HWC: A Logic Circuit Simulator with C-family Syntax

1 Introduction

HWC is a hardware specification language, designed for the simulation of digital logic. It aims to be a useful pedagogical tool (allowing students to simulate digital logic at a variety of levels, from simple sum-of-products up to simple CPUs), and it may, in the future, be useful for the simulation of realistic full-scale CPUs.

HWC has two central concepts: "parts" and "plugs." Parts can be thought, very roughly, as akin to objects in an OOP language: they have private state (including other parts), and contain code which performs various operations. Plugs, on the other hand, are very much like C structs: they have no internal logic – they simply represent a data structure. This means that each plug simply represents a set of bits, which can be connected to a similar pattern on another part. Basically, think of a "part" as a chip or IC, and an "plug" like a standard plug or connector.

(Terminology note: the term "part" is used both for the *type*, which defines the components of a part, and also for an *instance* of the type – that is, when the part is actually used in a specific situation. The term "plug" is used exclusively for an instance of a plug – that is, a real connector, somewhere in a part – while the term "plugtype" is used to refer to the type.)

HWC includes support for generics – meaning that the definition of parts and plugtypes can be parameterized with integers or the names of other parts and plugtypes. This makes it simple to write quite flexible code – for instance, to write a MUX which can connect any plugtype, and have any number of inputs.

2 Simple Example

The following code implements a single part, which implements a 4-input MUX. It has 4 data inputs, and one output, along with a 2-bit control input. It reads the control input, and then connects the correct input to the output. All inputs, as well as the output, use the same type – a plugtype named `Data`. (Note that, for simplicity, this example does not use generics.)

```
part MUX_4_Data {
    public Data in[4];
    public bit  control[2];
    public Data out;

    foreach(i; 0..4)    // 0 inclusive, 4 exclusive
        if (control == i)
            out = in[i];
}
plugtype Data {
    bit fieldA[10];
    bit fieldB;
    OtherPlugtype fieldC[2];
```

```
}
plugtype OtherPlugtype {
    ...
}
```

3 Limitations

HWC ignores all of the practical limitations of wiring (such as those addressed by VHDL, Verilog, or other professional hardware description languages); it doesn't care about propagation delay, hold times, fan-in/fan-out limitations, etc. Instead, HWC uses a simplified clocked model – where we treat each clock cycle as a single atomic time unit. In one clock cycle, each wire can carry exactly one bit, and that bit is always fully saturated (unless it is disconnected and floating). Likewise, HWC ignores issues of path length; it assumes that the actual physical implementation (defined in a more conventional hardware description language) has addressed this, and the clock is set appropriately. Likewise, HWC does not address questions of power or wire layout.

HWC does not model individual transistors; its lowest level of resolution is the logical gate.

4 Terminals

Every component – from a lowly logical gate to a very complex part – has a certain number of terminals, known as its "public plugs." These are, essentially, the pins of the microchip: they are the places where the outside world can connect to the chip.

The HWC language does not have any way to declare that certain terminals are input or output terminals; these invariants, instead, should be communicated in program comments.[1] [2] However, every public plug **must** include the type (either a simple type, a plugtype, or an array of either) and a name.

In addition, most parts will include non-public plugs. These have names and types just like the public plugs, but cannot be accessed by the outside world. Instead, these are used only for the internal logic of the part itself. (This is equivalent to using a temporary value in a conventional programming language.)

The syntax for declaring a plug is very similar to declaring a variable or field in C. Prefix the declaration with `public` to make the terminal a public plug.

```
public bit[16] inputA;
MyIntf temp;
```

1 An older version of the spec included a way to specify input and output bits, to be enforced by HWC. However, we discovered that non-trivial interfaces would often have bits in both directions – such as a "message" and an "ack." Other interfaces might have connections which might switch back and forth, being an input on one clock cycle and an output the next. Worse, parts might **use** an entire interface in both directions, reversing based on some unknown high-level logic. In time, we came to believe that specifying input/output in keywords was not workable for complex components.

2 Possible improvement for the future: we have considered using the "input" and "output" keywords to (optionally) decorate public plugs (nowhere else). Public plugs without "input" or "output" would make no presumptions about the direction of connections; input plugs would only allow signals coming **in** to the part, and output plugs would only allow signals going out.

5 Connections

A "connection" in HWC represents a wire (or a set of wires, known as a "bus") connecting two plugs. Most connections have an implicit direction, meaning that the programmer is indicating that the **only** direction that data can flow is from one end to the other. These connections are represented with the equals operator:

```
bit[16] foo;
bit[16] bar;
foo = bar;
```

This intentionally uses the syntax of the C assignment operator, and in HWC, it indicates direction: information can only travel from bar to foo, never in the other direction.

However, it is important to understand that HWC has a different temporal model than C. In C, assignments are executed in order, one after the other, and thus the order in which assignment statements are written has a large impact. By contrast, HWC represents a wiring diagram, and as such all of the connections happen **at the same time.** This means that the code snippet

```
a = b;
b = c;
```

Is **exactly equivalent** to the snippet

```
b = c;
a = b;
```

In both snippets, the programmer is indicating that data will flow from `c`, to `b`, and then to `a`[3].

6 Undirected Connections

As noted above, the = operator creates a directed connection[4], which indicates that data will only flow from right to left. (This intentionally mimics the assignment operator in C.)

However, since HWC models hardware, other types of connections are also possible. Undirected connections can be thought of as "dumb wires" which connect two terminals. These wires can carry data in either direction – although they can only carry it in **one direction per clock cycle.** These are declared using the `<->` operator. For example:

```
x <-> y;
```

On any given clock cycle, one of the following must be true:

- The value of `x` is set by some other logic; the value is driven across the wire, forcing `y` to the same value.

3 To be more precise, data **might** flow through that path – since it is possible that `c` might not ever be connected to anything which drives a value. However, the programmer is indicating that **if** a value is driven, the path is always `c->b->a`. Or, to quibble with the quibble: perhaps `c` is **not** connected to anything, but `b` is; in that case, the path is `b->a`, and `c` is not involved.

4 HWC doesn't assume that all directed connections would be implemented, in hardware, in such a way that the direction was enforced; instead, it is a simple sanity-check to help programmers find bugs. It is likely that many (if not most) of the directed connections would be implemented in hardware as simple wires – exactly as we describe the **concept** of an undirected connection.

- The value of `y` is set by some other logic; the value is driven across the wire, forcing `x` to the same value.
- Neither `x` nor `y` are set to any value; both float.

It is **illegal** for both `x` and `y` to be set (by external logic) on the same clock cycle, **even if they happen to be set to the same value.** If this circumstance occurs, HWC will report an error (since in hardware, this is a potential short-circuit).

Undirected connections may sometimes be used when the direction of data flow is fixed, but it is inconvenient to use the = operator. For instance, a plug might represent a complex interface, where some pins are used for input, and others for output; it would not be possible to connect the plug using a single = operator[5]. Another example might be generic code; if the generic code connects two inputs, it may not always be clear (to the generic code) what direction the data might travel.

However, we should note that *some* wires truly are bidirectional. They carry information in one direction on a certain clock cycle, and in the other direction on another. (While this is rare inside a chip, it often comes up in communication interfaces.)

7 Connecting Plugs

When a programmer defines a connection between any multi-bit plugtype (including arrays of bits), HWC treats it as a series of 1-bit connections. For instance, the two snippets below have **exactly** the same effect:

```
bit[4] foo;          bit[4] foo;
bit[4] bar;          bit[4] bar;
bar = foo;           bar[0] = foo[0];
                     bar[1] = foo[1];
                     bar[2] = foo[2];
                     bar[3] = foo[3];
```

The same is true of undirected connections: HWC treats an undirected connection between complex types as a set of (unrelated) 1-bit undirected connections. So, in the example above, if we used <-> instead of =, then it would be perfectly legal, on the same clock cycle, for data to travel one direction on some bits, and the other on other bits.

In an array of bits, this doesn't seem very interesting, since it is likely that all of the bits are used as a package, throughout the system. But in a complex interface, this could be very important. For instance, we might have a `valid` bit in a plugtype; if it's set, then the rest of the bits in the plug are also set – but if it is not set, then they will not be connected to anything. Or, we might even have a bit which indicates the proper **direction** that signals should be sent through a wire!

8 Conditional Connections

5 It is debatable whether this is good style or not; it may make a lot of sense to partition the input and output bits into two **different** interfaces, and connect each with an = operator. However, this spec recognizes the fact that some designs might prefer to use the <-> operator and a single, unified interface.

A conditional connection is a connection which is only made on some clock cycles. A programmer can define a conditional connection with an if statement, like below:

```
if (a == b)
    foo = bar;
```

In the example above, the value of the two plugs `a,b` will be compared; if the two are identical (in all bits), then `foo` will be connected to `bar`. The connection (if it is made) follows all the same rules as a non-conditional connection; you can even use a undirected connection if you want.

If the programmer nests `if()` statements, then the connection is only made if all of the conditions are true. For example:

```
if (a == b)
    if (c == d)
        foo = bar;
```

`if()` statements use the same syntax as they do in C – meaning that you can use (or omit) curly braces, and that you can use `else` and `else if` to express complex chains of conditions.

9 Memory

Memory is represented by variables with the following syntax:

```
memory(MyPlugtype) var;
```

The example above declares that there is a memory cell named `var`, which can store exactly one copy of the `MyPlugtype` interface. Like other variables, you can of course use basic types instead of interfaces – or arrays of either.

> **NOTE!** "memory" here does **not** refer to the main memory of a computer! Rather, "memory" refers to any storage unit, of any size, interpreted as any type, anywhere in the system. For instance, in a CPU, each register would be allocated as a memory cell, made up of many bits. Taken another way, main memory should be viewed as a special case of the memory feature of HWC – a special case where the memory is very large, and has a very simplistic strucutre.

Memory is read either by connecting to it with the = operator (with the memory on the right), or by using its name directly in another expression. Memory is written using the = operator, with the memory on the left-hand side. If you read and write from the same memory cell on the same clock cycle (which, of course, is quite routine!), then the read always reads the **old** value – that is, the value that was correct at the end of the previous clock cycle. Memory writes during this clock cycle never affect memory reads.

For example, the following snippet reads a memory cell, and stores its value into a plug:

```
memory(Data) someInfo;
Data copy = someInfo;
```

And this example writes to it:

```
someInfo.fieldX = 1;
```

This example uses a memory field in a larger expression:

```
memory(bit[32]) mask;
bit[32] workingVal = input & mask;
```

Finally, this example both reads and updates a memory cell:

```
memory(bit) toggle;
toggle = !toggle;    // it reverses each clock
```

Just like any other connection, the connections that write to a memory cell can be conditional. Just like any other terminal, it is always illegal to have more than one connection (conditional or not) which writes to the same memory cell on the same clock cycle – even if all of the connections write the same value.

