EE480 Assignment 4: Caching In

Implementor's Notes

Austin Langton, Adam Walls, Moustafa Seifalla

May 2, 2017

Introduction

Abstract: This project involves building a multi-cycle CPU implementation built on the SIK assembly language. This includes determining an encoding for each instruction, creating the actual processor and slowed memory along with a variable-sized cache for quicker instruction and data access, and testing it for various different coverages like line and toggle coverage.

General Approach The implementation of this project involved three primary steps:
1) Implementing a multi-cycle processor and slowed memory
2) Testing the design for coverage and proper functionality
3) Implementing a variable-size cache with prefetch capabilities

Interfacing with Main Memory

For our processor we used Dr. Dietz solution as well as his memory module. At first we had to get Dr. Dietz solution working, we had to change a few of his 'defines and turned some of them into registers in the modules.

The 1st thing we had to do was to tell the other process(thread) to wait until the current process from was retrieved its instruction from slow memory. To do this we had to look at the instruction register and we initialized them as waiting for instruction.

Then we allowed the first process that became active, to request an instruction. We implemented this by grabbing an instruction that we made, a signal that toggles (getInstrPid.) This signals which process is allowed to acces an instruction. This process needs to check if it has sent a load instruction request. If it has not sent a load instruction it will send one off, then when the other process becomes active and "sees" that, it will make sure that strobe is set to off, so that no new load request is sent.

Then we also had to check for wether the other proccess's MFC is 1. If so, it will place that data into the other process instruction register.

We also had to make sure we did not overwrite a data load request, So we used the signal ld(are we waiting on a data load?) and ldSt (for the store). If we were waiting on one of those two, the next clk cycle we turn off the strobe signal to signify that we are now waiting on a memory operation to complete. If the procces saw ld or ldStr was high then it would check for MFC and update the dest register and set ld to low and set strobe sent to low.

If strobe was set to high then we have a memory request, what type is determined by rnotw it is a read (i need something from the memory module). If rnotw is low then I am writing to memory.

We also had to add a Stage 0.5 to determine if we want to do an instruction load, memory data load, or an instruction write while simultaneously updating the cache.

Some Stage 0.5 operations:
1)toggle process id
2)It also determines the strobe sent variable and the strobe signal.
3)It will decide what to do with strobe, rnotw, addr(address of memory element you want to access), wdata, GetInstrPid and Instruction Register

The Cache

Our Cache is a fully mapped Cache. We use an index register to track where the next entry should be made. Every time a new cache element is made, it incremenets to the next entry in cache.

Data Cache is dirty when you have a a chache entry for a certain address and then you write to main memory over that address making that cahce entry invalid. To Handle this we will just always write through to memory and update the data in the cache entry.

The instruction cache is useful for speeding up loops because it will look in the cache and say: you need the instruction I have it here you dont need to request it from memory. Inruction Cache address fields is compared to the program counter of the current process to see if its valid. If not then we have a Cache miss then we have to wait by setting cachehit to low and set instruction register to an instrution wait opcode.

Data Cache is useful because if you need some variable that is loaded from main memory you can keep it in the cache. Data Cache is always given priority over Instruction Cache.

Notes about cache:
-Data Cache is always given priority over Instruction Cache.
-Valid === If something from memory has been put into the cache line
-Dirty === When CPU writes to the memory, but the cache contains old, obsolete data
-Not Dirty === When memory writes to the cache
-Cache line will be composed of the following: 1 valid bit, 1 dirty bit, 16 main mem address bits, 16 data/instr bits [33:0] cache [7:0] (index bits)
- As things are retrieved from memory, place them into the ir[pid] AND into the cache
- Instruction cache is ALWAYS clean Data cache on the other hand, is not

Problems and Solutions

We had a problem when checking the other processes MFC because we found that the MFC completed when the other process was active. MFC would output data from one clock cycle and then go back to garbage value.We had to make sure that each process's MFC was high.

/* TO DO (MORE PROBLEMS On Memory?) */

After we solved these problems we got a working version of utilizing the memory.

/* TO DO (MORE PROBLEMS On Cache?) */

As usual using Latex was a nightmare but we learned a lot by using it which I suppose is important.

Testing

/* TO DO */

Included Files

1) a5notes.pdf The Implementor's Notes
2) sik.aik Our Aik Encoding
3) ProcessorImplementation.v Verilog for our pipelined, multithreaded implementation of Sik with a Cache System
4) /*testing files */ testing files
/* TO DO (add testing files */