FPGA Implementation of a Three-Stage SMIPSv2 Processor

6.375 Laboratory 3
March 11, 2009

In the first lab assignment, you built and tested an RTL model of a two-stage pipelined SMIPSv2 processor, while in the second lab, you built a Bluespec model of a 3 stage pipeline. In the third lab assignment, you will be using various commercial EDA tools to synthesize, place, and route your Bluespec design. You will then emulate your design on an FPGA platform. In addition, you will attempt to optimize your design to increase performance and/or decrease area. This lab has two objectives: the first is to introduce you to the tools you will be using in your final projects, and give you some intuition into how high-level hardware descriptions are transformed into functional FPGA implementations. The second objective is to deepen your understanding of Bluespec to the point where you can achieve full parallelism in your Bluespec designs. This will require you to understand rule conflicts and to resolve them with Wire primitives. CircBSV.tar, which is posted on the resources page of the course web-site, has some good examples.

Unlike the previous two labs, this lab will be done in groups of two. As such, the deliverables for the lab will be slightly more time consuming than the previous labs. Some problems are open-ended, requiring thoughtful implementation. Do not wait until the last minute to attempt them. Good team work is essential in completing the labs in a timely manner. The final lab problem is the most difficult, but it is independent of the other problems; we recommend partitioning the implementation this problem and the other four.

The deliverables for this lab are (a) your optimized Bluespec source, driver software, and all of the scripts necessary to completely generate your FPGA implementation checked into CVS, (b) written answers to the critical questions given at the end of this document, and (c) an FPGA demonstration. The lab assignment is due at the start of class on Friday, March 13. As usual, you must submit the written answers by hand in class. Demonstrations will be scheduled during TA office hours the following Monday.

Before starting this lab, it is recommended that you revisit the Bluespec model you wrote in the first lab. Take some time to clean up your code, add comments, and enforce a consistent naming scheme. You will find as you work through this lab assignment that having a more extensive module hierarchy can be very advantageous; initially we will be preserving some module boundaries through the tool-flow which means that you will be able to obtain performance and area results for only some modules. By default, the Bluespec compiler flattens all module definitions into one large module. Use of the `synthesize` annotation before a module implementation will direct BSC to generate modular verilog. Please refer to the language guide (/mit/6.375/doc/bsc-reference-guide.pdf) for more information on the use of this directive. Unfortunately, preserving the module hierarchy throughout the tool-flow means that the CAD tools will not be able to optimize across module boundaries. If you are concerned about this you can remove the (* synthesize *) directive from your Bluespec code, reverting to the Bluepsec compiler's default behavior of generating flattened Verilog. Additionally, you can instruct the CAD tools to flatten the design before optimization.

Figure 1 illustrates the 6.375 tool-flow we will be using for the third lab. You should already be familiar with the simulation and compilation paths from the first and second labs. The Verilog code used by the FPGA CAD tools will be generated by the Bluespec compiler from your BSV code. We will use Synopsys Synplify Pro to *synthesize* the design. Synthesis is the process of transforming

an RTL model into a gate-level model. For this lab assignment, Synplify Pro will take your RTL model of the SMIPSv2 processor as input, along with a description of the FPGA cell library, and it will produce a Verilog net-list of FPGA cell gates. Unlike an ASIC synthesis flow, Synplify Pro will also map high-level constructs, such as memories, onto specialized FPGA blocks.

We will use Altera Quartus II to *place and route* the design. Placement is the process by which each cell is positioned on the FPGA, while routing involves wiring the cells together using the various FPGA communication channels. Notice that you will be receiving feedback on the performance and resource consumption of your design after both synthesis and place+route - the results from synthesis are less realistic but are generated relatively rapidly, while the results from place+route are more realistic but require much more time to generate. Place+route for your two-stage SMIPSv2 processor will take on the order of 15 minutes, but for your projects it could take up to an hour.

