Lab 6: RISC-V 6-stage Pipeline with Caches

Due: 11:59:59pm, Fri Mar 18, 2016

This lab is your introduction to realistic RISC-V pipelines and caches. In the first part of the lab, you will implement a six-stage RISC-V pipeline. In the second part, you will augment the pipeline with caches and external DRAM to run large benchmarks on actual FPGA.

Part A: RISC-V 6-stage Pipeline

1 Introduction

Add the 6.375 course locker, source the setup script, navigate to your top git repository directory from previous labs, download `lab6a-harness.tar.gz` from the course website

```
$ tar xf lab6a-harness.tar.gz
$ git add riscv-lab6a
$ git commit -m "Lab 6a initial commit"
$ cd riscv-lab6a
```

1.1 New BSV Files

1.1.1 Within `src/includes/`:

`FPGAMemory.bsv` A wrapper for block RAM commonly found on FPGAs. This has an identical interface as the DelayedMemory from the previous lab.

`SFifo.bsv` Three searchable FIFO implementations: one based off of a pipeline FIFO, one based off of a bypass FIFO, and the other based off of a conflict-free FIFO. All implementations assume search is done immediately before enq.

`Scoreboard.bsv` Three scoreboard implementations based off of searchable FIFOs. The pipeline scoreboard uses a pipeline searchable FIFO, the bypass scoreboard uses a bypass searchable FIFO, and the conflict-free scoreboard uses a conflict-free searchable FIFO.

1.2 New Assembly Test

`bpred_j_noloop.S` An assembly test similar to `bpred_j.S`, but the outer loop is removed.

1.2.1 Within `src/`:

`TwoStage.bsv` An initial two-stage pipelined RISC-V processor that uses a BTB for address prediction. Compile with `twostage` target.

`SixStage.bsv` An empty file in which you will extend the two-stage pipeline into a six-stage pipeline. Compile with `sixstage` target.

1.3 Testing Improvements

In the previous lab, the command `build -v <proc_name>` (run from the `scemi/sim/` directory) was used to build `bsim_dut` and `tb`. In this lab, this command builds `<proc_name>_dut` instead of `bsim_dut` so switching between processor types does not delete other processor builds.

Simulation scripts now require you to specify the target processor:

```
$ ./run_asm.sh <proc_name>
$ ./run_bmarks.sh <proc_name>
```

Simulating a single test requires you to run the correct simulation executable:

```
$ cp ../../programs/build/<assembly|benchmarks>/vmh/<test_name>.riscv.vmh mem.vmh
$ ./<proc_name>_dut > out.txt &
$ ./tb
```

2 Two-Stage Pipeline: `TwoStage.bsv`

`TwoStage.bsv` contains a two-stage pipelined RISC-V processor. This processor differs from the processor you built in the previous lab because it reads register values in the first stage and there is data hazard.

Discussion Question 1 (10 Points): Debugging practice! If you replace the BTB with a simple `pc + 4` address prediction, the processor still works, but it does not perform as well. If you replace it with a really bad predictor that predicts `pc` is the next instruction for each `pc`, it should still work but have even worse performance because each instruction would require redirection (unless the instruction loops back to itself). If you actually set the prediction to `pc`, you will get errors in the assembly tests; the first one will be from `cache.riscv.vmh`. What is the error you get? What is happening in the processor to cause that to happen? Why do not you get this error with PC+4 and BTB predictors? How would you fix it? You do not actually have to fix this bug, just answer the questions. (Hint: look at the `addr` field of `ExecInst` structure.)

3 Six-Stage Pipeline: `SixStage.bsv`

The six-stage pipeline should be divided into the following stages:

- Instruction Fetch – request instruction from iMem and update PC
- Decode – receive response from iMem and decode instruction
- Register Fetch – read from the register file
- Execute – execute the instruction and redirect the processor if necessary
- Memory – send memory request to dMem
- Write Back – receive memory response from dMem (if applicable) and write to register file

`IMemory` and `DMemory` instances should be replaced with instances of `FPGAMemory` to enable later implementation on FPGA.

Exercise 1 (20 Points): Starting with the two-stage implementation in `TwoStage.bsv`, replace each memory with `FPGAMemory` and extend the pipeline into a six-stage pipeline in `SixStage.bsv`. In simulation, benchmark `qsort` may take longer time (21 seconds on TA's desktop, and it may take even longer on vlsifarm machines).

