Final Project: Part 1
Developing a Non-Blocking Cache Hierarchy

1 Multi-Core SMIPS Processor

Figure 1 shows a diagram of a processor without cache coherency. 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.

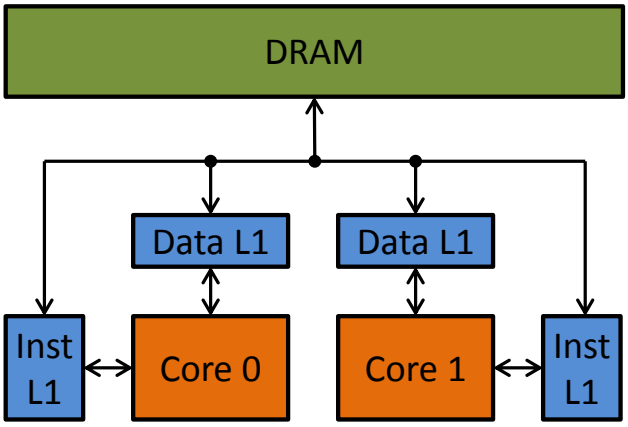

Figure 1: The initial flawed multi-core design.

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 non-blocking cache hierarchy with the MSI protocol implemented for the data caches.

2 Message Network

In order to get the MSI protocol to work, we need a message network for communications between the two caches and the parent protocol processor. This network will be made up of multiple message FIFOs and a message router module. A diagram of the network can be seen in Figure 2.

2.1 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 to upgrade/downgrade to, and data (only used for some responses). The code given with the lab provides a tagged union `CacheMemMessage` for these messages. A `CacheMemMessage` is either tagged as a `Req` with a value of `CacheMemReq`, or it is tagged as a `Resp` with a value of `CacheMemResp`. The associated typedefs are shown below:

```
typedef 2 NumCaches;
typedef Bit#(TLog#(NumCaches)) CacheID;
typedef struct{
    CacheID     child;
    Addr        addr;
    MSI         state;
```

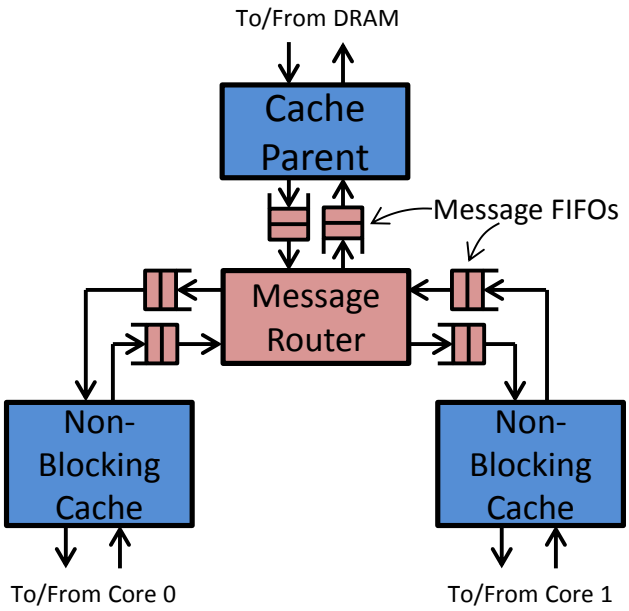

Figure 2: The protocol processors and the message network that implement the MSI protocol for cache coherency.

```
    CacheLine    data;
} CacheMemResp deriving(Eq, Bits, FShow);
typedef struct{
    CacheID      child;
    Addr         addr;
    MSI          state;
} CacheMemReq deriving(Eq, Bits, FShow);
typedef union tagged {
    CacheMemReq     Req;
    CacheMemResp    Resp;
} CacheMemMessage deriving(Eq, Bits, FShow);
```