Finally, after place+route we will tie the routed design and compiled C code together to produce a bit-file, again using Quartus II. This bit-file will be used to program the FPGA.

Disclaimer: This is the first time we are using FPGAs in 6.375, and we are using many new tool-chains and build paths. As a result, you may encounter “interesting” behaviors when attempting to complete the lab. If you encounter such behaviors, notify the TA immediately.

Another consequence of the novelty of the labs is that some aspects of the tools may be hard to understand. Should you have difficulty in figuring out what to do with a particular tool, contact the TA. The focus of the lab is digital design, not obscure compiler flags – do not get needlessly bogged down.

Each piece of the tool flow has its own build directory and we have scripted a basic tool chain for you. Please consult the following tutorials for more information on using the various parts of the tool flow.

- Tutorial 9: RTL-to-Gate Synthesis using Synopsys Synplify Pro
- Tutorial 10: Synthesis and Place and Route using Quartus II
- Tutorial 11: System Building with SOPC Builder
- Tutorial 12: Programming with the NIOS II IDE
- Tutorial 13: Importing IP with SOPC Builder **Required**

Getting Started

To begin the lab you will need to make use of the lab harness located in `/mit/6.375/lab-harnesses`. The lab harness provides makefiles, scripts, and test harnesses required to complete the lab. In order to get started, unpack the test harness, copy the src directory from your lab2 project and check everything into your CVS directory as lab3T. Assuming your lab2 code is located on 2008s/kfleming/lab2, the following commands extract the lab harness into your CVS directory, copy the relevant files from lab2 and adds the new project to CVS.

```
% cd 2008s/students/kfleming
% tar -xzvf /mit/6.375/lab-harnesses/lab3-harness.tgz
% cp lab2/src/*.* lab3/src/
```

