6.S195: Final Project
Multi-Core SMIPS Processor

November 25, 2013

Bitfile for Competition Due: Thursday December 10, 2013 at 11:59 PM

Git Repo Due: Wednesday December 11, 2013 at 3:00 PM

Final Presentation: Wednesday December 11, 2013 at 3:00 PM

1 Introduction

For the final project of 6.S195, you will work with a partner to implement a cache coherency protocol for a two-core SMIPS processor. You will also implement load-link (`LL`) and store-conditional (`SC`) instructions to enable atomic read-modify-write operations. To test the functionality and performance of your multi-core processor, you will write a multi-core test program and a benchmark. Finally you will make improvements of your choice to your two-core SMIPS processor to maximize its IPS for a competition against the other groups. You will present your project to the class on Wednesday December 11th.

2 Getting Started

To get the starting code for this project, copy the compressed file to a local directory and extract it by running the following commands.

```
$ cp /mit/6.s195/initial_project_code.tar.gz ./
$ tar -zxvf initial_project_code.tar.gz
$ rm initial_project_code.tar.gz
```

Once the TA has e-mailed you that you have access your group's git repo, one of you should clone the repo, copy the project files into it, add the files, and push the changes by running the following commands.

```
$ git clone /mit/6.s195/groups/group<n>/project
$ cp -r initial_project_code/* project/
$ cd project
$ git add *
$ git commit -m "Initial commit"
$ git push
```

At this point you and your partner can start using the git repo to share code with each other.

2.1 Initial processor

Figure 1 shows a diagram of the initial design you are given for this project. While this may look like a sufficient multi-core processor, it is not because the caches do not work together to achieve cache coherency.

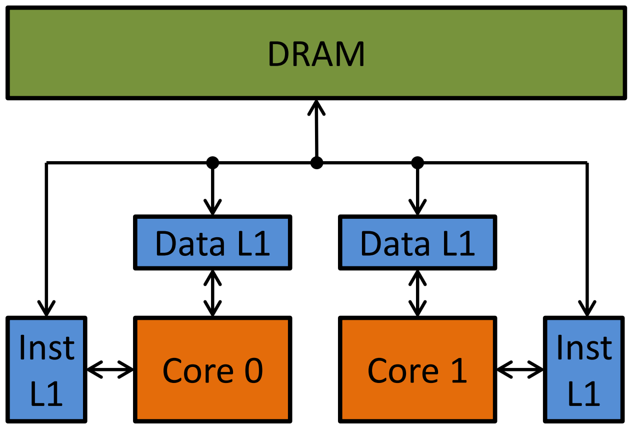

Figure 1: The initial flawed multi-core design.

2.1.1 Processor pipeline

The processor pipeline is a standard 6 stage pipeline. The decode and execute functions have already been updated to support the new instructions implemented in this project, so you will not have to modify it.

2.1.2 Coprocessor

The coprocessor has been expanded to support reading from one more register. Register 15 now contains the core ID, a number uniquely identifying the current core. Intuitively, core 0 has 0 as its core ID, and core 1 has 1 as its core ID. This is an easy way for the SMIPS processor to support multi-threaded programs without running an OS.

2.1.3 Caches

There are separate designs for the instruction and data caches. The instruction cache is specialized to be read only. Its response method just takes in the `PC` and there is no support for the cache to write back to the memory.

The provided data cache is different from the cache implemented in lab 8. This cache is write-back no-allocate, so it keeps track of dirty bits and only writes back to DRAM on cache misses and when dirty lines are evicted. This cache also improperly implements the `MSI` protocol and the `LL` and `SC` instructions; you will be correcting these implementations in this project.

3 Multi-Core SMIPS Processor

Figure 1 shows a diagram of the initial design you are given for this project. All of the caches talk directly with DRAM with minimal coordination. The only coordination performed in the network between the L1 caches and the DRAM is making sure DRAM responses make it back to the right L1 cache.

