6.375 2013 Tutorial I
Introduction to Bluespec

Richard Uhler

February 8, 2013

1 Administrative

Class Website:	http://csg.csail.mit.edu/6.375
TA Name:	Richard Uhler
TA Email:	ruhler@csail.mit.edu
TA Office Hours:	Monday, Thursday, 4:30-5:30pm, 38-301
38-301 lab access code:	Under investigation
Lab 1:	Posted, Due next Friday, Feb 15th

If you have not received any of the class emails, you are not on the email list. Send me your email and athena username so I can add you to the email list, give you access to the servers, and set up a lab repository for you.

2 One Rule at a Time

Bluespec SystemVerilog is the high-level hardware description language we will use extensively in this course. Hardware descriptions in Bluespec consist of two parts:

State A collection of *explicit* state elements, such as registers and FIFOs.

Rules A collection of *Guarded Atomic Actions* which modify the state.

Each rule has two parts: a guard and an atomic action. The guard specifies the condition under which the rule may be applied. If the guard condition is satisfied, the rule is said to be *enabled*. The atomic action describes how the state is modified when the rule is applied.

The key to Bluespec is the following execution model:

One Rule at a Time:
1. Choose *any* one enabled rule
2. Apply that rule's action to transform the state
3. Go back to (1)

One Rule at a Time is very important. It should always be in the back of your mind as you use Bluespec.

3 Hello World in Bluespec

Here is a simple Hello, World program in Bluespec:

```
module mkHelloWorld ();
    rule sayhello (True);
        $display("hello, world");
    endrule
endmodule
```

It has no state elements. It has a single rule which prints the line "hello, world".

Question: What will this Bluespec program do when you run it?

If you answered "Print the line 'hello, world' to the screen and stop", then review the previous section on One Rule at a Time. What will this Bluespec program do when you run it?

1. Choose an enabled rule: in this case, the rule `sayhello`
2. Apply that rule's action: prints "hello, world" to the screen
3. Choose an enabled rule: in this case, the rule `sayhello`
4. Apply that rule's action: prints "hello, world" to the screen
5. Choose an enabled rule: in this case, the rule `sayhello`
6. Apply that rule's action: prints "hello, world" to the screen
7. ...

This program will print the line "hello, world" to the screen over and over and over again until the end of time.

If we want the program to print "hello, world" only once, we can introduce a state element:

```
module mkHelloWorldOnce ();
    Reg#(Bool) said <- mkReg(False);

    rule sayhello (!said);
        $display("hello, world");
        said <= True;
    endrule

    rule goodbye (said);
        $finish();
    endrule
endmodule
```

1. Choose an enabled rule: Initially `sayhello` is enabled, because `!said` is `True`. `goodbye` is not enabled, because `said` is `False`, so the only enabled rule is `sayhello`.

2. Apply `sayhello`'s action: prints "hello, world" to the screen, and sets the register `said` to `False`

3. Choose an enabled rule: Now `said` is `False`, `sayhello` is not enabled, and textttgoodbye is enabled, so the only enabled rule is `goodbye`.

4. Apply `goodbye`'s action: Calls `$finish` to terminate the simulation.

4 Using the Bluespec Workstation

The Bluespec Workstation is Bluespec's IDE. You can use it to compile, link, explore, and simulate Bluespec designs. To use the Bluespec Workstation, log on to a linux athena machine. For example, if you are running linux on your own computer, you can log on remotely to a class server using your athena username and password:

```
localhost:~$ ssh -X ruhler@vlsifarm-03.mit.edu
```

To set up your environment with pointers to the class installation of Bluespec, add the 6.375 course locker and source the class setup script:

```
vlsifarm-03:~$ add 6.375
vlsifarm-03:~$ source /mit/6.375/setup.sh
```

Now you can launch the Bluespec Workstation by running the command `bluespec`:

```
vlsifarm-03:~$ bluespec&
```

Bluespec source code uses the `.bsv` extension. Bluespec Workstation project files use the `.bspec` extension. Save the Bluespec code for the hello world module in a file called `hw.bsv`. Launch the Bluespec Workstation. Create a new project. Set the top file to `hw.bsv` and the top module to mkHelloWorld. Compile, link, and simulate to see what happens.