Notice that the two-stage implementation uses normal register file and conflict-free scoreboard. However you could use pipelined or bypass versions of these components for better performance. Besides, you may also want to change the size of scoreboard. You may also want to check the scheduling of the rules to make sure it is what you expect. The scheduling dump can be found in `info_dut/mkProc.sched`.

Discussion Question 2 (5 Points): What evidence do you have that all stages are able to fire in the same cycle?

Discussion Question 3 (5 Points): In your six-stage pipelined processor, how many cycles does it take to correct a mispredicted instruction?

Discussion Question 4 (5 Points): If an instruction depends on the result of the instruction immediately before it in the pipeline, how many cycles is that instruction stalled?

Discussion Question 5 (5 Points): What IPC do you get for each benchmark?

3.1 Part A Submission

Check in all of your code for this part of the lab:

```
riscv-lab6a$ git add src/*
riscv-lab6a$ git add answers/lab6a
riscv-lab6a$ git status  # make sure your lab files are added
riscv-lab6a$ git commit -m "Lab 6a submission"
riscv-lab6a$ git push
```

Part B: Caches and DRAM

By now you have a 6-stage pipelined RISC-V processor. Unfortunately your processor is limited to running programs that can fit in a 256 KB FPGA block RAM. This works fine for the small benchmark programs we are running, such as a 250 item quicksort, but most interesting applications are (much) larger than 256 KB. Luckily the FPGA boards we are using have 1 GB of DDR3 DRAM accessible by the FPGA. This is great for storing large programs, but this may hurt the performance since DRAM has long latencies when reading data from them.

This part will focus on using DRAM instead of block RAM for main program and data storage to store larger programs and adding caches to reduce the performance penalty from long latency DRAM loads.

First you will write a translator module that takes memory requests from the CPU and translates them to memory requests for DRAM. This module will enable a larger storage space for your programs, but it will see a large decrease in performance since your are reading from DRAM almost every cycle. Next you will implement a cache to reduce the amount of times you need to read from the DRAM, therefore improving your processors performance. Lastly you will synthesize your design for an FPGA and run very large benchmarks that require DRAM and very long benchmarks that require an FPGA.

4 Part B Harness

Navigate to your git repository's top directory. Download `lab6b-harness.tar.gz` from the course website

```
$ tar xf lab6b-harness.tar.gz
$ git add riscv-lab6b
$ git commit -m "Lab 6b initial commit"
$ cd riscv-lab6b
```

5 Change in Testing Infrastructure

Since programming FPGA takes times (about 1min), it would take very long to run all tests if we re-program the FPGA to reset everything before running each test. To reduce the testing time, we only program FPGA once. After finishing one test, the software test bench (`scemi/Tb.cpp`) will initiate a soft reset of the processor states on FPGA, then write the VMH file of the new test program to DRAM, and start the new test. The software test bench will take in all the VMH files that we want to test as arguments, and perform tests using each of them. The software test bench will print out the name of the VMH file before starting each test. In simulation, we will also simulate the process of writing VMH files to DRAM, so the simulation time will be longer than before.

Below are example commands to simulate a processor named `withoutcache`, which we will build next, using assembly tests `simple.S` and `add.S`:

```
$ cd scemi/sim
$ ./withoutcache_dut > log.txt &
$ ./tb ../../programs/build/assembly/vmh/simple.riscv.vmh ../../programs/build/assembly/vmh/add.riscv.vmh
```

Here are the sample outputs:

```
---- ../../programs/build/assembly/vmh/simple.riscv.vmh ----
1196
103
PASSED

---- ../../programs/build/assembly/vmh/add.riscv.vmh ----
5635
427
PASSED

SceMi Service thread finished!
```

We also provide two scripts `run_asm.sh` and `run_bmarks.sh` to run all assembly tests and benchmarks respectively. For example, we can use the following commands to test processor `withoutcache`:

```
$ ./run_asm.sh withoutcache
$ ./run_bmarks.sh withoutcache
```

The standard outputs of BSV will be redirected to `asm.log` and `bmarks.log` respectively.

6 DRAM Interface

The VC707 FPGA board you will use in this class has 1 GB of DDR3 DRAM. DDR3 memory has a 64 bit wide data bus, but 8 64 bit chunks are sent per transfer, so effectively it acts like a 512 bit wide memory. DDR3 memories have high throughput, but they also have high latencies for reads.

