GAA-to-RTL Synthesis using the Bluespec Compiler

6.375 Tutorial 8
March 17, 2006

In this tutorial you will gain experience using the Bluespec Compiler (BSC) to automatically synthesize a register transfer level (RTL) hardware description from a guarded atomic action (GAA) hardware description. A GAA hardware description decomposes the design into many rules or actions. Each rule has a predicate and only fires when that predicate is true. Rules read a subset of the design's state, perform some operation, and then write a subset of the design's state. Each rule is atomic with respect to all other rules, and as a consequence a hardware designer can consider each rule in isolation assuming no other rules are firing in parallel. For this course we will be writing our GAA using the Bluespec System Verilog (BSV) hardware description language. The Bluespec Compiler takes BSV as input and generates an efficient RTL implementation which preserves the GAA semantics. Figure 1 shows how the Bluespec Compiler fits into the 6.375 toolflow.

The most straightforward RTL implementation of a GAA design would simply execute one rule each cycle. Although semantically correct, this implementation would be extremely slow. The Bluespec Compiler attempts to schedule multiple rules to fire in the same clock cycle while still maintaining correctness.

This tutorial begins by examining a greatest common divisor unit to illustrate the basics of the compiler and the language. The tutorial then uses several toy examples to show various scheduling issues. Finally, a multi-cycle (unpipelined) SMIPSv2 processor is used to demonstrate designing larger systems with Bluespec.

The following documentation is located in the course locker (`/mit/6.375/doc`) and provides additional information about the Bluespec System Verilog language as well as the Bluespec Compiler.

- `bsc-reference-guide.pdf` - Bluespec System Verilog language reference
- `bsc-user-guide.pdf` - Bluespec Compiler user guide
- `bsc-style-guide.pdf` - Patterns and idioms for designing with Bluespec
- `bsc-timing-closure.pdf` - Approaches for increasing performance of Bluespec designs
- `bsc-known-issues.pdf` - Know bugs and issues with the Bluespec Compiler
- `bsc-examples` - Directory containing several Bluespec examples

Getting started

Before using the 6.375 toolflow you must add the course locker and run the course setup script with the following two commands.

```
% add 6.375
% source /mit/6.375/setup.csh
```

For this tutorial we will be examining a greatest common divisor unit and a multi-cycle (unpipelined) SMIPSv2 processor as our example BSV designs. You should create a working directory and checkout the examples from the course CVS repository using the following commands.

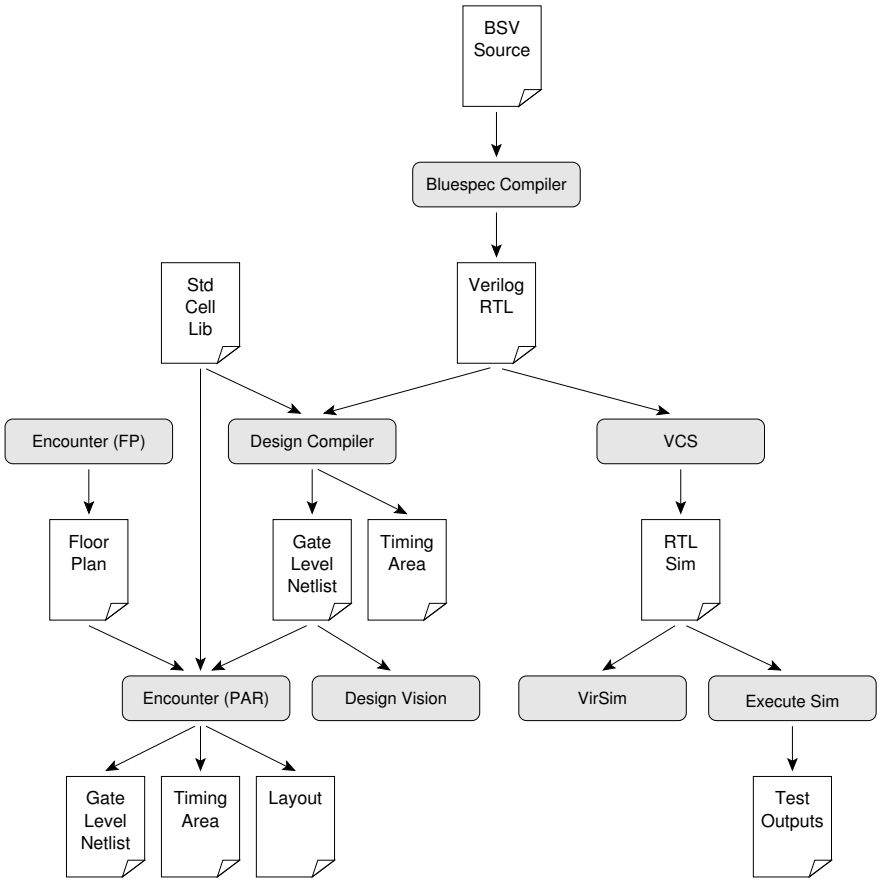

Figure 1: Encounter Toolflow

```
% mkdir tut8
% cd tut8
% cvs checkout examples/gcd-v
% cvs checkout examples/gcd-bsv
% cvs checkout examples/smipsv2-4mcycle-bsv
```

Before starting the tutorial, browse through the two example Bluespec projects. The actual designs will be discussed in more detail in the following sections. Notice that the directory structure is identical to our previous projects. The `src` directory contains our BSV source and the `build` directory contains various makefiles and scripts for running the tools. There is a new build directory named `bsc-compile` for running the Bluespec Compiler.

Running the Bluespec Compiler

In this section, we will be using the Bluespec compiler to synthesize a simple GCD unit. Take a closer look at the source code located in `examples/gcd-bsv/src`. The `IGcd.bsv` file contains an interface for a GCD unit while the `mkGcd.bsv` and `mkGcdWithAlu.bsv` files contain two different implementations of this interface. It is common to keep an interface and its implementations in separate files. The source directory also contains an appropriate test harness. The `mkGcdTH` module

in the `mkGcdTH.bsv` BSV file implements a simple state machine which pushes several tests into the GCD unit and verifies the results. The `mkGcdTH_wrapper.v` file is a Verilog wrapper to drive the clock and reset signals in our design. All of our Bluespec projects will need a toplevel Verilog wrapper.

Figure 2 is a *cloud diagram* and Figure 3 is the corresponding Bluespec code for the `mkGcd` module. A cloud diagram shows the rules, methods, and state present in the design and uses arrows to indicate the dataflow between these elements. Action-value method, action methods (such as the `start` method), and rules (such as `swap` and `subtract`) are all represented with clouds since all three constructs can change the state of a module. This is in contrast to value methods such as the `result` method which cannot change the state of a module.