2.2 Message FIFOs – `MessageFifo.bsv`

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. Each FIFO has its own enqueue method, but the dequeue logic choses the right FIFO to dequeue from (giving priority to the response FIFO if it is not empty). The message FIFO's interface is shown below:

```
interface  MessageFifo#(numeric type size);
    method Action enq_resp( CacheMemResp d );
    method Action enq_req( CacheMemReq d );
    method Bool hasResp;
    method Bool hasReq;
    method Bool notEmpty;
    method CacheMemMessage first;
    method Action deq;
endinterface
```

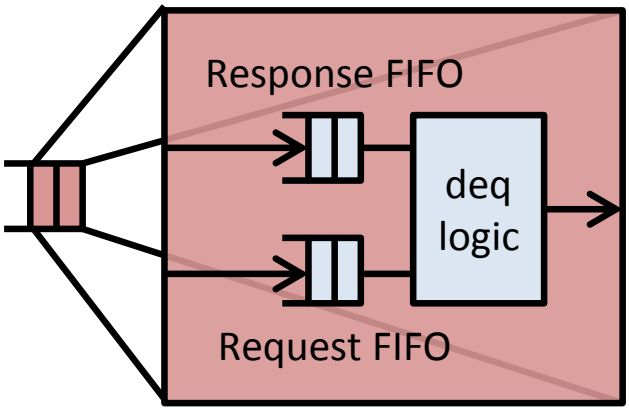

Figure 3: Detailed view of the message FIFO.

2.3 Message Router – `MessageRouter.bsv`

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

3 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.

3.1 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`.

3.2 Protocol communications

The MSI protocol involves communications between each of the L1 cache's protocol processors 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.

3.3 Parent Protocol Processor – `ParentProtocolProcessor.bsv`

The parent protocol processor is in charge of coordinating memory messages to and from the children caches, and it is in charge of communicating with the main memory. The parent protocol processor performs rules 2, 4, and 6 of the MSI protocol as introduced in class.

Your parent protocol processor will need to keep track of the state of each of the children caches. This includes:

- `child_state[index][cache_id]` – The current state of the cache line `index` in cache number `cache_id`.
- `child_tag[index][cache_id]` – The current tag for the cache line `index` in cache number `cache_id`.
- `waitc[index][cache_id]` – If the processor is waiting for a downgrade response for cache line `index` from cache number `cache_id`.

This module also needs to interact with main memory. When a cache is requesting an upgrade from I to S or M, it will need data, so this module is in charge of sending a read request to main memory to get the needed cache line. When a cache is sending a downgrade response from M to S or I, it will need to write back data, so this module needs to send a write request to main memory to write back.

3.4 Child Protocol Processor

The child protocol processor will be integrated with the non-blocking cache. The non-blocking cache will be in charge of performing rules 1, 3, 5, 7, and 8 of the MSI protocol as introduced in class.

The child protocol processor only needs to keep track of the state of its own cache. This includes

- `state[index]` – The current state of the cache line `index`.
- `tag[index]` – The current tag for the cache line `index`.
- `waitp[index]` – If the cache is waiting for an upgrade response for cache line `index`.

4 Non-Blocking Cache – `NBCache.bsv`

A note about the code from class: Yes, you were given code in the lecture for implementing a non-blocking cache, but for various reasons it didn't work. Also, that code was for a single core processor; your non-blocking cache will include cache coherency. For this project, you should first understand how the non-blocking cache should work, and then you should start from scratch. When you are creating state elements for your cache, make sure to use vectors of registers, not RegFiles (for the same reason you needed a vector of registers in lab 7).