The Sce-Mi interface generates a DDR3 controller for us, and it takes in `MemoryClient` interface to connect to it. The typedefs provided for you in this part use types from BSV's Memory package (see BSV reference guide or source code at `$BLUESPECDIR/BSVSource/Misc/Memory.bsv`). Here are some of the typedefs related to DDR3 memory in `src/includes/MemTypes.bsv`:

```
typedef 24 DDR3AddrSize;
typedef Bit#(DDR3AddrSize) DDR3Addr;
typedef 512 DDR3DataSize;
typedef Bit#(DDR3DataSize) DDR3Data;
typedef TDiv#(DDR3DataSize, 8) DDR3DataBytes;
typedef Bit#(DDR3DataBytes) DDR3ByteEn;
typedef TDiv#(DDR3DataSize, DataSize) DDR3DataWords;

// The below typedef is equivalent to this:
// typedef struct {
//     Bool          write;
//     Bit#(64)      byteen;
//     Bit#(24)      address;
//     Bit#(512)     data;
// } DDR3_Req deriving (Bits, Eq);
typedef MemoryRequest#(DDR3AddrSize, DDR3DataSize) DDR3_Req;

// The below typedef is equivalent to this:
// typedef struct {
//     Bit#(512)     data;
// } DDR3_Resp deriving (Bits, Eq);
typedef MemoryResponse#(DDR3DataSize) DDR3_Resp;

// The below typedef is equivalent to this:
// interface DDR3_Client;
//     interface Get#( DDR3_Req ) request;
//     interface Put#( DDR3_Resp ) response;
```

```
// endinterface;
typedef MemoryClient#(DDR3AddrSize, DDR3DataSize) DDR3_Client;
```

6.1 DDR3_Req

The requests for DDR3 reads and writes are different than the requests of `FPGAMemory`. The biggest difference is the byte enable, `byteen`.

- `write` – Boolean specifying if this request is a write request or a read request.
- `byteen` – Byte enable, specifies which 8-bit bytes will be written. This field has no effect for a read request. If you want to write all 16 bytes (i.e. 512 bits), you will need to set this to all 1's. You can do that with the literal `'1` (note the apostrophe) or `maxBound`.
- `address` – Address for read or write request. DDR3 memory is addressed in 512-bit chunks, so address 0 refers to the first 512 bits, and address 1 refers to the second 512-bits. This is very different than the byte addressing used in the RISC-V processor.
- `data` – Data value used for write requests.

6.2 DDR3_Resp

DDR3 memory only sends responses for reads just like `FPGAMemory`. The memory response type is a structure instead of just `Bit#(512)` so you will have access the `data` field of the response in order to get the `Bit#(512)` value.

6.3 DDR3_Client

The `DDR3_Client` interface is made up of a `Get` subinterface and a `Put` subinterface. This interface is exposed by the processor, and the Sce-Mi infrastructure connects it to the DDR3 controller. You do not need to worry about constructing this interface because it is done for you in the example code.

6.4 Example Code

Here is some example code showing how to construct the FIFOs for a DDR3 memory interface along with the initialization interface for DDR3. This example code is provided in `src/DDR3Example.bsv`.

```
import GetPut::*;
import ClientServer::*;
import Memory::*;
import CacheTypes::*;
import WideMemInit::*;
import MemUtil::*;
import Vector::*;

// other packages and type definitions

(* synthesize *)
module mkProc(Proc);
  Ehr#(2, Addr)  pcReg <- mkEhr(?);
  CsrFile        csrf  <- mkCsrFile;

  // other processor stats and components

  // interface FIFOs to real DDR3
  Fifo#(2, DDR3_Req)  ddr3ReqFifo  <- mkCFFifo;
  Fifo#(2, DDR3_Resp) ddr3RespFifo <- mkCFFifo;
  // module to initialize DDR3
  WideMemInitIfc      ddr3InitIfc  <- mkWideMemInitDDR3( ddr3ReqFifo );
  Bool memReady = ddr3InitIfc.done;
```

```
25  // wrap DDR3 to WideMem interface
    WideMem           wideMemWrapper <- mkWideMemFromDDR3( ddr3ReqFifo, ddr3RespFifo );
27  // split WideMem interface to two (use it in a multiplexed way)
    // This spliter only take action after reset (i.e. memReady && csrf.started)
29  // otherwise the guard may fail, and we get garbage DDR3 resp
    Vector#(2, WideMem)     wideMems <- mkSplitWideMem( memReady && csrf.started, wideMemWrapper );
31  // Instruction cache should use wideMems[1]
    // Data cache should use wideMems[0]
33
    // some garbage may get into ddr3RespFifo during soft reset
35  // this rule drains all such garbage
    rule drainMemResponses( !csrf.started );
37    ddr3RespFifo.deq;
    endrule
39
    // other rules
41
    method ActionValue#(CpuToHostData) cpuToHost if(csrf.started);
43    let ret <- csrf.cpuToHost;
      return ret;
45  endmethod

47  // add ddr3RespFifo empty into guard, make sure that garbage has been drained
    method Action hostToCpu(Bit#(32) startpc) if ( !csrf.started && memReady && !ddr3RespFifo.notEmpty );
49    csrf.start(0); // only 1 core, id = 0
      pcReg[0] <= startpc;
51  endmethod

53  // interface for testbench to initialize DDR3
    interface WideMemInitIfc memInit = ddr3InitIfc;
55  // interface to real DDR3 controller
    interface DDR3_Client ddr3client = toGPClient( ddr3ReqFifo, ddr3RespFifo );
57 endmodule
```