If you do not write to a memory cell on a given clock cycle, its value does not change; the same value will be presented on the next clock cycle.

Like any other connection, connections to a memory cell are tracked on a per-bit basis; that is, a memory cell that holds a complex plugtype is tracked internally as a plurality of 1-bit memory cells.

It is never legal to connect to a memory cell using an undirected connection.

All memory cells are initialized to hold the value 'false' (0), and will stay in that state until written to.

10 Flags

A 'flag' is a single-bit terminal with special connection rules. Flags are used for checking if a certain condition has occurred, in situations where it is difficult or impossible to have a single component which checks all of the possible trigger conditions. Arrays of flags are legal, including multi-dimensional arrays; however, flags can never be included as fields in a plugtype. Flags may be public plugs of a part.

To declare a flag, use the type `flag` instead of `bit`:

```
flag needFlush;
```

The semantics of flags are as follows:

- They use the same connection semantics as memory:
  - They can only be connected to with the = operator, never with <->
  - They can be read by using the name in an expression, or using = with the flag on the right side
  - They can be written by using the = operator, with the flag on the left side
- *Unlike* memory, the value read from a flag depends on what connects to the flag **this cycle**, not any previous cycle. (Flags have no memory.)
- **Any number** of connections can write to the same flag, on the same clock cycle. No error will be reported! However, the **only legal value to write** is 'true' (1)
  - Connections may be conditional

- If one or more connections write to the flag, then all reads of the flag will return 1
- If no connections write to the flag, then all reads of the flag will return 0

(Hardware designers will recognize a `flag` as modeling a wire with a pull-down resistor.)

11 Loops

HWC supports exactly one type of loop: `foreach()` over a compile-time range. `foreach()` loops are unrolled at compile time; thus, if a `foreach()` loop includes part or plug declarations, one copy of the part or plug is created for each iteration of the loop. This makes `foreach()` loops handy for building many duplicate copies of the same basic structure.

Because `foreach()` loops are unrolled at compile time, the bounds of the loop must be known at compile time.

One common use for `foreach()` loops is to compare a runtime variable to all possible values, and make conditional connections based on it:

```
foreach(i; 0..4)     // 0 inclusive, 4 exclusive
    if (control == i)
        out = in[i];
```

In this example, we iterate through the values 0,1,2,3 (the latter bound is exclusive). In each iteration of the loop, the integer variable `i` (see below for discussion of integer variables) is set to the proper value; note that it is **illegal** to modify the counter inside of the loop, and it is not possible to break out of the loop early[6].

Inside the body of the loop, we have an `if()` which compares `i` to the value of a terminal named `control`; presumably, it is a bit array with at least 2 bits. If `control` is equal to one of the 4 values, then the related conditional connection is made.

Loops are always "unrolled" at compile time – meaning that the compiler converts them into an equivalent form, which does not include any looping structure. The example above will be rewritten as follows:

```
if (control == 0)
    out = in[0];
if (control == 1)
    out = in[1];
if (control == 2)
    out = in[2];
if (control == 3)
    out = in[3];
```

6 We may consider adding a continue and/or break statement in the future, but I'm not yet convinced that it is necessary. If we implement break, that essentially makes all of the connections in the loop body conditional (based on whether or not we've broken out of the loop). If we implement continue, this makes some statements conditional – but only those statements **after** the continue (and they are skipped only if we did continue on **this pass** of the loop).

When a loop is unrolled, any structures or variables which are declared **inside** of it are duplicated – that is, each copy of the loop has its own copy of the element. For instance, imagine that we wanted to add many values together.

```
bits[32][4] aVals, bVals;
bits[32][4] out;
foreach(i; 0..4)
{
    Adder32 add;
    add.a = aVals[i];
    add.b = bVals[i];
    out[i] = add.out;
}
```

In the above example, the loop runs exactly 4 times. Each of those passes of the loop includes its own `Adder32` component, which handles the addition for that pair of numbers. In other words, there are 4 different `Adder32` parts.

Loops can be nested arbitrarily deep.

12 The `assert()` Statement

The `assert()` statement simply enforces that a given condition is true, on every clock cycle. HWC will check the condition (which must be a single bit) on every clock cycle, and report an error if the condition is false.

If an `assert()` statement is inside an `if()`, then HWC will only check the `assert()` condition when the `if()` statement is true. That is, the following snippets are equivalent:

```
if (x == y)
    assert(z);
```

```
assert(x != y || z);
```

13 Resolving Conditional Connections

At its core, HWC aims to model a static set of wires, connecting a fixed number of logical components. In this sense, there really is no concept of "time" – it's not true to say that something happens "first" and that something else happens "later." Rather, everything is happening in parallel.

Consider the following example code, which is legal syntax in both C and HWC:

```
a = b;
b = c;
```

In imperative languages such as C, it is well-known that time is implied; the first assignment (copying `b` into `a`) happens first. Only after this is complete do we copy `c` into `b`. Thus, what happens in the second assignment cannot possibly affect the first; thus, the value of `a` is entirely unrelated to `c`.

Things are very different in HWC. In HWC, the code above means that `a` and `b` are connected by a wire – but `b` and `c` are also connected, by another wire. Thus, `a` is

(indirectly) connected to `c`. Thus, on **every clock cycle,** all three terminals will always have **exactly the same value**[7]**.**

This is only a minor confusion – until we start talking about conditional connections. Consider the following code:

```
if (a == b)
     x = y;
a = foo;
b = bar;
```

This code compares the values of `a,b`. If they are identical, then it connects `x` to `y`. But it also connects `a` to `foo`, and connects `b` to `bar`. Which of these happens first? The answer, of course, is that they all happen in the same clock cycle – that is, there are a set of physical components which are constantly **attempting** to do their work, at all times. `a` is **always** connected to `foo`, because it's unconditional; the same is true for `b,bar`. The values of `a,b` are **always** being compared, and the physical components are **constantly** trying to connect (or disconnect) `x,y` based on the results of the comparison.

In the Real World, there would be a lot of noise in the output signal (`x`) before the circuit stabilized. In HWC, we simplify this model, and simply say that there is a dependency from `a,b` to the conditional connection. That is, the simulator must wait for the values of `a,b` to be known before it can decide whether or not the connection was made. Once those values are known, it can proceed.

Thus, there are **logical dependencies** throughout the simulated circuit. However, as we see above, these dependencies are **completely unrelated** to the order in which the statements of the program. That is, **all statements "happen" at the same time** – even if the simulator will have to deduce their answers in a certain order.

Unfortunately, not all configurations of all networks can be resolved. Circular dependencies are definitely possible, as in the example below:

```
if (a == b)
     x = 1;
if (x == y)
     a = 1;
```

The above example is definitely a bug, since there are **no clock cycles** in which the value of `a` or `x` can be clearly known. However, HWC doesn't report this as a bug, since there are other, more subtle, realistic scenarios where loops might happen. For instance, if a component is not in use (say, the 'enable' bit on some control interface is 0), then it might have any number of internal plugs which cannot take on sensible values. Instead of requiring that the programmer carefully disable all of this logic, we simply define that the affected plugs do not have a clearly-defined value, on some clock cycles.

14 Floating Values

7 See the footnote, far above, about this example. It is possible (though rare) for `a,b` to have the same value, but for `c` to not have a defined value; this happens when external components connect to `b` instead of `c`.

We've said above that wires can only carry a single "signal" (that is, a 0 or 1) per clock cycle, and that they can only carry it in a single direction. We've also said that we need to model dependencies – because each component will not give reliable outputs until its inputs have stabilized. But how do we model a wire **before** it has a reliable signal on it?

And is it possible for a wire to **never** have a signal on it, during some clock cycles?

In physical components, there is the concept of a "floating" terminal. This is a terminal which is not connected to any device which forces it to a specific value, and thus it's impossible to determine what voltage it will carry. This is critical for wires that might be used for both input and output (to prevent short circuits), but actually it's a general principle: what if you don't connect something?

Imagine a simple conditional connection in HWC:

```
if (a == b)
    x = y;
```

If `a` is not equal to `b`, then the connection between `x,y` is not established. But what if `x` is an input to some device – and `y` was the **only** connection that might be made to it? Or, maybe there are many possible connections, but on this clock cycle, none of them are completed? It is certainly possible that `x` might receive no signal.

Thus, HWC uses a special value, known as "floating," to indicate that a given value does not have any connection that drives it to a specific value. This is both the **initial** state of all bits in all plugs[8], and also the **final** state in some cases.

It is perfectly normal for a floating plug to be connected to another plug. Since floating means "no value driven yet," if the floating wire is connected to some wire with a specific value, then the signal travels down the wire, and the first wire is no longer floating. However, if both ends are floating, then they float together – they are known to have the same value, but this value is not known[9].

Inputs to devices may float. For instance, in the example above, if `a` was floating, then the condition `a==b` could not be resolved, and thus we could not decide whether to connect `x,y` or not. Or, one (or both) of the inputs to an AND gate might be floating – and this might affect the output.

We say that a floating input **might** affect the output because there are some cases where a result can be generated without knowing all of the values. Notably, if one input to an AND gate is false, then the output is false – even if the other input was floating. (Likewise, with true inputs to OR gates.)

15 Undefined Values

8 In this section, we will treat "plug" as shorthand for "one bit inside a plug," since connections are tracked at a per-bit resolution.

9 TODO: Footnotes above have mentioned the scenario where we connect `c->b->a`, but then only connect to `b`. In theory, `c` is still floating – although this is likely not what was intended. Should we treat this scenario as an error? Basically, that would make the rule "you cannot connect more than once **to** any plug (with directed connections), even if all but one of the connections are floating?"

Undefined values are related to "floating" values, but have an important distinction: they can participate in short circuits. Undefined values are values which are known to be[10] firmly set to **some** value (0 or 1), but the exact value cannot be determined.

Undefined values only come about at the **end** of the simulation of a clock cycle; we never track them until we have propagated all signals which we possibly can propagate. We then check the network for floating terminals, and record some of those floating signals as actually "undefined." Undefined signals are generated in the following circumstances:

- AND/OR/NOT/XOR gates (each bit is tracked independently):
  - If all inputs are floating or undefined, the output is undefined
  - If one input to an XOR is floating or undefined, the output is undefined
  - If one input to an AND/OR is floating or undefined, the output is undefined if the output cannot be deduced from the other input[11]
- Equality checks (== , !=):
  - The equality check is rewritten as a network of XOR and AND/OR gates, and then undefined signals are handled as stated above
- Conditional connections:
  - If the controlling condition for a conditional connection is floating or undefined, it is undefined whether the connection is made or not. Details are complex; see below.

Since "undefined" values (unlike "floating" values) represent saturated signals[12], they can cause errors when they are connected to other terminals. The following situations are all errors:

- An undefined value is connected to a wire which carries a signal (either true or false)
- An undefined value is connected to any other undefined value[13]

Undefined signals propagate through the network, just like any other signal; they flow through connections, and may end up as the inputs to other basic HWC components or to conditional connections.

