Lab 5: RISC-V Processors

Due date: Monday October 21st 11:59:59pm EST

In this lab you will implement five different pipelined R32I[1] RISC-V processors in Bluespec. Based on a working MultiCycle processor as a starting point, you will work on two-stage RISC-V pipelines and three-stage RISC-V pipelines. Each processor will run various software tests and benchmarks in simulation and/or on the FPGA.

1 The Processor Infrastructure

A large amount of work has already been done for you in setting up the infrastructure to run, test, evaluate performance, and debug your RISC-V processor in simulation and on the FPGA.

1.1 Initial Code

Add the 6.375 course locker, source the setup script, navigate to your git repository which contains the `audio/` folder from the previous labs, and run

```
$ tar xf /mit/6.375/lab2019f/lab5-harness.tar.gz
```

The code provided for this lab has three directories in it:

- `programs/` contains RISC-V programs in assembly and C.
- `connectal/` contains the infrastructure for compiling and simulating the processors.
- `src/` contains BSV code for the RISC-V processors.
- `bolib/` contains library of BSV objects used for the RISC-V processors

Within the BSV source folder, there are folders `src/types/` and `src/proclib` which contain the BSV code for all the modules used in the RISC-V processors. You will not need to change these files for this lab. These files are briefly explained in Table 1.

Folder	Filename	Description
`types`	`MemTypes.bsv`	Common types relating to memory.
	`ProcTypes.bsv`	Common types relating to the processor.
	`Types.bsv`	Common types.
`proclib`	`BTB.bsv`	Implementation of a Branch Target Buffer.
	`CsrFile.bsv`	Implementation of CSRs (including mtohost, which communicates with the host machine).
	`Ehr.bsv`	Implementation of EHRs as described in the lectures.
	`Exec.bsv`	Implementation of the instruction execution.
	`MemInit.bsv`	Modules for downloading the initial contents of instruction and data memories from the host PC.
	`MemorySystem.bsv`	Instantiation of I-Cache and D-Cache connected to WideMem.
	`MemUtil.bsv`	A collection of useful modules and functions about DDR3 and WideMem.
	`Reg6375.bsv`	Registers for 6.375
	`RFile.bsv`	Implementation of the register file.
	`Scoreboard.bsv`	Implementation of the counter-based scoreboards.

Table 1: BSV source code description

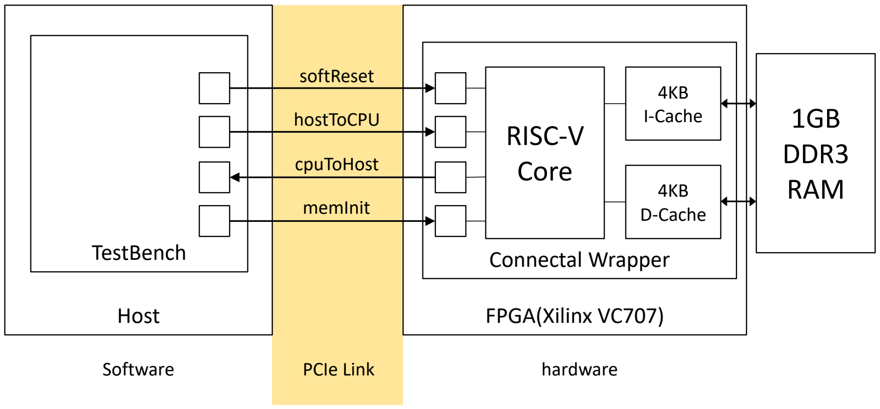

Figure 1: The Processor Test Setup

1.2 The Setup

Figure 1 shows the setup for the lab. When designing and debugging a processor, we will often need the help of another processor, which we call the host processor (labeled "Host" in Figure 1). To differentiate it from the host, we may refer to the processor you'll be designing (labeled "RISC-V Core" in Figure 1) as the target processor. The setup instantiates the processor from the specified processor BSV file and Connectal ports for the processor's softReset, hostToCpu, cpuToHost, memInit.

For each program run, we will first reset the RISC-V core using softReset, and load the memory with programs using memory initialization files (riscv files introduced in Section 3.1). Then we will direct the RISC-V core to begin the program. We will repeat this process for each program run.

The Memory Subsystem

Figure 1 also shows the memory subsystem of the processor which contains a 4KB Instruction Cache and 4KB Data Cache connected to a 1GB DDR3 DRAM. We have provided you this memory subsystem in Bluespec, which your processors connect to.