In the above example code, `ddr3ReqFifo` and `ddr3RespFifo` serve as interfaces to the real DDR3 DRAM. In simulation, we provide a module `mkSimMem` to simulate the DRAM, which is instantiated in `scemi/SceMiLayer.bsv`. In FPGA synthesis, the DDR3 controller is instantiated in the top-level module `mkBridge` in `$BLUESPECDIR/board_support/bluenoc/bridges/Bridge_VIRTEX7_VC707_DDR3.bsv`. There is also some glue logic in `scemi/SceMiLayer.bsv`.

In the example code, we use module `mkWideMemFromDDR3` to translate `DDR3_Req` and `DDR3_Resp` types to a more friendly `WideMem` interface defined in `src/includes/CacheTypes.bsv`.

6.5 Sharing the DRAM Interface

The example code exposes a single interface with the DRAM, but you have two modules that will be using it: an instruction cache and a data cache. If they both send requests to `ddr3ReqFifo` and they both get responses from `ddr3RespFifo`, it is possible for their responses to get mixed up. To handle this, you need a separate FIFO to keep track of the order the responses should come back in. Each load request is paired with an enqueue into the ordering FIFO that says who should get the response.

To simplify this for you, we have provided module `mkSplitWideMem` to split the DDR3 FIFOs into two `WideMem` interfaces. This module is defined in `src/includes/MemUtils.bsv`. To prevent `mkSplitWideMem` from taking action to early and exhibiting unexpected behavior, we set its first parameter to `memReady&&csrf.started` to freeze it before the processor is started. This also avoids scheduling conflicts with initialization of DRAM contents.

6.6 Handling Problems in Soft Reset

As mentioned before, you will perform a soft reset of the processor states before starting each new test. During soft reset, some garbage data may be enqueued into `ddr3RespFifo` due to some cross clock domain issues. To handle this problem, we have added a `drainMemResponses` rule to drain the garbage data, and have

added a condition that checks whether `drainMemResponses` is empty into the guard of method `hostToCpu`.

Suggestion: add `csrf.stared` to the guard of the rule for each pipeline stage. This prevents the pipeline from accessing DRAM before the processor is started.

7 Migrating Code from Part A

The provided code for this part is very similar, but there are a few differences to note. Most of the differences are displayed in the provided example code `src/DDR3Example.bsv`.

7.1 Modified Proc Interface

The Proc interface now only has a single memory initialization interface to match the unified DDR3 memory. The width of this memory initialization interface has been expanded to 512 bits per transfer. The new type of this initialization interface is `WideMemInitIfc` and it is implemented in `src/includes/WideMemInit.bsv`.

7.2 Empty Files

The two processor implementations for this part of the lab: `src/WithoutCache.bsv` and `src/WithCache.bsv` are initially empty. You should copy over the code from `SixStage.bsv` as a starting point for these processors.

7.3 New Files

Here is the summary of new files provided under the `src/includes` folder:

`Cache.bsv` An empty file in which you will implement cache modules.

`CacheTypes.bsv` A collection of type and interface definitions about caches.

`MemUtil.bsv` A collection of useful modules and functions about DDR3 and `WideMem`.

