HELLO WORLD

March 30, 2007

1 Introduction

You have just finished the Bluespec training course, and wish to do your first design in Bluespec. So you are sitting looking at a blank sheet of paper, or a blank computer screen, and wondering what on earth to write. The aim of this tutorial is to get you over this initial hurdle, and start you writing real Bluespec.

The first program which software people traditionally write in a new language is one which simply writes "Hello world" on the printer. Let us do the same. This means, of course, that our design is not going to be intended to turn into real hardware: it is merely going to produce output from a simulator. In this respect it is like a top-level test program, rather than a hardware device. (We'll have a design which produces real hardware as our next example.)

2 Version 1

A Bluespec design is a collection of packages—but in this simple design there will be only one package. So we begin by saying that we are writing a package[1] called "`FirstAttempt`"; and it is good practice to add a comment explaining what it is about. So we write:

```
package FirstAttempt;
   // My first design in the cool Bluespec language
endpackage
```

The compiler will object if the name of the package is not the same as the file it is stored in: that is, the package "`FirstAttempt`" must be stored in the file "`FirstAttempt.bsv`".

Type that into the computer, and run the Bluespec compiler on it:

```
bsc FirstAttempt.bsv
```

It should compile successfully, without any warnings.

A package may typically contain a few initial definitions of types and constants, and some module definitions. As an example of this, let us give a name to the string we are going to output:

```
package FirstAttempt;
   // My first design in the cool Bluespec language

   String s = "Hello world";
endpackage
```

Now let us think about the main module in this design. We must first consider how it is going to communicate with other hardware—either other modules in the design or hardware external to the design. That is, we must decide on the type of its interface. In this case the

[1] Actually the `package`/`endpackage` lines are not strictly necessary, but they are strongly recommended for all but the tiniest temporary test programs.

answer is easy: the module is *not* going to communicate with other hardware, so there is no interface to define.

By convention, the names of modules start “mk”, to distinguish them from their instantiations; so we can type in the shell of the module definition as follows:

```
package FirstAttempt;
   // My first design in the cool Bluespec language

   String s = "Hello world";

   module mkAttempt;
   endmodule
endpackage
```

Perhaps it is worth checking that this also compiles successfully.

Finally we add the rules, which define the module’s *internal* behavior (normally we would also have to add the interface methods, which define its *external* interactions, but in this example there aren’t any). In this example we have just one rule, which we shall call “say_hello”; this will use the system command “$display” to do the printing.

Since this module is to be “synthesized”, either to hardware or (in this case) to a simulator executable, we add the “synthesize” attribute; so the final first draft of our package is as follows.

```
package FirstAttempt;
   // My first design in the cool Bluespec language

   String s = "Hello world";

   (* synthesize *)
   module mkAttempt(Empty);
      rule say_hello;
         $display(s);
      endrule
   endmodule
endpackage
```

3 Simulation

First we run the BSV compiler tool bsc, to convert our package to Verilog RTL:

```
bsc -verilog FirstAttempt.bsv
```

This produces an RTL file called mkAttempt.v. From this we create and run the simulator executable:

```
bsc -o sim -e mkAttempt mkAttempt.v
./sim
```

Here, as explained in the *User Guide*, the "`-o sim`" specifies that the executable is to be called `sim`, and the "`-e mkAttempt`" says that `mkAttempt` is the top-level module in the design.

Try it and see what happens.

You will see that the rule executes on every clock cycle, so that there is an endless stream output of lines saying "Hello world".

4 Version 2

It is easy to make the output happen only once: we simply add a "`$finish`" command to the rule.

```
package FirstAttempt;
   // My first design in the cool Bluespec language

   String s = "Hello world";

   (* synthesize *)
   module mkAttempt(Empty);
      rule say_hello;
         $display(s);
         $finish(0);
      endrule
   endmodule
endpackage
```

Note that the actions within a rule execute simultaneously in hardware; but system tasks within a rule are performed by a simulator in the order they are written (for otherwise it would be very difficult to achieve the desired output). Thus the "`$display`" happens before the "`$finish`", as desired.

5 Version 3

The next version of this design is intended to output "Hello world" precisely five times. For this we shall need a register to act as a counter. This is declared in the module, before the rule. The counter must be able to count up to 5, so 3 bits will be sufficient, and the appropriate type is accordingly `UInt#(3)`—that is, an unsigned integer represented in three bits. We shall specify 0 as the register's initial value (which is restored whenever the design is reset). We add the register definition to the module as follows:

```
package FirstAttempt;
   // My first design in the cool Bluespec language

   String s = "Hello world";

   (* synthesize *)
   module mkAttempt(Empty);
      Reg#(UInt#(3)) ctr <- mkReg(0);

      rule say_hello;
         $display(s);
         $finish(0);
      endrule
   endmodule
endpackage
```

The rule must now be revised to increment the counter, and to finish at the appropriate time.

```
package FirstAttempt;
   // My first design in the cool Bluespec language

   String s = "Hello world";

   (* synthesize *)
   module mkAttempt(Empty);
      Reg#(UInt#(3)) ctr <- mkReg(0);

      rule say_hello;
         ctr <= ctr + 1;
         $display(s);
         if (ctr==4) $finish(0);
      endrule
   endmodule
endpackage
```

Remember that all actions within a rule happen simultaneously. So, even though the statement incrementing the counter is written above the if-statement, the latter sees the "old" value of `ctr`—thus, in order to produce five lines of output, the finish has to occur when that old value is 4.

