6.S195: Lab 8
Caches

November 13, 2013

Due: Friday November 22, 2013

Note: This lab is a direct continuation of Lab 7. You will start with the 6-stage pipeline with BTB and BHT you created for that lab.

1 Introduction

By now you have a 6-stage pipelined SMIPS processor with target and direction branch predictors. 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 256 MB of DDR2 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 lab 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 a wrapped 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.

2 DRAM Interface

The XUPV5 board has 256 MB of DDR2 DRAM. DDR2 memory has a 64 bit wide data bus, but 4 64 bit chunks are sent per transfer, so effectively it acts like a 256 bit wide memory. DDR2 memories have high throughput, but the also have high latencies for reads.

The DRAM controller used in this lab works similarly to the FPGAMemory modules. `ProcExample.bsv` connects the DRAM's interface to two fifos: `dramReqQ` and `dramRespQ`. `dramReqQ` takes requests of type `DDR2Request` which is defined and described below.

```
typedef struct {
    // writeen: Enable writing.
    // Set the ith bit of writeen to 1 to write the ith byte of
    // datain to the ith byte of data at the given address.
    // If writeen is 0, this is a read request, and a response
    //   is returned.
    // If writeen is not 0, this is a write request, and no
    //   response is returned.
    Bit#(32) writeen;

    // Address to read to or write from.
    // The DDR2 is 64 bit word addressed, but in bursts of 4
    // 64-bit words. The address should always be a multiple of
    // 4 (bottom 2 bits 0), otherwise strange things will happen.
    // For example:
    //   address 0 refers to the first 4 64 bit words in memory.
    //   address 4 refers to the second 4 64 bit words in memory.
    DDR2Address address;

    // Data to write.
    // For read requests this is ignored.
    // Only those bytes with corresponding bit set in writeen
    // will be written.
    DDR2Data datain;
} DDR2Request deriving(Bits, Eq);
```

`dramRespQ` returns values of type `DDR2Response` which is an alias for `Bit#(256)`.

The module `mkDDRWrapper` takes in the interfaces to the two DRAM fifos, and returns a more managable interface for the processor to use. This interface will allow an instruction cache and a data cache to use the same DRAM. The interface `DDRWrapper` and the associated interfaces and types are defined below.

```
interface DRAMWrapper;
    interface WideMemory iMem;
    interface WideMemory dMem;
    interface MemInitIfc init;
endinterface

interface WideMemory;
    method Action req(WideMemReq r);
    method ActionValue#(CacheLine) resp;
```

```
endinterface

typedef struct{
    Bit#(8)   write_en;
    Addr      addr;
    CacheLine data;
} WideMemReq deriving(Eq,Bits);

typedef Vector#(8, Data) CacheLine
```

`WideMemReq` is very similar to `DDR2Request` except `addr` is byte-addressed and the last 5-bits of the address should be 0, each `write_en` bit refers to 32-bit word, and the data taken in is a vector of 32-bit words.

The caches in this lab will use the following interface.

```
interface Cache;
    method Action req(MemReq r);
    method ActionValue#(MemResp) resp;
endinterface
```

This is just like `FPGAMemory` except there is no sub-interface for initialization. That means you should be able to easily replace your current `mkFPGAMemory` module constructors with a cache constructor (assuming you change types and remove references to the initialization sub-interface of FPGAMemory).

From here, if you could translate `MemReq` to `WideMemReq` and `CacheLine` to `MemResp` appropriately, then you could use the DRAM right away. For the first exercise, that is exactly what we are going to do, but first you need to get the code for lab 8.

3 Starting Off

To get your personal git repository for lab 8 run the following command.

```
$ git clone $GITROOT/lab8.git
```

This will create a folder called `lab8` with some support code for this lab.

Next, copy all the `*.bsv` files from `lab7/src` to `lab8/src`. Add all your source files to the git repo by running the following commands.

```
$ cd lab8
$ git add src/*
$ git commit -am "Initial code from lab 7."
$ git push
```

At this point neither simulation or fpga synthesis will work because the dut wrapper files assume a different processor interface than the one from lab 7. This will all be fixed in the first exercise.

