Memory cells can be bits or *plugtypes*, except they have special semantics: When they are read from, the value that was written to the memory cell during the **previous** clock cycle is read. When they are written to, the value that will be read **next** clock cycle is changed. As such, memory cells can be modeled as having a *read* side and a *write* side. A memory cell being on the left side of a connection statement means that the *write* side is being written to. Conversely, a memory cell on the right of a connection means that the *read* side is being read from. The *read* side of a memory cell can never be changed during a clock cycle; they are modified at the end of a clock cycle based on the *write* side of the memory cell. At the start of the **first** clock cycle, the *read* side of all memory bits have a value of zero. Conversely, at the start of **every** clock cycle, the *write* side of all memory bits have a value of *floating*. In syntax, the keyword *memory* is used to modify the *plugtype* that the memory cell is an instance of. Note that a *part* cannot be a memory cell, since a *part* represents more than a collection of bits.

```
memory(bit[2]) memory_2bit_example;
```

Figure 4: An example of a memory cell declaration. Note how it acts as a wrapper around the bit declaration syntax.

There are two types of connections in HWC: Directed (`=`) and undirected (`<->`). Directed connections set the bit(s) on the left side of the operator to the value of the bit(s) on the right side. As mentioned before, this is done to imitate the use of the equality symbol in *C*-style languages. In contrast, HWC's undirected connections allow the bit(s) on both sides of the operator to influence the other. As such, a value set on either side of the operator will result in an attempt to set the other. This can be dangerous since, as stated before, a wire's value can only be set once per clock cycle. In hardware, the concept of an undirected connection is utilized infrequently, but HWC allows for them because no physical wire is directed electrically. However, in practical usage, most wires are used as if they were directed, so our directed connections are merely an abstraction of the gates that connect and disconnect wires in hardware. Since *plugtypes* are simply collections of bits, both directed and undirected connections are allowed between *plugtypes* of the same type. In the future iterations, HWC will allow for *plugtypes* of different types to be connected via a cast, since *plugtypes* are just collections of wires that can be connected in arbitrary ways.

```
wireA = wireB;
wireC <-> wireD;
```

Figure 5: An example of a directed and undirected connection, respectively. wireB's value would be assigned to wireA, while either wireC or wireD's value could be assigned to the other.

Connections can further be modified to be conditional connection. This is done using the if-statement syntax from the *C*-family of languages. As its name suggest, conditional connections allow a connection to occur if and only if some condition is met. Here, if-statements serve as another abstraction of the gates in hardware that give a wire its electrical charge based on a series of input charges.

```
if(wire1 == wire2) {
      wire3 = wire4;
}
```

Figure 6: An example of an if-statement with a directed connection. Note that, for a single statement within the condition, curly braces are not required but can be used.

The singular type of loop in HWC is the for-loop. It requires an iterator variable, and then two constant or compile-time variables that represents the bounds of the for-loop. Constant or compile-time variables must be used because the compiler needs to know the bounds of the loop during compilation, in order to correctly create the *wiring diagram*. This is because HWC has no concept of time, and so for-loops cannot be used for temporal iteration. Instead, during the creation of the *wiring diagram*, for-loops are unrolled into enumerated collections of statements based on the bounds of the loop. Said unrolling is done by copying and pasting the statements for every number within the bounds of the loop and replacing the iteration variable with the current number (see example below). In HWC's for-loop syntax, the lower bound is inclusive and the upper bound is exclusive.

```
for(iter; 0..4)
      arrayA[iter] = arrayB[iter];

Is unrolled in the wiring diagram as:

arrayA[0] = arrayB[0];
arrayA[1] = arrayB[1];
arrayA[2] = arrayB[2];
arrayA[3] = arrayB[3];
```

Figure 7: An example of for-loop syntax and how a for-loop is unrolled during HWC *compilation. Note how iter is replaced by numbers within wiring diagram.*

Integer constants can be used in expressions which would normally require arrays of bits, such as comparison and directed connections. When an integer is used that way, HWC automatically converts the integer to its binary representation during compilation, with the size of the representation implicitly determined by the context. This adheres to HWC's paradigm that any component can be represented as a series of operations on 1-bit wires.

```
if(4_bit_wire == 15)

The above user code is converted into the wiring diagram as:

if(4_bit_wire == 0b1111) // 4 bits
```

Figure 8: An example of how integer constants are converted into a binary representation during HWC *compilation.*

Like in the *C* family of languages, we require a *part* called *main* in HWC code. In relation to hardware, *main* represents the actual component being built. Other *parts* in the code serve as *subparts* utilized by *main*. If a *plugtype* or *part* is not used directly or indirectly in *main*, then the compiler simply ignores them. As such, the *wiring diagram* for a file is defined as all the wires and components necessary to implement *main*. However, unlike *C*-styled languages, simulation of HWC code does not begin at *main*, nor any *part* in particular.