Below is some pseudo-code for the non-blocking cache you will be creating:

```
request method:
    if ld request:
        if hit in store queue -> return hit
        if hit in data cache -> return hit
        otherwise:
            insert into load buffer
            send upgrade request if necessary & possible
    if st request:
        if hit in data cache and empty store queue -> update data cache
        otherwise:
            insert into store queue
            send upgrade request if necessary & possible

handling message from memory:
    if response:
        update cache according to response
        save response to register
        go to load hit state
        load hit)
            if hit in load buffer -> remove entry and return hit
            if no hit in load buffer -> go to store hit state
        store hit)
            if hit at head of store queue -> dequeue hit and update data cache
            if no hit at head of store queue -> go to store req state
        store req)
            resend request for head of store queue if necessary & possible
            go to load req state
        load req)
            if conflicting address in load buffer -> resend request if possible
            go to ready state
    if request:
        handle request according to rules 5 and 7

*each mention of sending an upgrade request (rule 1) implies sending a downgrade response
if necessary (rule 8).
```

4.1 Internal Modules

There are two internal modules used by the non-blocking cache to keep track of loads and stores. The first is a load buffer, and the second is a store queue. Implementations of each are provided with the initial code.

4.1.1 Load Buffer – `LdBuff.bsv`

The load buffer is provided for you and it has the following interface:

```
interface LdBuff#(numeric type size);
    method Maybe#(Tuple2#(Bit#(TLog#(size)), LdBuffData)) searchHit(Addr x);
    method Maybe#(LdBuffData) searchConflict(Addr x);
    method Action remove(Bit#(TLog#(size)) x);
    method Action enq(LdBuffData x);
endinterface
```

The methods do the following:

- `searchHit(x)` – looks for the next load hit corresponding to the cache line of address `x`. If there is a hit, it returns a valid `Tuple2` of the index in the load buffer and the `LdBuffData` corresponding to the matching load. If there is no hit, it returns an invalid value. In order to remove this hit from the load buffer, you need to use the index returned from this method in the `remove` method.
- `searchConflict(x)` – looks for a load request to the same index as address `x` but a different tag. This is used to find new upgrade responses to send.
- `remove(x)` – removes an entry from the load buffer. This should only be used with indexes obtained from the `searchHit` method.
- `enq(x)` – used for enqueuing new items into the load buffer. See `LdBuff.bsv` for the definition of the type `LdBuffData`.

4.1.2 Store Queue – `StQ.bsv`

The store queue is provided for you and it has the following interface:

```
interface StQ#(numeric type size);
    method Maybe#(Data) search(Addr x);
    method Action enq(StQData x);
    method Bool empty;
    method Action deq;
    method StQData first;
endinterface
```

The methods do the following:

- `search(x)` – looks for the most recent store to address `x` and returns the data if there is a match. If there is no match it returns an invalid value.
- `enq(x)` – used for enqueuing new items into the store queue. See `StQ.bsv` for the definition of the type `StQData`.
- `empty` – returns true if the store queue is empty. It is necessary to check this before updating the cache for a store hit.
- `deq` – dequeues the first element from the store queue.
- `first` – returns the first element in the store queue.

4.2 Advice

4.2.1 Handling tagged union `CacheMemMessage`

Messages from the parent protocol processor come as values of the tagged union type `CacheMemMessage`. To best handle these messages, use a case statement with pattern matching like shown below.

```
case( cache_mem_message ) matches
    tagged Resp .resp:
    begin
        // handle CacheMemResp resp
    end
    tagged Req .req:
    begin
        // handle CacheMemReq req
    end
endcase
```

4.2.2 Handling rules 1 and 8

Rules 1 (upgrade-to-y request) and 8 (downgrade-to-y request) work together to get a desired line in the cache with a specified MSI state. If a different line is occupying the space where the desired line will be, then rule 8 is used to remove that line. Rule 1 is used to send an upgrade for an invalid slot or a valid slot with a lower state. Since rule 1 enqueues a request, rule 8 enqueues a response, and message FIFOs can handle a request and a response in a same cycle, then rule 1 and rule 8 can be performed in the same cycle.