There are a couple key syntactic issues which are important to keep in mind when writing in BSV. The first is that each BSV file should contain one and only one package, and that package should have the exact same name as the file. For example, the `mkGcd.bsv` file contains one package named `mkGcd`. At the beginning of each package we use `import` statements to tell the compiler which packages (and thus which BSV files) will be used by the current package. For example, the `mkGcd` package imports the `IGcd` package since it contains the GCD interface. All BSV type names *must* begin with an uppercase letter and all module names *must* begin with a lowercase letter. For example, the `IGcd` interface, `Reg#(Int#(32))` interface, and the `Int#(32)` type all begin with an uppercase letter, while the `mkGcd` module begins with a lowercase letter. Although not required, we suggest that you use an uppercase `I` prefix for all interfaces. We will use the `TH` suffix to indicate files and modules which make up a test harness.

We will run the Bluespec Compiler manually and then learn later how to automate the process with makefiles. Begin by creating a temporary build directory and copying the BSV source files. We must copy the BSV source files since the Bluespec Compiler currently assumes that the source files are located in the same directory from which the compiler is executed.

```
% pwd
tut8
% cd examples/gcd-bsv/build
% mkdir temp
% cd temp
% cp ../../src/*.bsv .
```

We can now use the following two commands to compile each BSV file into Verilog. You can learn more about the possible command line options in the *Bluespec System Verilog User Guide* (`bsc-user-guide.pdf`) or simply use `bsv -help`.

```
% pwd
tut8/examples/gcd-bsv/build/temp
% bsc IGcd.bsv
% bsc -verilog Gcd.bsv
% bsc -verilog -g mkGcdTH mkGcdTH.bsv
```

The `-verilog` command line option tells the compiler to use the Verilog code generation backend. Although there is a C code generation backend, we will not be using it in this course. The `-g` command line option is used to specify the toplevel module in the design. In addition to the

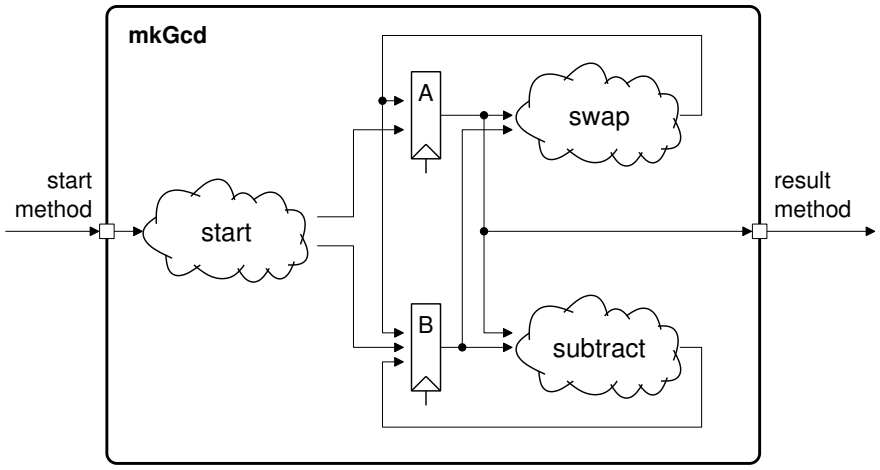

Figure 2: Cloud diagram for the `mkGcd` BSV module

```
package mkGcd;
import IGcd::*;

(* synthesize *)
module mkGcd( IGcd );

  Reg#(Int#(32)) a <- mkRegU;
  Reg#(Int#(32)) b <- mkReg(0);

  rule swap (( a < b ) && ( b != 0 ));
    a <= b; b <= a;
  endrule

  rule subtract (( a >= b ) && ( b != 0 ));
    a <= a - b;
  endrule

  method Action start( Int#(32) a_in, Int#(32) b_in ) if ( b == 0 );
    a <= a_in; b <= b_in;
  endmethod

  method Int#(32) result() if ( b == 0 );
    return a;
  endmethod

endmodule

endpackage
```

Figure 3: Bluespec System Verilog source for a greatest common divisor unit

Verilog RTL, the compiler will also generate intermediate `bi` and `bo` files. You should not need to directly use any of these intermediate files.

The Bluespec Compiler includes its own dependency tracking infrastructure to determine which BSV files are required to build a given toplevel module and which of these BSV files are out of date and thus need to be recompiled. The `-u` command line option enables the Bluespec dependency tracking infrastructure. For example, the following commands rebuild the design in one step.

```
% rm -rf *.bi *.bo *.v
% bsc -u -keep-fires -verilog -g mkGcdTH mkGcdTH.bsv
```

We have also added the `-keep-fires` command line option so that the compiler will generate extra signals to help in debugging. We will learn more about these signals later in this section. Try running the Bluespec Compiler again with the `-u` option. The compiler will indicate that the packages (i.e. the source BSV files) are up-to-date and that there is no need for recompilation.

Examining the Generated Verilog RTL

Take a look at the generated verilog for the GCD unit located in `mkGcd.v` (see Figure 4). Notice that the compiler generates Verilog-1995; it will not generate any Verilog-2001 constructs. The `mkGcd` Verilog module includes ports which correspond to the Bluespec interface methods. The `start` method has been implemented with two input ports for the `a_in` & `b_in` arguments and an `EN_start` & `RDY_start` pair for the method control flow. The Bluespec Compiler specifies the *micro-protocol* implemented by these *enable* and *ready* signals. The module signals to the caller that it is ready by setting the ready signal to one. The caller can then actually call the method by setting the enable signal to one. The caller is responsible for checking the ready signal before setting the enable signal. Bluespec action-value and action methods have both an enable and a ready signal. This is in contrast to Bluespec value methods which only have a ready signal. For example, the `result` method is a value method and thus is implemented with just a ready signal. When the `RDY_result` signal is one then the value is valid. The compiler adds a `CLK` port (for the clock signal) and `RST_N` port (for the active-low reset signal) to all generated modules.

Because we set the `-keep-fires` command line option, the compiler will generate a `CAN_FIRE` & `WILL_FIRE` signal pair for each action-value method, action method, and rule. Without the `-keep-fires` option, the compiler would optimize many of these signals away. The `CAN_FIRE` signal will be one whenever the corresponding method or rule is able to fire that cycle. In other words, the `CAN_FIRE` signal reflects the status of the predict (including implicit and explicit conditions) for the corresponding method or rule. The `WILL_FIRE` signal will be one whenever the corresponding method or rule actually does fire that cycle. The `CAN_FIRE` signals and `WILL_FIRE` signals are the inputs and outputs for the Bluespec generated scheduler. Notice that the scheduler is just combinational logic inside the `mkGcd` module. If a rule is able to fire but does not because of a conflict then the `CAN_FIRE` signal will be one but the `WILL_FIRE` signal will be zero. You should be able to recognize the predicates for the `subtract` and the `swap` rules. By examining the `WILL_FIRE` signals we can see that the compiler has generated a schedule such that both rules always fire when enabled.