As mentioned previously, HWC is based upon combinatorial logic and HWC has no sense of time during a clock cycle because all operations run in parallel. This means that the ordering of operations within a chunk of HWC code has no bearing on the operation of said code. Take the list of directed connections:

```
x = y;
y = z;
```

A programmer familiar with the *C* family of languages would assert that x has no relation to z. Since the order of statements matters in *C*-style languages, x would be assigned y's value and then y assigned z's value. However, in HWC, x's value is now intrinsically connected to z's value. This is because HWC emulates how hardware has no temporal ordering in its operation; all components in hardware are running simultaneously at all times. Instead of a sense of time, HWC has a sense of logical dependency. Suppose the value of z is set somewhere. HWC would now know the value of y, since z is connected to y. The value would then propagate to x, which was dependent on y. In this sense, HWC is abstracting away the real-time of operation of circuits by propagating values within discrete clock cycles.

What follows is a short list of possible expressions in HWC. Note that HWC has a variety of conventional operators for performing calculations on and comparisons between integer constants, but these are illegal for use during run-time and have been omitted for simplicity.

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

Figure 9: Possible expressions in HWC*. Note that we use "expr" as shorthand for "expression".*

HWC EXAMPLES

```
part main
{
	public bit input;
	public bit output;
	output = input;
}
```

Figure 10: Simple HWC *code.*

Above is sample Hardware C code that the authors call "simple_part". Those with knowledge of *C*-styled languages are invited to pause and examine the code above. This is hoped to illustrate the intuitive design of HWC.

The above code is perhaps the most trivial instance of HWC code. Its singular *part*, *main*, has two *public* wires, *input* and *output*, each of a single bit. When a value is given to *input*, that same value is transferred to *output*.

```
part simple_memory
{
	private memory(bit) state;
	public bit out;
	out = state;
	state = !state;
}
```

Figure 11: An example of HWC *code with a memory cell.*

Above is a slightly more complex example of HWC code. The *part* "simple_memory" has a *public* bit *out* and a *private* memory cell *state*.

Recall that memory cells in HWC are composed of two separate sides: A read side and a write side. In the statement "*out = state*", *state* is on the right side of the directed connection, so the value of the read side of *state* is written to *out*. In the statement "*state = !state*", both the write and the read side of *state* are being used. In plain English, this line of code simply inverts the value of *state* by the end of every clock cycle. Below is a small table that enumerates the values of the three wires in "simple_memory" at the end of each clock cycle.

Clock cycle	out	state (read bit)	state (write bit)
1	0	0	1
2	1	1	0
3	0	0	1
4	1	1	0

Figure 12: The values of all wires in simple_memory at the end of the first four clock cycles.

Note that since *state* is a *private* bit, no other *part* can write a value to it. On the other hand, another *part* containing an instance of "simple_memory" might attempt to set the *out* plug of its "simple_memory" instance, since *out* is a public *plugtype*. However, this would short-circuit the component, because the value of *out* would be written to twice in a single clock cycle. As such, the *part* "simply_memory" is of little practical use; it only helps determine whether the current clock-cycle is odd or even-numbered. A more useful example of a *part* follows below.

```
part MUX_4in_bit32
{
	public bitBus[4] in;
	public bitBus    out;
	public bit[2] control;

	for (i; 0..4)
		if (control == i)
			out = in[i];
}

plugtype bitBus
{
	bit[16] upper;
	bit[16] lower;
}
```

Figure 13: An example of how a MUX could be written in HWC.

The previous page contains the authors' favorite example of HWC code. As its name suggests, "MUX_4in_bit32" is a multiplexer (MUX) that chooses one output between four 32-bit inputs based on two bits of control. This example illustrates how HWC can be used to accurately represent components of already-existing physical hardware, which bolsters HWC's goals of being both an educational tool and a method of prototyping new hardware designs.