Components	Capacity	Description
Cache	4KB	Direct Mapped Cache Non-blocking (able to accept new requests) for *cache hits*; Blocking (unable to accept new requests) for *cache misses* .
DDR3 DRAM	1GB	80-cycle access latency

Table 2: Memory Subsystem Property

The components of the memory subsystem have the properties shown in Table 2. It is **important** to keep in mind these properties because they will impact the performance of the benchmarks running on your processors.

2 Build the Processor

In this lab you will work on several versions of a processor, which are defined in `TwoStageEhr.bsv`, `TwoStageRedir.bsv`, `TwoStageBtb.bsv`, `ThreeStage.bsv` and `ThreeStageBypass.bsv`. To build a processor for simulation, say `MultiCycle.bsv`, run in `lab5/` directory:

```
lab5$ make build_bluesim VPROC=MULTICYCLE
```

[1]RV32I is the base integer 32-bit variant Instruction Set Architecture (ISA).

This will create a new folder `bluesim` and the compilation may take a minute or two. To remove this directory for clean rebuilding, you can run:

```
lab5$ make clean
```

You may want to clean up the project between every processor build. The reason is that if you compiled for MultiCycle successfully and then compile for TwoStage, but your two-stage compilation fails at type checking, there is a chance that the build script just proceeds and the resulting output for TwoStage is for the MultiCycle because the TwoStage never managed to compile.

You can also build a processor for the course FPGA (Xilinx VC707 Board) by running

```
lab5$ make build_vc707g2 VPROC=<MULTICYCLE|...>
```

This will take a long time (>1hr) and you might need several modifications in your designs to meet the timing constraint. You will only need to use the FPGA for the last section of this lab.

3 Processor Software Test Bench

The `program` directory contains RISC-V programs to test the processor for correctness and performance.

Assembly Tests

The `programs/assembly/src` directory contains source code for assembly tests.

Filename	Description
`simple.S`	Contains the basic infrastructure code for assembly tests and runs 100 NOP instructions.
`bpred_bht.S`	Contains many branches that a branch history table can predict well.
`bpred_j.S`	Contains many jump instructions that a branch target buffer can predict well.
`bpred_ras.S`	Contains many jumps via registers that a return address stack (RAS) can predict well.
`cache.S`	Tests a cache by writing to and reading from addresses that would alias in a smaller memory.
`<inst>.S`	Tests a specific instruction.

Table 3: Assembly Test source code description

Each assembly test will print the cycle count, instruction count, and whether the test passes or fails. An example output for simple.riscv.vmh on a MultiCycle processor is

```
814
103
PASSED
```

The first line is the cycle count, the second line is the instruction count, and the last line shows if the test passes. The instruction count is larger than the cycle count because we read the instruction count CSR (instret) after reading the cycle count CSR (cycle). If the test fails, the last line will be

```
FAILED exit code = <failure code>
```

You can use the failure code to locate the problem by looking into the source code of the assembly test.

It is highly recommended that you re-run all the assembly tests after making any changes to your processor to verify that you didn't break anything. When trying to locate a bug, running the assembly tests will narrow down the possibilities of problematic instructions.

Benchmarks

We provide two sets of benchmarks of different data sizes.

Benchmark	Description	Input Data Size
`towers.c`	Tower of Hanoi.	7 Discs
`median.c`	1-D three element median filter	One 256-element `int` array
`multiply.c`	Software multiplication	Two 128-element `int` arrays
`qsort.c`	Quicksort on	One 256-element `int` array
`vvadd.c`	Vector-vector addition	Two 128-element `int` arrays

Table 4: Small Benchmark source code description

Benchmark	Description	Input Data Size
`median.c`	1-D three element median filter	One 10240-element `int` array
`multiply.c`	Software multiplication	Two 5120-element `int` arrays
`qsort.c`	Quicksort on	One 10240-element `int` array
`vvadd.c`	Vector-vector addition	Two 5120-element `int` arrays

Table 5: Big Benchmark source code description

`programs/smallbenchmarks` directory contains source code for benchmark programs whose working sets fit in the 4KB cache (Table 4).

`programs/bigbenchmarks` directory contains source code for benchmark programs whose working sets *exceed* the 4KB cache (Table 5).