By default, the compiler flattens the entire design into a single Verilog module. Although the generated net names include some of the original BSV hierarchy information, this flattened design can be very difficult to debug. We can use the `(* synthesize *)` attribute before a BSV

```
module mkGcd( CLK, RST_N, start_a_in, start_b_in, EN_start, RDY_start, result, RDY_result);
  input           CLK, RST_N;
  input  [31 : 0] start_a_in;   // Start method input operand
  input  [31 : 0] start_b_in;   // Start method input operand
  input           EN_start;     // Start method enable signal
  output          RDY_start;    // Start method ready signal
  output [31 : 0] result;       // Result method return value
  output          RDY_result;   // Result method ready signal

  // Register A            // Register B
  reg  [31:0] a;           reg  [31:0] b;
  reg  [31:0] a$D_IN;      wire [31:0] b$D_IN;
  wire        a$EN;        wire        b$EN;

  wire a_SLT_b___d3 = (a ^ 32'h80000000) < (b ^ 32'h80000000) ;

  // Start action method
  wire RDY_start       = ( b == 32'd0 ) ;
  wire CAN_FIRE_start  = EN_start ;
  wire WILL_FIRE_start = EN_start ;

  // Result value method
  wire        RDY_result = ( b == 32'd0 );
  wire [31:0] result     = a ;

  // Rule RL_subtract
  wire CAN_FIRE_RL_subtract  = !a_SLT_b___d3 && b != 32'd0 ;
  wire WILL_FIRE_RL_subtract = CAN_FIRE_RL_subtract ;

  // Rule RL_swap
  wire CAN_FIRE_RL_swap  = a_SLT_b___d3 && b != 32'd0 ;
  wire WILL_FIRE_RL_swap = CAN_FIRE_RL_swap ;

  // Register A mux and enable logic
  always @( EN_start or start_a_in or WILL_FIRE_RL_swap or b or WILL_FIRE_RL_subtract )
  begin
    case (1'b1) // synopsys parallel_case
      EN_start              : a$D_IN = start_a_in;
      WILL_FIRE_RL_swap     : a$D_IN = b;
      WILL_FIRE_RL_subtract : a$D_IN = a - b;
      default: a$D_IN = 32'hAAAAAAAA /* unspecified value */ ;
    endcase
  end
  assign a$EN = EN_start || WILL_FIRE_RL_swap || WILL_FIRE_RL_subtract ;

  // Register B mux and enable logic
  assign b$D_IN = EN_start ? start_b_in : a ;
  assign b$EN   = EN_start || WILL_FIRE_RL_swap ;

  // Register State
  always @( posedge CLK ) begin
    if      ( !RST_N ) b <= 32'd0;
    else if ( b$EN   ) b <= b$D_IN;
    if      ( a$EN   ) a <= a$D_IN;
  end

endmodule
```

Figure 4: Generated Verilog RTL for the `mkGcd` Bluespec Module

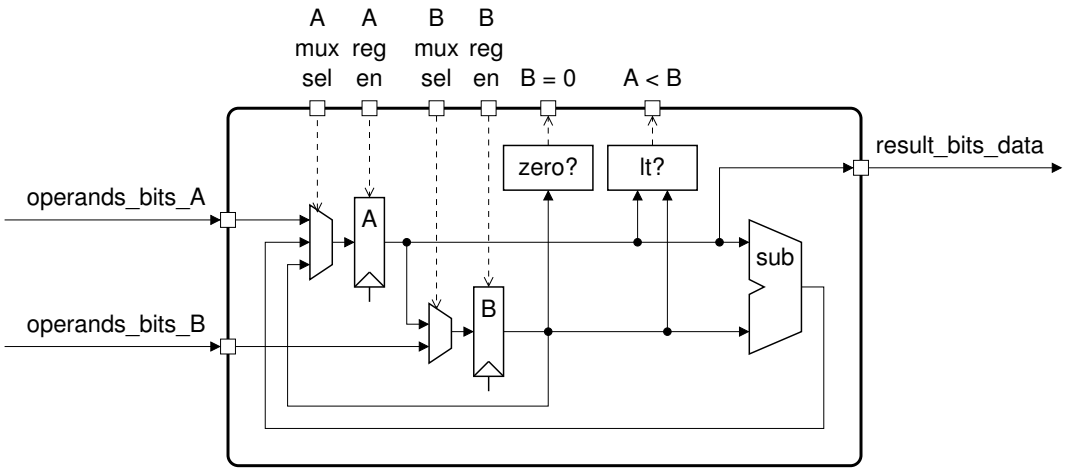

Figure 5: Datapath for hand written RTL model of greatest common divisor unit

module definition to tell the compiler that we want it to create a separate Verilog module (in a separate Verilog file) for that specific BSV module. In this example we used a `(* synthesize *)` attribute for the `mkGcd` module and as a consequence we generated two separate Verilog modules: the `mkGcd` module in `mkGcd.v` and the `mkGcdTH` module in `mkGcdTH.v`. If we were to comment out the `(* synthesize *)` attribute, then the compiler would generate a single flattened Verilog module. We will attempt to preserve the module hierarchy as much as possible with regular use of the `(* synthesize *)` attribute. Unfortunately, the Bluespec Compiler cannot create separate Verilog modules if the corresponding BSV modules have parameters or if the module interface has parameters. Even so, we should try to use the `(* synthesize *)` attribute whenever possible.

Now let's examine the generated combinational and sequential logic and compare it to the hand written Verilog RTL located in `examples/gcd-v/src`. Figure 5 shows the hand written datapath. Can you find the A & B registers and the A & B muxes in the generated Verilog? Notice that for the A mux, the compiler has generated a case statement where the case expression is a constant and the case items are expressions. Although this is an awkward way to represent a mux, it is legal Verilog (see Section 9.5 of the *Verilog 2001 Language Description* (`verilog-language-spec-2001.pdf`)). Now look for the less-than comparison and the zero equality check in the generated Verilog. The compiler has factored out the combinational logic for the less-than comparison, but the zero equality check hardware is duplicated four times in the expressions for `RDY_start`, `RDY_result`, `WILL_FIRE_RL_subtract`, and `WILL_FIRE_RL_swap`. Hopefully the RTL-to-Gates synthesis tool will optimize this into a single zero equality check.

Although the Bluespec Compiler will do its best to refactor common combinational logic, sometimes we need to be more explicit about what resources we would like to share among the rules. We can do this by creating a helper combinational module as shown in the `mkGcdWithAlu` module. The `mkGcdAlu` helper module includes only value methods and no state. We have added two new attributes to the `mkGcdAlu` module. The `(* always_ready *)` attribute tells the compiler to optimize away the ready signal since all of the module's methods should always be ready. The compiler will statically check if all of the methods are indeed always ready, and it will produce an error if this is not so.

The following commands will rerun the compiler with the new version of the GCD unit.

```
% rm -rf *.bi *.bo *.v
% perl -i -pe 's/mkGcd\(\)/mkGcdWithAlu\(\)/' mkGcdTH.bsv
% bsc -u -keep-fires -verilog -g mkGcdTH mkGcdTH.bsv
```

If you examine the generated Verilog for the `mkGcdWithAlu` module you will notice that zero equality check logic is no longer duplicated. Instead the various methods and rules share the combinational `mkGcdAlu` module to perform the less-than comparison, the zero equality check, and the subtraction. This idiom of refactoring shared combinational logic into a helper module is useful way to reduce the area of a design.