This example contains a *plugtype* which we call *bitBus*. It contains 32 bits in total, half of which it titles *upper* and the other half titled *lower*. *bitBus* acts as the 32-bit input and output type for the MUX. In a normal array in HWC, bits with a greater index are more "upper" when determining things like how the bit array converts to an int. So, we could easily replace *bitBus* with "bit[32]" in "MUX_4in_bit32", and it would work exactly the same. However, we added *bitBus* to demonstrate the potential of *plugtypes* to assist in readability via named elements.

Now, we examine the statements inside "MUX_4in_bit32". We hope the declarations to be self-evident in their purpose, so we turn our attention towards the for-loop and its conditional connection (if-statement). Recall that, upon compilation, for-loops are unrolled into separate statements. Additionally, recall that integer constants are rewritten into an array of bits upon compilation. So, the for-loop within "MUX_4in_bit32" could be rewritten as:

```
if (control == 0b00)
       out = in[0];
if (control == 0b01)
       out = in[1];
if (control == 0b10)
       out = in[2];
if (control == 0b11)
       out = in[3];
```

Figure 14: An example of how the for-loop in MUX_4in_bit32 is unrolled during compilation.

As such, "MUX_4in_bit32" simply checks what value the controls bits have and, depending on the value, connects the corresponding element of *in* to *out*. This is where the idea of a "conditional connection" comes from; while there are four possible connections for *out* in a given clock cycle, only one will be selected based on a condition. Note that *out* can be connected to one of the elements of *in* without needing to specify either of the *upper* or *lower* bit arrays of *bitBus*. This is because, in HWC, a user may connect two plugs if they have the same type. HWC simply interprets this as connecting each of the individual bits, one by one, as a measure of convenience for coders.

HWC AS A COMPILER

The creation of Hardware C utilized a standard compiler pipeline. A lexer scans an ".hwc" file and passes the parse tokens it reads from the file to a parser. Said parser builds a parse tree based on these tokens, which it then passes to a semantic analyzer. The semantic analyzer does four passes through the parse tree, converting it into a semantic tree by performing a more thorough analysis of the code and checking for user errors along the way. The semantic tree is organized in such a way that it is easily converted into a *wiring diagram*, which is the output of the compiler. The HWC simulator can then read through the *wiring diagram* in order to illustrate the component's operation over several clock cycles. We will examine each step of the compilation process in greater detail below. Aside from the lexer and parser phases, which were written in their respective languages, all of the compiler's code was written in *C*.

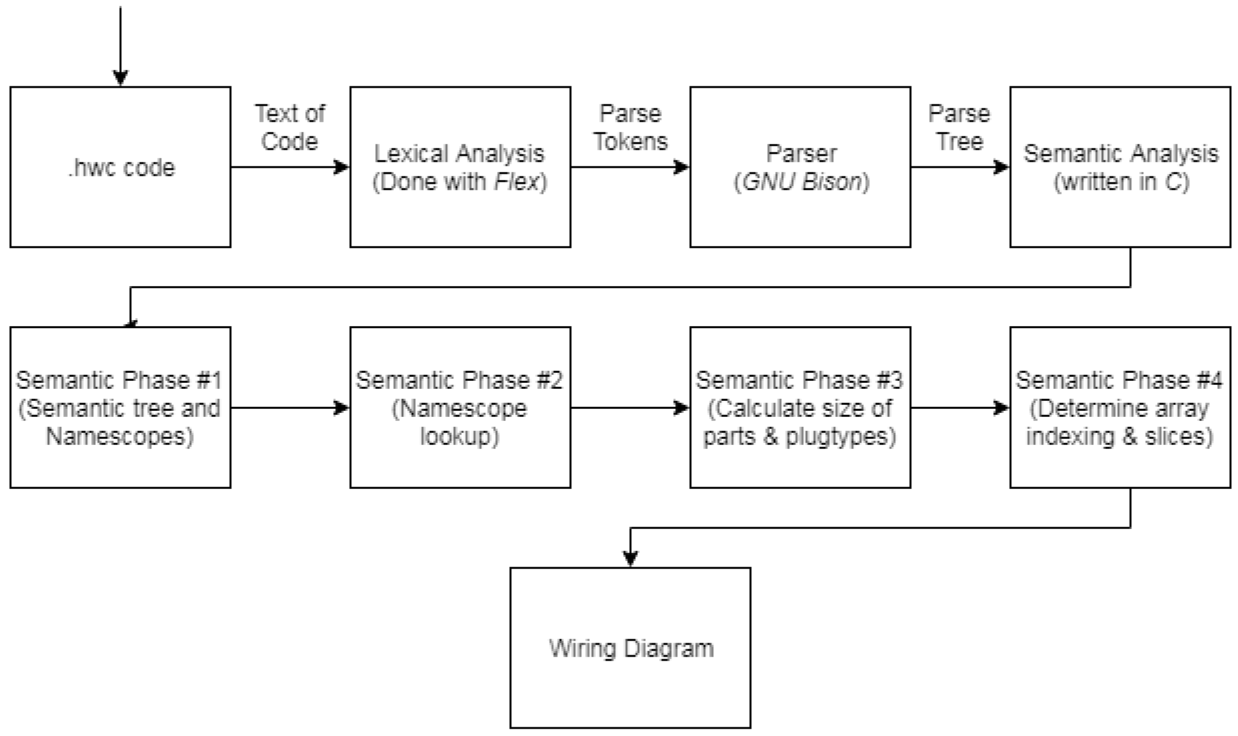
.hwc code
Text of Code
Lexical Analysis (Done with *Flex*)
Parse Tokens
Parser (*GNU Bison*)
Parse Tree
Semantic Analysis (written in C)
Semantic Phase #1 (Semantic tree and Namescopes)
Semantic Phase #2 (Namescope lookup)
Semantic Phase #3 (Calculate size of parts & plugtypes)
Semantic Phase #4 (Determine array indexing & slices)
Wiring Diagram