Each benchmark will print its name, the cycle count, the instruction count, the return value, and whether it passes or fails. An example output for the towers benchmark on a MultiCycle processor is

```
-- benchmark test: towers --
Benchmark tower
Cycles = 59016
Insts  = 6096
Return 0
PASSED
```

If the benchmark passes, the last two lines should be Return 0 and PASSED. If the benchmark fails, the last line will be

```
FAILED exit code = <failure code>
```

Performance is measured in instructions-per-cycle (IPC), and we generally want to increase IPC. For our pipeline we can never exceed an IPC of 1, but we should be able to get close to it with a good branch predictor and proper bypassing.

3.1 Compiling the Assembly Tests and Benchmarks

Our test bench runs RISC-V programs specified in riscv binary (*.risv) format.

A Makefile is provided under each directory for generating programs in the .riscv format. Compile all the assembly tests by running the following:

```
    lab5$ cd programs/assembly
assembly$ make -j8
```

This will create a new directory, `programs/build/assembly`, which contains compilation results for all assembly tests. The directory contains the following subdirectories,

- `bin/` contains all the .riscv files, which will be used during processor memory initialization.
- `dump/` contains all the dumped assembly codes.

Similarly, go to the `programs/<smallbenchmarks|bigbenchmarks>` folders directly and run the `make` command in the respective folder to compile all benchmarks. The compilation results will be in `programs/build/<smallbenchmarks|bigbenchmarks>` directories.

3.2 Using the Test Bench

Our test bench is software running on the host processor which interacts with the RISC-V processor using Connectal as shown in Figure 1. The test bench starts the processor and handles toHost requests until the processor indicates it has completed, either successfully or unsuccessfully. For example, the cycle count in the test output are actually toHost requests from the processor to print an integer, and the requests are handled by the test bench by printing the integer out. The last line (i.e. PASSED or FAILED) of the test output is also printed out by the test bench based on the toHost request which indicates the end of processing.

To run the test bench, first build the project as described in Section 2 and compile the RISC-V programs as described in Section 3.1. In simulation, our RISC-V processor always loads the file `program` to initialize the memory.

For example, to run the median benchmark on the processor in simulation you could use the commands from the top directory:

```
lab5$ ln -sf programs/build/smallbenchmarks/bin/median.riscv program
lab5$ ./bluesim/bin/ubuntu.exe 1>log
```

This creates a file `log` that contains the trace of your execution.

For your convenience, we have provided scripts `run_sw.sh`, which run all the assembly tests and benchmarks automatically. The standard output (`stdout`) will be redirected to `logs/<test name>.log`.

3.2.1 Test bench output

There are two sources of outputs from RISC-V simulation. These include BSV `$display` statements (both messages and errors) and RISC-V print statements.

BSV `$display` statements are printed to `stdout` by `bsim_dut`. BSV can also print to standard error (`stderr`) using `$fwrite(stderr, ...)` statements. The scripts `run_tests.sh` redirects the `stdout` to the `logs/<test name>.log` file.

RISC-V print statements (e.g., `printChar`, `printStr` and `printInt` functions in `programs/benchmarks/common/syscall.c`) are handled through moving characters and integers to the `mtohost` CSR. The test bench reads from the cpuToHost interface and prints characters and integers to `stderr` when it receives them. The messages from `stderr` show up on your screen together with messages from `stdout`, but they are not logged into a file.

4 Evaluating Processor Designs

In this lab, we will work on six processor designs which include the MultiCycle processor. Once your processor passes all *assembly tests*, you can evaluate your processor design. For each processor you will analyze processor's circuit by synthesizing the processor using yosys. You will also run *small benchmark* software on the processor in simulation to evaluate the processor.

We have prepared an Excel worksheet (`lab5.xlsx`) on which you will fill the evaluation data of each processor design.

4.1 Synthesize the Processors

You can synthesize your processor by running:

```
lab5$ synth src/[processor_bsv_filename] mkProc -p src:src/proclib:bolib/ -l multisize
```

This will report the critical path delay, and area of your circuit. Please note that there are two metrics for the area, and we will only use the "Area (excluding the memory)" for the evaluation data.

4.2 Instructions Per Cycle

Processor performance is often measured in instructions per cycle (IPC). This metric is a measure of throughput, or how many instructions are completed per cycle on average. IPC is computed by dividing a number of instructions by how many cycles it took to execute those instructions. A MultiCycle processor will require several cycles to complete a instruction therefore its IPC is smaller than 1.0. DRAM latency penalty due to cache misses will further decrease IPC of MultiCycle.

