Pipelined LC3b Processor

ECE411 Spring 2017
April 29th
Group2

Eric Johnson	erjohns3
Ben LeVine	bslevin2
John Pickar	pickar2

Introduction

The LC3*b* Processor is a simple processor whose instruction set and layout are taught to undergraduates in many universities across world. To demonstrate our knowledge obtained through the University of Illinois Urbana-Champaign's Computer Engineering program, we developed a pipelined LC3*b* processor and a multi-level memory hierarchy, both written in SystemVerilog. In doing so, we have demonstrated our knowledge and abilities in constructing pipelined designs, a design to increase processing throughput, and in memory hierarchy, a design which allows for faster data access, data reuse, and less reads and writes with physical memory.

Project Overview

As dictated by LC3*b* requirements, a 16-bit datapath was used and memory was designed to contain 2^{16} locations. In order to fit specification, each address holds one byte of data, allowing for acceptable addresses from `0x0000` to `0xFFFF`. We meets these requirements and fully implemented the LC3b instruction set (RTI is not supported). Furthermore, the processor is connected to two split L1 caches for instructions and data, which are in turn connected to an arbiter which communicates to an L2 cache. The L2 cache is wired to physical memory.

The pipelined stage is split into 5 stages: fetch, decode, execute, memory operation, and write back. The split L1 caches contain two ways, has a set index of 8, and holds 16 byte per line. Likewise, the L2 cache has four ways, a set index of 8, holds 16 bytes per line, and uses a pseudo-LRU policy to evict lines. The L1 caches are connected to the L2 cache through an arbiter.

Design Descriptions

Overview

The project was constructed over five different portions broken up within a period of three months time. Initially, a paper diagram was constructed to map out the planned implementation. From there, the five checkpoints, which were one to two weeks apart, were makers for certain features to be implemented. Between each checkpoint, a meeting was had with our mentor TA (Krishna Srinivasan) to discuss our progress and implementation.

Milestones

1. The software design of the pipelined architecture was started. All five stages of the pipeline were constructed; the processor at the time did not take into account data hazards, memory stalls, or branch prediction. Thus, it utilized magic memory in order to avoid cache misses and stalls. Because there was no hazard prevention or branch prediction, NOPs had to be inserted between commands for success operation. At the time, the LC3*b*α instruction set was implemented (ADD(i), AND(i), NOT, LDR, STR, and BR). In order to achieve proper register loading and mux selection, a control rom took the value from `IR`, analyzed it, and decoded the proper signals based on the bit values.

   To verify correct performance for this checkpoint, given test code was run and analyzed for correct performance.

2. The LC3*b* ISA was fully implemented at this checkpoint (Note that the instruction RTI is not supported by this processor). In addition, an instruction cache was connected to the fetch stage in order to pull

instructions. Likewise, a data cache was connected to the memory operation stage in order to manipulate data. The two caches were linked by a newly implemented arbiter to simulated memory (200 ns delay for access).

The arbiter was implemented to take one request at a time (the other request would wait) and process it with the next level memory. Once finished, it would switch to the other request in order to prevent starvation. In our design, the data cache was serviced first if both request arrived at the same time.

With to the new memory hierarchy, stalls were introduced to the pipeline due to memory delays. Proper consideration had to be taken into account so stages wouldn't load with incorrect data. Furthermore, with the introduction of LDI and STI, additional logic had to take place in the memory operation stage in order to account for two memory accesses. This was done by adding a new register that would load to the value of `1` when the first operation completed and reset to `0` when the second responded and the whole pipeline loaded.

As like the previous checkpoint, given test code was used to verify correct operations.

3. Hazard detection, forwarding, and a 4-Way Set-Associative pseudo-LRU L2 cache were implemented for this checkpoint. For data forwarding, the register file in decode was set to load on the negative edge of the clock, thus making it a transparent register file (this allowed our pipeline to store results in write back and read them in decode during the same cycle). In addition, the passed `IR` value for stages decode, execute, and memory operation were analyzed by a forwarding unit.

The forwarding unit decides whether or not to pass back data from the finished memory stage to the start of execute (for the upcoming memory stage), from the finished memory stage to the end of decode (for the upcoming execute stage), or from the finished execute state to the end of decode (for the upcoming execute stage). This allows forwarding from MEM -> MEM, MEM -> EX, and EX -> EX. In order to determine if data should be forwarded, it checked if source registers for the upcoming data matched destination registers for the data being processed. If it was a match, it would forward When passing data back to the end of decode (upcoming EX), if both of the destination registers from the sources matched, the data from execute was used as it was the most recent.

In addition to the forwarding policies, memory jumping was implemented. Operations in the execute stage and earlier were allowed continue down the pipeline and jump over anything stalled in the memory stage if it did not rely on it or a memory operation (Note: see advanced features for more information regarding this function).

The L2 cache was implemented by extending the L1 cache to four ways. In the process, the offset writer located in the L1 cache to address 2 bytes of data at a time was no longer necessary. In order to manage the four ways, a pseudo-LRU policy was implemented. This policy used three bits to determine which set to evict. By reading the first bit, it could tell if it should evict way one or two, else three or four. Depending which two were chosen, one of the remaining bits was used to pick the pseudo least recently used one.

Similar to the last two checkpoints, the verification of this material was done by given test code.

4. Branch implementation, performance counters, and an eviction write buffer were written and verified for this checkpoint. Branch prediction

was started in the fetch stage, where if the upcoming command was determined to be branch, it's target value was calculated and loaded into the PC on the next load (thus forcing a static branch taken policy). It was later determined in the memory stage if the branch was correct; if it was not, the registers at the start of decode, execute, and memory are reset to `0` on the upcoming load.