`SimMem.bsv` DDR3 memory used in simulation. It has a 10-cycle pipelined access latency, but extra glue logic may add more to the total delay of accessing DRAM in simulation.

`WideMemInit.bsv` Module to initialize DDR3.

There are also changes in `MemTypes.bsv`.

8 `WithoutCache.bsv` – Using the DRAM Without a Cache

Exercise 2 (10 Points): Implement a module `mkTranslator` in `Cache.bsv` that takes in some interface related to DDR3 memory (`WideMem` for example) and returns a `Cache` interface (see `CacheTypes.bsv`). This module should not do any caching, just translation from `MemReq` to requests to DDR3 (`WideMemReq` if using `WideMem` interfaces) and translation from responses from DDR3 (`CacheLine` if using `WideMem` interfaces) to `MemResp`. This will require some internal storage to keep track of which word you want from the cache line that comes back from main memory. Integrate `mkTranslator` into a six stage pipeline in the file `WithoutCache.bsv` (i.e. you should no longer use `mkFPGAMemory` here). You can build this processor by running `build -v withoutcache` from `scemi/sim/`, and you can test this processor by running `./run_asm.sh withoutcache` and `./run_bmarks.sh withoutcache` from `scemi/sim/`.

Discussion Question 6 (5 Points): Record the results for `./run_bmarks.sh withoutcache`. What IPC do you see for each benchmark?

9 `WithCache.bsv` – Using the DRAM With a Cache

By running the benchmarks with simulated DRAM, you should have noticed that your processor slows down a lot. You can speed up your processor again by remembering previous DRAM loads in a cache as described in class.

Exercise 3 (20 Points): Implement a module `mkCache` to be a direct mapped cache that allocates on write misses and writes back only when a cache line is replaced. This module should take in a `WideMem` interface (or something similar) and expose a `Cache` interface. Use the `typedefs` in `CacheTypes.bsv` to size your cache and for the `Cache` interface definition. You can use either vectors of registers or register files to implement the arrays in the cache, but vectors of registers are easier to specify initial values. Incorporate this cache in the same pipeline from `WithoutCache.bsv` and save it in `WithCache.bsv`. You can build this processor by running `build -v withcache` from `scemi/sim/`, and you can test this processor by running `./run_asm.sh withcache` and `./run_bmarks.sh withcache` from `scemi/sim/`.

Discussion Question 7 (5 Points): Record the results for `./run_bmarks.sh withcache`. What IPC do you see for each benchmark?

10 Running Large Programs

By adding support for DDR3 memory, your processor can now 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 to wait for the 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.

Exercise 4 (0 Points, but you should still totally do this): Before synthesizing for an FPGA, lets try looking at a program that takes a long time to run in simulation. The program `./run_mandelbrot.sh` runs a benchmark that prints a square image of the Mandelbrot set using 1's and 0's. Run this benchmark to see how slow it runs in real time. Please don't wait for this benchmark to finish, just kill it early using ctrl-c.

10.1 Synthesizing for FPGA

You can start FPGA synthesis for `WithCache.bsv` by going into the `scemi/fpga_vc707` folder and executing the command:

```
$ vivado_setup build -v
```

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

10.2 Running on the FPGA

Ensure that your design has met timing by checking the timing report. Log onto one of the FPGA servers listed in the Resources section of the course website. Check that no one else is using the FPGA with `w` and `top` commands. Program the FPGA and run the benchmarks.

```
fpga_vc707$ programfpga xilinx/mkBridge.bit
fpga_vc707$ ./run_bmarks.sh
```

Exercise 5 (10 Points): Synthesize `WithCache.bsv` for the FPGA and program the bitfile. Get the results for the benchmarks and add them to your answers text file.

Discussion Question 8 (10 Points): How many cycles does the Mandelbrot program take to execute in your processor? The current FPGA design has an effective clock speed of 50 MHz. How long does the Mandelbrot program take to execute in seconds? Estimate how much of a speedup you are seeing in hardware versus simulation by estimating how long (in wall clock time) it would take to run `./run_mandelbrot.sh` in simulation.

10.3 Part B Submission

Check in all of your code for this part of the lab:

```
riscv-lab6b$ git add src/*
riscv-lab6b$ git add answers/lab6b
riscv-lab6b$ git status  # make sure your lab files are added
riscv-lab6b$ git commit -m "Lab 6b submission"
riscv-lab6b$ git push
```