Simulating the Generated Verilog RTL

We can use Synopsys VCS to simulate the generated Verilog RTL. See *Tutorial 1: Simulating Verilog RTL Using Synopsys VCS* for more information about using VCS. The following command will compile the Verilog into an simulator executable and then run the simulator.

```
% vcs -PP mkGcdTH.v mkGcdWithAlu.v mkGcdAlu.v ../../src/mkGcdTH_wrapper.v
% ./simv
```

The test harness will print out some information about whether or not each test passed or failed. We can use the Synopsys VirSim waveform viewer to visualize the GCD unit in action. The following command will start the waveform viewer with the appropriate VPD file.

```
% vcs -RPP +vpdfile+vcdplus.vpd
```

Add the following signals to the waveform viewer.

- `mkGcdTH_wrapper.gcdTH.gcd.RST_N`
- `mkGcdTH_wrapper.gcdTH.gcd.CLK`
- `mkGcdTH_wrapper.gcdTH.gcd.RDY_start`
- `mkGcdTH_wrapper.gcdTH.gcd.EN_start`
- `mkGcdTH_wrapper.gcdTH.gcd.start_a_in`
- `mkGcdTH_wrapper.gcdTH.gcd.start_b_in`
- `mkGcdTH_wrapper.gcdTH.gcd.WILL_FIRE_RL_subtract`
- `mkGcdTH_wrapper.gcdTH.gcd.WILL_FIRE_RL_swap`
- `mkGcdTH_wrapper.gcdTH.gcd.a`
- `mkGcdTH_wrapper.gcdTH.gcd.b`
- `mkGcdTH_wrapper.gcdTH.gcd.RDY_result`
- `mkGcdTH_wrapper.gcdTH.gcd.result`

Figure 6 shows the waveforms in more detail. You can see that on reset the start method is ready and enabled. The input operands 27 and 15 are clocked into the A and B registers. The `WILL_FIRE` signals show the `subtract` and `start` rules firing and the A and B registers being

updated appropriately. When B is zero, the `result` method and the `start` method become ready and new input operands are clocked into the A and B registers.

The primary methodology for debugging our Bluespec designs will be to analyze the generated Verilog and to carefully examine the `RDY`, `EN`, `CAN_FIRE`, and `WILL_FIRE` signals. We can use these signals to determine if rules are firing when desired and if methods are enabled when needed.

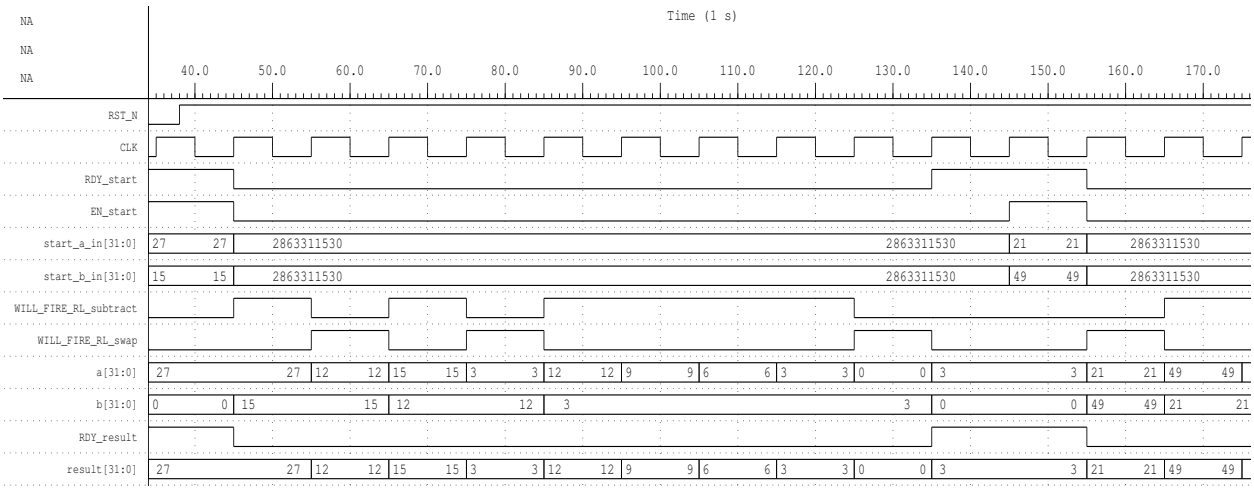

Figure 6: Waveforms for the GCD Unit

Automating the Process with Makefiles

We can automate the BSV build process using makefiles. Take a look at the build directories located in `examples/gcd-bsv/build`. The `bsc-compile` build directory is used to run the Bluespec Compiler, while the `vcs-sim-rtl` build directory is used to run Synopsys VCS. The following commands will build the GCD unit and test it.

```
% pwd
tut8/examples/gcd-bsv/build
% cd bsc-compile
% make compile
% cd ../vcs-sim-rtl
% make sim-rtl
% make run-tests
```

There is a toplevel makefile which can run both the Bluespec Compiler and Synopsys VCS. The following commands will clean all of the build directories, automatically run the Bluespec Compiler and VCS if needed, and then test the GCD unit. The toplevel `clean` target will delete generated content in *all* the subdirectories so use it carefully.

```
% pwd
tut8/examples/gcd-bsv/build
% make clean
Are you sure you want to do a FULL clean? [Y/N] Y
% make run-tests
```

Using Toy Examples to Illustrate Rule Scheduling

In this section we will use several toy examples to illustrate how the Bluespec Compiler schedules multiple rules to fire in the same clock cycle while still preserving the correctness of the program. Before presenting the toy examples, we need to review the general strategy for rule scheduling. Assume we have a design with two rules named `r1` and `r2`. Due to GAA semantics all implementations of the design must *appear* to execute the two rules in some *logical sequential order*: either rule `r1` first and then rule `r2` (denoted as `r1` < `r2`), or rule `r2` first and then rule `r1` (denoted as `r2` < `r1`). If an implementation actually did execute all rules sequentially (i.e. one rule each cycle) it would be extremely slow). We want to exploit the concurrency inherent in the hardware to execute multiple rules in parallel. Two rules can be executed in parallel if their effect on the design's state when executed in parallel is the same as a specific logical sequential ordering. If no such logical ordering exists then we say the rules *conflict* and thus they cannot be executed concurrently while still preserving correctness.

The Bluespec Compiler determines if two rules can be executed in parallel by examining how the rules' *read sets* and *write sets* intersect with each other. A rule's *read set* is the set of state from which the rule reads, and a rule's *write set* is the set of state to which a rule writes. Table 1 illustrates the sixteen possible ways the read and write sets for two rules can intersect.

