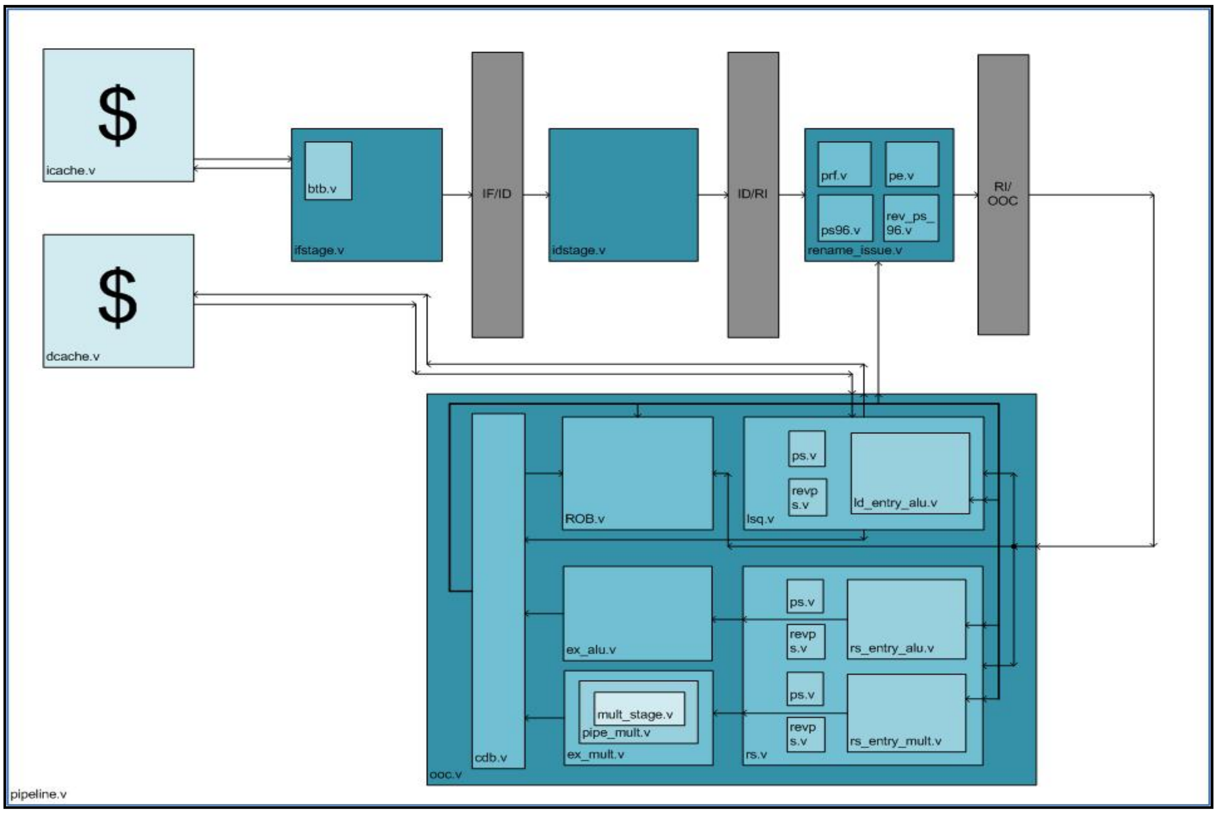
The VeriDifficult Alpha Processor

Vikas Khurana
Niranjan Ramadas
Sunwoo Kim

Table of Contents

1. Description of Processor

Our processor was designed primarily for performance while maintaining correctness and a complexity not too great for a three person group. As such, our processor is superscalar and features an out of order core that implements Tomosulu's third algorithm, a branch predictor and branch target buffer, a load-store queue, a non-blocking data cache and a pre-fetching instruction cache.



1.1 Basic Design

The following are the basic features implemented in our processor.