Figure 15: A diagram of the compilation steps of HWC*, beginning with the user's* HWC *code and ending with the wiring diagram.*

The first step of our compiler is lexical analysis, which we performed using the lexer software *flex*. During compilation, a lexer reads the text of the given code and generates parse tokens based on what it reads. The HWC lexer looks for three major categories of tokens: Identifiers and keywords, numeric constants, and operators. The rules for what qualifies as an identifier or keyword in HWC and what qualifies as a numeric constant follow the same structure as *C*-styled identifiers, keywords, and numeric constants. As for operators, the lexer looks for the set of symbols covered at the end of the **HWC LANGUAGE** section. Additionally, the HWC lexer ignores block comments using the */* */* syntax and one-line comments using the *//* syntax.

Next, the HWC lexer passes the parse tokens it generates to the parser, which uses HWC's grammar to allocate memory for the parse tree while populating it. The parser used for HWC was *GNU Bison*. The grammar of HWC can be understood through three overarching concepts: Statements, declarations, and expressions. Declarations are, as their name implies, used to declare the existence of something, like a *part* or a *plugtype*. Declarations are a subset of statements, but they have a special purpose as an anchor to which expressions and other statements can refer back. Note that only declaration statements are valid within *plugtypes*, while any statement type is valid within a *part*. Statements are instructions for the compiler on how to generate the *wiring diagram*. Expressions are used to construct statements, and can be simplified into combinations of certain keywords, identifiers, and operators. Below is a more technical description of HWC's grammar that has been simplified in order to give the reader a clearer idea of how the grammar operates.

```
file_decl:
  part_decl
  plugtype_decl

part_decl:
  "part" IDENT '{' stmts '}'

stmts:
  '{' stmts '}'
  "subpart" decl_stmts
  "public" decl_stmts
  "private" decl_stmts
  expr '=' expr ';'
  "for" '(' IDENT ';' expr ".." expr ')' stmts
  "if" '(' expr ')' stmts
  "if" '(' expr ')' stmts "else" stmts
  "assert" '(' expr ')' ';'

plugtype_decl:
  "plugtype" IDENT '{' decl_stmts '}'

decl_stmts:
  expr decl_list ';'
  "memory" '(' expr ')' decl_list ';'

decl_list
  IDENT
  IDENT ',' decl_list
```