Improving IPCs of a processor towards 1.0 would require pipelining the processor, and other various architectural techniques such as better branch predictions.

5 Multi-Cycle Processor

The provided code, `src/MultiCycle.bsv`, implements a three-cycle RISC-V processor connected to realistic memory, which has a request/response interface. That is, to read from the memory, you use the request method and in a later cycle, the response method will be ready and can be read by firing the response method. (Latency insensitive)

Exercise 1 (5 points): Build the MultiCycle processor for simulation, and run assembly test and small benchmarks on the processor. Fill in the MultiCycle column of `lab5.xlsx` with the data you observed, and calculate the IPCs for each benchmark.

```
lab5$ make build_bluesim VPROC=MULTICYCLE
lab5$ ./run_sw.sh
```

Synthesize your processor by running:

```
lab5$ synth src/MultiCycle.bsv mkProc -p src:src/proclib:bolib/ -l multisize
```

Discussion Question 1 (10 points): What is the critical path (starting point, end point) of your MultiCycle Processor?

6 Two-Stage Pipelined Processor

6.1 Fixing the Two-Stage Pipelined Processor using EHR

`TwoStage<Ehr|Redir>.bsv` contain an implementation of a functional two-stage pipelined processor that correctly handles control hazards, but it is not properly pipelined.

The reason the processor is not properly pipelined is because the `doFetch` and `doExecute` rules conflict, so they cannot run in the same cycle.

Discussion Question 2 (5 points): Why do rules `doFetch` and `doExecute` conflict?

To resolve the rule conflict, you can simply uses Ephemeral History Registers (`EHR`) for `pc` and `epoch`.

Exercise 2 (20 points): Fix the two-stage pipelined processor (`TwoStageEhr.bsv`) by using EHRs. Fill in the TwoStageEhr column of `lab5.xlsx` with the data you observed, and calculate the IPCs for each benchmark.

Run the following to test your processor

```
lab5$ make build_bluesim VPROC=TWOSTAGEEHR
lab5$ ./run_sw.sh
```

Note: Please run `make clean` between different processor designs as explained in Section 2.

Synthesize your processor by running:

```
lab5$ synth src/TwoStageEhr.bsv mkProc -p src:src/proclib:bolib/ -l multisize
```

Discussion Question 3 (10 points): How does the critical-path delay compare with MultiCycle processor? What is the critical path (starting point, end point) of your TwoStageEhr processor.

6.2 Fixing the Two-Stage Pipelined Processor using Rule Splitting

You can also fix the Two-stage processor by splitting the conflicting part of rule `doExecute` into rule `doRedirection`, such that:

- Rule `doExecute` saves the misprediction condition and redirected PC in two registers.
- Rule `doRedirection` is executed only if the saved misprediction condition is true, and it updates the `pc` and `epoch` registers.

Exercise 3 (20 points): Fix the two-stage pipelined processor (`TwoStageRedir.bsv`) by splitting the conflicting part of rule `doExecute` into rule `doRedirection`. Fill in the TwoStageRedir column of `lab5.xlsx` with the data you observed, and calculate the IPCs for each benchmark.

Run the following to test your processor

```
lab5$ make build_bluesim VPROC=TWOSTAGEREDIR
lab5$ ./run_sw.sh
```

Note: Please run `make clean` between different processor designs as explained in Section 2.

Synthesize your processor by running:

```
lab5$ synth src/TwoStageRedir.bsv mkProc -p src:src/proclib:bolib/ -l multisize
```

Discussion Question 4 (10 points): How do the IPCs of small benchmarks compare with `TwoStageEhr`? Why do IPCs increase/decrease compared with `TwoStageEhr`? Which is the critical path? How does the critical path compare with `TwoStageEhr`?

6.3 Next Address Prediction

Now, let's use a more advanced next address predictor. One such example is a branch target buffer (BTB). It predicts the location of the next instruction to fetch based on the current value of the program counter (the PC). For the vast majority of instructions, this address is PC + 4 (assuming all instructions are 4 bytes). However, this isn't true for jumps and branches. So, a BTB contains a table of previously-used next addresses ("branch targets") that were not simply PC+4, and the PCs that generated those branch targets.

`BTB.bsv` contains an implementation of a BTB of 2^{logn} entries. Its interface has two methods: `predictedNextPC` and `train`.

```
interface NAP#(numeric type logn);
    method Word predictedNextPC(Word pc);
    method Action train(Word pc, Word nextPC);
endinterface
```

- The method `predictedNextPC` takes the current PC and it returns a prediction.
- The method `train` takes a program counter and the next address for the instruction at that program counter and adds it as a prediction if it is not PC+4.