When we cannot resolve a conditional connection (because the controlling condition is floating or undefined), we apply the following rules:

- If the connection uses the = operator, then the left-hand side is set to undefined, but the right-hand side is unaffected.
- If the connection uses the <-> operator, **both** sides are set to undefined[14]

---

10 Or more precisely, firmly believed to be.

11 What happens if both of the inputs to a gate are connected to the same floating or undefined value? Then the output might be deduced. However, we choose not to check for this condition, because proving that two inputs are equal in all possible situations is almost certainly an NP-complete problem.

12 (that is, wires that are forced to one of the two possible extreme voltages)

13 As we discussed below, HWC will **NOT** check for "duplicate" undefined signals; such situations are errors, even if it would be impossible, in real hardware, for a short-circuit to result.

14 Again, this is an excessively cautious strategy that HWC uses because the "precisely correct" strategy would likely be NP-complete. There are situations where it would be physically reasonable to only record **one** of the sides as undefined (for instance, if the opposite side had a non-floating value). But to detect **all** of these situations would likely be NP-complete.

We set one (or both) ends of a conditional connection to undefined, this may cause errors – either because one of them is already set to undefined (undefined-undefined is an error), because one of them is already set to a specific value, etc. In that case, HWC will report an error – because the circuit can sometimes result in short-circuits.

16 Runtime vs. Compile-Time

HWC programs think about expressions (and statements) in two ways: runtime, and compile-time. Runtime expressions and statements (the vast majority of any HWC program) deal with the actual values carried on wires; the values of these expressions change over time (one value per clock cycle, but can be different on different clock cycles). Compile-time expressions and statements are fixed at compile time.

To make the code easier to read, HWC requires that (nearly) all compile-time statements start with the `static` keyword. For example,

```
static int width = 3;
```

declares a compile-time variable named `width`, and sets it to 3. Similarly,

```
static if (a == b)
```

is an `if()` statement which **must** be resolved at compile time; thus, `a,b` must both be compile-time expressions.

16.1 Implicit Casting (compile-time → runtime)

While it is obviously impossible to include any runtime values into compile-time expressions, HWC allows for implicit casting of compile-time expressions to runtime expressions. This is possible because the only two compile-time types are `int` and `bool`.

A `bool` is implicitly cast to a `bit`, any time that it is used in a runtime expression or statement[15]. An `int` will also implicitly cast to a `bit`, but this is only valid if the value of the int is either `0` or `1`.

An `int` is implicitly cast to an array of `bits` (representing the binary encoding of the value), but there are several key details:

- While an `int` can hold a negative value, it cannot be implicitly cast to an array of `bits`. If the user attempts this, it is a compile-time error.
- The *length* of the array is determined from context (which is either an connection statement, or an operator of some sort); the `int` is converted to the right number of bits to match the other side of the expression or statement. Arrays can thus be as small as 1 bit long. (A zero-length array is not allowed, even if the value of the `int` is zero.)
- If the integer is too large to be stored in the available bits, it is a compile-time error.

15 TODO: Should it also allow for implicit casting to a bit[1]? I'm disinclined, but uncertain.

Implicit casting of `int` allows the following sorts of runtime expressions or statements:

```
terminal = 3;

if (runtimeBitArray == compileTimeConstant) ...

assert (value == false);

foo = bar | 0x00f0;
```

17 Runtime Statements

HWC supports the following runtime statements:

```
expr = expr;
```

This defines a directed connection between the two terminals; the value from the right side is sent to the left. The terminals must be the exact same type. (If a compile-time expression is used, it must be on the right side, and it is implicitly cast to the runtime type, see above.)

```
expr <-> expr;
```

This defines an undirected connection between the two terminals. The terminals must be the exact same type, and compile-time types are not allowed.

```
assert (expr);
```

This expression must resolve to a single `bit`. It is evaluated at runtime, and will throw a runtime error if the value is 0.

NOTE: If the expression is a compile-time expression, this will still be treated as a runtime `assert()` (that is, the error will only be reported at runtime), even though the result will always be the same.

```
if (expr) stmt

if (expr) stmt else stmt
```

A runtime `if()` statement makes any connections which happen inside it **conditional.** Moreover (unlike the `static if()` statement below), declarations in **both** branches of the `if()` statement are evaluated, and each branch also defines its own name scope. (This means that names declared in the 'true' branch do not conflict with names declared in the 'false' branch. It also means that it is impossible to access names declared inside the `if()` statement, from outside the `if()` statement.)

If `if()` statements are nested, then any connection statements inside them are only executed if **all** of the enclosing `if()` statements have true conditions.

Runtime `if()` statements may contain compile-time statements; the compile-time

statements are evaluated as normal. The only thing that the runtime `if()` affects is the set of connections which (presumably) are made somewhere inside.

NOTE: If the condition is a compile-time expression, then this will still be treated as a runtime `if()` (including all of the rules about how names are handled), even though, at runtime, the `if()` will always make the same decision.

18 Compile-Time Statements

HWC supports the following compile-time statements[16]:

```
static int IDENT = expr;
```

```
static bool IDENT = expr;
```

These statements declare and initialize compile-time names. Note that it is **illegal** to modify any compile-time value after it has been initialized.[17]

```
static if (expr) stmt
```

```
static if (expr) stmt else stmt
```

These are `if()` statements which must have a condition which is `bool`. Since these are resolved at compile time, a not-executed branch of an `if()` statement is actually *removed from the design*, and names declared inside the statement(s) are visible in the scope that contains the `static if()` statement. Note that this is **unlike** the runtime `if()` statement, where both branches of the `if()` are included in the design, and where the statement(s) for new name scopes. For example, the following code is legal:

```
static if (a == b)
    public bit[16] foo;
else
    public bit[32] foo;
foo[0] = 1;    // OK to access foo[] outside the if()
```

Note that the statement(s) in the `if()` may be single statements, or blocks with `{}`. They may contain compile-time and/or runtime statements, and/or declarations.

```
static assert (expr);
```

This `assert()` is checked at compile time; it is a compile-time error if it is false. The type of the expression must be `bool`.

```
foreach (IDENT; expr..expr) stmt
```

This is the only HWC loop; the `IDENT` will be used as the name of an `int` (the name must not already be in use in this scope). Both expressions must be `int` expressions; they are the bounds (inclusive, exclusive) of the loop. The loop is unrolled, and the name

16 Note that this list specifically **excludes** part and interface declarations.
17 TODO: Add a rule about declare-before-use.

is set to the proper value for each unrolled copy of the loop.

As with `static if()`, the statement inside the loop may be a single statement or a block. Unlike `static if()`, the body of a loop is a new name scope, and declarations made inside it are thus inaccessible from outside the loop. However, any such declaration is **duplicated** – with exactly one copy of the declaration for each unrolled copy of the loop.

19 Runtime Expressions

HWC supports the following runtime expressions:

`IDENT`

An identifier is a primary expression (that is, an expression with no operators). It may sometimes be a runtime expression (when it refers to a terminal[18] in the current name scope). It may also be a compile-time expression (when it refers to a part or subcomponent in the current name scope, or a compile-time variable).

`expr . IDENT`

Selects a field from an expression. If the base expression is a terminal (or a field inside one), then IDENT must be the name of a field in that type. If the base expression is a part or subcomponent, then the IDENT must be the name of one of the public plugs of that part. In either case, the new expression is a runtime expression!

`expr [ intExpr ]`

Selects one element out of an array. The base expression must be a runtime expression that refers to an array; the index must be an `int` expression.

NOTE: HWC does **not** allow for runtime indices – however, it is easy to build a MUX which will allow you to select from the elements of an array, given a runtime bus as the control.

`expr [ intExpr .. intExpr ]`

Slices a range out of an array. Has the same basic rules as array indexing, except that it returns a new array, which is a subset of the original. The indices are (inclusive,exclusive).

`(expr)`

Parentheses, to enforce order of operations.

`expr == expr`

`expr != expr`

18 NOTE: The terminal might be a memory cell or a flag.

Compares all of the bits of two expressions and returns a single bit, which is true (or false, for !=) iff all of the bits are identical.

The two expressions must have the same type; implicit typecasting of compile-time types to runtime types is supported. If the type is a complex type (such as a user plugtype, or an array), then the bits are all compared individually.

```
! expr
```

Performs logical negation of an expression. Is only valid for a single bit (or a length-1 bit array). Returns the same type.

```
~ expr
```

Performs bitwise negation of an expression. Is valid for any type. Returns the same type.

```
expr & expr

expr && expr
```

Performs a bitwise AND of all bits in the expressions. Both expressions must have the same type (implicitly cast if necessary); the result has the same type. User types are allowed.

For runtime inputs, the two operators are identical.

```
expr | expr

expr || expr

expr ^ expr
```

Performs a bitwise OR/XOR of the inputs. See AND above for rules.

```
(bit[])expr
(bit[x])expr
(bit[x][y])expr
```

Casts any runtime expression to an array of bits of exactly the same size.

```
(TYPE)expr
```

Casts an array of bits (of exactly the right size) to a plugtype or array type. Will produce a compile-time error if the size of the input array is incorrect. The original expression (before the cast) **must** be an array of bits (or a single bit); no other types are allowed.

(If a programmer wants to cast one complex type to another complex type, they must first cast it to an array of bits, and then to the new type.)

20 Compile-Time Expressions

HWC supports the following compile-time expressions:

`IDENT`

See the identical runtime expression description above.

`NUM`

A numeric literal. Only integers are supported, but both decimal and hexdecimal formats are allowed. We use the well-known C syntax for these literals, except that underscores are allowed (and ignored) anywhere in any numeric literal. Negative numbers are interpreted as negation-of-literal, so negative hex is supported implicitly.

`true`

`false`

A logical literal, of type `bool`.

`(expr)`

Parentheses, to enforce order of operations.

`-expr`

Negation of an `int`.

`expr + expr`

`expr - expr`

`expr * expr`

`expr / expr`

`expr % expr`

Performs a matematical operation; the inputs must both be `int`[19]. The ordinary order of operations and associativity rules apply.

`expr == expr`

`expr != expr`

Performs an equality comparison; the inputs must have the same type, but may be `int` or `bool`. (Note that there is no implicit casting between `int` and `bool`.) The result is type `bool`.