Figure 16: A simplified version of HWC*'s grammar.*

As stated, we simplified this grammar for the purpose of clarity. For example, the normal grammar has contingencies for optional or empty declarations or statements, in places like *parts* or *plugtypes*. The formal grammar also allows for unit tests, which we plan to add in future iterations of the language.

```
expr:
  expr2
  expr2 "==" expr2
  expr2 "!=" expr2
  expr2 "<" expr2
  expr2 ">" expr2
  expr2 "<=" expr2
  expr2 ">=" expr2

expr2:
  expr3
  expr2 '&' expr2
  expr2 "&&" expr2
  expr2 '|' expr2
  expr2 "||" expr2
  expr2 '^' expr2
  expr2 '+' expr2
  expr2 '-' expr2
  expr2 '*' expr2
  expr2 '/' expr2
  expr2 '%' expr2

expr3:
  expr4
  '!' expr3
  '~' expr3

expr4:
  expr5
  expr4 '[' expr2 ']'
  expr4 '[' expr2 ".." expr2 ']'
  expr4 '[' ".." expr2 ']'
  expr4 '[' expr2 ".." ']'

expr5:
  expr6
  expr5 '.' IDENT

expr6:
  expr7
  '(' expr ')'

expr7:
  IDENT
  NUM
  "true"
  "false"
  "bit"
```

Figure 17: A list of the various ways HWC *can interpret an expression. Note how the hierarchy of expressions corresponds to their precedence.*

The parser, having created its parse tree, passes the root of the tree to the semantic analysis, so it can generate the *wiring diagram*. The semantic analysis has four phases, each of which involves a pass through the parse tree or the resulting semantic tree. These phases serve to both check the user's code for errors and to extract data needed for the *wiring diagram*. In order, the phases (1) build *namescope* objects and convert the parse tree into a tree that is more easily searched by the semantic analyzer, (2) use said *namescope* objects to verify the validity of identifiers, (3) calculate the size of every type and the offset of every sub-type, and (4) check the indices used in array expressions. After the fourth phase for all of the *parts* and *plugtypes* required to build *main* is completed, a *wiring diagram* is produced.

The first semantic phase builds a new version of the parse tree while creating *namescope* objects. The *namescope* for a set of statements describes the current range of valid names for said statements. *namescopes* form a tree-like structure, where the valid names for a *namescope* include the valid names for the *namescope*'s parent, but not the *namescope*'s children. Each declaration found during the first semantic phase is added to the current *namescope*. Whenever a new list of statements begins, a new *namescope* is made with the old *namescope* as its parent. We resume using the parent of a *namescope* whenever a list of statements end. We generate an error when a name is declared that already exists in the *namescope*. For example, if a *plugtype* named *bitBus* has already been declared in the file, we prevent the user from declaring another *plugtype* or *part* named *bitBus*. While creating *namescopes*, we convert the parse tree into a semantic tree, which has all the same information as the parse tree but accounts for *namescopes* and is easier for the semantic analyzer to search through.

The second semantic phase utilizes the *namescope* objects created in the first phase to do name-lookups for declarations and expressions. For example, the declaration statement "*public* *bitBus currentBus*" does a name-lookup in the corresponding *namescope* for the name *bitBus*. If the name *bitBus* is valid, a pointer to the declaration of *bitBus* is added to the statement's object. Likewise, the if-statement "*if(currentBus == 32)*" checks its *namescope* for the name *currentBus* to determine if the if-statement is valid or not, and a pointer back to the declaration statement is created if so. In both of these cases, if the *namescope* says the name is invalid, the compiler generates an error for the user. For declaration of bits, such as "*public bit[2] control*" from "MUX_4in_bit32", a dummy *plugtype* called "BitType" is used so the semantic analyzer views "bit" as a valid name with a corresponding declaration.

The third semantic phase calculates the size of every *part* and *plugtype* using the declaration pointers found in the previous phase. To do this, the semantic analyzer iterates through each *part* and *plugtype* declaration and calculates its size based on the other declarations within. This makes the process recursive, since the size of one type usually relies on the size of another, so flags are used to mark which types have been, have not been, or are in the process of being measured. Care is taken by this phase to avoid recursive definitions (like when type A relies on the size of type B and type B relies on the size of type A) and an error is given to the user if this occurs. The "base case" for sizes is the *bit*, which has a size of 1. Once we know the size of all the sub-types in a type, we are able to fill in the size of the type, which helps us fill in the size of other types. In calculating the size of a type, we also fill in offset information for each type's sub-types, where the offset of a sub-type describes its related range of wires in its enclosing type. This moves the compiler towards its goal of modeling everything in HWC as a series of connected 1-bit wires.