For more information on how to use the Bluespec Workstation, consult the bluespec user guide, which is available at `$BLUESPECDIR/../doc/BSV/user-guide.pdf`:

```
vlsifarm-03:~$ display $BLUESPECDIR/../doc/BSV/user-guide.pdf
```

5 Exercise

Exercise: Implement a Bluespec module which performs multiplication by repeated addition. Use a register of type Bit#(16) for the multiplicand, a register of type Bit#(16) for the multiplier, and a register of type Bit#(16) for the product. Set the initial value of the multiplicand and multiplier to your favorite numbers, such as 7 and 5, and set the initial value of the product to 0. Implement a rule 'sum' which is enabled as long as the multiplier is greater than 0. The 'sum' rule should increment the product by the multiplicand and decrement the multiplier by one. Implement another rule which displays the final product when the multiplier reaches 0 and terminates the simulation.

6 Reference Documents

$BLUESPECDIR/../doc/BSV/user-guide.pdf Describes how to use the Bluespec Workstation for compiling, linking, and simulating Bluespec designs.

$BLUESPECDIR/../doc/BSV/reference-guide.pdf Bluespec Language Reference.

$BLUESPECDIR/../doc/BSV/bsv-by-example.pdf Book which teaches Bluespec with many examples.

7 Tips for Lab 1

7.1 Registers vs. Variables

Registers in Bluespec are state elements which persist across rule executions. To apply a rule's action, first all registers are read, the body of the action is executed to determine the new values for registers, and finally the registers are updated with those new values.

Variables are just names for things, they are not associated with any storage elements. The value of a variable is based syntactically on the previous line of code where the variable was set.

For example, consider the following module:

```
module mkFoo ();
    Reg#(Bit#(32)) rx <- mkReg(7);
    Reg#(Bit#(32)) ry <- mkReg(5);

    rule regrule (True);
        ry <= rx + ry;
        rx <= rx - ry;
        $display("rx: ", rx, " ry: ", ry);
    endrule

    rule varrule (True);
        Bit#(32) vx = rx;
        Bit#(32) vy = ry;
        vy = vx + vy;
        vx = vx - vy;
        ry <= vy;
        rx <= vx;
        $display("vx: ", vx, " vy: ", vy);
    endrule
```

```
endmodule
```

When the `regrule` is applied, assuming `rx` is 7 and `ry` is 5, the first step is to read the register values. Conceptually the rule then becomes:

```
rule regrule (True);
    ry <= 7 + 5;
    rx <= 7 - 5;
    $display("rx: ", 7, " ry: ", 5);
endrule
```

The statements of the action are then executed from top to bottom to determine the new values of the registers:

```
rule regrule (True);
    ry <= 12;
    rx <= 2;
    $display("rx: ", 7, " ry: ", 5);
endrule
```

The final result of executing the rule is `ry` is set to 12, `rx` is set to 2, and the line "rx: 7, ry: 5" is printed to the screen.

Contrast this with what happens when `varrule` is executed, assuming the same initial state. First register values are read:

```
rule varrule (True);
    Bit#(32) vx = 7;
    Bit#(32) vy = 5;
    vy = vx + vy;
    vx = vx - vy;
    ry <= vy;
    rx <= vx;
    $display("vx: ", vx, " vy: ", vy);
endrule
```

Then the rule statements are executed from top to bottom, with variables replaced with their most recent definition:

```
rule varrule (True);
    Bit#(32) vx = 7;
    Bit#(32) vy = 5;
    vy = 7 + 5;
    vx = 7 - (7 + 5);
    ry <= (7 + 5);
    rx <= (7 - (7 + 5));
    $display("vx: ", (7 + 5), " vy: ", (7 - (7 + 5)));
endrule
```

The end result is ry is set to 12, rx is set to -5, and the line "vx: 12 vy: -5" is printed to the screen.

Variables may be reassigned any number of times within a rule. A register may only be set once by a rule.

7.2 Different Kinds of Assignments

Equals (=) Used for variable definition. For example:

```
rule myrule (True);
    Bit#(32) v = 3;
    $display(v);
    v = v + 5;
    $display(v);
endrule
```

Less Than Equals (<=) Used for register assignments within a rule. For example:

```
Reg#(Bit#(32)) r <- mkReg(3);
rule myrule (True);
    r <= 5;
    ...
```

`<=` is actually just syntactic sugar for calling a register's `_write` method:

```
rule myrule (True);
    r._write(5);
    ...
```

Less Than Dash (<-)(outside of rule) Used for instantiated state elements. For example:

```
module mkFoo ();
    Reg#(Bool) r <- mkReg(True);
    ...
```

Less Than Dash(<-)(within a rule) Used to get the result of an action method:

```
module mkFoo ();
    rule foo (True);
        Bit#(32) x = 4;
        $display(x);
        x <- multiplier.getResult();
        $display(x);
    ...
```