Lab 10,11,12 Multi-core Processor

Over these three labs you will build a multi-core processor. Labs 10 and 11 are intermediate labs and will only be tested to ensure progress. Lab 12 is when you must demonstrate the completed design. You should remove MMIO from your design for these labs.

The following are to be completed throughout the course of labs 10 through 12.

1. A block diagram of your multi-core memory hierarchy.(Dcache x2 + coherence controller) A state diagram and block diagram of your updated Dcache. A state diagram and block diagram of your coherence controller.

   All diagrams must be in explicit detail and in electronic format for any points. No hand drawn or photo-copied/scanned diagrams will be accepted.

   > NOTE
   >
   > 
   > Any evaluation turned in without block diagram(s) of the design will receive a grade of 0.

2. Parallelize the merge sort algorithm in MIPS assembly and verify it functions correctly.
3. Build a coherence controller which implements the MSI protocol. Implement the load-linked (LL) and store-conditional (SC) instructions.
4. Connect two pipelines, two L1 cache (I+D) hierarchies, the memory, and the coherence controller to create a complete dual-core processor. Verify functionality in source.
5. Synthesize your multi-core processor. Verify the synthesized version of your multi-core is functional.
6. Verify your design in hardware.

1 Background

The multi-core processor will start a new project. You will want to create a `project4` directory in the `ece437` directory. You will need to setup this directory correctly. Re-read lab1's handout for a reminder on how to run dirset.

In this lab you will design a multi-core processor by replicating your pipeline design and cache hierarchy, and connecting them to a coherence controller.

Your threading model will be very simple. Each processor will start at a predetermined address and start executing the thread located at that address. The lecture notes will serve as a excellent reference for using locking primitives and programming models. Be sure to use these when writing your sort algorithm.

NOTE

Remember the top of the address space is 0x7ffc, and both processors must maintain their own separate stack.

2 Software

You are to program a parallel version of merge sort. The values may be divided up among the processors, and sorted. The merge must use locks or flags to correctly merge the data.

The purpose of this assignment is to get some idea about parallel programming and to understand how your dual-core will be designed to run such a program. This is not a software course, so do not make this program too complicated.

WARNING

You probably do not want to do the recursive merge sort in assembly. Since we only have two processors, you only need to divide and conquer once. That means, split the input array into half, make each processor sort half of the array with any sort algorithm you like, and then merge two halves to obtain the result.

The input data is provided on the course website. Use your program to sort this data .

To simulate a dual-core design use the simulator program that you have used in lab3 to simulate your assembly files (the asm simulator). The command to run the simulator in dual-core mode is as follows.

```
>sim -m
```