Notice that whether or not the read sets for the two rules intersect (i.e. the two rules read the same state) has no bearing on the possibility of executing those rules concurrently. The table introduces some additional terminology to describe the various ways the two rules can interact. Two rules are *conflict free* (CF) if each rule's write set does not intersect with the other rule's read and write

Type	**r1 Rd** ∩ **r2 Rd**	**r1 Wr** ∩ **r2 Rr**	**r1 Wr** ∩ **r2 Wr**	**r1 Rd** ∩ **r2 Wr**	**Possible Logical Sequential Orderings**	**Example**	
CF	X				`r1` < `r2` or `r2` < `r1`	`r1` : x <= 5	`r2` : y <= 6
SC1	X			Y	`r1` < `r2`	`r1` : x <= y	`r2` : y <= 6
SC2	X		Y		`r1` < `r2` or `r2` < `r1`	`r1` : x <= 5	`r2` : x <= 6
SC1	X		Y	Y	`r1` < `r2`	`r1` : x <= x+1	`r2` : x <= 6
SC1	X	Y			`r2` < `r1`	`r1` : x <= 5	`r2` : y <= x
C	X	Y		Y	none	`r1` : x <= y	`r2` : y <= x
SC1	X	Y	Y		`r2` < `r1`	`r1` : x <= 5	`r2` : x <= x+1
C	X	Y	Y	Y	none	`r1` : x <= x+1	`r2` : x <= x+2

Table 1: Possible interactions between the read sets (Rd) and write sets (Wr) of two rules named `r1` and `r2`. A Y indicates that the corresponding sets do intersect, nothing indicates that the corresponding sets do not intersect, and a X indicates that it doesn't matter if the corresponding sets intersect or not. CF = Conflict Free, SC1 = Sequentially Composable with One Possible Ordering, SC2 = Sequentially Composable with Two Possible Orderings, C = Conflict.

sets. If `r1` and `r2` are conflict free then executing them in parallel has the exact affect on the state as if we had executed `r1` before `r2` or `r2` before `r1`. In other words, either logical sequential ordering is an acceptable explanation of the parallel execution. Two rules *conflict* (C) if there is no logical sequential ordering which explains the parallel execution of those rules. If both rules read and write the same state then it is not possible to execute them in parallel. For example, if rule `r1` is (x <= y) and rule `r2` is (y <= x) then we cannot fire these rules in parallel. If we did we would essentially be performing a swap with out temporary state, but since our GAA semantic model is completely serialized this is not possible. We could of course perform a swap by putting both expressions in the same rule.

Two rules which are not conflict free and also do not conflict are called *sequentially composable* (SC). Although the read and write sets of sequentially composable rules intersect, there is still a possible logical sequential ordering which will adequately explain their parallel execution. For example, if rule `r1` is (x <= y) and rule `r2` is (y <= 6) then execution them in parallel has the same effect on the design's state as if we executed `r1` before `r2`. This is because in hardware we do all the reads before we do any writes, so executing them in parallel causes both rules to get the original value of y. Notice that `r2` < `r1` is *not* an acceptable logical sequential ordering, since this ordering would require rule `r1` to see the new value of y. This is just not possible in hardware; we cannot forward the new value of y to rule `r1` within the same clock cycle.

Just because two rules write the same state does not imply that those rules conflict. Two rules can be *mutually exclusive* meaning that the predicates for the two rules cannot be true at the same time. Mutually exclusive rules cannot be enabled at the same time and thus regardless of whether or not they write the same state they do not conflict. There is also a more subtle example of two non-conflicting rules which write the same state. A special form of sequential composability occurs if just the write sets of the two rules intersect (noted in the table as SC2). In this case although the final state is different depending on the logical sequential ordering we choose, either logical sequential ordering is an acceptable explanation of the parallel execution of these rules.

It is important to note that using read/write sets to determine which rules can be executed in parallel is a conservative approach. For example, if rule `r1` is (x <= y) and rule `r2` is (y <= x) then we conservatively assume that this is a conflict. But what if the x and y registers both hold the same value? Then rule `r1` and rule `r2` can indeed execute in parallel, and either logical sequential ordering is an acceptable explanation for their parallel execution. It is very difficult (possibly impossible) for the compiler to statically determine this type of scheduling. If we wanted to exploit this parallelism the scheduler hardware would become significantly more complicated. Thus the Bluespec Compiler restricts itself to the more conservative analysis depicted in Table 1.

Figure 7 shows an example module template and five different definitions for rule `r1` and rule `r2`. Create a temporary build directory and then use your favorite text editor to create six BSV files with the examples in Figure 7 named `ex1.bsv`, `ex2.bsv`, etc. Note that since the Bluespec Compiler requires that the finale name and package be the same, you will need to substitute the appropriate example number for `N` in the template.

Module Template

```
package exN;

module exN( Empty );

  Reg#(Bit#(8)) z <- mkReg(0);
  Reg#(Bit#(8)) x <- mkReg(0);
  Reg#(Bit#(8)) y <- mkReg(0);

  rule incZ;
    if ( z < 15 )
      z <= z + 1;
    else
      $finish;
  endrule

  // Insert rules r1 and r2 here

endmodule

endpackage
```

Example 1

```
rule r1 ( z > 5 );
  x <= x + 1;
endrule

rule r2 ( z > 10 );
  y <= y + 2;
endrule
```

Example 2

```
rule r1 ( z > 5 );
  x <= x + 1;
endrule

rule r2 ( z > 10 );
  x <= x + 2;
endrule
```

Example 3

```
rule r1 ((z > 5) && (z <= 10));
  x <= x + 1;
endrule

rule r2 ( z > 10 );
  x <= x + 2;
endrule
```

Example 4

```
rule r1 ( z > 5 );
  x <= x + 1;
endrule

rule r2 ( z > 10 );
  y <= x + 2;
endrule
```

Example 5

```
rule r1 ( z > 5 );
  x <= y + 1;
endrule

rule r2 ( z > 10 );
  x <= y + 2;
endrule
```

Figure 7: Various examples illustrating different rule interactions

Use the Bluespec Compiler as follows for each example (where N is the example number).

```
% pwd
temp
% ls
ex1.bsv ex2.bsv ex3.bsv ex4.bsv ex5.bsv
% cp /mit/6.375/tools/bluespec/current/lib/Verilog/main.v .
% bsc -show-schedule -show-rule-rel RL_r1 RL_r2 -show-rule-rel RL_r2 RL_r1 \
      -verilog -g exN exN.bsv
```

The `-show-schedule` command line option tells the compiler to output information about the rule and method scheduling in the design. For each rule, the compiler identifies the predicate and a list of *blocking* rules. If rule `r1` blocks rule `r2` then these rules conflict; furthermore, if these rules are both enabled during the same cycle then rule `r1` will take priority over rule `r2`. The format for rule schedule information is shown below.