1.1.1 Reservation Station

Our reservation station is divided into two 16 entry groups. One group is allocated for ALU instructions and the other is allocated for multiplication instructions. When an instruction enters the reservation station, its inputs are driven to both groups. If the instruction is not a multiplication, a memory operation, a halt, or no-op, it is placed into an ALU RS entry. If it is a multiplication it is placed into a multiplications RS entry. The entries chosen to accept the incoming instruction within each group is determined using two priority selectors per group. One selects the big endian priority; the other selects the little endian priority. Since each group has two priority selectors, the RS can handle any combination of ALU and multiplication instructions. If the priorities of either group point to the same entry, a stall signal is sent out of the RS. After an entry is written to, it monitors the Common Data Bus (CDB) to catch any invalid values it has. If any physical register number (PRN) in the entry matches a valid PRN on the CDB, the RS entry grabs the value. When all the dependencies in any RS entry are met, it lets the RS module know by outputting a ready-out signal. The RS then makes use of four more priority selectors (two for each group) to choose entries to send to the functional unit. Entries are sent every cycle, except in the case where the CDB is saturated, in which case the priority selector is disabled. In the case of a branch mispredict, the entire RS is reset.

1.1.2 Re-order buffer

The re-order buffer (ROB) consists of 64 entries. The structure was created using register arrays. In order to maintain order, the ROB makes use of head and tail pointers. In our case, we use two head pointers and two tail pointers so that the processor can be issued two commands and can commit two commands. On issue, the ROB checks to make sure the instructions coming in are valid. If they are, it places them at the two tail pointers. On the next clock cycle, the tail pointers are incremented by two, and two more instructions are inputted. In the case where only one of the input instructions is valid, that instruction is placed at the first tail pointer, and both tail pointers are incremented by only one. Each ROB entry has an execution done bit. An entry cannot commit unless the execution done bit is set to 1. If a halt or no-op instruction is inserted, its execution done bit is automatically set to one. Otherwise, for all other instruction types, the execution done bit is set to 0 on issue. The ROB then monitors the CDB. If the ROB number on the CDB is valid, the execution done bit for that ROB number entry is set to one. When an entry hits the head of the ROB, if the execution done bit is a 1, it commits. Branch misprediction is calculated when the ROB entry hits the head of the ROB. A misprediction signal is sent out of the ROB if at the head of the ROB execution is done and the branch prediction does not equal the actual branch direction, or if execution is done and the predicted target address does not equal the actual target address. Finally, the ROB stalls when the next tail pointer equals the current head pointer.

1.1.3 Rename-Issue

Rename-Issue eliminates false dependencies by taking inputs from the ID/RI pipeline register and renaming all the inputted architected registers into physical registers. It chooses the physical registers from a list of 96 physical register numbers (PRN). The destination registers are granted new PRNs that are determined by the register alias table's (RAT) free list. After the destination registers receive their PRNs, the PRNs are written into the RAT on the next clock cycle. When an instruction finishes execution, the Rename-Issue stage looks at the CDB for a destination PRN and a value. It then grabs the value and places it into the 96 entry physical register file (PRF). When an instruction commits from the ROB, the rename-issue module receives an architected register number (ARN), a PRN, and an old PRN. This old PRN is then kicked out of the Retirement RAT (RRAT). When the Rename-Issue stage receives a mispredict signal, it copies the RRAT's free list into the RAT's free list, as well as the contents of the RRAT into the RAT. In addition, it validates and invalidates certain entries in the PRF to restore correct state.

1.1.4 Functional Units

There are four main functional units. There are two units for ALU functions, and two units for multiplication functions. The ALU units calculate arithmetic instructions that are not multiplications or load/store address. The ALU units handle branch calculations as well. The ALU functional units complete execution in one cycle. The multiplication function units complete execution in 8 cycles. Though the multiplication functional unit takes 8 cycles to complete, it is pipelined, and can therefore accept a new input every cycle. When a unit finishes execution, it attempts to broadcast its value on the CDB. If stalled by the CDB, a functional unit will not accept any new values. Also, in the case of a branch misprediction, all functional units will flush out their registers.

1.1.5 Common Data Bus

The Common Data Bus (CDB) takes results from 4 different functional units and 2 memory values and outputs them on a bus. The CDB gives priority first to load-store queue (LSQ) results, then multiplication results, and finally ALU results. If there are more than two inputs coming into the CDB, it chooses two, and then sends out a stall signal to the functional units whose values are not going to be broadcasted. The only result that is not broadcast on the CDB is a store. Stores do not have a destination PRN, but they do have a ROB number. Instead of putting this ROB number on the CDB, we instead forward it directly to the ROB. This bypasses the CDB and saves a cycle on every store.

1.2 Advanced Features

The following are the advanced features in our processor.