The `predictedNextPC` method should be called to predict the next PC, and the `update` method should be called after a branch resolves. The execution stage requires both the PC of the current instruction and the predicted PC to resolve branches, so you need to store this information in a pipeline register or FIFO.

Exercise 4 (20 points): Copy your working `TwoStageRedir.bsv` to `TwoStageBtb.bsv`, and add a properly-sized BTB to the processor(`TwoStageBtb.bsv`). Fill TwoStageBtb column in `lab5.xlsx` with the data you observed, and calculate the IPCs for each benchmark.

Run the following to test your processor

```
lab5$ make build_bluesim VPROC=TWOSTAGEBTB
lab5$ ./run_sw.sh
```

Note: Please run `make clean` between different processor designs as explained in Section 2.

Synthesize your processor by running:

```
lab5$ synth src/TwoStageBtb.bsv mkProc -p src:src/proclib:bolib/ -l multisize
```

Discussion Question 5 (10 points): What is the size of your BTB? After adding the BTB, how do the IPCs compare with TwoStageRedir? Will adding a BTB to the TwoStageEhr design help its IPC? Explain why.

7 Three-Stage Pipelined Processor

7.1 Fixing the Three-Stage Pipelined Processor

To improve the two-stage design, let's implement a three-stage pipelined processor with following stages:

- The *Fetch stage* initiates a instruction memory read request and sets the PC to the predicted next-PC value (PC+4).
- The *Decode stage* decodes the fetched instruction and reads its source operands from the register file.
- The *Execute stage* executes the instruction, reads or writes to the data memory and updates the register file as needed.

This design is like the one described in Lecture 12. Unfortunately, since the Decode and Execute stages can execute concurrently, there can be a *data hazard* in this processor pipeline: the Decode stage can read a stale value from the register file, which has not been yet updated by an earlier instruction that is still in the Execute stage.

One can resolve this data hazard by tracking all outstanding register file writes into a hardware structure called a *Scoreboard*, and stalling the Decode stage when the index of one of the source registers is found in the scoreboard. When an instruction writes to the register file, the item should be removed from scoreboard, and the Decode stage can then proceed.

The Scoreboard has the following interface:

```
interface Scoreboard#(numeric type size);
    method Action insert(Maybe#(Bit#(5)) dst);
    method Action remove(Maybe#(Bit#(5)) dst);
    method Bool search1(Maybe#(Bit#(5)) src1);
    method Bool search2(Maybe#(Bit#(5)) src2);
endinterface
```

- `size` is the number of outstanding register write indices that the Scoreboard can hold.
- method `insert` inserts a destination register index into Scoreboard. An `Invalid dst` is treated as a NOP on the register file write. Each `Valid` or `Invalid dst` occupies a slot in the Scorebard and a search for an `Invalid dst` will return `False`.
- `method remove` removes the an outstanding register write index `dst` from Scoreboard.
- `methods search1` and `search2` will match `src` register indices with a `Valid` register index stored in the Scoreboard, and returns `True` if a match is found. A search for register `0` is always `False`.

`ThreeStage.bsv` contains a non-functional three-stage pipelined processor that does not handle hazards correctly. Specifically, the code in `ThreeStage.bsv` has three issues:

1. Rule `doDecode` does not have the necessary logic to *stall* the Decode stage on a data hazard. In rule

`doDecode`, a new instruction `inst` from `iMem` (Instruction Memory) *should not* be processed in case the previous instruction had stalled. Due to the request-response interface of `iMem`: once `iMem.resp()` is called, the value it returns is not available in `iMem` anymore—subsequent `iMem.resp()` calls return the data for subsequent load requests. Therefore, if `doDecode` needs to stall (due to a data hazard), it needs to save the fetched instruction in `fetchedInst` to avoid losing it. Consequently after stall, `doDecode` should use the instruction previously saved into `fetchedInst` register instead of calling `iMem.resp()`.

2. Rules `doExecute` and `doLoadWait` do not have the necessary logic to remove the oldest item from the Scoreboard in the Execute stage when an instruction finishes execution. Specifically, these rules should call `sb.remove` in the following two cases:
   - For an instruction with `Valid dst`, `sb.remove` and `rf.wr` should be called atomically, which would be guaranteed if they were called together in the same rule.
   - For an instruction on the wrong path of execution (i.e., a mispredicted instruction), `sb.remove` should also be called.