`Rule: r2`	Rule name
`Predicate: ! (z.read <= 10)`	Rule predicate (including implicit conditions)
`Blocking rules: (none)`	Which rules conflict with & have priority than this rule

The `-show-rule-rel` command line option tells the compiler to output information about a specific pairwise rule relationship. Notice that the `RL_` prefix is required when specifying rule names. For each pair of rules, the compiler identifies if the predicates are disjoint (or in other words if the two rules are mutually exclusive), if the rules are conflict free, and if the rules are sequentially composable. The format for the pairwise rule information is shown below.

`Scheduling info for rules ‘‘RL_r1’’ and ‘‘RL_r2’’;`	
`  predicates are not disjoint`	Indicates if rules are mutually exclusive
`    no <> conflict`	If no ¡¿ conflict, then rules are conflict free?
`    no < conflict`	If no ¡ conflict, then rules are not sequentially composable?
`    no resource conflict`	See BSC user guide
`    no cycle conflict`	See BSC user guide
`    no <+ conflict`	See BSC user guide

After compiling the Verilog we can use a generic Verilog wrapper file provided with the Bluespec installation to create an executable simulator with VCS.

```
% cp /mit/6.375/tools/bluespec/current/lib/Verilog/main.v .
% vcs exN.v /main.v +define+TOP=exN
% ./simv +bscvcd
% vcs -RPP +vcdfile+dump.vcd
```

The scheduling information provided by the compiler is a little confusing, so it is important to interpret it correctly. Take a close look at the scheduling information displayed by the compiler for each example.

Example 1: Conflict Free

The compiler notes that the rules are not mutually exclusive and that they are conflict free. We can see this because their are no blocking rules and there is no <> conflict. Simulate the example and verify that rules are able to fire in parallel when their predicates are satisfied

Example 2: Conflict
From Table 1 we know that these rules should conflict. The scheduling information reports that rule `r2` blocks rule `r1`. The pairwise rule information shows that the rules are not conflict free. It also shows that there is a < conflict for `r1` relative to `r2` and a < conflict for `r2` relative to `r1`. This essentially means that two logical orderings are required: `r1` < `r2` and `r2` < `r1`. Since both of these logical orderings cannot be satisfied at the same time, these rules conflict. The compiler also displays a warning because it had to make an arbitrary decision concerning which rule should take priority. Usually these warnings indicate a design error since we should not be relying on the compiler to arbitrarily pick the correct priority for the rules. Later in this section we will see how to use a scheduling attribute to explicitly tell the compiler the desired rule priorities. Simulate the example and verify that when `CAN_FIRE` is true for both rules, `WILL_FIRE` is only true for rule `r2`.

Example 3: Mutually Exclusive
Compare the compiler output from example three to that from example two. These are identical examples except for the rule predicates. In example three we have made the predicates mutually exclusive and as a consequence there is no longer a conflict between these two rules. Simulate the example and verify that only one rule's `CAN_FIRE` signal is true per cycle.

Example 4: Sequentially Composable (SC1)
From Table 1 we know that these rules are sequentially composable (SC1) and that the appropriate logical sequential ordering is `r1` < `r2`. We can see that in the pairwise rule output from the compiler. It shows that the rules are not conflict free, but that a < conflict for `r1` relative to `r2` exists. Notice that there is no < conflict for `r2` relative to `r1` exists. Essentially the compiler is telling us that these rules are sequentially composable with a logical ordering of `r1` < `r2`. We can also see this in the "Logical execution order" section of the scheduling output. Simulate this example and verify that both rules fire in the same cycle when they are both enabled.

Example 5: Sequentially Composable (SC2)
From Table 1 we know that these rules are sequentially composable (SC2) and that either logical sequential ordering is appropriate. Although the compiler scheduling output is a little confusing, the compiler does produce a useful warning which states that rule `r2` will appear to fire before `r1` when both fire in the same clock cycle. Use VCS to simulate this example and verify that these two rules do indeed fire in the same cycle when they are both enabled.

We can use various scheduling attributes to help influence the scheduling of the rules in the design. These attributes are especially important when we want to avoid having the compiler make an arbitrary decision.

`(* descending_urgency = ''r1,r2'' *)`
This attribute should be placed before any rule is defined in a module. It specifies that if rule `r1` and rule `r2` conflict, then the compiler should give priority to rule `r1`.

`(* fire_when_enabled *)`
This attribute should be placed immediately before a rule definition. It asserts that the rule *must* fire when it its predicate (both the implicit and explicit conditions) is true. The compiler will statically check to see if this rule has a conflict which would prevent it from firing, and if so the compiler will report an error.

`(* no_implicit_conditions *)`
This attribute should be placed immediately before a rule definition. It asserts that any implicit conditions for this rule must always be true. In other words the explicit conditions should completely define the predicate for the rule. The compiler will statically verify that this condition is satisfied and report an error of necessary.

`(* always_enabled = ``method1,method2'' *)`
This attribute should be placed before a module definition. The compiler will not generate an enable signal for the listed methods, and thus the methods must be executed on every cycle. The compiler will statically verify that the method is indeed called every cycle. If no methods are listed, then this attribute applies to all the methods in the module.

`(* always_ready = ``method1,method2'' *)`
This attribute should be placed before a module definition. The compiler will not generate a ready signal for the listed methods, and thus each method's predicate must always be true. The compiler will statically verify this and report an error if necessary. If no methods are listed, then this attribute applies to all the methods in the module.

As a final example of rule and method scheduling, let's examine the compiler output for the GCD unit. The `schedule.rpt` make target in the `gcd-bsv/build/bsv-compile` build directory will generate a report containing the schedule information for all other rules and methods in the design. The following commands generate the scheduling report. It is important to make sure that the design will completely compile without errors before trying to generate a scheduling report since this make target needs to rebuild the design from scratch.

```
% pwd
tut8/examples/gcd-bsv/build/bsc-compile
% make compile
% make schedule.rpt
```

Examine the `schedule.rpt` file and find the scheduling information for the `subtract` and `swap` rules. You should be able to determine that these rules are mutually exclusive. You should also be able to clearly see the predicate for each of these rules.

An SMIPSv2 Multi-Cycle Processor using BSV

In this section we will take a look at a SMIPSv2 processor written in Bluespec System Verilog. Start by browsing through the BSV source code in `examples/smipsv2-4mcycle-bsv`. Figure 8 how the rules in the core interact. The core interface includes tohost/fromhost methods as well as a request/response main memory interface. The core implementation contains a multi-cycle (unpipelined) SMIPSv2 processor, a blocking instruction cache, a blocking data cache, and a memory arbiter.