Performance counters were added and are accessible through LDR and STR commands. The counters increment whenever an event happens (such as a cache miss). To access the values, reads to the values from `0xFFE0` and higher stored the current value from the counter accessed into the desired register. Using STR would reset that counters values regardless of the value provided.

The eviction write buffer was also added between the L2 cache and simulated memory. When the L2 cache evicted a line, it would write that line to the write buffer in a single cycle. Then, the L2 cache would continue to read in a new value and when that was completed, the buffer would then write the new data to memory. This allowed for faster processing of data as the memory delay penalty for writing would necessarily be seen.

For testing branch prediction, a file was written that tested many different branches in a row (taken and not taken) along side addition to make sure register values weren't updated on a miss. The pipeline was analyzed to make sure the correct values were processed. After this was complete, the branch predicted and branch miss values from the counters were loaded into the register file and then cleared to confirm correct counter operation.

For the eviction write buffer, the L2 Cache was thoroughly tested at the same time. The program written would force a load to every single cache line in the L2, and then write an easily identifiable value

to it. It then proceeded to flush the L2 cache with new data, in which values were sent to the buffer. It was able to be seen that the data was evicted to the buffer, the read then occurring, and the write then taking place. After all values were flushed, a read to the simulated memory to access the easy identifiable value was called to confirm correct L2 and eviction write buffer storing.

5. This final checkpoint implemented the features listed in *Advanced Design Options* if not stated elsewhere.

Advanced Design Options

Memory Stage Leapfrogging

This advanced feature allowed us to have commands that did not rely on a memory operation or forwarded data requirement to jump over the memory stage and into writeback from execute (EX -> WB). This feature was implemented in checkpoint three and takes careful consideration of if the memory stage should load the `CC` register. In order to implement the design, the memory stage checked if the execute stage relied on the value it was working with, if it needed a memory operation, or if it would modify the PC through an offset (the offset is sent back in the memory stage). If it did not need any of these things, it was allowed to jump passed and the load `CC` flag in the memory stage was disabled if set.

Testing of this data was done by running multiple instructions passed LDI, STI, and other instructions that stalled the memory stage. It was observed that dependant values or ones that needed to modify the memory stalled, while others passed and set the `CC` correctly as was the intended behavior.

As for performance, it was shown that implementing these policies had a minimal effect on frequency (122.91 MHz -> 122.74 MHz). In addition, we found it performed slightly faster taken into account frequency and run time. However, these values were dependant on the program being run. Loads and stores with non-dependent or branching instructions following would cause this features to greatly improve overall performance.

Four way set associative L2 cache with pseudo LRU policy

See checkpoint three for discussion of implementation. Note that this feature was implemented in checkpoint three and is single cycle on hit or miss. As a result, there is no difference in performance that can be provided. It was able to be tested by the test listed in checkpoint four for the eviction write buffer and analyzing to make sure the correct values for the LRU were seen.

Hardware Prefetching (Basic)

A module that prefetches the next line on a read miss was implemented and inserted after the L1 instruction cache. It was observed during testing that it would cause multiple additional data cache delays, thus slowing down the pipeline. As a result, this feature was not used in our performance version of the CPU for the competition code. We believe it would be useful in an environment that uses sequential arithmetic operations without too many branches or memory operations.

Additional Observations

In addition to the advanced features, we also tried inserting another eviction write buffer between the arbiter and the L2 cache. While this resulted in less runtime, our clock frequency was no longer as high

(122.74MHz -> 114.76 MHz) and thus scrapped. It should also be known, that if we had a register after the memory stage to stall the execute stage for one cycle for data forwarding as well as a multi-cycle L2 cache, it theoretically would have greatly increased performance due to both areas being large critical paths.

Below is the data values recorded and adjusted for frequency from various tests.

Simulation runtime from testomatic using 200ns delay on memory
Time units are in ps (picoseconds) if unlisted
Scores is the cube root of (c1a * c2a *c3a)

Test	Frequency	SDC Settings	mp3-final	mp3-final A
Default	122.91 MHz (8.137 ns)	8.500/4.250	851190000	691968133
MEM_Skip	122.74 MHz (8.148 ns)	8.350/4.175	846810000	689921786
ArbEWB + MS	114.76 MHz (8.714 ns)	8.900/4.450	846790000	737879052
L1Prefetch+MS	121.17 MHz (8.253 ns)	8.350/4.175	849660000	701213172

comp_1	comp_1 adj	comp_2	comp_2 adj	comp_3	comp_3 adj	Score
1250560000	1017459931	729660000	593653894	6349630000	5166080873	1461292915
1246610000	1015650970	728730000	593718429	6349630000	5173236109	1461153263
1246580000	1086249565	728410000	634724643	6349910000	5533208435	1562538290
1247000000	1029132624	1461280000	1205975077	6349710000	5240331765	1866621028

Default is the design without memory leapfrogging (MEM_skip or MS.

Conclusion

We were able to achieve the task of creating a pipelined LC3*b* with a multi-level memory hierarchy. In additional, we implemented multiple advanced design features such as skipping the memory stage if not required for a specific instruction. Although some of our advanced features may have brought forth performance degradation we were able to observe the tradeoff of performance between frequency and the amount of clock cycles used to complete a benchmark. With the completion of this project,

we have demonstrated our understanding and skills to create hardware with performance taken into consideration.