Massachusetts Institute of Technology – Laboratory for Computer Science
**IAPBlue A Mini Course on Bluespec**
`http://www.csg.lcs.mit.edu/IAPBlue`

Lab How-To 13 Jan 2003

This document describes how to compile and simulation a Bluespec design. Some of the logistics is specific to our computers, but most of it is still relevant if you choose to work on your own computer.

We have four linux computers available for running the Bluespec compiler. Two of them are on the CSG network and are called `watermelon` and `mcn02`. The other two are on the Theory network and are called `falcon` and `egret`. The hosts are all part of the `lcs.mit.edu` domain.

The Bluespec compiler will be available on all of these computers. The Verilog tools that we describe below are only for the CSG computers.

## Running the Bluespec Compiler

The compiler binary is called `bsc`, for "Bluespec compiler". This compiler is automatically in your path on `watermelon`. On the other computers, you may need to make an alias or add it to your path. On those computers, the compiler is found in:

```
        /net/watermelon/export/bsc/bsc
```

To get a list of command-line flags, run `bsc -help` or just `bsc -h`. To view a few extra "hidden" flags, you can add the argument `-v` for "verbose", which when used with `-h` lists a few additional debugging flags. We will use these debugging flags to print out intermediate results from the compiler so that we can get an understanding of the compilation process. Typically, the compiler takes a Bluespec design and returns either a Verilog or C implementation. The debug flags will let us view the TRS and scheduling information that the Verilog or C is generated from.

To generate Verilog code for a design, use the following command:

```
        bsc -verilog -g  module_name  filename
```

For example, in Lab 1 we provide the file GCD.bs which defines a module `mkGCD`. To produce the verilog file `mkGCD.v`, run the command:

```
        bsc -verilog -g mkGCD GCD.bs
```

If you would like to see the stages that the compiler is passing through as it produces the output, you can add the `-verbose` or `-v` flag.

To compile to C, replace `-verilog` with `-c`.

Before the compiler produces Verilog or C output, it generates an intermediate form called ATS (or Abstract Transition System). This is the TRS description that we have been discussing in lectures. To get insight into what the compiler is doing, it can be useful to view the ATS. One way to have the compiler dump the ATS is with the `-ats` flag:

```
        bsc -verilog -g mkGCD -ats GCD.bs
```

However, this flag does not give us all the intermediate information we would like. For instance, the conflict information between rules is not provided. We can obtain this information using the `-squery` flag. This flag takes the names of two rules and provides statistics about the interaction of those two rules. In the GCD example, we can ask for conflict information between rules `Swap__1` and `Subtract__2` with the command:

```
bsc -verilog -g mkGCD -squery Swap__1 Subtract__2 GCD.bs
```

We can also compare the internal rules with the interface methods:

```
bsc -verilog -g mkGCD -squery Swap__1 start_ GCD.bs
```

To view all rule conflict information, one can type `-squery \* \*`.

For Verilog modules that are generated, we might be interested in the conflicts between the I/O signals of the Verilog module. This information is needed by any designs which use that module, because the larger design needs to know which signals it can exercise in the same cycle and which it can't. The compiler includes this information in the header of the Verilog files that it generates. If we look at the header of the generated `mkGCD.v` file, we see:

```
// Method conflict free info:
// [result_ < start_]
```

Finally, you may find the `-show-stats` flag to be useful. When this flag is used, the compiler prints information about the number of state elements used in the design, the number of rules, the numbers of inputs and outputs, and the types of scheduling structures that were used to produce the scheduler.

For compiling a design spread over multiple files, you will want to include the `-u` flag. This tells the compiler to also compile any files that the current file depends on. The compiler will even check if those files have been touched and need to be recompiled.

### The bottom line

For most projects, you can compile the entire thing with the single command:

```
bsc -u -verilog -g  toplevel_module  toplevel_filename
```

If you would like to know more about the compilation process and the flags available to the compiler, refer to the Bluespec Compiler User Manual, available on the course website.

## Simulating a Bluespec design

As the previous section explained, the Bluespec compiler can produce Verilog code or C code, depending on the flags that the user provides. The Verilog output can be synthesized to produce hardware or it can simulated, with a Verilog simulator, to analyze or verify the design. Similarly, the generated C code can be compiled and run to simulate the firing of rules in the Bluespec design.