With this limited amount of coordination, if core 0 writes to an address that is contained in the cache for core 1, core 1 will not see that update until the line is evicted and read again. If core 1's program is in a loop that is waiting for that address to be changed, then core 1 may never see the change because the cache line may never be evicted.

To fix this problem, you will implement the MSI protocol for cache coherency. The MSI protocol will allow core 1 to see the update from core 0 in the previously described case by forcing an eviction in core 1. Figure 2 shows a diagram of your cache structure with the MSI protocol implemented for the data caches.

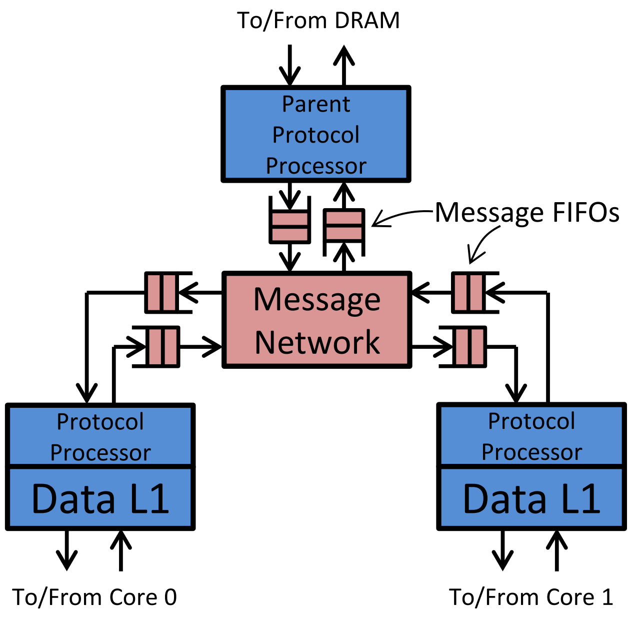

Figure 2: The protocol processors and the message network that implement the MSI protocol for cache coherency.

Only the data caches will have a cache coherency protocol because we are assuming we are not dealing with self-modifying code. In most processors, cache coherency in the case of self-modifying code is handled by interrupts anyways.

4 MSI protocol

The MSI protocol is a method to preserve cache coherency. MSI refers to the three states a cache line can be in: Modified, Shared, and Invalid. `I` means the current cache line is not valid for reading or writing. `S` means the current cache line is valid for reading, but it may be in other caches too, so it is not valid for writing. `M` means the current cache line is only in this cache, so it is valid for reading and writing. The states `M`, `S`, and `I` can be thought of as an order `M > S > I`. A transition from a lower state to a higher state is an upgrade. A transition from a higher state to a lower state is called a downgrade.

4.1 MSI enumeration in Bluespec

The provided code includes an enumeration type for MSI status values. Part of its definition is shown below.

```
typedef enum { M, S, I } MSI deriving( Bits, Eq, FShow );
instance Ord#(MSI);
    function Bool \< ( MSI x, MSI y );
    function Bool \<= ( MSI x, MSI y );
    function Bool \> ( MSI x, MSI y );
    function Bool \>= ( MSI x, MSI y );
    function Ordering compare( MSI x, MSI y );
    function MSI min( MSI x, MSI y );
    function MSI max( MSI x, MSI y );
endinstance
```

Like usual, the `MSI` enumeration is derived as an instance of `Bits` and `Eq`, but there are two other type classes that MSI is in: `FShow` and `Ord`. `FShow` allows for the name of a value of `MSI` to be printed using the `fshow` function as shown in the following two examples.

```
MSI y = req.y;
// displaying simple message
$display("Processing downgrade-to-", fshow(y), " request");
// displaying more complicated message
Int number = 7;
String message = "Hello World"
$display("number: %d   state: ", number, fshow(y), "   message: %s", message);
```

Having `MSI` as an instance of the `Ord` typeclass allows you to compare `MSI` values in the same way as shown in lecture. The comparison `y < M` is valid and returns true for `y == S` and `y == Y`. `MSI` was made an instance of `Ord` by defining the comparison operators on values of `MSI`.