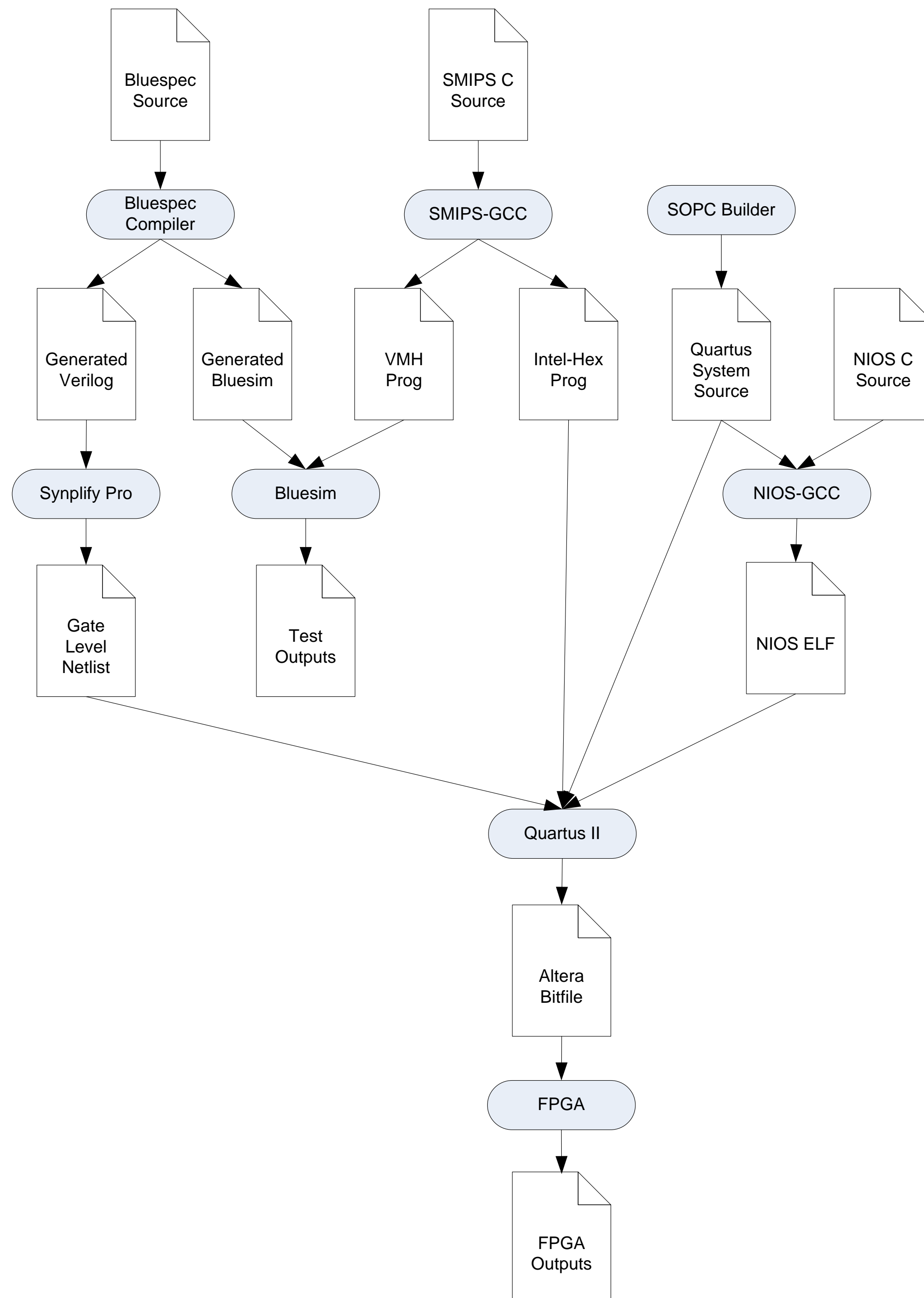

Figure 1: 6.375 Toolflow for Lab 3

```
<copy over any local tests if they exist>
<make sure you can still compile and run all the assembly tests>
% find lab3 | xargs cvs add
% cvs commit -m "Initial checkin"
```

The resulting `lab3` project directory resembles the structure you used in `lab2`. The primary change to the original directory structure is the introduction of the `syn` and `par`, which contain files necessary to build the physical implementation of your design. Although we have scripted the tool-flow, both Quartus and Synplify Pro have GUIs which you can use to examine these files. The `build` directory contains the following subdirectories which you will use when building your FPGA design.

- `bdir` - Bluespec generated intermediate files
- `vdir` - Bluespec generated Verilog
- `simdir` - Bluespec generated Bluesim simulation files
- `cdir` - Bluespec generated C codes (not used)
- `syn` - Synplify Pro Synthesis
- `par` - Quartus II Place and Route

We have provided you with a basic tool-chain, which should be capable of building your design, running it on the FPGA, and collecting run results. **You will have to make modifications to these script files as you push your design through the physical tool-flow.** The following command will build a bit-file, complete with code, which can be used to program the FPGA.

```
% pwd
2008s/students/kfleming/lab3/build
% make bitfile
```

Building an FPGA bitfile is a very computationally intensive task. For your simple three-stage SMIPS processor, synthesis and place+route can take anywhere from 10 minutes to 20 minutes. Budget your time accordingly.

FPGA System

Figure 2 shows a diagram of the Lab 3 system. In addition to the SMIPS processor, the system has a second, NIOS co-processor. In the system, the NIOS processor drives the SMIPS processor, collects performance information about the SMIPS processor, and acts as a rudimentary debugger for the SMIPS processor. As no cache-coherence mechanism is in operation, we recommend halting the SMIPS processor before taking any action from the NIOS that could affect the state of the SMIPS.

You will implement Lab 3 on the terasiC DE2-70 board, shown in Figure 3, which features a medium-sized Cyclone II FPGA and rich set of peripherals. Of the peripherals, we will use only the UART for `printf` and will ignore the rest of the peripherals for the lab, although for your projects, you may use any peripherals you see fit.