1.2.1 Superscalar

Our pipeline is two-way superscalar. That means that it fetches two instructions, issues two instructions, and possibly commits two instructions. Theoretically, this would bring a CPI as low as .5. However, due to memory latency, data dependencies, pipeline stalling and branch misprediction, a CPI of .5 is not possible in our pipeline.

1.2.2 Tomosulu’s Third Algorithm

Our pipeline made use of Tomosulu’s Third Algorithm to execute code out-of-order. Tomosulu’s Third employs the use of a PRF instead of an ARF. This essentially cuts out the need for the ROB to store values. Instead, the ROB simply serves as a retirement placeholder, making sure that instructions are committed in order. In addition, the ROB also retains its place as a branch direction resolver. The PRF in Tomosulu’s Third keeps record of all the physical registers and their value. Whenever something is finished executing, the value is stored in the PRF. The validity of the data is then asserted with a valid bit.

1.2.3 Pre-fetching Instruction Cache

The instruction cache can be thought of as containing two components; the actual cache memory itself, and the controller which links it to memory and the pipeline. In our case, the cache memory is a 512 line direct mapped cache. It is able to return data the same cycle it is accessed if the access is a hit. The controller for the instruction cache grabs instructions as they are needed. If something is in the cache memory, it outputs the value the same cycle it looks for it. Otherwise, if something is not in the cache memory, the controller sends the appropriate signals to memory to grab the value needed.

Our instruction cache controller also has the ability to pre-fetch instructions. When a cache miss occurs, the cache controller starts sending additional addresses to the memory after the cache miss address is sent. Each address sent in this manner is equal to the previous address incremented by 8. The cache controller continues to send a new address each cycle until the first cache miss finally comes back from memory with the correct instruction. Then, the controller takes each subsequent instruction that returns from memory and inserts them into the cache.

1.2.4 Load-Store Queue

The Load-Store Queue (LSQ) has 16 load entries and 8 store entries. It can accept two memory instructions per cycle. The load entries are structured similar to the reservation station structure, and the store queue is structured similar to the ROB. Loads that are put into a load entry grab a color from the store queue. The color is just the value of the store queue tail pointer. In this manner, every load that enters the load queue is colored with a store. In the case that there are no stores in the store queue, the loads are colored with a 'no-color' bit. The color lets the load know when the store in front of it has committed. When the store does commit, it broadcasts its color to all the load entries, and the entries that match in color set their color dependency bit to 1. After a load gets placed into a load entry, it sits until all its dependencies are met. Once the dependencies are met (including the color dependency), it sends the load to memory. In this way, loads can still execute out of order (they are sent as soon as their dependencies are met), but correctness is maintained by waiting for any stores in front of it to commit first. The relationship between the LSQ and the data cache is important. When the LSQ sends a load to the data cache, it also sends the ROB number of the load. When the value is ready, the data cache sends it back along with the ROB number. That way, even though a load can be sent every cycle, the LSQ knows which returning values correspond with which load.

Stores inserted at the tail pointer of the store queue. After insertion, the tail pointer is incremented. Stores are sent to memory when they hit the head of the ROB, all its dependencies are met, and it is at the head of the store queue. After they are sent to memory, the store queue verifies that the store completed. After the store does complete, it sends the ROB number of the store directly to the ROB. By bypassing the CDB, stores commit one cycle quicker. When the ROB number of the store gets to the ROB, the ROB commits the store. After the store gets committed, the head pointer in the store queue increments.

1.2.5 Non-blocking Data Cache

The data cache has two components to it, much like the instruction cache. It is composed of a cache memory and a controller. The cache memory has 512 lines and is direct mapped, write-through, and write-allocate. It also has the ability to return data from it the same cycle it is accessed, provided the access is a hit. The controller communicates with the LSQ, and handles both stores and loads. The data cache controller also has priority over the instruction cache controller in regards to accessing memory.

Our data cache (d-cache) controller is non-blocking, meaning it can process other memory transactions even when there is a cache miss. In order to do this, we implemented a 15 entry cache buffer. When loads enter the d-cache controller, if they are a miss in the cache memory, they are put into a buffer in addition to being sent to memory. The loads are indexed into the buffer by the 4 bit response they get from memory, and that entry is validated. When loads enter the buffer, their ROB number and their target addresses are stored.