19 HWC does not provide built-in mathematical operators for runtime types; the user must implement some of their own. However, it seems likely that a standard library of common types will be developed quickly.

```
expr < expr

expr > expr

expr <= expr

expr >= expr
```

Performs an inequality comparison; the inputs must both be `int`. The result is type `bool`.

```
expr << expr

expr >> expr
```

Performs a bit-shifting operation[20]. It is a compile-time error if either operand is negative, although zero is legal for either side.

```
expr & expr

expr | expr

expr ^ expr
```

Performs a bitwise AND/OR/XOR between two integers. Note that NOT is not supported, since integers do not have a fixed number of bits. The normal order of operations and associativity rules apply (AND higher than XOR, XOR higher than OR). Also note that these operators **cannot** be used with booleans.

```
! expr

expr && expr

expr || expr
```

Performs logical operations between `bool` expressions. The normal order of operations and associativity rules apply. NOTE: The bitwise negation is **not** allowed on compile-time expressions; it is undefined on `bool`, and would produce an infinite string of bits on `int`!

```
sizeof(typeOrExpr)
```

Returns the size (in bits) of any plugtype. Can be used as the index into a declaration of any part or plug, or anywhere else that an `int` expression is normally allowed.

If the parameter is a runtime expression, then this refers to the size of the *plugtype* of the expression – which is fixed at compile time. The runtime *value* is ignored in this

20 Note that HWC does **not** provide built-in shift or rotate operators for bit fields, since there are a variety of ways those might be implemented. Instead, users should implement bit-shifting by connecting sub-arrays to other sub-arrays, and then setting the remaining bits to 0 or 1 as appropriate. Of course, we expect to see a standard library implementation of common options to develop rapidly.

expression.

`typeof(part), typeof(plug)`

Returns the type of a part or plug. Currently, this is not something that can be stored – not even in a compile-time variable. It is only useful as part of cast expressions or as parameters to a generic type.

21 Functions

HWC supports two types of functions: compile-time and runtime. A compile-time function takes compile-time parameters, and returns either `int` or `bool`. No side-effects are allowed in a compile-time function[21], meaning that they obey strict functional semantics.

Runtime functions are shorthand for parts; they take some number of parameters (any mix of compile-time and runtime expressions), and return a *runtime* expression. The runtime expression is, in effect, the 'out' plug of the implicitly-declared part. For example, using runtime functions, we may replace the following code snippet:

```
FooPart thing;
thing.a = 123;
thing.b = someControlVal;
dest = thing.out;
```

With the following:

```
dest = FooAsFunc(123, someControlVal);
```

TODO: syntax (including deciding whether to pre-specify the return type)

TODO: semantics

TODO: discussion of "output params" in runtime functions. Maybe use `<->` as an attribute to indicate bi-directionality? Or maybe use "input/output" ?

22 Connection Parameters

TODO

23 Member Functions (of parts)

TODO

24 Generics

TODO

---

21 You may declare a compile-time variable inside a function, but you may not modify it – and you cannot modify any global-scope compile-time variable.
TODO: discuss global-scope compile-time variables!

25 TODO

TODO

26 TODO

TODO

27 TODO

TODO

28 Generics

The examples of parts and interfaces above were all hard-coded to use particular bit sizes; however, HWC also allows for parameterized parts and interfaces. There are several types of parameters:

- integer
- boolean
- interface type (basic type or user-defined)
- part type
- connection

29 Simple Functions

HWC supports the use of functions to perform declarations or numeric calculations inline

in a single HWC statement. All functions are executed at compile time; some resolve to runtime values (bool or integer), while others resolve to parts which have implicit connections.

The simplest form of function is one that returns a compile-time value. These functions can only have integer and boolean parameters[22]. They can only return integers and booleans:

```
static function FIBONACCI(int i)
{
    static assert(i >= 0);

    static if (i <= 1)
        return 1;
    return FIBONACCI(i-2) + FIBONACCI(i-1);
}
```

TODO: I just added the 'static' prefix to the function declaration. Shall we remove all of the 'static' keywords inside???

The function FIBONACCI above calculates Fibonacci numbers. It works pretty much like a C function; however, notice that (following in the HWC style) all if()s in the function must be static if() because they are resolved at compile time. HWC does not enforce that all of the possible code paths return the same type (or even that they return anything at all); however, for each actual call of the function, the compiler will require that the chosen code path return a compile-time value (boolean or integer).

Functions are not allowed to have any side-effects, and cannot access any outside variables; they must calculate their return value entirely from their parameters – thus, they are "pure" functional code. In theory, the compiler could record the return value from each one in order to save time if they are called again; however, that has not yet been implemented.

Functions can include foreach() loops:

```
static function FACTORIAL_foreach(int i)
{
    static assert(i >= 1);

    int retval;
    retval = 1;
    foreach(val; 2..i+1)
        retval = retval*val;

     return retval;
}
```

TODO: shall we remove the loop, and simply require all static functions to be functional?

22 One could imagine uses for interface type or part type parameters (such as using the size of an interface to determine some value used in the function); however, they are not currently allowed by the language as it is not clear that the value added is worth the added complexity.

The function FACTORIAL_foreach() above calculates factorial using a foreach() loop. Of course, the same could be accomplished with recursion.

30 Part Functions

In addition, HWC supports "part functions" – these are functions which return a part, rather than a value. More precisely, a part function instantiates a new part in the context where the function call was performed, and the return value from the function is one of the public interfaces into the part. Likewise, some of the parameters passed to the function are typically also runtime interfaces – these "connection parameters" are connected as defined by the calling code.

Part functions make it possible for the programmer to define their own "syntax sugar:"

```
part function XOR_func_16(
                    bit[16] connection input1,
                    bit[16] connection input 2)
{
     return (input1 & input2) | (~input1 &
~input2);
}

part XOR_4_16
{
     public bit[16] input1;
     public bit[16] input2;
     public bit[16] input3;
     public bit[16] input4;
     public bit[16] output;

     output = XOR_func_16(
                 XOR_func_16(input1,input2),
                 XOR_func_16(input3,input4));
}
```

The part function XOR_func_16 above has two connection inputs; they are both bit[16]s. It calculates the exclusive-or of the two inputs, and returns the value.

The part XOR_4_16 above (which is an ordinary part, **NOT** a part function) calculates the XOR of four 16-bit inputs. To do so, it calls XOR_func_16() three times. This means that three copies of the XOR_func_16() part are implicitly declared as part of the XOR_4_16 part; the inputs and outputs of each function part are connected according to the call graph implicit in the code.

In order to prevent confusion (so that users do not believe that "execution" of a part function ends early), part functions are only allowed to have a single return statement, and it always must be the very last statement in the part function. (This also helps to automatically enforce that the part function returns the same type of interfaces on every clock cycle.) The return statement is always required, and cannot be conditional[23]. Since

23 Of course, the interface which is returned is connected to other logic inside the part function – and that

there is no way to reference the part function after it has been "called" in the program, no other public interfaces are allowed in a part function – although private interfaces are allowed and often very useful.

Except for (a) the existence of connection parameters, (b) the return statement, and (c) the exclusion of other public interfaces, the syntax and semantics of a part function are exactly like those of a part. It can include other parts, any number of connections (including conditional connections), foreach() loops, and static if()s.

Of course, static if()s imply compile-time parameters; thus, a part function can include any mix of connection parameters and other types of parameters:

```
part_function XOR_16_invertible(
                bool INVERT,
                bit[16] connection input 1,
                bit[16] connection input 2)
{
        bit[16] xor;
        bit[16] retval;

        xor = XOR_func_16(input1,input2);

        static if (INVERT) {
                retval = ~xor;
        } else {
                retval = xor;
        }

        return retval;
}
```

The part XOR_16_invertible above calculates the XOR of two 16-bit inputs; however, if the REVERSE parameter is set to true, then it will return the binary negation of the result, instead of the natural result.

Sometimes, a compile-time bool can be replaced with a runtime bit connection to achieve the same purpose:

```
part_function XOR_16_dynInvertible
                (bit connection invert,
                bit[16] connection input 1,
                bit[16] connection input 2)
{
        bit[16] xor;
        bit[16] retval;

        xor = XOR_func_16(input1,input2);
```

logic might include conditional connections; thus the "return" statement might return an interface which is not connected to anything. But the "return" itself is never conditional.

```
        if (invert) {
                retval = ~xor;
        } else {
                retval = xor;
        }

        return retval;
}
```

The part XOR_16_dynInvertible above is exactly the same as XOR_16_invertible, except that the boolean INVERT parameter has been replaced with a single-bit runtime connection. This will generate a part which makes a conditional connection based on the value of the 'invert' connection[24].

Not all functions have input connections:

```
part_function clock(int RATE, int PULSE_WIDTH)
{
        static assert(RATE > 1);
        static assert(PULSE_WIDTH < RATE);

        ...

        return clock_interface;
}
```

The part function clock above represents a simple clock; it generates a pulse (of configurable width) every so-many clock cycles. It has no need for an input connection, and so simply communicates through its return interface.

Note that we have not shown exactly what clock_interface is above. We might imagine that it is something as simple as a single bit (giving the clock value), but it might be more advanced – for instance, including fields for resetting the clock, masking the clock, etc.

Like all other interfaces in HWC, the inputs and return value from a part function can

24 You may worry that the use of a conditional connection (controlled by a runtime value) is inefficient if it so happens, in your program, that the condition is always the same. Do you need to implement two versions of each function part – one that has static parameters, and another that has runtime connections? Almost certainly not. If you hard-code one of the parameters to a compile-time value (0 or 1), then it is a trivial optimization for the compiler (or a post-processor on the wiring diagram) to discover that the "conditional" connection is not conditional at all – and to thus convert the dynamic if() into a static if(). The HWC language spec does not guarantee that the compiler will perform this optimization – but as a practical matter, one can reasonably expect it.

In general, use runtime connections for all parameters to part functions which could reasonably allow them – only use compile-time parameters for parameters which absolutely must be known at compile time (those which control the number of elements in the part).

(Similarly, in a non-function part, it is generally advisable, when possible, to replace boolean parameters with single-bit public interfaces.)

theoretically transmit signals in either direction; the return interface can accept input signals, and the input connections can drive outputs. Programmers are encouraged to write part functions which act in a way which will not be surprising to their users (by using inputs mostly for input values and return values mostly for outputs), but the language does not enforce this.

31 'auto' Connections

All examples of connection parameters to this point have specified the type of the interface used in the connection. However, HWC also supports 'auto' connections; these connections permit any type of interface, making it possible to build more generic part functions:

```
part_function XOR_func_auto(
            auto connection input1,
            auto connection input2)
{
      return ( input1 &  input2) |
           (~input1 & ~input2);
}
```

The part function `XOR_func_auto` above is another example of an XOR part function; this one uses auto connections, and thus can use any type of input. However, a syntax error will arise (inside the part function itself) if the types of the two inputs are not identical – since the binary ANDs inside the part function require that both inputs be of the same type.

32 part "main"

All HWC programs must include exactly one part with the name "main." The wiring diagram produced by the compiler represents this part.

Main may have parameters, but if it does, all of the parameters must be specified to the compiler, so that the part can be instantiated.

Main may have (but is not required to have) external interfaces. If it has external interfaces, then the simulator will make those available to the user as inputs to, and outputs from, the wiring diagram.