The processor includes three rules: a `pcgen` rule which issues instruction requests to the instruction cache, a `exec` rule which does all the work involved in instruction execution, and a `writeback` rule which writes load data returning from the data cache into the register file. The processor is unpipelined which means that only one instruction is executed at a time. A state register indicates which rule is currently active. ALU instructions only fire the `pcgen` rule and the `exec` rule which

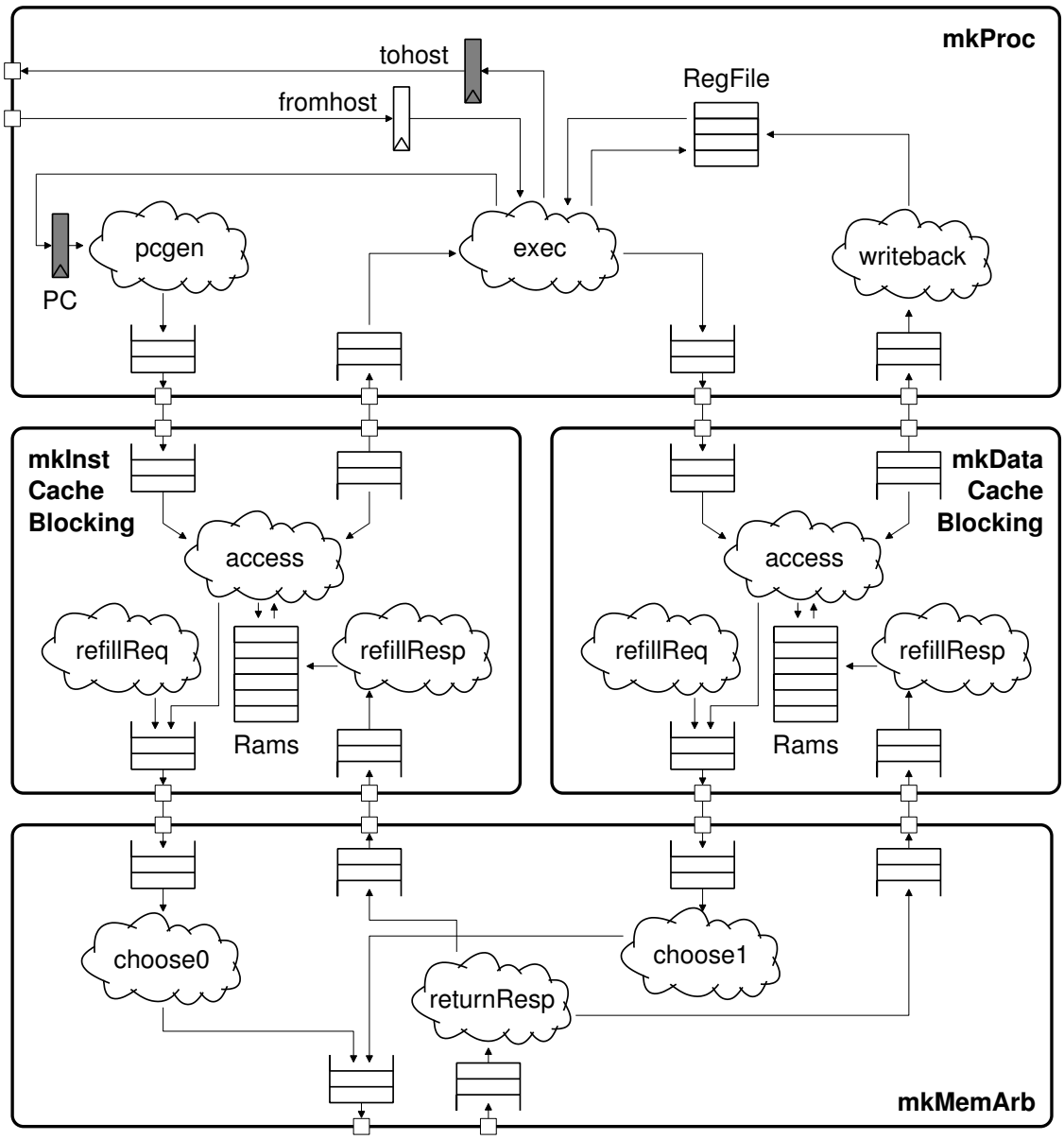

Figure 8: Cloud diagram for SMIPSv2 multi-cycle (unpipelined) processor with blocking caches

writes the ALU result into the register file. Load/Store instructions fire all three rules. There is no register file write conflict since the `exec` rule and the `writeback` rule are mutually exclusive. The `exec` rule also takes care of updating the PC.

The processor includes two ports to memory: an instruction port and a data port. These ports are written using the subinterfaces and the Bluespec library `Client/Server` interface. For more information consult the *Bluespec Language Reference* (`bsc-reference-guide.pdf`). The memory interface supports tagged loads and stores. The tag can help the processor manage memory systems which return responses out-of-order. For the this tutorial, all requests are returned in-order so the tag is not absolutely required. Although the processor uses the tag for load requests to indicate the destination register, we could just as easily use a separate queue internal to the processor to manage the register writeback specifiers.

The only difference between the instruction cache and the data cache, is that the instruction cache does not support stores. Both caches have single 32-bit word cache lines and are blocking. A blocking cache means that the cache completely finishes processing one memory request before starting the next memory request. Each cache includes a state machine which invalidates all cache lines on reset. The `access` rule processes an incoming memory request and checks the tag RAM to see if it the desired data is currently in the cache. If the request is a hit, then the `access` rule gets the appropriate data from the data RAM and enqueues a memory response in the response queue. If the request is a miss, then the `access` rule first checks to see if the victim cache line is valid and if so it enqueues a store memory request into the queue to the memory arbiter. The next cycle the `refillReq` rule will enqueue a refill request into the queue to the memory arbiter. If the victim cache line is invalid, then the `access` rule can immediately enqueue a refill request into the queue to the memory arbiter. When the refill request eventually returns from main memory, the `refillResp` rule writes the cache line in the data RAM. The next cycle the `access` rule will now hit in the cache and return the data to the processor. It is essential that the cache lines are marked invalid when the processor is reset. To achieve this, the cache includes a small state machine which steps through and invalidates each cache line. As a consequence, the processor cannot start execution for several hundred cycles after reset.

The memory arbiter uses a round-robin arbiter to decide which cache can access the off-chip main memory port. The explicit condition for the `choose` rules includes some state which indicates which request should have access to the main memory port. After making a request, the state is changed so that the other request will have priority on the next cycle. The memory arbiter uses the tag in its main memory requests to indicate which cache made the request. The arbiter can then use this tag when the response returns to know where to send the response.

The core implementation uses a simple pattern for the design of the various module interfaces. Low-level modules such as the register file use standard methods, but higher-level modules such as the processor, the caches, and the memory arbiter exclusively use `Get/Put` subinterfaces. This approach allows us to use the `mkConnectable` function to structurally connect high-level modules.

Many of the FIFOs in the design are actually bypass FIFOs (`BFIFOs`). These FIFOs include a combinational path from the `enq` method to the `deq` method so that these methods can fire in parallel. The corresponding logical sequential ordering is `enq` < `deq` which means that we can pass data from the `enq` method to the `deq` method within the same cycle. Bypass FIFOs eliminate extra dead cycles between the various high-level module interfaces.