The 38-301 lab machines have been configured to interface directly with the DE2 boards for both programming and running the lab test-benches. On these machines, the following command should run the test programs on the FPGA and collect results over the serial line. Running each test-bench takes about 20 seconds.

```
% pwd
2008s/students/kfleming/lab3/build
% make run-asm-tests-fpga && run-bmarks-smips-fpga
```

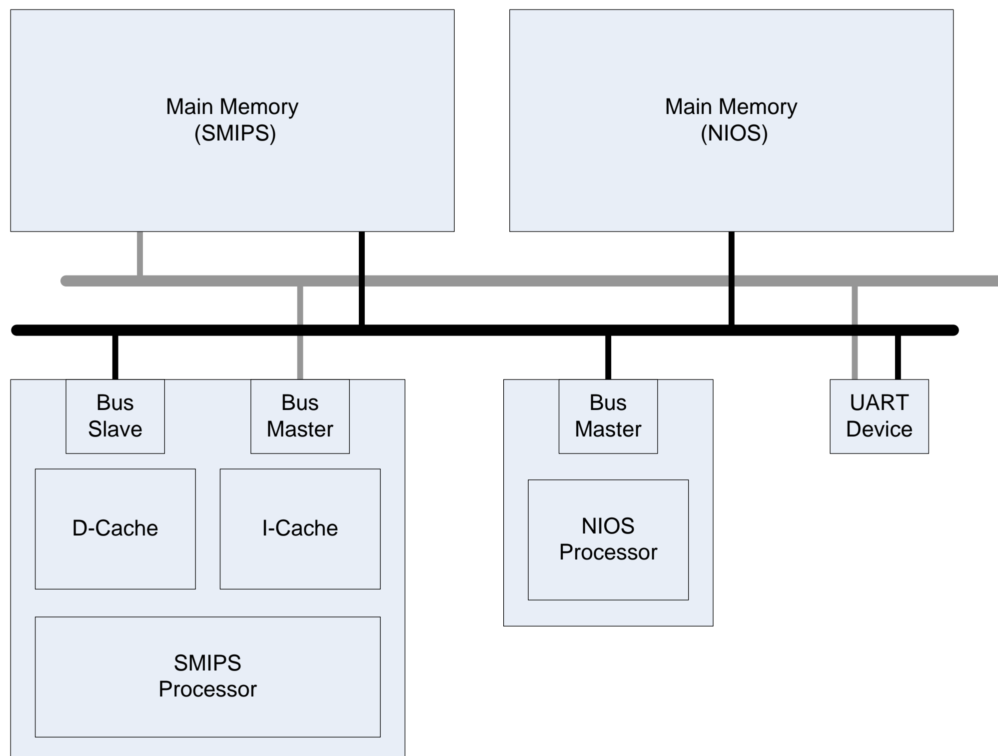

Figure 2: 6.375 System for Lab 3

During each test, two programs are run within the FPGA. The first is the SMIPS benchmark program, which you should not have to edit or compile yourself. The second program in the system is the SMIPS driver program, which is run on the NIOS-II processor. We have provided you with a simple driver program, located in `par/software`, which starts the SMIPS and waits until the SMIPS writes to the `toHost` register.

Although no modification is necessary to the provided program, you may want to make alterations to this program as you test out the new features of your implementation. Altera provide an Eclipse based IDE for editing NIOS programs, which can be invoked by typing `nios2-ide` into the command line. We have provided you with a basic software implementation:

```
% pwd
2008s/students/kfleming/lab3/par/software
% ls
  smipsV2  smipsV2_syslib
```

`smipsV2` contains the main source code, while `smipsV2_syslib` will build any necessary system libraries. You must *import* these two projects into the IDE, which can be done from the `File` menu. You may be asked to provide a *ptf* file. This file is generated by SOPC Builder and contains some metadata about the NIOS system:

```
% pwd
2008s/students/kfleming/lab3/par/
% ls *.ptf
```