4.2 Protocol communications

The MSI protocol involves communications between each of the L1 cache's protocol processor and the parent protocol processor at the memory as shown in Figure 2. In practice, the L1 cache is combined with its protocol processor, but the parent protocol processor is implemented separately from the DRAM. Also the children protocol processors have different functionality than the parent protocol processor.

The caches and the parent protocol processor can each issue requests and responses. Upgrade requests are initiated in the L1 caches and sent to the parent protocol processor for processing, and downgrade requests are sent by the parent protocol processor to the children caches. Each request will be answered by a response message once the request has been fulfilled.

The slides from lecture 22 have a detailed description of the 8 different situations that need to be handled for both incoming and outgoing messages. The message structure and network is described in depth in Section 5.

4.3 Data storage for MSI protocol

The MSI protocol relies on the caches and the memory to do some book keeping about what state different addresses are in, and if there are any expected responses for a given address.

4.3.1 Cache-side

The cache keeps track of the status of its own cache lines.

- `state[a]` – This stores a value of type `MSI` for each cache line.
- `waitp[a]` – This tracks expected responses from the parent for each cache line. Its data type is `Maybe#(MSI)`.

4.3.2 Memory-side

The parent protocol processor keeps track of the cache lines in each of its children.

- `child[i][a]` – This stores the status of address `a` in cache `i`.
- `waitc[i][a]` – This tracks expected responses from child `i` for address `a`. Just like the child-side `waitp[a]`, its data type is `Maybe#(MSI)`.

It takes too much memory to store information for each possible address, so the parent should implement these two with indexes and tags just like the children cache. You will have to figure out exactly how to implement the tags and when in the protocol to change them.

5 Message Network

The MSI protocol requires a network between the protocol processors of the L1 caches and the parent protocol processor to pass messages back and forth. This network will be made up of message FIFOs and a message network module. A diagram of the network can be seen in Figure 2.

5.1 Message structure

The messages sent between caches and the protocol processor contain information about the type of message, the source/destination cache (`child`), the address, the MSI state, and data (only used sometimes). The code given with the lab provides a structure `CacheMemMessage` for these messages as shown below.

```
typedef enum { Req, Resp } ReqResp deriving( Eq, Bits, FShow );
typedef 2 NumCaches;
typedef Bit#(TLog#(NumCaches)) CacheID;
typedef struct{
    ReqResp   cmd;
    CacheID   child;
    Addr      addr;
    MSI       state;
    CacheLine data;
} CacheMemMessage deriving(Eq,Bits);
```

5.2 Message FIFOs

In order for the MSI protocol to work, the message FIFO needs to give priority to responses over requests. One way to implement this is to have two FIFOs, one for responses and one for requests as shown in Figure 3. The enqueue and dequeue logic puts the messages in the right FIFO, and choses a FIFO to dequeue from (giving priority to the response FIFO if it is not empty).

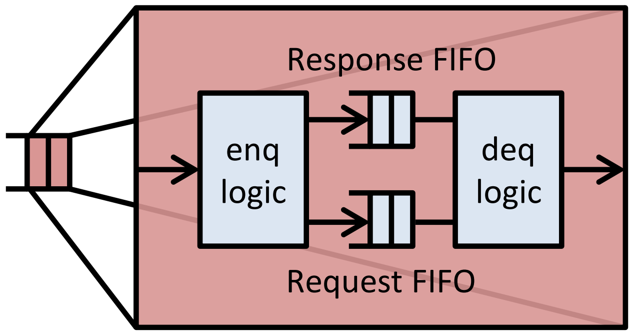

Figure 3: Detailed view of the message FIFO.

5.3 Message network module

The message network module will take messages from the two caches and send them to the protocol processor one at a time. It should also give priority to responses over requests. The message network module will also have to take messages from the parent and send them to the right cache.

6 Atomic Read-Modify-Write Operations

Atomic read-modify-write (RMW) operations are essential to programming for multi-core processors. Load-link (`LL`) and store-conditional (`SC`) are two primitive instructions in the SMIPS ISA that work together to enable atomic RMW operations. Their assembly code and function are very similar to `LW` and `SW`, but they have hidden side effects that allow for atomic RMW operations.

