MIPS Multicore Processor

CPE404 Final Project Report

Jared Hayes

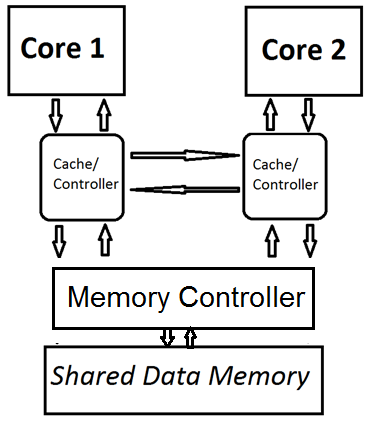
Nick Repetti

Instructor: Dr. Sarah Harris

8 May 2015

As more time passes, single processor performance growth has decreased. We cannot further increase processor performance without making costly sacrifices in other areas, such as power consumption. In response to this, companies have started to focus their efforts on multicore processor designs. Multicore processors utilize parallel processing to compute problems faster by splitting up the computations between multiple cores.

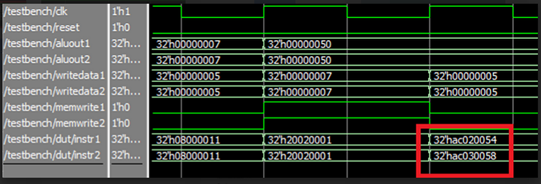
We are designing a MIPS multicore processor with shared centralized memory in Verilog. This multiprocessor is built off of a pipelined MIPS single core processor that we have already designed prior to this project. This project only demonstrates the implementation of two cores for the sake of simplicity, although, more cores could be easily instantiated by replicating our process multiple times. Each core communicates to a shared data memory. This will allow a core to have access to data that has been processed by another, so that each core can process separate tasks while having access to all necessary data. However, the need to constantly access memory will hinder performance and may neutralize the performance benefits of multicore processing. Therefore, each core will have its own cache that will store repeated data accesses to dramatically reduce how often a core will need to go to memory. A cache controller module in each core will communicate between the core, data memory and the caches of other processors.

.

**Process**

*Duplicating the MIPS Single Core*

The first step is to duplicate the single core processor and run both cores separately. In order to accomplish this, we instantiated a second device under test (DUT) in our top module for the second core. Each core runs on the same clock, but has distinct hardware. Although they share the same data memory, they have distinct instruction memory modules in order to run two separate programs at once. In order to test this phase, we created two separate programs that performed basic tasks. The first program ran on the first core and stored a value from register 2 into memory address 84 and the second core ran a second program that stored a value from register 3 into memory address 88. The simulation shows the two separate instructions being performed simultaneously in machine code.



*Creating an L1 Direct-Mapped Cache*

Each data entry in the cache must have the following components: data, set bits, tag and a valid bit. The cache takes in an instruction and extracts these bits from the machine code of that instruction. Since instruction addresses have a four byte offset, the lowest two bits of the instruction is ignored as they are used to reflect this fact. The set bits are determined by the desired size of the cache. For this project we decided on a cache with 32 entries, so we have 5 set bits [6:2]. The remaining bits of the instruction make up the tag of the cache entry. The set bits act as an index to identify entries in the cache. The tag bits of an incoming instruction is compared with the tag bits of entries in the cache to determine if that space in memory has already been recorded. The valid bit is used to determine if an entry in the cache still contains valid data. When the current instruction is trying to access a value in memory that is already in the cache it registers a “hit” signal and takes the data directly from it. If that value is not present in the cache, it is registered as a miss and has to get the data from memory and fill the cache with the necessary information.

*Cache Controller/Memory Module*

We implemented a writeEnable signal and a readMem signal for the cache controller. WriteEnable is set high when we are writing to the cache and readMem is set high when the cache misses and the processor must get the data from main memory. A hit is registered when writeEnable/readMem is low, the tag bits of the current instruction matches the tag bits of one of the entries in the cache, and that entry is valid. If writeEnable is asserted then it records the tag, data and valid status of the instruction into the cache. If readMem is asserted then it does the same, except it gets its data from memory.

When a core is attempting to write to memory, we check if its writeEnable signal is high and that its hit signal is low. We also check to make sure that another core is not attempting to write to memory at the same time. If that is the case, then we give certain cores higher priority. If the hit signal is low and the writeEnable signal is also low, then that core reads from memory. It does not matter if they both attempt to read from memory at the same time.