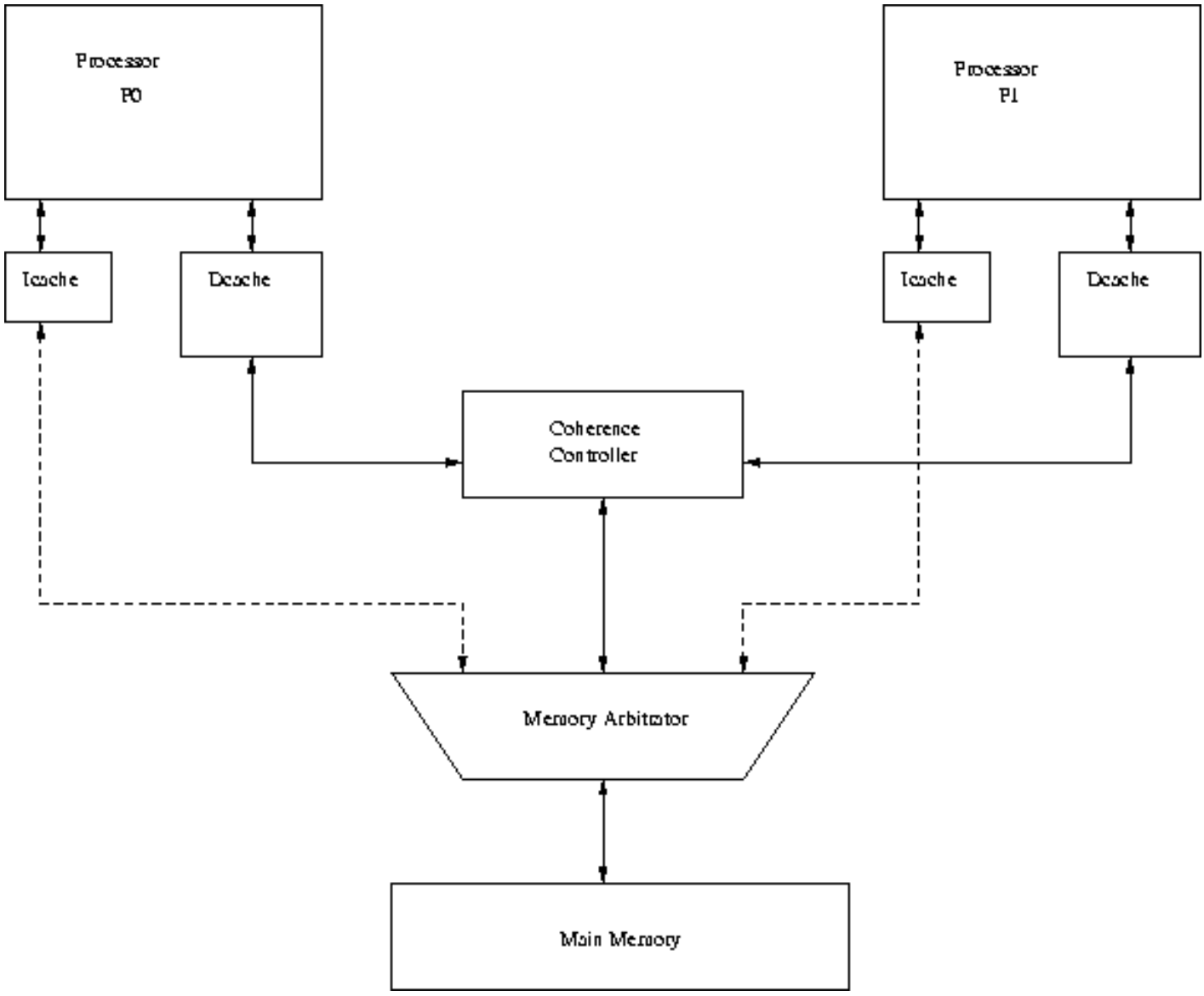
NOTE

The sim -m mode assumes that your program for first processor starts at 0x0000 and program for second processor starts at 0x0200 by default. You can still use "-t" to get the detailed trace.

3 Multi-core Design

Figure 1 is a diagram of the basic dual-core design. The key part is the memory hierarchy. This hierarchy must be able to control the flow of data between the two processors' caches and the main memory in a coherent manner.

Figure 1 Dual-core Components



The component that controls the flow of information between the Dcaches and main memory is the coherence controller. This component will check the memory accesses made by each processor's Dcache and determine where the data is and what action needs to occur to get the requested piece of data.

3.1 Load Link and Store Conditional

You have to implement these two new instructions. The opcode for Load Link (LL) is 0x30 and the opcode for Store Conditional (SC) is 0x38. These instructions are used together to implement an atomic read-modify-write operation. This facilitates the implementation of synchronization primitives such as locks.

The LL and SC instructions work in conjunction with a 'link' register. Every core has its own link register. When a core (P0) executes LL, it loads the requested memory address into its special 'link' register. When P0

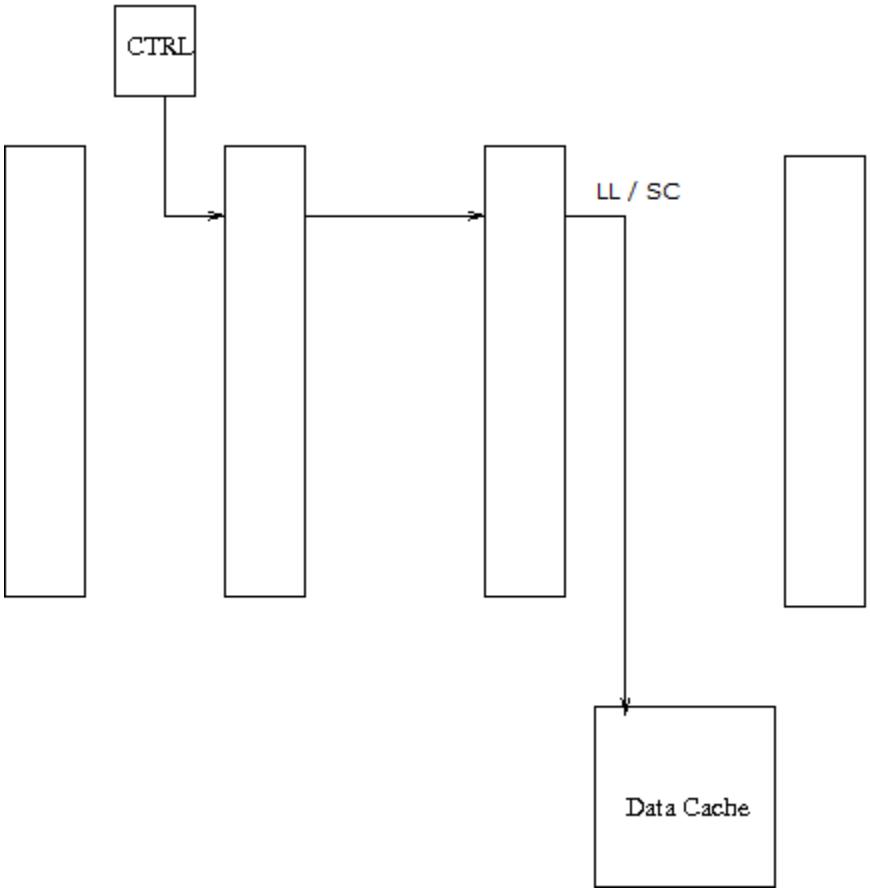
executes SC, if the target memory address matches the valid value stored in its 'link' register then the SC succeeds. If in the meantime any other core (say P1) performs a memory write operation to the same memory location (ie. to the address in P0's link register), then the 'link' register of P0 is invalidated. In that case SC executed by P0 fails. For Example, any SW or SC from P1 will need to check the link location of P0, and if there is a match, the 'link' register will need to be invalidated. In other words, any BusRdX request coming from P1 to P0 should check P0's link register for possible invalidation.