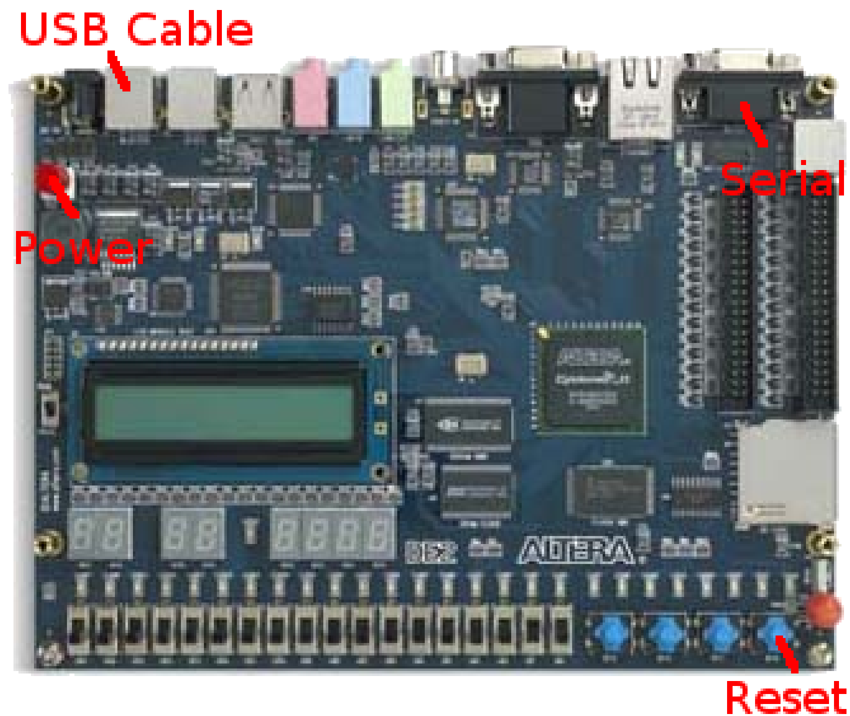

Figure 3: DE2-70 Board

`smipsV2System.ptf`

Building the `smipsV2` project will generate the hex files necessary to program the FPGA, once par is completed.See Tutorial 12 for more information on how to program the NIOS.

Debugging with CBus

Unlike the VCS or BlueSim simulators, where `$display` and waveform viewers may probe every wire at any time, it is not very easy to determine what is going on within a physical circuit. Often to debug physical systems, designers will run certain signals out to pins on the board. However, for complex systems, this approach may be too slow, low-level, and error-prone to be workable.

We therefore introduce the `CBus` as an alternative debugging mechanism. `CBus` is a simple bus mapping utility that can construct a bus interface across an entire design. We have implemented a collection of `CBus` modules, each of which has a some notion of address. For example, to add a read only register to the `CBus`, invoke `mkCBusWideRegR(addr,reg)`. This call will map `reg` into the `CBus` starting at address `addr`. The current address mapping can be found in the file `FPGATypes.bsv`. You may extend this mapping as you wish, but be aware that registers larger than `AvalonDataWidth`, 32 in this case, will be mapped across successive addresses. `CBus` performs no address conflict check, so it is up to you to ensure that your register mapping is valid. The source for our `CBus` modules and some C drivers for the modules can be found in the BSVCLib.

The `CBus` module collect builds a list of all `CBus` modules in the system and exposes this list using

`exposeCBusIFC`, typically in a top level module. At the top level, read and write queries may be fed into the `CBus`, which then maps the requests on to the modules it has collected. As a result of this collection behavior, modules which have `CBus` must be declared with the following syntax:

```
module [ModWithCBus#(AvalonAddressWidth,AvalonDataWidth)] mkProc( IProc );
```

Modules containing `CBus` are not separately synthesizable – should you require a synthesis boundary you must expose the `CBus`.