### Simulating in C

If you run the Bluespec compiler with the `-c` flag instead of `-verilog`, you will produce a C source file and a header file. For example, the following command:

```
bsc -c -g mkGCD GCD.bs
```

will generate the files `mkGCD.h` and `mkGCD.c`. The compiler will also compile them into an object file `mkGCD.o`.

The C code produced for a module doesn't run on its own. It has to be compiled into a larger program (or *simulation environment*) that will clock the design and properly handle its inputs and outputs. The Bluespec compiler comes with a generic simulator, which has all the basic features we need to debug our designs. (One could imagine using a more sophisticated simulator. And in fact, the generic simulator has hooks for extending it with additional features. If you would like to explore adding extensions to the simulator, read the Compiler User Manual and the documentation for the Traffic Light Tutorial, which makes use of the simulator hooks to allow user interaction with the model.)

## Making the simulator binary

To compile C object files into a C simulation, run the Bluespec compiler with the list of object files and use the `-e` flag to indicate the toplevel module of the design. So for the GCD example, you would type:

```
bsc -e mkGCD mkGCD.o
```

This will link the modules into the generic simulation environment and produce a binary called `a.out`. To give the program a more sensible name, use the `-o` flag:

```
bsc -e mkGCD -o GCD mkGCD.o
```

## Running the simulation

To run the simulation, simply execute the binary:

```
./GCD
```

However, without any flags, the output will be uninteresting. You will need to use the following flags, described in section 6, "Executing a Bluespec program," of the compiler user manual. Specifically, section 6.2 gives the flags for the C back end.

```
-v        verbose output
-s        show the state
-r        show the selected rules
```

With `-v`, you will see a list of rules which are enabled (that is, their conditions are satisfied) and you will see a list of the rules which the scheduler picked to execute. The flag also inserts a separator ("`----------`") between the output of each cycle, so that you can tell which rules are in which cycle.

The `-r` flag prints the rules which fire on each cycle. This flag is thus redundant with the `-v` flag, but can be used alone to print out the rules without any of the other information.

The `-s` flag prints the state of the system on every cycle. It is useful to use with the `-v` flag so that you can see what state led to the enabling of the rules. There are two additonal flags which can be used in conjunction with `-s`.

```
-t        show the types of state elements
-q        avoid ambiguity by listing the package that each type is defined in
          (this is called "qualifying" the type)
```

Showing the state of Arrays is not possible, since there may be too many values. Instead, you can use the following flag which prints out the value of any array write that happens:

```
-u        show array updates
```

The C simulation will end when no rule can fire. If rules can always fire, then the simulation will run until you kill it. You can also tell the simulator to halt after a certain number steps using the `-m` flag:

```
-m steps         the maximum number of cycles to run the simulations
```

Finally, there are facilites for dumping signals from the simulation to a file and for exercising the inputs during simulation. The C back end has the ability to write VCD files, like one would produce in a Verilog simulation, and which can be viewed in a waveform viewer:

```
-V vcdfile
-F refile
-d drvfile
```

The `-V` flag lists the file to dump the signals to. The `-F` flag names a file containing regular expressions, which are used as a filter to select specific signals to be dumped (rather than dumping the entire simulation!). The `-d` flag provides the name of a driver file which contains input signals to drive the simulation with. The formats of these files are described in section 6.2 of the compiler user manual.

## Simulating in Verilog

When run with the `-verilog` and `-g` *mod* flags, the Bluespec compiler produces a RTL verilog file that corresponds to the module *mod*. The verilog file is called *mod*`.v`.

To compile Verilog for the GCD example, one would use the following command:

```
bsc -verilog -g mkGCD GCD.bs
```

and the result would be a file called `mkGCD.v`.

### Interface to the Verilog module

If you look in the generated file `mkGCD.v`, the Verilog module `mkGCD` will have the following interface:

```
input CLK;
input RST_N;
input [31 : 0] start__1;
input [31 : 0] start__2;
input E_start_;
output start__rdy;
output [31 : 0] result_;
output result__rdy;
```

`CLK` is the clock. `RST_N` is a negative asserted reset.