When an address is read with `LL`, the read address is tracked for changes. If the data at the address changes before the `SC` instruction, the store will not be performed and the `SC` instruction will return failure. If the data at the address did not change between `LL` and `SC`, then the store will happen, and it will return success. This read-modify-write operation is ensured to be atomic because it will only happen if there are no changes to the data in memory between `LL` and `SC`.

An example of some code that uses `LL` and `SC` to perform `MEM[$a0]++` atomically can be seen below.

```
LOOP:
    LL      $t0, 0($a0)  # load MEM[$a0]
    ADDI    $t0, $t0, 1  # increment
    SC      $t0, 0($a0)  # try to store to MEM[$a0]
                         # $t0 will contain 0 if not successful
    BEQ     $t0, 0, LOOP # try again if not successful
```

If the location `MEM[$a0]` was updated between `LL` and `SC`, then the `SC` instruction would not be successful and this code would try again.

6.1 Link address register

The atomicity of `LL`/`SC` pairs is tracked with a link address register in the cache. The link address register stores values of type `Maybe#(Addr)`. Valid values signal linked addresses, and invalid values signal broken links.

The link address register is written to by `LL`, and it is set to invalid when the cache line containing the link address is evicted. The store from the `SC` instruction is performed only if the target address for the store is still contained in the link address register.

6.2 Implementing `LL` and `SC`

Unlike other instructions, Implementing `LL` and `SC` requires modifications to your processor pipeline and data cache.

6.2.1 Pipeline modifications

The provided code includes all necessary pipeline modifications for `LL` and `SC`. The decode and execute functions were modified to handle `LL` and `SC`. The memory pipeline stage was modified to request loads and stores for `LL` and `SC` using the newly added `MemOp` values `Ll` and `Sc`. Additionally, the write back stage was modified to wait for a response from every `Sc` memory request.

6.2.2 Cache modifications

The provided cache implementation ignores the conditional portion of `SC` and always performs the write and returns success. You will need to upgrade the cache to recognize `MemOp` values of `Ll` and `Sc` as special instructions. `Ll` requests will have to update the link address register. `Sc` requests will have to check the link address register before attempting to write and return success or failure through the `resp` method. If the `SC` was successful, the cache should respond with a `1` for the write. If the `SC` failed, the cache should respond with `0`.

7 Multi-Core Software

Most software that runs with multiple threads relies on the operating system to deal with new threads for the program. To simplify our SMIPS multi-core processor, we write our multi-core programs with two main functions: `main0` and `main1`. The start code in `programs/lib/start.c` loads the core's id from the coprocessor, changes core 1's stack pointer (so core 0 and core 1 do not use the same stack), and calls main 0 and main 1 on core 0 and core 1 respectively. The SceMi test bench keeps track of data written back from each core, and prints it for the user to see, along with what core the message came from.

For single-core programs, the start code runs the program on core 0 and has core 1 immediately return successfully.

7.1 Multi-core programs

There are four multi-core example programs included with the code for this project. These are run with by the `./run_mc_programs` script. These do not test specific functionality like the assembly tests, and they do not test performance like the benchmark tests. They just are example multi-core programs to give you some rough feedback about the functionality of your processor.

- `coreid` – Just prints each core's core id and returns success.
- `hello` – A simple producer/consumer hello world program. One thread copies `"Hello World!"` from a `char` array to an `int` array. The second thread takes the `int` array and prints it to the screen.
- `incrementers` – This runs two threads incrementing a shared variable. Each thread keeps track of how many times it has incremented the shared variable. The shared variable is not incremented atomically, so the shared variable does not usually show the number of times the two cores think they incremented it.
- `incrementers_atomic` – Does the same thing as `incrementers`, but it does so with `LL` and `SC` so the shared count is incremented atomically.

These programs are all written in C, so you should be able to use them as a framework to make your own multi-core programs.