3. Finally, rules `doFetch` and `doExecute` conflict just like they did in the two-stage pipelined processor (and this conflict can be fixed in the same way).

Exercise 5 (20 points): Fix the three issues in `ThreeStage.bsv`. Fill in the ThreeStage column of `lab5.xlsx` with the data you observed, and calculate the IPCs for each benchmark.

Hint: You can first attempt to fix the first two issues which will get you a functionally correct processor.

Run the following to test your processor

```
lab5$ make build_bluesim VPROC=THREESTAGE
lab5$ ./run_sw.sh
```

Note: Please run `make clean` between different processor designs as explained in Section 2.

Synthesize your processor by running:

```
lab5$ synth src/ThreeStage.bsv mkProc -p src:src/proclib:bolib -l multisize
```

Discussion Question 6 (10 points): How do IPCs compare with `TwoStageRedir`? How does the critical-path delay compare with `TwoStageRedir`

7.2 Improve the Three-Stage Pipelined Processor using Bypassing

Data hazards can be resolved by adding bypass paths from later pipeline stages to earlier stages. We can use a bypass register file, which `wr` is scheduled before `rd`, i.e., the effect of write will be observed by read in the same clock cycle. When bypassing a register file, a `Scoreboard` module of either `mkScoreboard` or `mkBypassingScoreboard`. is also needed to remove data hazards appropriately. Please read the rule schedules of those modules in the comment of `Scoreboard.bsv`.

Exercise 6 (20 points): Copy your working `ThreeStage.bsv` to `ThreeStageBypass.bsv`, and improve it using bypassing. Fill in the ThreeStageBypass column of `lab5.xlsx` with the data you observed, and calculate the IPCs for each benchmark.

Hint: This will only require you to change a couple of module instantiations without major code change.

Run the following to test your processor

```
lab5$ make build_bluesim VPROC=THREESTAGEBYPASS
lab5$ ./run_sw.sh
```

Note: Please run `make clean` between different processor designs as explained in Section 2.

Synthesize your processor by running:

```
lab5$ synth src/ThreeStageBypass.bsv mkProc -p src:src/proclib:bolib -l multisize
```

Discussion Question 7 (10 points): How do the IPCs of the bypassed Three-stage processor compare with ThreeStage without bypass? How does the critical-path delay compare with your Three-stage pipelined processor without bypassing?

8 Running Large Programs

Your processor can also run larger programs than the small benchmarks we have been using. Unfortunately, these larger programs take longer to run, and in many cases, it will take too long for simulation to finish. Now is a great time to try FPGA synthesis. By implementing your processor on an FPGA, you will be able to run these large programs much faster since the design is running in hardware instead of software.

8.1 Synthesizing for FPGA

You can build any processor in this lab for an FPGA (Xilinx VC707 Board) by running

```
lab5$ make build_vc707g2 VPROC=...
```

This command will take a lot of time (about one hour) and a lot of computation resources. You will probably want to select a bdbm server that is under a light load.

8.2 Running on FPGA

Before running on an FPGA, please make sure that your design has met timing by checking the timing report, `vc707g2/bin/top-post-route-timing-summary.txt`. Log onto one of the FPGA servers listed in the Resources section of the course website. Run the following to run benchmarks on the FPGA.

```
lab5$ ./fpgarun_sw.sh
```

`fpgarun_sw.sh` will first program the FPGA and then run various software on your processor.

Exercise 7 (20 Points): Synthesize the Three-Stage processor for the FPGA, and run big benchmarks on the processor.

```
lab5$ make build_vc707g2 VPROC=THREESTAGEBYPASS
lab5$ ./fpgarun_sw.sh
```

Discussion Question 8 (10 points): What are the IPCs of the Three-stage processor for big benchmarks? How do they compare with their corresponding small benchmarks? Explain the changes of IPCs against small benchmarks.

9 Submission

We only need your processor designs in `src/`, discussion answers and a complete lab5.xlsx for this lab. If you have added any new files related to your processors you also need to include them in your submission.

Check in your code for the lab5:

```
lab5$ git add src/*
lab5$ git add lab5.xlsx
lab5$ git add discussion.txt
--- Add any new file that you created for the processors ---
lab5$ git status # make sure your lab files are added
lab5$ git commit -m "Lab 5 submission"
lab5$ git push
```