The following commands will build the processor, run the assembly tests, and then evaluate the performance of the processor.

```
% pwd
tut8/examples/smipsv2-4mcycle-bsv/build
% make run-asm-tests
% make run-bmarks-perf
```

As with previous projects, you can use Synopsys VirSim to view waveforms corresponding to a specific program execution. For example, the following commands generate a VPD file for the `smipsv2_addiu.S` test program and then launch the waveform viewer.

```
% pwd
tut8/examples/smipsv2-4mcycle-bsv/build
% cd vcs-sim-rtl
% ./simv +exe=smipsv2_addiu.S.vmh
% vcs -RPP +vpdfile+vcdplus.vpd
```

In addition to using the waveform viewer, we have also provided some infrastructure for producing text traces of the processor. If you examine the BSV source for the core you will see various uses of the `traceTiny()` and the `traceFull()` functions. These trace functions output a trace tag and some trace data. This is the information you will see being displayed at the console when you run the simulator. The `bsv-trace.pl` Perl script can turn this trace output into a clean text trace format with one cycle per line. The script takes a configuration script as input which describes how to transform the trace output. For example, the following commands will produce the trace output which is partially shown in Figure 9.

```
% pwd
tut8/examples/smipsv2-4mcycle-bsv/build/vcs-sim-rtl
% ./simv +exe=smipsv2_lw.S.vmh > trace.out
% bsv-trace.pl proc-trace.cfg trace.out
```

The trace output shows the current PC and which rule is executing in the processor (`P` = `pcgen`, `X` = `exec`, `W` = `writeback`). The trace output has columns for the instruction cache, the data cache, the memory arbiter, and the main memory respectively. The current design has a single cycle main memory.

Notice that ALU instruction only fire the `pcgen` and the `exec` rules, while the load instruction fires all three rules. The load instruction hits in the data cache (indicated with a `h`), and the overall latency is five cycles. You should be able to see that the instruction fetch is hitting in the cache (indicated with a `h`) for several instructions until the processor tries to fetch the instruction at address 0x0000124c. This request misses in the instruction cache (indicated with a `M`) and causes a refill request to go through the memory arbiter and out to main memory. When the response from main memory returns to the instruction cache, the cache processes the refill (indicated with a `R`) and now the original instruction fetch hits in the cache.

We encourage you to make use of the text tracing infrastructure in your own designs, since it can help create more informative traces than the raw waveforms.

We can use the same infrastructure as in previous labs to synthesize and place+route the design. The following commands will run the appropriate tools. We can also just use the `enc-par` make target and the toplevel makefile will run the synthesis and floorplanning steps as necessary.

```
% pwd
tut8/examples/smipsv2-4mcycle-bsv/build
% make dc-synth
% make enc-fp
% make enc-par
```

```
CYC: 2128 pc=0000122c P  [100|      ] [   |      ] [   |   |   ] [   |   ]
CYC: 2129 pc=            [   |100  h] [   |      ] [   |   |   ] [   |   ]
CYC: 2130 pc=         X  [   |      ] [   |      ] [   |   |   ] [   |   ] sll r0, r0, 0
CYC: 2131 pc=00001230 P  [100|      ] [   |      ] [   |   |   ] [   |   ]
CYC: 2132 pc=            [   |100  h] [   |      ] [   |   |   ] [   |   ]
CYC: 2133 pc=         X  [   |      ] [104|      ] [   |   |   ] [   |   ] lw r4, 0x0004(r2)
CYC: 2134 pc=            [   |      ] [   |104  h] [   |   |   ] [   |   ]
CYC: 2135 pc=         W  [   |      ] [   |      ] [   |   |   ] [   |   ]
CYC: 2136 pc=00001234 P  [100|      ] [   |      ] [   |   |   ] [   |   ]
CYC: 2137 pc=            [   |100  h] [   |      ] [   |   |   ] [   |   ]
CYC: 2138 pc=         X  [   |      ] [   |      ] [   |   |   ] [   |   ] lui r29, 0xff00
CYC: 2139 pc=00001238 P  [100|      ] [   |      ] [   |   |   ] [   |   ]
CYC: 2140 pc=            [   |100  h] [   |      ] [   |   |   ] [   |   ]
CYC: 2141 pc=         X  [   |      ] [   |      ] [   |   |   ] [   |   ] ori r29, r29, 0xff00
CYC: 2142 pc=0000123c P  [100|      ] [   |      ] [   |   |   ] [   |   ]
CYC: 2143 pc=            [   |100  h] [   |      ] [   |   |   ] [   |   ]
CYC: 2144 pc=         X  [   |      ] [   |      ] [   |   |   ] [   |   ] bne r4, r29, 0x0013
CYC: 2145 pc=00001240 P  [100|      ] [   |      ] [   |   |   ] [   |   ]
CYC: 2146 pc=            [   |100  h] [   |      ] [   |   |   ] [   |   ]
CYC: 2147 pc=         X  [   |      ] [   |      ] [   |   |   ] [   |   ] addiu r5, r5, 0x0001
CYC: 2148 pc=00001244 P  [100|      ] [   |      ] [   |   |   ] [   |   ]
CYC: 2149 pc=            [   |100  h] [   |      ] [   |   |   ] [   |   ]
CYC: 2150 pc=         X  [   |      ] [   |      ] [   |   |   ] [   |   ] addiu r6, r0, 0x0002
CYC: 2151 pc=00001248 P  [100|      ] [   |      ] [   |   |   ] [   |   ]
CYC: 2152 pc=            [   |100  h] [   |      ] [   |   |   ] [   |   ]
CYC: 2153 pc=         X  [   |      ] [   |      ] [   |   |   ] [   |   ] bne r5, r6, 0xfff5
CYC: 2154 pc=0000124c P  [100|      ] [   |      ] [   |   |   ] [   |   ]
CYC: 2155 pc=            [   |     M] [   |      ] [100|   |   ] [100|   ]
CYC: 2156 pc=            [   |      ] [   |      ] [   |   |100] [   |100]
CYC: 2157 pc=            [   |    R ] [   |      ] [   |   |   ] [   |   ]
CYC: 2158 pc=            [   |100  h] [   |      ] [   |   |   ] [   |   ]
CYC: 2159 pc=         X  [   |      ] [   |      ] [   |   |   ] [   |   ] lui r4, 0x0000
CYC: 2160 pc=00001250 P  [100|      ] [   |      ] [   |   |   ] [   |   ]
CYC: 2161 pc=            [   |     M] [   |      ] [100|   |   ] [100|   ]
CYC: 2162 pc=            [   |      ] [   |      ] [   |   |100] [   |100]
CYC: 2163 pc=            [   |    R ] [   |      ] [   |   |   ] [   |   ]
CYC: 2164 pc=            [   |100  h] [   |      ] [   |   |   ] [   |   ]
CYC: 2165 pc=         X  [   |      ] [   |      ] [   |   |   ] [   |   ] addiu r4, r4, 0x12b0
```

Figure 9: Trace output from SMIPSv2 processor running the `smispv2_lw.S` test