7.2 Making your own test for MSI protocol

There are no tests provided to specifically test that the MSI protocol has been implemented properly. You can run the multi-core example programs above to lightly test your processor, but to run a more thorough test on your processor, you need to make your own test. Your test should cover as many cases as possible out of the 8 situations of the MSI protocol shown in lecture 22. To see how many cases your test covers, insert `$display` statements in your cache corresponding to each of the 8 situations so you know what situation is being handled at each moment. Furthermore, your program should check for expected behavior and return a non-zero number on failure.

7.3 Making your own benchmark

There are also no true multi-core benchmarks provided with the initial project code. To get a good sense for how your processor handles multi-core applications, you need to create a multi-core benchmark. Your new benchmark could be an entirely new program, or it could be as simple as a multi-core implementation of an existing single-core benchmark.

7.4 Building and running your new multi-core program

To build your new multi-core program, you will first have to add it to the makefile. First add your program to the following line:

```
# List of MC programs
mc:= coreid hello incrementers incrementers_atomic <program>
```

Now add the dependencies for your program:

```
# MC program dependencies
build/<program>.mc.exe: <source files for program>
```

Now that your program has been added to the makefile, you can build it with `make` and run it with `./run_mc_programs` in either the `scemi/sim/` or `scemi/fpga/` directories.

7.5 Tips for multi-core programs

- Look at the existing multi-core programs before writing your own programs.
- Shared data between cores need to be marked with `volatile` so the compiler knows to read the data from memory before using it instead of keeping a local copy in a register for long periods of time.
- If you want to achieve maximum performance from your program, do not forget about cache lines. If your two threads are constantly writing to the same cache line (even if they are writing to different addresses) they will be stalling a lot while trading off `M` status between the two threads. To avoid this, you can initialize long vectors of integers and use two locations that are 8 apart; they will be guaranteed to be in different lines.
- Keep your instructions and data types simple. The current SMIPS processor does not implement all of the instructions that the compiler may use.
- The stack pointers for core 0 and core 1 are neighbors. If your program uses an excessive amount of stack storage or recursion, your stacks may collide and cause unexpected results.

8 Competition

There will be a competition between all the teams working on the final project. Whoever has the highest average instructions per second (IPS) for their final design will win.

8.1 Constraints

Your final processor must be a two-core SMIPS processor that is compatible with the provided SceMi simulation and FPGA infrastructure and able to run the provided benchmarks and programs. It must be synthesized on an XUPV5 FPGA with the default memory clock speed. Other than that, you are free to make any changes.

8.2 Submission

Before December 10th at 11:49 PM, you must submit a bitfile for your final competition design, along with the `mkBridge.srp` synthesis report, to the TA for judging.

8.3 Judging

Your processor will be tested on an FPGA against a set of single-core and multi-core benchmarks. Some of these benchmarks you will have seen before, and some of these benchmarks will be new to you.

8.4 Ideas

Here are some ideas for ways to speed up your processor:

- Implement associative caches.
- Implement a non-blocking cache.
- Implement a more advanced branch predictor.
- Add an L2 cache to the parent protocol processor.
- Allow for data from downgrade-to-S responses to provide data for upgrade-to-S requests without passing through DRAM.

9 Project Presentation

You will present your project on the last day of class, December 11th, at 3:00 PM. Your presentation will cover some portion of your benchmark, your test, and/or the improvements you made to your processor for the competition. More details about the presentation will be available at a later date.

10 Project Requirements

This project has 8 parts as listed below.

1. Implement the MSI protocol for cache coherency.
2. Implement the `LL` and `SC` instructions in the L1 caches.
3. Implement any additional changes to the processor to maximize your average IPS for the competition.
4. Synthesize and run your design on the XUPV5 FPGA.
5. Write a test for the MSI protocol.
6. Write a multi-core benchmark.
7. Write a project report (about 2-4 pages) covering how you met the above requirements.
8. Present your project on the last day of class.

All of these parts are due on the last day of class: December 11th by 3:00 PM.