The LL instruction will act similar to a load (lw) instruction. It will load the value at the address into a register. Additionally it will write the address into the 'link' register.

The SC instruction will act similar to a store (sw) instruction. If the address matches that in the 'link' register, the sc will store to the memory location. Otherwise, the sc will not modify the memory. The instruction also returns an appropriate condition code to notfiy the processor of its success or failure. Check 'asm -i' for the rtl details.

You need to keep in mind that sc needs to write back the condition code to a register. Think about how this might affect forwarding. The 'link' register should be located inside the data cache. It is recommended you add a new signal coming out of your ID stage and filter it through your pipeline registers to the mem stage. This signal will tell the cache that the current memory read or write is linked. Also, remember that the link register will be invalidated due to any memory write access at that location by any other core. You will want to have a way to determine if the 'link' register is valid.

Figure 2 LL and SC Signal



3.2 Coherence Controller

Your coherence controller will implement the MSI coherence protocol discussed in lecture. Your Dcache already keeps track of the state of each block with the valid and dirty bits. If a block is valid and dirty it is in modified state. If a block is valid and not dirty, it is in shared state. Finally, if a block is not valid, it is in the invalid state. Again, we are assuming you do not overwrite program code, or else you will need to add the Icache to the mix.

Every time the processor reads or writes to the Dcache it must send the action, address, and data to the coherence controller. This controller will arbitrate which transaction to complete. It will then check the other processor's Dcache to determine what action needs to occur. Refer to the MSI state diagram in the lecture notes for transactions.

The coherence controller will take a memory access request, check if the data is in the other processor's cache. If it is, it will signal the Dcache to take the appropriate action. If it is not in the other processor's cache it will retrieve the value from main memory and have the requesting Dcache take the appropriate action.

You will find it useful to arbitrate based on a priority. This priority can be processor ID, or in our case you pick one processor and call it P0. If any simultaneous access requests are received by the coherence controller it will pick the processor with the highest priority (lowest pid) to serve (in our case P0).

Remember, you must handle the case when both processors are attempting an LL/SC pair. This means that 'link' registers should be invalidated on a SC instruction when there is a match. This creates a "first to sc" race condition between processors on a given address. In other words, if the two processors are holding links to the same memory location, then the first one that performs a SC on that location wins and the other must re-acquire the lock. Make sure your ll/sc implementation is consistent with the MSI protocol.

NOTE

There is also a reference reading under "Links" on the course website explaining how ll/sc works with coherence.

WARNING

For grading purpose, Please name your main Coherence Controller design file as coco.vhd.

3.3 Snooping

Snooping is what the coherence controller will do to determine what is in a Dcache. The coherence controller looks for tag, and state when it snoops. In some cases it will need the actual data in the data ram. This snooping will tie up the Dcache data ram(it is assumed your tags are stored in the data ram), which means the processor does not get to use the Dcache while the coherence controller is snooping around.

You will need to modify the Dcache to allow the coherence controller access to the data ram. When the coherence controller is accessing the cache data ram the processor can not use the cache, thus it must generate a memwait on any processor read/write (lw/sw) accesses until it is done snooping.

3.4 Memory Arbitrator

This is the priority mux that decides which stage gets to access main memory. This is similar to the one that was used in the uniprocessor with caches. It is modified slightly so that the coherence controller gets top priority. Then the first processor (P0) Icache followed by the second processor (P1) Icache. This order may be subject to change.

3.5 Program Counters

The two processors must start at different addresses. The first processor will start at address 0x0000 and the second processor will start at address 0x0200. These addresses are subject to change.

The best way to do this is have a mux with both starting addresses and have a constant bit to set the PC to either 0x0000 or 0x0200 when you assert nreset.

4 Turning In

Again, make sure your `Makefile` works, make sure your cpu.vhd and tb_cpu.vhd are the same as given on the website (do not commemt any signals out in the entity), make sure your own top level file is called mycpu.vhd, your main cache files are called dcache.vhd and icache.vhd, and your main coherence controller file is called coco.vhd before you do your submission. Type the following from your project4 directory:

```
~/ece437/project4> make clean
~/ece437/project4> make lab12
```

Lastly, from your `project4` directory, type:

```
~/ece437/project4> submit -p lab12
```