6 Other Versions

This section shows a few alternative versions, some of which work and some of which don't.

6.1 Version 4

Another way of writing this is to use two rules, as follows:

```
package FirstAttempt;
   // My first design in the cool Bluespec language

   String s = "Hello world";

   (* synthesize *)
   module mkAttempt(Empty);
      Reg#(UInt#(3)) ctr <- mkReg(0);

      rule end_run (ctr==5);
         $finish(0);
      endrule

      rule say_hello (ctr<5);
         ctr <= ctr + 1;
         $display(s);
      endrule
   endmodule
endpackage
```

Check that these two versions give the same results.

6.2 Version 5

We might next consider splitting the rules still further, separating incrementing the counter into a rule of its own. This would lead to the following design:

```
package FirstAttempt;
   // My first design in the cool Bluespec language

   String s = "Hello world";

   (* synthesize *)
   module mkAttempt(Empty);
      Reg#(UInt#(3)) ctr <- mkReg(0);

      rule end_run (ctr==5);
         $finish(0);
      endrule

      rule say_hello (ctr<5);
         $display(s);
      endrule

      rule inc_ctr;
         ctr <= ctr + 1;
      endrule
   endmodule
endpackage
```

What would be the effect of doing this? In fact, it would continue to give the correct result; but notice that we are beginning to live dangerously. If the increment and the display were in the same rule, then we can be sure that they happen either together or not at all. If the increment is in a rule of its own, it is counting clock cycles, not lines of output. If the display rule were held up for any reason (admittedly unlikely in this particular example), the test would finish after the prescribed number of cycles, however many lines had been output by then.

The moral is that things which are required to happen together should be put into the same rule—simultaneous operations should not be achieved by relying on two rules firing in the same clock cycle.

6.3 Version 6

In our next version we get still more sloppy—we omit the condition on the rule "`say_hello`". We reason that the "`end_run`" rule will cause the test to terminate at the appropriate time, so there is no need to cease counting explicitly.

```
package FirstAttempt;
   // My first design in the cool Bluespec language

   String s = "Hello world";

   (* synthesize *)
   module mkAttempt(Empty);
      Reg#(UInt#(3)) ctr <- mkReg(0);

      rule end_run (ctr==5);
         $finish(0);
      endrule

      rule say_hello;
         $display(s);
      endrule

      rule inc_ctr;
         ctr <= ctr + 1;
      endrule
   endmodule
endpackage
```

Now the result is unpredictable. As we know, the behavior of the design always corresponds to the rules being executed "atomically"—that is to say, it is as if the rules executed in a single clock cycle were actually executed one at a time, one after the other. In this example all three rules fire in the final cycle, and the compiler tool has to choose whether "`end_run`" is to go before "`say_hello`" or after it. With the design as written above, and today's version of the tool, the "`say_hello`" rule will go first, and we shall get six lines of output instead of the desired five. It could be that rearranging the order of the rules would lead to a different outcome—but of course it would be foolish to rely on this, and this version should be considered bad.

6.4 Version 6

Finally we try the two-rule version with the condition of "`say_hello`" omitted:

```
package FirstAttempt;
   // My first design in the cool Bluespec language

   String s = "Hello world";

   (* synthesize *)
   module mkAttempt(Empty);
      Reg#(UInt#(3)) ctr <- mkReg(0);

      rule end_run (ctr==5);
         $finish(0);
      endrule

      rule say_hello;
         ctr <= ctr + 1;
         $display(s);
      endrule
   endmodule
endpackage
```

This version will in fact always give the right result—the order in which the two rules appear to fire "within" the final cycle is no longer arbitrary. But the reason for this is somewhat subtle[2], and so this version should also be regarded as bad, just because its correctness is unnecessarily difficult to understand.

[2]The rule "`say_hello`" writes to the register `ctr`, whereas "`end_run`" merely reads it; so the scheduler must arrange that the overall effect must be as if "`end_run`" executed first (for otherwise, in the actual hardware, where all the reads of the clock cycle happen before any of the writes, "`end_run`" would see a stale value of `ctr`. Thus the "`$finish`" happens before the extra "`$display`" would have happened, so the number of lines output is the same as with the previous version.