We attach our `CBus` interface to a soft-core co-processor, which provides us with debug facilities like `printf`. We could also implement more complex interactive debugging mechanisms using the same interface.

A number of debugging facilities and performance counters have been implemented for you using `CBus` and given you some simple C drivers for the software co-processor. Feel free to implement any debugging interfaces you require, but take care that your `CBus` address mappings do not conflict.

Lab Hints and Tips

This section contains several hints and tips which should help you in completing the lab assignment.

Tip 1: Start Early!

While compilation of the SMIPS processor for FPGA does not take a long time relative to other FPGA designs, it does lengthen the design feedback path substantially. If you start the lab late, you may not have enough time to debug the lab due to long compilation times.

Tip 2: Always Test Your Processor After Making Modifications

When pushing your processor through the physical tool-flow, it is common to make some changes to your RTL and then evaluate their impact on area, power, and performance. Always retest your processor after making changes and before starting the physical tool-flow. You can use the `run-asm-tests` make target to quickly verify that your processor is still functionally correct. Keep in mind, that a fast or small processor which is functionally incorrect is worse than a slow or large processor which works!

Tip 3: Reporting Clock Period

As discussed in the tutorials, you need to specify a clock period constraint during both synthesis and place+route. The tools will try and meet this constraint the best they can. If your constraint is too aggressive, the tools will take a very long time to finish. They may not even be able to correctly place+route your design. If your constraint is too conservative, the resulting implementation will be suboptimal. While the lab design can handle multiple clock domains, for this lab we will require that your design be capable of running at 50 MHz.

Even if your design does not meet the clock period constraint it is still a valid piece of hardware which will operate correctly at some clock frequency (it is just slower than desired). If your design