`start__1` corresponds to the first argument to the `start` method from the Bluespec design (the x input). `start__2` is the second argument to the `start` method (the y input). `E_start_` is the enable signal for the `start` method. `start__rdy` is the ready signal for the `start` method.

`result_` is the value of the `result` method from the Bluespec design. `result__rdy` is the ready signal for the `result` method.

### Primitive modules

The GCD module instantiates two primitive modules, `RegUN` and `RegN`. These modules are defined in the `/usr/lib/Bluespec/Verilog/` directory on the CSG machines and in `/net/watermelon/export/bsc/bsc-latest/lib/Bluespec/Verilog` on the other machines.

### Top-level simulation shell

In order to simulate the GCD design a testbench is required. You can write a testbench in Bluespec or in Verilog. We have provided a sample Verilog testbench for the GCD example. It is called `GCDTest.v` and is available with the other files for Lab 1.

The testbench instantiates the `mkGCD.v` module and passes it all required signals (clks, data inputs, etc.). It asserts the `start_en` signal when the inputs to the GCD circuit are ready. When the GCD has been computed (`result_rdy` is asserted) it prints the result.

### Simulating the design in verilog

Set the following environment variables:

```
setenv PATH ${PATH}:{/home/jj/cadtools/synopsys/vcs/vcs6.2/bin}
setenv MANPATH "/home/jj/cadtools/synopsys/vcs/vcs6.2/man:$MANPATH"
setenv LM_LICENSE_FILE 1975@catfish
```

You can generate an executable simulator by invoking the vcs compiler:

```
vcs     -Mupdate                                \
        +libext+.v                              \
        -y .                                    \
        -y /usr/lib/Bluespec/Verilog            \
        -PP                                     \
        GCDTest.v                               \
        -o GCDTest
```

This compiles the `GCDTest.v` and `mkGCD.v` files into an executable: `GCDTest`.

### Viewing waveforms

Invoke the `vcs` GUI with the command:

```
vcs -RPP &
```

To open the waveforms:

- Click on Waveform.
- Select Open from the File menu.
- Select vcdplus.vpd from the window that pops up.
- Hit OK.

You can select the signals you want to view:

- Click on Hierarchy
- Select the signal you want.
- Click on Add.

### Simulator documentation

Documentation on the verilog simulator is available on the CSG computers at:

```
/home/jj/cadtools/synopsys/vcs/vcs6.2/doc/UserGuide/vcs.pdf
```

---

## Synthesis of Verilog circuits

If you would like to compile a Verilog circuit to examine its timing and area properties, you may use the Synopsys Design Compiler on the CSG computers.

Here is how you would compile the mkGCD.v design:

Our target library will be TSMC 0.18um.

Make sure you have a copy of the `.synopsys_dc.setup` file in your directory. This file is provided with the files for Lab 1. Create a `syn` directory for output results.

You will need to set the following environment variables:

```
setenv SNPSLMD_LICENSE_FILE 1975@catfish
setenv SYNOPSYS /home/jj/cadtools/synopsys/syn
setenv PATH ${PATH}:{/home/jj/cadtools/synopsys/syn/linux/syn/bin}

setenv TSMC_HOME /home/jj/cadtools/lib/tsmc018g
setenv TSMC_SYM  ${TSMC_HOME}/aci/sc/symbols/synopsys
setenv TSMC_SC  ${TSMC_HOME}/aci/sc/synopsys
setenv TSMC_IO  ${TSMC_HOME}/fe_tpz973g_220a/TSMCHOME/digital/synopsys/tpz973g_220a
setenv TSMC_SCR  ${TSMC_HOME}/fe_tpz973g_220a/TSMCHOME/digital/synopsys/script
```

To run the compiler and synthesize your design, you will need the script `GCDSyn.scr`, which we have also provided among the files for Lab 1. Once you have this script, you can type the following command:

```
dc_shell -f GCDSyn.scr
```

The synthesis tool will produce output files in the `syn` directory. The two important results are in `mkGCD.timing`, which lists the minimum clock period in nanoseconds, and `mkGCD.area`, which lists the size of the design in square micrometers.