33 File Types, Packaging, and Imports

HWC programs are made up of two types of files. The root file of any compilation must be a single .hwc file; this file must include part 'main.' This file may import any number of .hwp files, known as "packages;" each package makes additional HWC declarations, but may never include a part 'main.' Imported packages may import other packages, to any depth. Import loops are permitted, and are automatically resolved by the compiler.

Unlike C, which allows multiple .c files, the HWC compiler only supports a single .hwc input file; it performs the entire compilation based on this file, and produces a wiring

diagram which represents 'main'. Any code (in the .hwc or any package) which is not eventually part of 'main' is discarded.

Unlike C (where headers are imported as raw text), each .hwp file forms an independent name space. To access the declarations inside an imported package, the importing file must use the fully qualified name. For example, if a file imports the package “foo::bar” and wants to reference the interface BAZ inside it, it must refer to that interface as “foo::bar::BAZ.” (Note that all referenced packages must be explicitly imported; it is **not** sufficient to simply use the fully-qualified name to implicitly import the package.)

If a package imports another package, the names of the latter package are **not** visible to files which import the former package; instead, they must import the latter package directly. For example, imagine that the main .hwc file imports package “foo::bar,” and “foo::bar” imports the package “baz::fred.” The names inside “baz::fred” are not accessible to the main .hwc file unless it explicitly imports “baz::fred” as well.

Import names represent a directory structure. That is, the import name “foo::bar” means that the compiler will search the standard import paths for a file named “foo/bar.hwp”. The default import path is the current directory; however, the compiler may support arguments to alter the import path (adding or removing directories) or otherwise changing the import-search algorithm.

All import statements must be at the head of the file, before any other statements. (Whitespace and comments before the imports, or in the midst of the imports, are allowed.)

34 Aliases

Since fully-qualified names are often inconvenient to use, HWC supports alias statements. Aliases create a new name, in the current file, which is an alias of a named declaration in another package. For instance, if “foo::bar” imports the package “baz::fred” and wants to make the interface type “WILMA” accessible inside its own name space, it may use the following alias statement:

```
alias baz::fred::WILMA;
```

As this alias statement does not specify a new name, the alias is given the name “WILMA”; from there on in the package, code can reference “WILMA” without needing the fully-qualified name. In addition, code which imports “foo::baz” may reference the name “foo::baz::WILMA,” which is identical to “baz::fred::WILMA.”

Alias statements may also provide a new name for the imported declaration:

```
alias baz::fred::WILMA as BARNEY;
```

This creates an alias in the local file, but this alias is named “BARNEY.” In all other respects, it works the same way as the “WILMA” alias before.

Aliases cannot have parameters; however, they may refer to parameterized types. If they do so, they must choose either to (a) make no mention of the parameters; or (b) to supply

all parameters.

If the alias does not mention any parameters, then the newly-aliased type has the exact same parameters as the original; if it gives parameters, then the newly-aliased type has no parameters:

```
alias foo::bar::BAZ as BAZ_1;
alias foo::bar::BAZ(true, 100, my_part) as BAZ_2;
```

In the alias statements above, we assume that "foo::bar::BAZ" is a parameterized type. The first statement creates a new alias in the local file named "BAZ_1," which has the same parameters as "foo::bar::BAZ," while the second statement creates a new alias in the local file named "BAZ_2", which has no parameters.

Note that when making a non-parameterized alias of a parameterized type, giving a new name is mandatory – although it may be the same as the name of the original type:

```
alias foo::bar::BING(bit, 10) as BING;
```

Aliases may be used to make shortcuts or alternate names to locally-defined types:

```
alias my_long_parameterized_type(1,2,3,4) as asdf;
alias i_dont_want_to_type_this_often as jkl;
```

The alias statements above creates simple, short names which represents a long expressions, each of which resolves to a local type.

It is illegal to make any declaration (including an alias) which shadows another name visible in the same scope. (TODO: move this to a more general location in this doc!)

As will be discusseed below, parameterized types can include default values for some or all of their parameters. Sometimes, however, it may be desirable to declare a new parameterized type which is a "child" of the original parameterized type, and provides new defaults. The only way to do this is to wrap the original type in a new parameterized declaration, and this is only possible with parts and part functions (not interfaces):

```
part BAZ_simplified(part PART)
{
	public intf_foo foo;
	public intf_bar bar;

	foo::bar::BAZ(true,100, PART) baz;
	baz.foo = foo;
	baz.bar = bar;
}
```

While it is possible to wrap one interface inside another, it is not possible to hide the fact, because interfaces do not have internal connections:

```
interface DERIVED(interface INTF)
{
	BASE(INTF,true) base;
}
```

Currently, aliases can only refer to either (a) names in the local file, without any qualification; or (b) fully-qualified type names in other packages. An alias cannot be used to create an alias of an import package, or of a directory of import packages.

Currently, HWC does not have syntax to alias all names from a remote package into the local scope; each name must be aliased individually.

Aliases may never be conditional.

35 Name Scope

Names are only accessible in the scope in which they are declared. In general, curly braces {} create a new scope, and thus names declared within a block are not accessible outside of the block. The notable exception to this are blocks associated with static if() statements: these do **not** create a new scope, since the parts and interfaces inside those statements are often accessed from outside the block[25]:

```
part static_if_example(bool arg1, bool arg2)
{
	public bit[16] input;
	public bit[16] output;

	static if(arg1) {
		PART_TYPE_A step1;
	} else {
		PART_TYPE_B step1;
	}

	static if(arg2) {
		PART_TYPE_A step2;
	} else {
		PART_TYPE_B step2;
	}

	step1.input = input;
	step2.input = step1.output;
	output      = step2.output;
}
```

In the example above illustrates why static if() blocks do not create a new name scope. In this example, two different sub-components are declared: step1 and step2. However, the type of each component is controlled by boolean parameters, and thus the declarations must be inside static if() blocks. Yet the components must be accessible from outside the static if() blocks. If static if() created a new name scope, then this sort of component might not be possible.

25 TODO: should we consider removing this exception? Would it be practical to require that the static if() perform all of the connections necessary to the outside world? That might be difficult to make work if you had two parts, in two different static if() statements, which wanted to connect to each other...

foreach() loops create a new name scope; each interface or part declared inside the loop is only visible in that pass of the loop. (This is important because there may be many copies of the same interface or part, with the same name: one from each iteration of the loop.)

Declarations are allowed inside a foreach() loop; the names declared inside the loop are only accessible to code inside the same loop:

```
part foreach_name_example(int COUNT) {
	public bit[16] input;
	public bit[16] output;

	bit[16][COUNT+1] steps;

	steps[0] = input;
	foreach(i; 0 .. COUNT) {
		MY_PART nestedPart;
		nestedPart.input = steps[i];
		steps[i+1]	= nestedPart.output;
	}
	output = steps[COUNT];
}
```

In the example above, a foreach() loop is used to chain many copies of some part together. An array of private interfaces is used to form the links between the various steps; each step is declared as a component inside an iteration of the foreach() loop. Each such component is named 'nestedPart'; this name is only accessible within that iteration of the foreach() loop.

Like foreach() loops, runtime if()s create a new name scope.

TODO: keep writing the paragraph above.

TODO: mention (again) that files create new name scope.

TODO: mention default parameters and the named-value method of passingh parameters

TODO: mention typeof() syntax

TODO: mention that a new scope is created (in a runtime if() or a foreach() loop) even if the curly braces are omitted.

TODO: mention that “bare” curly braces (not associated with any if() or foreach()) are forbidden

TODO: mention name aliasing (forbidden)

TODO: mention that all names are public in the scope they are declared (so all names in a

file scope are accessible to all other files which import the package)

TODO: mention that names inside a part or interface declaration are **never** accessible from outside the declaration – except for the public interfaces.

36 Overloading Forbidden; Default Arguments Allowed; Named Parameter Passing

TODO

TODO: keep writing from here

TODO: write a formal grammar definition, and a formal specification of the language

IDEA: We could add member functions to interfaces...these are equivalent to calling a function, passing the interface as the connection parameter. There is probably a comparable reasonable implementation of member functions in parts.

TODO: split into 5 major sections:

- Introduction/Overview
- Compiler specification
- Simulator specification
- Compiler implementation

- Simulator implementation

HWC allows the user to define compile-time variables. HWC only allows the following types:

- "int" (unlimited precision signed integer)
- "bool" (boolean value)
- fixed-size array of another type (multidimensional arrays allowed)

Variables are strongly typed, and must be declared explicitly. The declaration may include an initializer; if it does not, then the values default to 0 for int's and false for bool's. Arrays cannot be given initializers in their declarations; the fields in the array all take their default values.

Variables may be modified any number of times. When a variable is referenced, it takes whatever value it had most recently. If a variable is modified inside a loop (see below) - and referenced in the same loop - then each reference uses the most recent assignment to the variable.

All variable declaration and assignment statements must include the "static" keyword, to indicate that they are evaluated at compile-time.

Variables may be declared at file scope (that is, outside of any specification) or specification scope (inside a specific specification).

File-scope variables may be referenced, but never modified, inside a specification in that file. Moreover, once a variable is referenced by a certain specification, it is illegal to modify the value of that variable later in the file. (Otherwise, it would not be clear whether the specification should have used the old or new value of the variable!)

Variables may never be referenced outside of the scope where they were declared. See NAME SCOPE below for more information.

37 Static if()

HWC allows for "static if()" - that is, an if() whose condition is a compile-time expression. This lets the user to conditionally compile certain parts of a file.

static if() is allowed at file scope (in which case it may include file-scope variable declarations, file-scope loops (see below), and even specifications). It is quite legal to declare the same specification or variable name twice in the same file - provided that

static if() is used to ensure that only one is actually in use.

static if() at file scope (outside any loop) is always evaluated exactly once – when the file is parsed.

static if() is allowed inside a loop (see below). In this case, it is evaluated once per pass of the loop. That is, the compiler first unrolls the loop, and only later evaluates each of the static if() statements found inside the loop. In some cases, the condition of a static if() may resolve to true in some passes of the loop, and false in others.

static if() is also allowed inside a specification. In this case, it is executed once per instantiation of the specification. Likewise, static if() is allowed inside loops inside a specification; it is executed once per iteration of the loop.

NOTE: static if() is *NOT* a scoping block. That is, it is permissible to define a name inside a static if() block, and then to reference that name outside of the block.

38 Foreach Loops

HWC only supports a single type of loop: the compile-time foreach loop. In a foreach loop, the progammer gives start and end values (both compile-time values), and a variable name. HWC unrolls the loop, setting the variable to the proper value in each pass of the loop. It is illegal for code to modify the loop variable.

Since HWC loops always are of this simple form, it is impossible to write a loop which iterates forever.

HWC iterates over the range [start,end); if end<=start, then the loop resolves to the empty statement.

NOTE: Loops may be nested arbitrarily deep.