The fourth, and final, semantic phase verifies that all indices used in array expressions are valid. In the third phase, the exact size of types and arrays of said types were calculated. Knowing their size, indexing and slice operations into arrays can now be validated. When the fourth phase finds an invalid index or slice expression, an error is reported to the user.

A *wiring diagram* is produced after the four phases of the semantic analyzer have been completed. The *wiring diagram* is an itemized list of all the base types that would be required to implement the HWC code in physical hardware. In that way, it resembles a parts list. The *wiring diagram* contains the total number of bits in the component as a whole, as well as the total count of memory cells, logic operations, connections between wires, and assertions. Beneath each count is a list of descriptions for the various elements that contributed to the count. For each memory cell, the size of the cell and the bit index of their read and write side is listed. For each logical operation, the type of logic (AND, OR, NOT, XOR), the bit index of its input(s) and output, and the overall size is listed. For each connection, the size and the bit index of the start and end of the connection is listed. For each assert, a single bit is listed. This bit will be checked by the simulator at the end of every clock cycle to ensure that it has a value of 1, and an error will be reported to the user otherwise. This leads to a *wiring diagram* with the following basic structure (# indicates where a number ought to go):

```
version: 1.0

bits #

memory count #
	memory size # read # write #

logic count #
	logic TYPE size # a # b # out #

connection count #
	connection size # to # from #

assert count #
	assert #
```

Figure 18: A template for the wiring diagram currently produced by HWC.

Note that, for the logic operator NOT, the above "b #" which represents the second input is not included because NOT has only one input. This will be illustrated shortly.

Earlier, a *part* named "simple_memory" was presented in the **HWC EXAMPLES** section. In order to provide a concrete example of a *wiring diagram*, below is both the HWC code for said *part* and the resulting *wiring diagram*.

```
part simple_memory
{
	private memory(bit) state;

	public bit out;

	out = state;
	state = !state;

}
```

Figure 19: The simple_memory part.

```
# HWC Wiring diagram
version: 1.0

bits 4

memory count 1
	memory size 1 read 0 write 1

logic count 1
	logic NOT size 1 a 0 out 3

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

assert count 0
```

Figure 20: The wiring diagram that is currently produced from the part simple_memory. Note how "b #" is not used in the logic statement because of the logic operator NOT, as mentioned above.

The advisor for this thesis worked on the simulator for HWC code. 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.

HWC APPLICATIONS

Hardware C has two intended applications: A tool for prototyping innovative digital logic and an aid in helping students learn about how circuits in hardware work. As a tool for prototyping, a hardware designer can use HWC's unique model of circuits as products of combinatorial logic to divide a component’s operation into discreet time steps. Then, the one-to-one relation of HWC code to a hardware implementation can be used to construct a feasible version of the component, with the *wiring diagram* acting as a list of parts. As a pedagogical aid, the *C*-based design and the ability of HWC to represent actual circuits aims to transfer a computer science student's knowledge of programming into knowledge of how hardware operates.

HWC was originally designed to be a tool for prototyping theoretical circuit designs. This purpose influenced the paradigms of HWC; for example, modeling each clock cycle as a discreet time step abstracted away the continuous nature of the CPU, allowing for easier design. Additionally, the base syntax of HWC contains only components with a singular implementation, like an AND gate or an equality check. Any component with multiple feasible implementations must be constructed from the ground up (or imported from already existing HWC code). This allows for a one-to-one relation between HWC code and implementation in hardware, meaning the *wiring diagram* of HWC can easily be used to guide the construction of a physical component.

Over time, HWC began to be viewed as a tool to help computer science students learn about computer hardware. By being able to directly convert a piece of hardware into a *C*-styled representation, it was reasoned that students would be able to more easily understand how said hardware operates.

HWC has already been demonstrated in classes in order to gauge its current effectiveness. The purpose and general syntax of HWC was explained to students, who were then asked to implement already-existing hardware components in HWC with minimal assistance. The goal of this exercise was to measure the intuitive nature of HWC, and whether students could determine the relation between HWC and hardware design. Below is the author's implementation of a half-adder followed by a selection of half-adder implementations that groups of students came up with.

```
part halfAdder_author
{
	public bit[2] in;
	public bit    out;
	public bit    carry;

	out   = in[0] ^ in[1]
	carry = in[0] & in[1]
}
```

Figure 21: The author's implementation of a half-adder in HWC.

```
part HalfAdder {
	public bit a;
	public bit b;
	public bit sum;
	public bit carry;

	sum = a^b;
	carry = a & b;
}
```