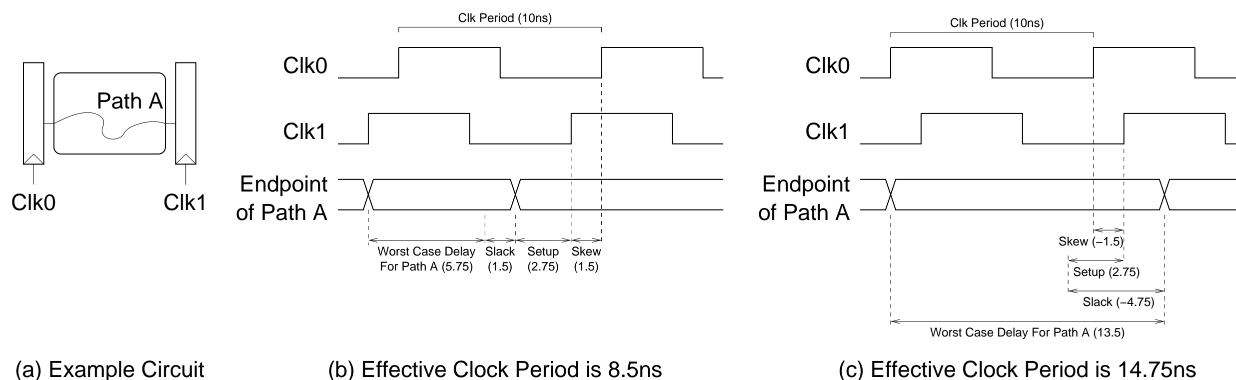

Figure 4: Determining your hardware's effective clock period

does not meet timing the tools will report a *negative slack*. Similarly, a design which makes the timing constraint but does so with a positive slack can run *faster* than the constrained clock period. For this lab we are more concerned about the *effective* clock period of your design as opposed to the clock constraint you set before synthesis. The effective clock period is simply the clock period constraint *minus the worst slack* ($T_{clk} - T_{slack}$). The synthesis and place+route timing reports are all sorted by slack so that the path with the worst slack is listed first. To determine the effective clock period for your design simply choose the smaller of the rising and falling edge slacks. Figure 4 illustrates two examples: one with positive slack and one with negative slack. In this example, our clock period constraint is 10 ns. In Figure 4(b), the post-place+route reports indicate a positive slack value of 1.5 ns and thus the effective clock period is 8.5 ns. In Figure 4(c), the post-place+route reports indicate a negative slack value of 4.75 ns and thus the effective clock period is 14.75 ns. Notice that the effective clock period in Figure 4(c) is *not* equal to the worst case combinational critical path (i.e. 13.5 ns). This is because we must also factor in setup time and clock skew.

Tip 4: Design Good Test-benches

Even with easy-to-use CBus interface and some detailed knowledge of what is going on inside of the processor, it is still challenging to obtain information from the FPGA. Save yourself the headache – use the FPGA to identify problems in your system, but try to debug in simulation. If new functionality is added, as in the critical thinking questions below, introduce a new test-bench to exercise it. Unit-testing modified modules is another good strategy to ensure that your changes are correct.

In complex systems, it is very easy for a small bug to cascade across the system quickly, obscuring its true origin. Although it is tempting to hack at a suspected bug immediately, a better approach is to introduce a test-vector that establishes the cause of the bug.

Tip 5: Use CVS to your advantage

It may make sense to archive functional versions of your processor in case something goes wrong. CVS tags are useful in this respect, allowing you to label certain code revisions for future reference.

Critical Thinking Questions

Question 1: Evaluate Your Baseline Processor

Push your baseline processor (no branch predictor, RWires) through the physical tool-flow and report the information listed below. You may have to synthesize some modules separately to obtain resource usage numbers. It is acceptable to remove `CBus` modules to obtain these results.

When reporting resource usage, be sure to list RAM and multiplier usage in addition to logic and register usage.

- Post-synthesis resource consumption of the register file, and data-path (excluding register file) `rpt_mkCoreFPGA.areasrr`
- Post-synthesis resource consumption of processor from `rpt_mkCoreFPGA.areasrr`
- Post-synthesis critical path and corresponding effective clock period in nanoseconds from `smipsV2.srr`
- Post-place+route resource usage of processor (only the SMIPS, not the entire system) `smipsV2System.map.rpt`
- Post-place+route critical path and corresponding effective clock period in nanoseconds from `smipsV2System.sta.rpt`

Your post-place+route numbers will probably be significantly worse than your post-synthesis numbers. Explain why the place+route tool is; reporting more area and a longer clock period than the synthesis tool. In Question 2 of the first lab you made some predictions concerning the critical paths and the area of your design. Were these predictions correct? If they differ how has your intuition changed after pushing your processor through the physical tool-flow?

Run the processor benchmarks on the FPGA. Does you processor run immediately? Did you meet the 50 MHz clock rate? How much extra debugging was necessary?

Question 2: Breakpoints

Breakpoints are often used for debugging software, but software breakpoints can also be used as an aide in debugging hardware on the FPGA. For example, if we want to test the effects of a new instruction in isolation, we may use breakpoints to step through its execution, monitoring machine state and checking for correctness. We can also use software breakpoints in a more traditional manner to collect precise performance information from the machine: to gather performance information about a particular function we could set breakpoints at its entry and exit.

In this exercise, you will use `CBus` to implement a breakpoint mechanism for your SMIPS processor. If the breakpoint is enabled, then the processor should halt when the breakpoint instruction is to be executed. All preceding instructions should complete execution as normal, but the breakpointed instruction should have no visible effect on the system until after the breakpoint is cleared. When implementing your breakpoint, pay attention to the way that you handle branch instructions.

Once you have implemented and tested your breakpoint mechanism, describe its architecture. How do you ensure the proper semantics of the breakpoint? Now that we can gather performance

information precisely, how well does your simulation IPC match up to your FPGA IPC? Explain any difference.