39 Runtime if()

HWC supports if() statements which are evaluated at runtime; to declare one, simply write "if()" without the "static" keyword. The input condition of a runtime if() must always be a single bit. This bit can be explicitly taken from the output of any value, or may be the result of an arbitrarily complex runtime expression. (For instance, equality tests are allowed, as are boolean operations between bits. See IMPLICIT LOGIC below.)

NOTE: It is illegal to pass a compile-time value (even a boolean) as the condition to a runtime if(). If a compile-time value is available, use static if().

Runtime if() statements are only valid inside a part specification, and variable assignment or declaration statements are *NOT* allowed inside a runtime if() block. On the other hand, loops and static if() statements *ARE* legal inside a runtime if(); HWC expands those statements according to the regular rules.

The key reason for a runtime if() is to allow for conditional connections between

components. This is how HWC implements MUXes and most other switching logic. A connection statement (see below) is conditional if it resides inside a runtime if() block; the connection is made only if the runtime condition is true.

40 Connections

Each HWC part contains many connections. A connection may be thought of as a wire which connects two terminals inside the part. A terminal may be a pin of some nested part, an external pin of the current part, a power line (ground or +Vcc), or simpy a named terminal inside the part (in which case we expect to see another wire connecting the named terminal to something else).

Connections may be condition or unconditional. Unconditional connections are connections which are made outside of any runtime if() block; these connections are always valid, on every clock cycle. Conditional connections are connections which are made inside a runtime if block; these connections are made only on clock cycles when the runtime if() block is true.

Connections are bidirectional, meaning that a value may be driven on either end of the connection; the connection ensures that (for this clock cycle) the value is driven to the terminal on the other end.

It is valid to connect two or more input pins together. This simply ensures that if any one of the inputs is driven (that is, it is connected to some non-floating output), then all of the connected inputs will have the same value. If none of the pins are connected to any non-floating output, then all of the connected inputs will float.

However, it is dangerous to connect two non-floating outputs together, as a short cicuit may result in a real chip. As described below in RUNTIME VALUES, there are three types of non-floating values: true, false, and undetermined. If you connect two or more non-floating values together, and there is any combination of true and false, then the HWC simulator will immediately report a fatal runtime error. However, if all of the values are the same, or if some undetermined values are present, then the simulator will report a warning, indicating a possible (but not certain) problem with the design.

See UNDETERMINED VALUES below for more disscussion.

Since connections only last a single clock cycle, a specification must re-define all of the connections which are required for each clock cycle. There is no way to make a connection automatically persist across multiple clock cycles. (Even unconditional connections are considered, by HWC, as connections which are re-established on every clock cycle.)

NOTE: The above description accurately defines how a single-bit connection works. However, HWC does not require the programmer to connect each bit individually; the programmer may connect an entire bit array, interface, or array of interfaces to another. (HWC also allows you to connect an integer to a bit array; HWC converts the integer to its binary expression, and thus connects the bit array to the proper source rails.) HWC

decodes any multi-bit connection as a series of bitwise connections; each connection is treated separately. Thus, some of the bits in the connection may carry values in one direction; some may carry it in the other; some may end up floating this clock cycle.

41 Runtime Values

Each clock cycle, every terminal in the part takes a certain value. HWC allows for four different values: true, false, floating, and undetermined.

True and false have their ordinary meanings: the terminal is driven, by some component, either to +Vcc (true) or ground (false).

"Floating" means that the terminal is not drive by any of its connections (or, that it is not connected to anything). The HWC simulator determines that a terminal is floating either (a) when it determines that it is not connected to anything, or (b) determines that it is connected to other terminals, but none of the terminals in the set are connected to anything which drives a value.

(a) The HWC simulator has a list, for each terminal, of all of the possible connections to that terminal. This is a list of all of the unconditional and conditional connections to that terminal. As the simulator runs, it keeps track of which connections have actually been made (or not). If a terminal eventually drops to 0 possible connections (because all of the connections were conditional, and all conditions were false), then the terminal has a floating value.

(b) Just because a terminal is connected to another terminal does not mean that either is driven to a specific value. If a set of terminals are all connected together (directly or indirectly), then all of them will have the same value (floating or not). However, as in (a), we can track the set of all possible connections to terminals *OUTSIDE* of this set. If all of the terminals are only connected to each other (not to any terminal which drives a value), then all of them are floating.

Of course, some terminals (the outputs of logic gates) are internally driven. These terminals never float (though they may, as described below, be undetermined).

42 Undetermined Values

When the input to a logic gate floats, then the HWC simluator may not be able to strictly determine the value of its output. However, this doesn't mean that, in a real circuit, that the value would float. Rather, it means that, in a real circuit, the value would take some undetermined but specific value.

The HWC simulator models such output terminals with the "undetermined" value. "Undetermined" means that the terminal is *NOT* floating, but that it takes an unknown value. Undetermined values often propagate: if one or more inputs to a gate is undetermined, then its output may be undetermined as well.

When a terminal is driven by multiple connections, there is a possibility of a short circuit; when one or more of those of those connections is the undetermined value, then HWC

cannot ensure that a short circuit would not happen in a real chip. However, HWC also is not sure that a short circuit would actually result; perhaps the various connections would eventually all drive the same value. Thus, HWC will report a warning to the user. Perhaps, later, we will implement tools for exhaustive testing of such worrisome conditions.

NOTE: We can track some combinations of undetermined values by indexing undetermined values as they are created. That is, each time that we create a new undetermined value, we give it a unique (for this clock cycle) index value; when this propagates through a logic gate, the output keeps the same index. (When a logic gate has two undetermined inputs, then the output is a new, third value. When an undetermined value hits a NOT gate, the output keeps the index, but is marked as "negation of index X.") This allows us to determine special cases, such as where a terminal is driven by two connections, both to the same (or opposite!) undetermined value; or where a logic gate has both of its inputs be the same (or opposite) undetermined value. (TBD whether or not the simulator will implement this).

43 Undetermined Connections

Sometimes, a floating or undetermined value controls a conditional connection. In this case, it is impossible for HWC to determine whether or not the connection actually is made. HWC has to handle several different possibilities:

(a) If both ends of the undetermined connection are floating, then nothing needs to happen; the connection doesn't matter.

(b) If both ends of the undetermined connection are driven to the same value, (true or false), then HWC will post a warning (similar to the one posted if the connection was definitely in place here).

(c) If both ends are driven to *different* values, then HWC will post a warning, indicating that a short circuit will result if the connection is actually made.

(d) If both ends are "undetermined", then HWC will post a warning (similar to the one posted if a terminal is driven by multiple undetermined values).

(e) If one end is floating and the other is not, then the floating end will be driven to undetermined. (Note that this is a slightly wider variant of undetermined" than that created by logic gates: the terminal may actually float, depending on whether the connection exists or not.

TODO: what happens when a single terminal has several undetermined connections???

TODO: discuss "input" & "output" terminals (that is, terminals which are inputs to logic gates, and those which are output from). output terminals *NEVER* float; they are

always true, false, or undetermined. This has implications for undetermined connections: an undetermined connection to an output terminal *always* drives the other terminal to "undetermined" (even before we decide on a designation for the output terminal) because we know that the output terminal will never float.

44 Physical Implementation of Connections

TODO write this

TODO continue cleanup from here

- that is, that all possible connections were

conditional, and all chose not to connect this cycle - or (b) that the

terminal is connected to other terminals, but none of the terminals are connected to anything which drives a value.

"Floating" means that the terminal is not driven by any of its connections (or, that it is not connected to anything). The HWC simulator sets all terminals to floating at the beginning of each clock cycle, and moves them to other states only when the simulator finds an active connection to another terminal which is in one of the other 3 states. If HWC eventually determines that there are no connections which drive it to some other state (that is, there are no connections at all to this terminal, or none of the connections drive a value), then the simulator will report "floating" as the terminal's final state for this clock.

"Undetermined" means that the terminal takes a specific value, but HWC is unable to determine what that value is. Undetermined values arise from logic gates with floating or undetermined inputs. For instance, if both of the inputs of an OR are floating/undetermined (or if one is false and the other is floating/undetermined), then the HWC simulator cannot determine what state the output might take. It drives the "undetermined" state on the output.

Note the difference between "floating" and "undetermined": floating means that no value has been driven (yet), and undetermined means that some value has certainly been driven but we do not know what it is. If a terminal is driven to the "undetermined" value by one of its connections, it is a fatal runtime error for any other of its connections to drive any value (even undetermined) - because, in a real-world implementation, a short circuit might result.

Also note that the simulator differntiates between "temporary" floating values, and terminals which are firmly known to float this clock cycle. Temporary floating values simply mean that the simulator has not yet figured out what value they are driven to - a fixed value of true, false, or undetermined may eventually arise. Logic gates only produce the undetermined value when its inputs are firmly known to float this clock cycle.

TODO: what happens when the control of a conditional connection is undetermined or floating? I suppose that that is OK, so long as both of the terminals are floating...however, if either of them are not floating, then the other side should be marked "undetermined". But we can't mark both sides "undetermined" the moment that we find the undetermined connection, since it is valid for one (though not both) to be driven to some value. Ick.

TODO: will HWC always examine all of the undetermined terminals? Will it look for short circuits within them? For some simulations, we may wish to simply treat those sections as "dead" sections and overlook all short-circuit detection. Should we declare

that some parts need this checking, and others do not???

TODO: We should *NOT* track into the disconnected portions of the chip. Imagine a perfectly valid chip design, which is able to correctly handle any input whatsoever. But if you leave every single input floating, then every (or nearly every) chip inside would go to the undetermined state. This would almost certainly cause terminal errors, but they are spurious; by definition, the chip works correctly in all situations. The issue is that HWC doesn't model all of the possible combinations of states properly; it simply declares some terminals as undetermined, and (falsely) think that absolutely every possible combination is possible. This isn't really true; the logic of the chip limits the possible sets, and thus prevents short circuits.

TODO: by that logic, should we *ever* track undetermined states? Isn't that generally a major problem?

TODO: maybe setting known-mismatched values is a terminal error...setting known-matched or some-undetermined is a warning? And we snapshot the combination of terminals which have the warning, for offline analysis?

TODO: maybe we can audit subsets of the code, and prove that certain subsets are OK? Maybe using input assertions to limit the possibilities?

TODO: need a section on error analysis

45 Undetermined Values

As described above, when a terminal has a value of "undetermined," it means that the terminal takes a particular fixed value (that is, it is not floating), but the simulator was

unable to determine what value it would be.

TODO

46 Value Resolution

The HWC simulator must be able to resolve arbitrarily complex networks of logic. In some devices, there may be a clear dependency tree, where inputs flow gracefully to outputs in a heirarchical fashion. Others, however, may be much more complex, even including looping structures.