Figure 22: A student's half-adder implementation that represents the input as two separate bits instead of as an array.

```
part HalfAdder
{
	public bit a;
	public bit b;
	public bit carryOut;
	public bit sum;

	sum = (a != b);
	carryOut = (a && b);

}
```

Figure 23: A student's half-adder that calculates sum with an inequality check rather than the XOR operator. This is valid syntax.

```
part HalfAdder {
	//Can we parameterize parts?
	public bit a,b, sum, carry;
	sum = (a != b);
	carry = (a & b);
}
```

Figure 24: A student's half-adder that declares multiple bits in a single declaration.

```
part HalfAdder
{
    public bit a;
    public bit b;

    public bit s = a ^ b;
    public bit c = a & b;
}
```

Figure 25: A student's half-adder that initializes the output bits during their declaration.

```
Part ONE_BIT_HALF_ADDER
{
	Public input_1;
	Public input_2;
	Public output;
	Public carry_out;

	Output = input_1 + input_2;
	If (input_1 == input_2 == 1)
	{
		Carry_out = 1;
	}
}
```

Figure 26: A student's half-adder that has bitwise addition and chains comparisons. Note that neither of these are valid syntax.

```
part  halfAdder
{
	public bit a;
	public bit b;
	public bit output;
	Public bit carry;

	output = (a ^ !b) | (!a ^ b);
	carry = a ^ b;
}
```

Figure 27: A student's half-adder that has confused the XOR symbol (^) as the AND symbol (&).

While they varied in naming conventions and general design, most of the students group were able to create a half-adder that would function in HWC. Adding to this accomplishment, the groups only had twenty minutes of instruction about HWC before they began writing their code. We hope this demonstrates how students with knowledge of *C*-styled languages are able to intuit the syntax of HWC.

Students were also encouraged to provide feedback about the design of HWC. In figure 24, a student asks if parameterizing *parts* is possible. This is based on the idea that *parts* are similar to functions in *C*, in that they take some amount of bits as input and produce some sort of result using them. We have considering allowing this in future versions of HWC. Another student, in discussion, questioned how a memory bit differs from a regular bit. This is a reasonable confusion, as the way HWC utilizes memory has no exact parallel in *C*. As such, certain aspects of HWC seem to require more explanation to students than others.

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. Over time, it was realized that its *C*-styled syntax and its one-to-one statement-to-implementation relationship could be used to explain hardware to computer science students through the familiar lens of *C*. The language has two principal abstractions of hardware: *parts*, which act as components, and *plugtypes*, which act as the connections between *parts*. Memory cells written to at the end of each clock cycle are read at the beginning of the next clock cycle, allowing a circuit to evolve across discreet time steps. When HWC code is compiled, a *wiring diagram* is produced, which details all of the wires and connections required to create the *part* called *main*. For this thesis, the compiler to convert HWC code into its corresponding *wiring diagram* was written. The compiler utilizes a lexical analyzer, a parser, and then four phases of semantic analysis to create an appropriate *wiring diagram*. The *wiring diagram* is the main output of the current iteration of HWC, and acts as an example of what the language can accomplish.

Yet there is still much ambition for the future of HWC. Many planned features of the language have not yet been implemented; most urgently, support for multi-dimensional arrays must be added. Additional features include compile-time variables and operations to support them, casting between *plugtypes*, and allowing components to be imported from other files. We have also considered expanding upon the foundations of the language to include new concepts like parameterized *parts*, or *plugtypes* with non-declaration statements. Furthermore, a graphic version of the HWC simulator has been conceptualized. This version would show the text of the user's code and would highlight different characters and lines as values propagate throughout the circuit. This GUI simulator could be used to further HWC's pedagogical goal, by allowing a student to see how values propagate in a HWC representation of a component. It could also be used as a visual debugging tool or as a development environment for HWC. With these ideas and many more additions in mind, we have laid out the foundations of HWC with our eyes turned towards its future growth.

CODE

The complete source of the HWC compiler and simulator can be found at the links below.

https://github.com/JLHonors/HWC

This repository holds the progress of HWC as of this thesis. The directory "hwcCompile" contains the majority of the code written for this thesis. The directory "thesis" contains this thesis and supporting files that were used in its writing.

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

This repository is reserved for future development of HWC.