4 Using the DRAM Without a Cache

Now that you have the code for lab 8, take a look at `ProcExample.bsv`; this file contains a skeleton for a processor using DRAM. You need to modify your `Proc.bsv` to use the same interface as `ProcExample.bsv`, and you need to add the dram code from `ProcExample.bsv`. Once you have modified `Proc.bsv` you are ready to implement `mkTranslator` in `Cache.bsv` to get your processor to use the DRAM.

Exercise 1 (10 Points): Implement `mkTranslator` in `Cache.bsv` to translate from `MemReq` to `WideMemReq` and from `CacheLine` back to `MemResp` appropriately. You will need to add a state element to this module to keep track of what what word in the output `CacheLine` to select for `MemResp`. Once you have implemented it, incorporate it into your pipeline by replacing references to `mkFPGAMemory` with `mkTranslator`. Since there is a DDR2 functional model for simulation included in the lab 8 repo, you can simulate this design in the same way you simulated labs 6 and 7. Record benchmark performance in `discussion.txt`.

If you want to get this processor running on the FPGA, you will have to add the following rule to `mkProc`.

```
    rule drainMemResponses( !cop.started );
        dramRespQ.deq;
    endrule
```

This rule clears the DRAM response queue when the processor is not running. This is needed because the processor implementation without a cache will have DRAM traffic when the processor is reset between programs, and outstanding DRAM responses will mess up the processor when it is started again for the next program.

5 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. There are many different types of caches that you can implement, but for this lab, you will be implementing a direct mapped cache that writes-through on write hits and does not allocate on write misses.

Exercise 2 (20 Points): Implement a `mkCache` to be a direct mapped cache that writes-through on write hits and does not allocate on write misses. Use the `typedefs` in `CacheTypes.bsv` to size your cache. Incorporate this cache in your pipeline and simulate the processor with the cache. Record benchmark performance in `discussion.txt`.

6 Running Large Programs

Now that you have a processor with a significant amount of memory, you can run larger benchmarks. Unfortunately larger benchmarks take many processor cycles to complete, and therefore will take too long to simulate. Luckily you have access to FPGAs that can run much faster than simulation, so you are going to compile your processor for the FPGAs.

Exercise 3 (5 Points): Synthesize your processor from exercise 2 for the FPGA by running `xilinx build -v xupv5_dut` in `scemi/fpga`, and build the test bench executable by running `build -v tb`. Run either `w`, `top`, or `ps -af` to see if someone else is using the FPGA currently, if it is free, run `programfpga` to program the FPGA. Run the standard `run_assembly` and `run_benchmarks` to test your processor and caches initially. Record the normal benchmark performance in `discussion.txt`.

There are much larger versions of the original benchmarks contained in `/mit/6.s195/large-programs`. Instead of working on hundreds of memory locations, these benchmarks work with millions of memory locations.

Exercise 4 (5 Points): Run these larger benchmarks on your processor using `run_large_benchmarks`. These will take much longer to run on the FPGA, but most of the time will be spent waiting for the memories to be programmed through the SceMi interface. The actual execution will be very fast. Record the number of cycles and instructions for these larger benchmarks in `discussion.txt`.

7 Discussion Questions

Answer these questions in `discussion.txt`.

Exercise 5 (10 Points):

1. How does the latency of the DDR2 simulation model compare to the latency of the actual DDR2 DRAM on the FPGA?
2. What advantage does your write-through policy for write hits have over a write-back policy? What advantage does a write-back policy have over write-through?
3. Do you think you could see an improvement in cache performance by using a smaller cache line? What about a larger cache line? Assume you are still using the same number of bits in your cache.

8 Submitting the lab

When you are done with this lab, commit all your changes and push your changes back to be graded by running the following commands:

```
$ git commit -am "Done with lab 8"
$ git push
```