For instance, consider a device which has several small parts arranged in a loop. Each of the devices has a control bit, some amount of memory, and two ring interfaces (that is, interfaces which allow it to communicate with its adjacent devices). The ring interfaces are unconditionally connected to each other. Each ring interface is connected to a monitor, outside the ring, which reads the value being passed around the ring.

Each device is a MUX: when its control bit is off, the device connects its input-side ring interface to its output-side ring interface. However, when its control bit is on, it connects its internal memory to its output, and ignores its input.

Notice that, based on which control bits are on or off, the entire ring might be floating, the entire ring might be connected to the memory of a single node, or different parts of the ring might be connected to different memories. Moreover, since any node could be the "selected" node (that is, driving its memory onto the ring), there is no fixed order of evaluation which HWC can use to determine the value of the various terminals in the network.

Thus, the HWC simulator uses a depth-first evaluation model. In every clock cycle, we expect at least a few terminals to be driven: external input pins, the output terminal of internal memory bits, and any values which are hard-wired to either ground or +Vcc. The simulator sets each of these in turn, propagating the driven values to all of the unconditionally connected terminals.

When a driven value arrives at the input to a logic gate, HWC evaluates whether or not

the output of the logic gate can now be determined. If it can, then the simulator recurses deeper into the logic network, propagating the new value; if not, then the input is saved, but the logic gate is deferred until the other input(s) can be determined.

Likewise, when a driven value arrives at a terminal representing a conditional connection, the connection will be established (if appropriate). If one of the connected terminals is already driven to a value, then the connection will propagate this to the other terminal; otherwise, the connection is recorded for later use in the same clock cycle (if one of the two conneted terminals is eventually driven to a value).

When a conditional connection is *NOT* made (because the control bit is false), the simulator will check to see if this is the last possible connection to either terminal; it may be possible to set one or both terminals to the final-floating value (which might cause propagation from there into logic gates).

When the depth-first search runs out of terminals to drive, HWC will set all remaining terminals to "undetermined."

NOTE: It is possible to devise a circuit which gives a determined value, even from undetermined inputs. For instance, the expression (A or !A) always resolves to true, even if A is undetermined. However, the HWC simulator does not recognize such conditions, since (in general) the only way to do so would be to iterate over all possible values of the undetermined inputs.

47 Functions

A function is a special type of parameterized part, which can be easily declared and connected using a simple runtime expression.

A function is called using the same syntax as is used for part instantiation:

```
     function_name(param1, paramNameN: paramN, paramName2: param2, ... )
```

Unlike part parameters, which must all be compile-time expressions, some of parameters to a function may be runtime parameters, designated (in the specification) with the "connection" keyword. In this case, the parameter fulfills two purposes. First, typeof(parameter) is used to instantiate the specification (so two function calls, where the parameters are the same type, will be treated as identical instantiations of the specification). Second, the part will automatically have a public interface of the proper type, with the parameter name as the name of the interface. Code inside the part may connect to this interface exactly as if it was explicitly declared.

Connection parameters may have an explicit type, or may be generic. Connection parameters with explicit type only accept references to runtime expressions with the matching type. Generic connection parameters have the "auto" type, and allow connections from any runtime type. It is up to the implementation to dedeuce the type,

and provide an appropriate implementation. HWC provides only very limited facilities for type deduction, however. length(runtime expression) gives the length of an array, and is a syntax error if the expression is not an array. typeof(runtime expression) gives the type of a runtime expression, and is useful for declaring interfaces which match an input type. No other type deduction is provided.

Functions may declare, at most, a single public interface. This interface (if it exists) must be named "return". A function call is a runtime expression (just like a reference to a interface of some part); the value of the function call expression is simply the return interface from the function (if any). This return value may, optionally, be connected to other elements inside the calling part.

Function calls are only allowed inside part declartions. When HWC detects a function call inside a runtime expression, it automatically instantiates the part and adds an unnamed nested part to the current part definition. It then connects the various connection parameters to the input expressions provided by the user. It then uses the "return" interface from the function as the value of the current runtime expression.

(If a function call occurs inside an if block, then the connections to the function call - just like all other connections in the block – are conditional, and will only be made when the condition is true.)

Since function calls evaluate to ordinary runtime expressions, you may chain function calls together, connect them to interfaces, or do even connect them to each other. (When you connect one function call to another, note that you are connecting their return values to each other.)

Finally, note that the interfaces of a function call - both its connection parameters and its return - are ordinary HWC interfaces. While we conventionally expect the parameters to be inputs and the return to be an output, HWC does not enforce this. Each interface simply connects parts inside the function to parts outside, and values can (potentially) travel in either direction. Thus, function calls with multiple outputs are quite possible; simply pass one or more of the output interfaces as connection parameters to the function.

48 Implicit Functions

HWC provides syntax sugar for a few commonly-called library functions:

HWC Syntax	Function Call
A == B	EQUALS(A,B)
A != B	NOT(EQUALS(A,B))
~A	NOT(A)
A & B	AND(A,B)
A | B	OR(A,B)
A ^ B	XOR(A,B)

A >> B	RSHIFT(A,B)
A >>> B	BARREL_RSHIFT(A,B)
A << B	LSHIFT(A,B)

49 Name Scope

HWC has three different name scopes: file scope, specification scope, and loop scope. Note that if() and static if(), while they use curly braces, do *NOT* define new name scopes. So, names declared inside a static if() are accessible outside it. (No declarations are allowed inside a runtime if().)

File scope contains all of the names which are defined in a file but ouside of any specification or loop. These names include specification names, file variables, import names, and aliases. All of these names are accessible from specifications in this file, although they may not be created or modified inside a specification. Code outside the file may access specification names and aliases, but cannot access import names or variables. (Note that this means that code outside this file normally cannot access the files imported by this file; the other file must perform its own import. However, if this file defines an alias for an import, then the other file can use the alias to indirectly view the imported file.)

Specification scope contains all of the names which are defined inside a certain instance of a specification. These include public interface fields, variables, private interface names, and component parts. (Private interface names and component parts are not allowed in interfaces - only parts and functions.) Public interface fields are accessible from all code. All other names in specification scope are only accessible inside that instance of the specification: they cannot be accessed even by other instances of the same specification.

Loop scope contains all of the names declared in the context of one iteration of a loop. This includes the iterator variable (which the user cannot modify) and any variables or private interfaces declared inside the loop. All of the names vanish when the current iteration of the loop ends. For this reason, specifications, aliases, imports, and public interfaces may never be declared inside of any loop.

TODO scan the rest of this document to make sure that the above paragraph is matched elsewhere.

Names are never allowed to shadow one another. That is, it is never legal to declare a new name which duplicates one which is already accessible in that context. So, if you declare a name at file scope, the same name can never be used as a name in a specification or loop in that file.

However, it is perfectly legal to use the same name in sibling scopes. For instance, it is legal to declare the same name in two differnt files (even if one file imports the other). Likewise, it is legal to declare the same name in two different specification scopes.

50 Parameterization

Any specification may be parameterized. This is declared by adding a list of parameters in () following the specification name (similar to a C function declaration). Specifications which need no parameters may either use empty parentheses, or may omit the parentheses altogether.

There are six types of parameters:

- bool
- int
- interface
- part (parts and functions only)
- connection (functions only)
- auto connection (functions only)

bool parameters take a simple boolean value. The value must always be a compile-time expression; it is never valid to pass a runtime expression (even a single bit).

int parameters take an integer value. Integers are always compile-time values.

interface parameters take the name of an interface. The parameter name, when referenced inside the specification, is effectively an alias of whatever interface is provided by the caller. Interface parameters must always provide fully instantiated names. This means that, if the caller passes a name of a parameterized interface as the parameter to another specification, then it must provide all of the parameters for the interface it passes.

part parameters are just like interface parameters, except that they take part references instead of interfaces. Since parts can only be used inside other parts, part parameters are only valid in part specifications.

connection parameters are only used in functions. A connection parameter declaration gives a fully-instantiated interface name, the keyword 'connection', and the parameter name. The caller of the function must provide a runtime expression whose type matches the declaration. HWC then implements an automatic interface of that type on the function-part, and connects the runtime expression to the interface.

auto connection parameters are like connection parameters, but a little more complex. auto connection parameters are declared like connection parameters, except that the type given is the keyword 'auto'. auto connection parameters effectively function as two parameters: an interface parameter (which HWC deduces from the type of the runtime expression passed as the parameter), and a connection parameter (which HWC connects to the runtime expression). Inside the specification, the interface may be accessed by the expression

typeof(<paramName>)

while the connection (that is, the interface associated with the connection) may be accessed by the expression

<paramName>

51 Default Parameters

Any parameter declaration (except connection and auto connection parameters) may provide a default value. (Unlike C++, which requires that all of the parameters are right-aligned in the parameter list, HWC allows parameters with and without defaults to be intermixed.)

When code refers to the specification (meaning that it must be instantiated), the referring code may provide values which override the defaults, or it may omit that parameter (meaning that the default is used).

52 Instantiating a Specification

HWC does not "instantiate" a specification until it is actually used. The compiler starts with the "main" part, and instantiates all of the interfaces and parts referenced in it; this process recurses until all parts and interfaces in the entire tree have been instantiated.

Specifications which are not parameterized can only be instantiaed once; the first time that HWC finds a reference to this part or interface, it will instantiate the spec, and save it for later re-use.

However, specifications which have parameters might be instantiated many times, with many different combinations of parameters. Each time that the compiler encounters a reference to the spec, it will compare the current reference's parameters to previous instantiation(s). If the parameter combination has been used before, then the compiler will simply re-use the old instantiated spec. If the combination is new, it will re-run the spec with the new parameters.

Note that connection parameters are not counted as parameters in the comparison. Connection parameters always have the same type, and thus do not affect the internal implementation of the function. For similar reasons, auto connection parameters are treated as parameters in as much as they provide an interface parameter; however, inasmuch as they provide a connection parameter, they are ignored.

TODO: add support for variables, parts, and private interfaces in loop scope. They simply cannot be referenced after that iteration of the loop ends.

53 Anonymous Interfaces

HWC allows for the declaration of anonymous interfaces

TODO: defer this description. But eventually, the implementation will simply be to add an anonymous interface declaration expression rather than a custom statement.

TODO: write this

TODO: see notes in spec

TODO: continue writing/cleaning up the spec from here

ANONYMOUS INTERFACES

C allows the declaration of anonymous structs, which are structs declared inline, inside another struct. This often makes the code clearer, since you don't have to declare the nested fields in another location in the file.

For the same reason, HWC allows the declaration of anonymous interfaces, using the syntax:

```
interface { <fields> } <name> [, <name> ...] ;
```

Anonymous interfaces may be used inside an interface specification, or inside the input or output interfaces of a part.

HWC does *NOT* allow the declaration of anonymous parts.

ANONYMOUS COMPONENTS

Anonymous components are components inside a part specification which are not explicitly listed as parts, but which are automatically added by the compiler to handle boolean computation or comparison.

