EE480 Assignment 4: Caching In

Implementor's Notes

Austin Langton, Adam Walls, Moustafa Seifalla

May 3, 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. In order to do this, we had to change a few of his 'defines and turned some of them into registers in the modules. His solution also never initialized the process ID register, which indicates which process is active. Once Dr. Dietz's solution was working, it became necessary to implement the slow memory module and request delayd instructions from it.

A stage 0.5 was added to control memory operations. It did several things including determine if an instruction load, memory data load, or an instruction write was necessary while simultaneously updating the cache.

The 1st thing we had to do to implement the slow memory module was to tell the other process(thread) to wait until the current process received its instruction from slow memory. To do this we had to look at the instruction register and we initialized them as waiting for instruction so that the processor knew to fetch an instruction for the first process.

In order to send a memory read or write, we had to control a few signals serving as inputs into the memory module: - strobe signaled to the memory module that have a memory read or write was being made - rnotw determined what type of memory peration it was * If it is high, then it signaled a read (I need to load something from the memory module). * If rnotw is low then it signaled a write to mem-

ory via a store instruction.
- addr determined which address from memory needed to be read or written to - wdata was used for writes when rnotw was low and contained the data to be written to memory

Then, we allowed the first process that became active to request an instruction. The processor first checks if a load is being made, then it checks several other signals: getInstrPid, strobeSent, and the instruction register for the current process. getInstrPid signals which process is allowed to request an instruction from memory. strobeSent is a two-element array which indicates whether each process has sent a load instruction request. If the process with instruction load permissions is active and 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 at the positive edge of the next clock tick.

We also had to check for whether the other proccess's MFC is 1. If so, it will place that data from the rdata output register of the memory module into the other process's instruction register.

Another issue we checked was making 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 destination register and set ld to low and set strobe sent to low.

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 direct-mapped Cache. We split the caches in half so that each process got half of the cache space. For instructions, it takes the address associated with the program counter and takes a modulo operation of the cache's capacity divided by 2 on the instruction's main memory address as the index into the cache. If it is process 0, it will be placed into the lower half of the cache. If it is process 1, it will be placed in the upper half of the cache. This prevents each processes instruction from being overwritten in the cache memory due to the direct-mapping implementation. For example with a cache size of 8 and instructions at memory addresses 0x0000 and 0x8000, the first instruction will be placed in the instruction cache index 0 while the other will have an instruction cache index of 4.

The instruction cache is useful for speeding up loops because it will look in the cache and say: you need the instruction from memory, but I have it here you. So, there's no need to request it from memory. Intruction Cache address fields is compared to the program counter of the current process to see if its valid. If not, there is a cache miss making it necessary to wait by setting cacheHit to low and set the 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

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.

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 check if MFC was high while the process that requested it was ”asleep”. So, we had both processes check the MFC output from the memory module

One final issue was making the memory module compatible with the soon-to-be-implemented caches. The cache needed addresses for memory fetches in order to properly operate. So, an extra addrOut register was added so that at the completion of every memory fetch, addrOut would output the address where a memory read, i.e. load, was made.

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

The first problem with cache was figuring out how to implement the direct mapping in order to make entries into it. Another was what the actual insides of the cache needed top look like like where the instructions/data need to go and what else were included within a cache block.

As usual using Latex was a nightmare but we learned a lot by using it which I suppose is important. As it turns out, debugging a processor of this complexity is a frustrating process. It proved to be quite a humbling experience.

Testing

The testing for this processor was done by checking to see if the instructions executed as intended. The first problem was discovering that the torf register was not being set as expected. This was caused by the stack pointer for the registers to go out of bounds of the register file bounds and would go to -1. This had to be fixed in one of the later stages of the pipeline. After that was remedied, it was on to fix the next bug in the long list of bugs encountered during testing.

The first few bugs caused the processor to loop forever without ever reaching a halt state. In order to fix this, a count variable was set in the testbench so that after 10000 clock ticks, the testbench would stop regardless if a halt state had been reached. This greatly sped up the debugging process and finding why the

processor never reached a halted state.

The next problem encountered for this aspect was discovering that the aik specifications were improperly set up. The opcodes for Jump, JumpF, and JumpT were all mixed up. So, when testing was going on and the processor was not jumping as expected for an absolute jump, it was found that the processor was getting a JumpT instruction instead of the intended absolute Jump instruction.

Verifying the cache sped up the processor was surprisingly pain free. It was obvious that it greatly sped up the processor for instruction fetches.

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)Test_Program_Description.txt Contains the assembly instructions associated with each test program
5)storeProg.vmem Contains the assembly opcodes showing the operation of loads and stores
6)testProg1.vmem Contains the assembly opcodes the cache speeding up the processor
7)testProg2.vmem Contains the assembly opcodes the cache speeding up the processor
8)testProg4.vmem Contains the assembly opcodes the cache speeding up the processor