*Establish Cache Coherency*

In the event that a core attempts to read shared data after another core writes to that same address, there is a possibility that the data between the cores will not be coherent. Therefore, it is important to implement some sort of cache coherency protocol.

**Performance Analysis**

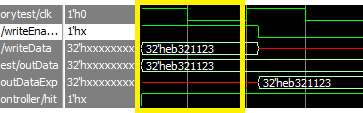
Multicore processors can substantially improve performance on many applications. In the modern world, programs can utilize multicore processors to help designate different tasks to be executed simultaneously. For example, apps that run on smart phones can take advantage of this by keeping one core specifically handling user input, while the other core(s) can be used to perform background tasks. This is immensely important as the main UI should never have to wait for some other task to be finished before the user can continue to use the phone’s other features.

We developed a test program to demonstrate the power of multicore. It basically reads in a bunch of numbers on one core, and has the other core run statistics at the same time. The second core is then able to output the results immediately after the first core is done reading in the last input. Unfortunately, there was a problem with the Hazard Unit on the MIPS Pipeline which was causing unexpected stalls during certain branch instructions. This caused our simulation to not produce the expected output.

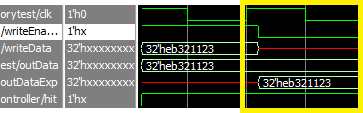
**Single Core Cache Simulation**

The following section will cover the simulation results of our project.

*Write/Read*

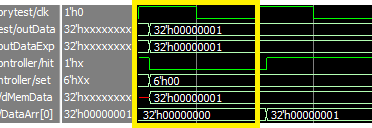


WriteEnable is high and we are writing some arbitrary data to memory, which will also be stored into the cache.



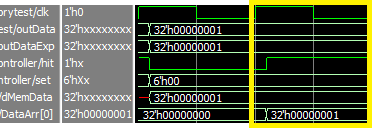
Here we are reading, so WriteEnable is low. The outData matches the expected value as Hit is high.

*Cache Miss*



Here we are attempting to do a memory read from an address that is stored in memory, but not in the cache. The proper data is generated at the output, but it came from the Data Memory rather than the cache. This can be seen as the Hit signal is low.

*Cache Hit*

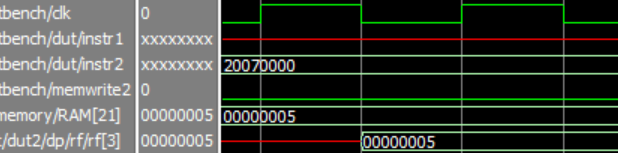


Here we are attempting to read from that same memory address, but this time it is in the cache because of the previous instruction. The expected data is at the output, and you can see that it came from the cache because the Hit signal is high.

**Dual Core Simulation**



Both cores are running separate instructions at the same time. Core 1 writes the value 0f 0x5 into memory address which is located in RAM[21].



Core 1 has completed its instructions, and Core 2 begins to read the value from memory address located in RAM[21]. It stores the correct value, 0x5, in $3 (rf[3]) of its own Register File.



Core 2 then adds 0x12 to $3 and then writes it back to the same memory address, located at RAM[21].

A major problem arises now, because Core 2 has overwritten a previous value that Core 1 has stored in its cache. When Core 1 goes to read the value, it will read it from its own cache instead of the actual value which is only in memory. This is where cache coherency comes into play.

**Future Work and Improvements**

We were unable to implement the cache coherency protocol in order to get the simulations to work effectively. The protocol we were implementing was the Write-Through Invalidate Protocol. Every time a core writes to a variable, it must invalidate the other cores’ cache block as well. This is accomplished through a state variable located internally to each cache block. The writing cache will change the state variable of every other core’s cache block. The cache block can either be valid, or invalid. Based on this state variable, the cache will either read from its given block, or it will be forced to read from memory.

Future work on this project would include implementing the Cache Coherency Protocol. There is enough work done on the other parts of this project that it could be used a starting point for any future group trying to implement Dual Core. Other improvements could be made by implementing an even more advanced Snoopy protocol. The project could then be placed on a DE2 board to demonstrate the power of running with dual cores.

**Appendix**

The following files are the important files that we implemented or modified in order to complete our project. They can all be found in the accompanied zip folder. All other files in the zip folder are relevant to the MIPS Pipeline.

*cachecontrol.v*

*memorycontrol.v*

*mipsmem.v*

*top.v*

*mipstest.v*