Example (bitwise AND):

```
 <in the inputs section>
   MyInterface input1;
   MyInterface input2;

 <in the outputs section>
   MyInterface output;

 <in the wiring diagram>
   (input1 & input2) => output;
```

Example (comparison):

```
 <in the inputs section>
   MyInterface input1;
   MyInterface input2;
```

```
<in the outputs section>
  bit eq;

<in the wiring diagram>
  (input1 == input2) => eq;
```

If the first example above, we have two inputs of identical type. We want to AND them together, and output the result. We could do this by explicitly declaring an AND gate (HWC supports parameterized AND, which can take any input type); we would connect each input to an input of the AND, and wire the output of the AND to our output. However, the one line of code in the wiring diagram accomplishes the same thing; the compiler will automatically add the AND component on our behalf, and wire it up as needed.

Similarly, the second example shows the compiler automatically adding a comparator.

HWC only support anonymous components for the following simple operations:

AND	&
OR	|
XOR	^
NOT	!
EQ	==
NEQ	!=

Note that it does not support less than, or the other inequality comparators, since those are typically implemented with an ALU. This is too complex for an anonymous component; the user should implement an ALU (or choose a standard one), and explicitly declare it.

VARIABLES

Variables come in four types:

- bool
- int
- bool array
- int array

A bool represents a boolean value; it can take the value TRUE or FALSE. It is initialized to FALSE. if() conditions always require boolean values as inputs; there is no automatic conversion of integers to booleans.

An interger represents an arbitrary-precision non-negative integer value (see below for details).

It is possible to build arrays of either integers or booleans; the length of the array must be declared when the array is declared, and is fixed from then on. However, note that the length may be controlled by a variable, and thus the length of the variable may vary based on parameters to the specification. The compiler will abort if you attempt to index outside the bounds of an array variable.

HWC does not support floats, characters, strings, structs, pointers,

multidimensional arrays, or any other type, other than the 4 listed above.

54 Integer Values

All integer variables (and temporary values) in HWC are arbitrary-precision unsigned ints. This means that, while HWC can model any non-negative finite integer, HWC cannot model negative numbers. This rather startling hole is intentional; as you see below, HWC includes implicit casting of integers to binary representation, and if we support signed integers, then it would be unclear when (or if) to sign-extend integer values. Similarly, signed and unsigned values introduce significant complexities if we want to support compile-time binary operations between integers. If you need to support signed values, you may pair an integer with a boolean value, or (if possible) offset the integer so that it stays in the positive range. (One way to express a signed integer is as the difference of two unsigned values.)

HWC uses arbitrary-precision integers, rather than a fixed integer size, to maximize expressiveness. We want the programmer to be able to perform arbitrary calculations (particularly bit shifts) without worrying about compile-time overflow.

HWC supports the all of the basic mathematical and comparison operations:

```
+ - * / %

== != < <= > >=
```

It also supports most bitwise logical operators:

```
& | ^
```

However, it specifically does *NOT* support binary negation, since there is no upper limit to the size of the integer value. An integer in HWC theoretically has an infinite number of digits (although finite value). Thus, if we allowed binary negation, then the result would have infinite value.

If a user needs to perform binary negation, he must specify the number of bits to negate; negation can then be accomplished using XOR:

```
newVal = oldVal ^ ((1 << bits)-1);
```

55 Implicit Conversion of Integers

Integer values may be used in expressions where we would normally expect runtime bit arrays. For instance, the following lines of code hard-code a particular output to a certain value, determined at compile time:

```
int myVal = ...;

myVal => outputField;
```

When an integer value is used in a runtime context, HWC automatically converts it to a binary value, exactly fitting the expected field. However, if the integer value is too large to fit into the expected number of bits, then HWC will report a compile error.

Note that implicit conversion only works when the context expects an array of bits. Integers cannot be implicitly converted to interface types. (Of course, you may split the integer into fields using integer bit operations, and then connect each field to the various fields of the interface.)

Most often, implicit conversion is used with loops, testing whether an input matches a value:

```
foreach(i; 0..maxValue)

{

  (input == i) => targetElements[i].control;

}
```

In the above example, we assume that 'input' is a bit array input, which is attempting to select from options (such as a MUX control field). targetElements[] is an array of parts, each of which has a single-bit 'control' field (such as switches). The code above declares 'maxValue' many anonymous comparators, each comparing the input against a different fixed value (the value of i at that particular iteration of the loop). Each comparator produces a single bit output, which is tied to the correct control input.

NAMES AND SCOPE

Like C, HWC supports nested scoping. However, the HWC model is much more

limited than C.

First, unlike C, shadowing is illegal in HWC. "Shadowing" is when a name

in a deeply-nested scope is the same as the name of something else, in a

shallower scope. This is a syntax error in HWC.

Second, while any curly-brace code block in C generates a new scope, this is not true in HWC. HWC doesn't have a preprocessor, like in C; instead, if uses if() statements, evaluated at compile time, to conditionally include or exclude elements. Thus, fields declared inside an if() block need to be accessible for later use. Thus, if() blocks (and, in fact, most blocks) are *NOT* scoping blocks.

There are only three scoping blocks in HWC:

- File
- Specification
- Loop

That is, each file is its own scope. Two files may declare elements with the same name without creating any problems; code inside one file may refer directly to names declared in that file, but must use the full package prefix syntax when referring to any name declared in any imported package. (That is, there is no automatic searching of imports for names.) If a file makes use of the same name from an imported package over and over, and using the full package prefix is annoying, the file may use an alias to copy the name into the local scope. Note, however, what that does: it creates a new name in the local file scope (which can thus be accessed directly, without any package prefix), which happens to be an alias of something defined in a different package.

The file scope may only include the names of part and interface specifications. HWC does not allow if() statements at the file scope, so it is not possible to conditionally compile different versions of part or interface specifications; however, each specification may include parameters which control which features might exist *inside* the specification. (See below.)

Specification scope contains all of the variables and fields declared inside a given specification (except those declared inside a loop); it also includes the parameters (if any) of the specification. Variables must be declared outside of any if() block, and thus cannot be conditional; however, when declaring array variables, it is possible to pass another variable (or parameter) as the array size; thus, it is possible to conditionally choose (or, to calculate from parameters) the apprpropriate size of the array.

Fields (that is, interfaces and component parts), on the other hand, may be declared inside an if() block (allowing conditional inclusion), subject to a few restrictions:

- Fields may never be declared inside loops (except for anonymous parts).
- If the same name is declared conditionally in multiple lines in the specification, they must be in exclusive sections of an if/else tree to ensure that at most one of those declarations can apply.

Unlike file and specification scope, there can be multiple layers of loop scope, reflecting nested foreach() loops. Loop scope contains the iterator for that loop, plus any variables declared inside the body of the loop.

As with specification scope, variables in loop scope may not be inside if() blocks inside the loop; however, it is permissible for the loop itself to be inside an if(). Look at the following example:

```
foreach(i; 0..10)
{
  int foo;   // legal.  Resides in loop scope.
}
if(condition)
{
  int bar;   // ILLEGAL!!!  Inside an if()
  foreach(i; 0..10)
  {
    int foo;   // legal.  Resides in loop scope.
    if(contidition2)
    {
      int baz;   // ILLEGAL!!!  Inside an if()
    }
  }
}
```

Note, in the example above, that two different variables are declared as 'foo'. This is legal, because the two variables reside in *different* (not nested) loop scopes.

UNIT TEST

TODO: Should we embed all of the unit tests into the main object file, and then have the simulator run the unit tests as part of its normal simulation run?

Each HWC source file (.hwc or .hwp) may, optionally, include a single part name "unittest". This part must not be parameterized. This part performs unit tests on any parts which the file declares.

Unit tests are generated by running the HWC compiler on the file with the --unittest flag. Instead of looking for a "main" part, the compiler will compile the unittest part. It will generate a wiring diagram, identical to normal compilation.

To run the unit test, load the wiring diagram into the simulator; the simulator will expect the part to include (at least) the following interfaces:

- "done" (output : bit[0])
  The unit test sets this bit to TRUE whenever the "result" field is valid.
- "result" (output : bit[x]) where x>0
  This reports the result of the unit test. If the value is 0, the unit test passed; any nonzero value is an error code which should be reported to the programmer.

The unittest part may include any additional interfaces (for debugging purposes, for instance). If the unittest part includes any input interfaces, the simulator will drive them to 0 throughout the duration of the test.

SIMULATION

The HWC simulator is designed to allow the execution, analysis, and debugging of HWC models. It reads the .hwo files generated by the compiler, and then executes the model as directed by the user.

As the simulator runs, it will log any errors that it encounters. Errors fall into two broad categories: fatal and non-fatal. Fatal errors indicate serious problems which either make the simulation unrunnable, or which might result in damage to the chip in the real world. Example: two elements driving different values to the same fan-in element (which would result in a short circuit in real life). Non-fatal errors indicate flawed logic 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 issues which often indicate logic errors, but which sometimes are harmless and can be ignored. 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, running only a single model, with no external logic. Eventually, however, we hope to expand it with many features, such as running multiple chips (wiring them together inside the simulator, even if they were not in the source files), emulating

external hardware elements (to allow emulation of real CPUs), and perhaps other features.

My long-term goal is to build a reasonably functional simulation of a PowerPC computer, running Linux (or perhaps a microkernel?), where I can test and execute real programs. (I hope to test ACALL!)

HIGH LEVEL MODEL

TODO

NOTEWORTHY HOLES

HWC is designed to be an expressive but minimal language. It does not aim to be, or even approach, a full-power general purpose programming language. It only attempts to make it easy to design large, heirarchical, parameterized hardware components. It makes a number of simplifications which may be surprising. Each of these was noted above, but I thought I would put them all together in a list here:

- No negative integers
- No floats

- No character values or strings
- No structs or multidimensional arrays
- No pointers or variable references
- No functions

It would be, of course, possible to rectify these problems. However, I left each of these out intentionally, in order to simplify the language and thus the compiler. My theory is that if you need more advanced processing, you should perform that processing in a wrapper program, and then pass your computed results as parameters to the main part.

TODO: INTEGRATE INTO THE TEXT:

- Forward declaration: only allowed for specifications. Never allowed for variables, fields, and *DEFINITELY NOT* variable values.
- Variable values: you may change a variable, after using it, even if you use it for something important, like an array size. However, the old use will never see the change; the old declarations stay as they were.
- I've thought about allowing aliases to refer to fully-instantiated remote specifications, but it makes the compiler and language more complex. Pull it out of this spec (if it ever made it in, I'm not sure). Also, specify that an alias *MUST* refer to a spec in an imported file, just to make the compiler easier to write.

TODO: CHANGE:

- Eliminate rules about var and field declaration locations. Just require that the code outputed, when we instantiate, has the right syntax.

TODO: ADD:

- Implicit conversion of bool to bit