Question 3: Evaluate a Simple Branch-Predictor

If you have not done so already, implement the simple branch predictor described in Lab 2 handout, and verify its correctness by running the assembly tests and benchmarks. Although predictors generally increase the IPC, they may negatively impact the area or speed of your processor. In this question we want to evaluate whether or not the branch predictor is a beneficial addition to the processor. Report the change in post-place+route area and timing after adding the branch predictor. Factoring in the advantages of the predictor (increased IPC) and the possible disadvantages (increased area and clock period), do you think adding the predictor is a good idea? Given that you baseline processor has poor rule concurrency (you probably won't have gotten pcgen and exec to fire in the same cycle), the increase in IPC is most likely negligible. What does this say about the benefits of action-level speculation in the absence of action-level concurrency?

Question 4: Refining using RWires

Identify the source of conflict between the rules implementing the three pipeline stages of your processor. Through a combination of rule decomposition (you may want to think about having a separate exec rule for branch and jump instructions) and RWires, you should be able to increase your IPC to something close to 1 for the long benchmarks.

RWires may be set each cycle by a writer and read later in that cycle. RWire uses a `Maybe` tag to denote whether it has been written in a given cycle. Writes are not store; if a value is not read in the cycle that it is written, it is lost. RWires have the following interface:

```
interface RWire#(type valType);
     method Action wset(valType val);
     method Maybe#(valType) wget();
endinterface
```

Recall that the schedule for the RWire is as follows:

```
wset < wget
```

You should also consider using `mkRegFileWCF`. This `RegFile` schedules `upd` and `sub` as conflict free. While this will eliminate scheduling conflicts, you will have handle the case where `upd` and `sub` touch the same address.

Using these constructs, create a version of your design which has the following scheduling property:

```
writeback < execute < pcGen
```

To do this you will need to mechanically transform your design as described in class. After identifying the state elements responsible for the rule conflicts, eliminate the conflicts using `RWire`.

Toggle your branch-predictor and report the FPGA IPC numbers with and without branch prediction. With the addition of action-level concurrency (more rules firing in parallel), how does the branch predictor (action-level speculation) affect performance? What throughput does this version of your design achieve? Synthesize this version of the design as you did for Question 1. How have the results changed and what was the cause of the biggest change? In your opinion is the difference worth it? How does this method compare to experimentally sizing your FIFOs as in Question 1 of Lab 2?

Question 5: Interfacing with Devices

Up until this point, we have assumed that memory is the only external device with which the SMIPS processor interacts. However, to be useful, systems also need to interface with I/O devices. Typically, in embedded systems, these I/O devices are memory-mapped, that is, they have registers which the processor can address, read, and write just like memory locations. Unlike memory locations, operations on memory mapped devices may have side-effects, for example, a write to a certain address might toggle an LED on or off. Since these operations have different semantics than memory operations, they must be handled differently. In particular these operations cannot be cached – if the cache absorbs the toggling of the LED address, the LED will not blink.

Modify your SMIPS core to handle memory mapped I/O. There are several different ways of implementing this feature, but we recommend simply treating some segments of the address space as memory-mapped. By implementing memory mapped I/O you should be able to have your SMIPS processor write "Hello World!" to the on-board serial port (examine the `serial` benchmark for some software to handle this). We will require a successful demonstration of an operating "Hello World!" program to pass this lab. The on-board serial port is shared between the SMIPS processor and the debugging co-processor – be sure to carefully orchestrate the control of this resource.

Document your architecture and explain your reasons for choosing it. Which modules needed modification? We have not provided you with a well-defined test environment for this question. Describe your testing strategy. What modifications did you make to the original test-harness?

Once you have "Hello World" running, re-evaluate the original SMIPS benchmarks. Has the IPC changed? Did the critical path or resource consumption change?