There are four different situations where rules 1 and 8 are used in the non-blocking cache: load miss, store miss, resend request from head of store queue, and resend load request from load buffer. To reduce the complexity of performing rules 1 and 8 and these four situations, you can write some functions to help you. The following shows two function prototypes for sending an upgrade request with rules 1 and 8.

```
function Bool can_send_upgrade_req( Addr a, MSI y );
    // Returns true if the cache can send an upgrade to y request for address a.
    // If a downgrade response is necessary, this includes if that downgrade is possible too.
endfunction

function Action send_upgrade_req( Addr a, MSI y );
    return (action
        // If rule 8 is necessary, this sends an downgrade response for the old cache line
        // In all cases, this sends an upgrade request for the new cache line or for the new state
    endaction);
endfunction
```

The function `can_send_upgrade_req` should return true if it is possible to use rule 1 (and possibly rule 8) to get the cache line corresponding to address `a` with state `y`. This function should return false if the cache line is already in the desired state. The function `send_upgrade_req` sends the request for rule 1 (and possibly the response for rule 8) to get the cache line corresponding to address `a` with state `y`. By using these two functions, you can significantly reduce the complexity of your written design.

These two functions should be defined within the `mkNBCache` module so it can read and write the internal state of that module.

4.2.3 Handling memory responses

One of the most complicated things in the non-blocking cache design is the state machine that is used to handle memory responses. When a response comes back from memory, you need to first look for hits in the load bufer, then look for hits in the store queue, and then send more requests if possible. The states in this state machine can be expressed by the following enumeration:

```
typedef enum { Ready, LdHitState, StHitState, StReqState, LdReqState } NBCacheState deriving (Bits,
    Eq);
```

The actions for each of these states are described in the pseudo-code outline of the non-blocking cache above.

While the non-blocking cache is performing these actions, it needs to stop handling incoming requests from the processor and all incoming messages from the memory. Therefore the state needs to be `Ready` in order to do these actions.

This state machine has many optimizations that can be performed to trim some computation time from it. For example, if a line is upgraded from I to S, then after the `LdHitState` you can skip the `StHitState` go straight to the `StReqStateState`. Likewise, if a line is upgraded from S to M, then you can skip the `LdHitState` and go straight to the `StHitState` when starting the FSM. Also, the two states `StHitState` and `StReqState` can be combined into a single state that performs both tasks.

5 Part 1 Requirements

In this part, you are expected to implement the four modules listed below to create a coherent, non-blocking, cache hierarchy.

1. Message FIFO
2. Message router
3. Parent protocol processor
4. Coherent non-blocking cache

Each of these modules have a test bench provided in the initial code. These (and possibly other) test benches will be used to check the functionality of your implementations.

5.1 Initial Code

The initial code for this project is in your group's git repository. You can get it by running the following command:

```
git clone /mit/6.175/groups/group<N>/project-part1.git
```

where `<N>` is replaced with your group's assigned number. If you are not part of a group, the initial code is also available on the course website.

The root directory contains a BSV source file for each module to implement, and it contains various other BSV source files for required typedefs, functions, and library modules. There are also folders containing tests for the modules.

1. `message-fifo-test` – Tests the message FIFO in `MessageFifo.bsv`
2. `message-router-test` – Tests the message router in `MessageRouter.bsv`. This test also uses the message FIFO.
3. `ppp-test` – Tests the Parent Protocol Processor in `ParentProtocolProcessor.bsv`. This test also uses the message FIFO.
4. `nb-cache-mini-test` – Tests the non-blocking cache in `NBCache.bsv` with load/store hits/misses one at a time. This test also uses the message FIFO.
5. `nb-cache-test` – Tests the non-blocking cache in `NBCache.bsv` with more advanced test cases. This test also uses the message FIFO.

To test any of these tests, go into the directory, run `make`, and run the generated executable. If the test passed, it will print `PASSED`. If it failed, it will print `FAILED`. To get more information about where your processor is failing a test, change the `debug` variable inside the testing module to `True`, and recompile.