When a load comes back from memory, its value is sent out of the controller. The ROB number of the load is accessed by indexing into the buffer using the memory tag of the load. This ROB number is also sent out of the controller. The buffer entry is then invalidated. If a tag comes back from memory and that buffer entry is invalid, no ROB number is sent out of the d-cache controller. In the case where a value is returning from memory the same time a cache hit is being processed, both values are sent to the LSQ at the same time.

1.2.6 Branch Target Buffer

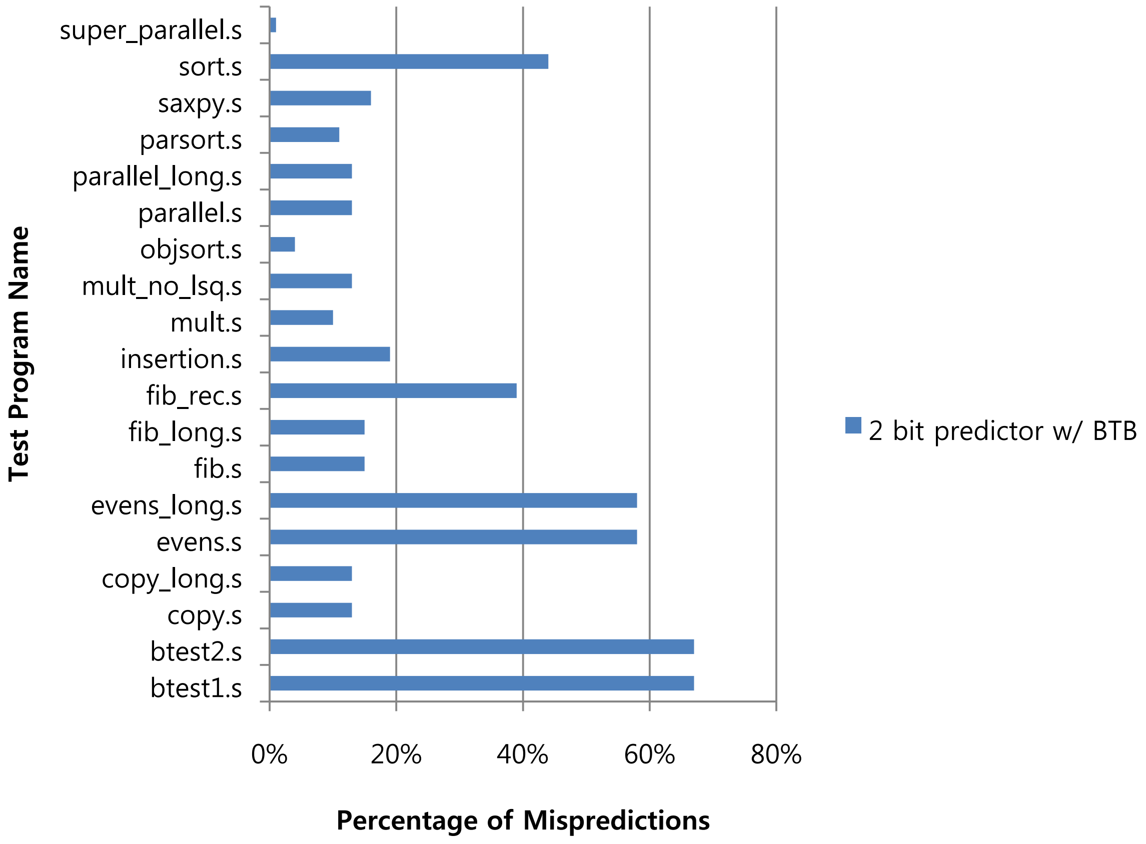
The 64 entry branch target buffer (BTB), alongside the two-bit branch predictor, lives inside the fetch module. This allows us to predict branches without any penalty. When an instruction gets to the fetch stage, 6 bits from the instruction’s PC (specifically, bits 7 to 2) are used to index into the BTB. There, the instruction’s PC is compared with the PC in the BTB entry. If they match, and the entry is valid, the prediction that corresponds with that BTB entry is sent alongside the instruction out of the fetch stage. If the prediction is ‘taken’, the target PC at that entry is sent, and the current PC register is updated with the new PC value.

The BTB is updated whenever there is a branch mispredict. When the mispredict signal is high, the BTB grabs the PC of the mispredicted branch and uses bits 7 to 2 to index into a BTB entry. There, the incoming target PC replaces the target PC of that entry, and the 2 bit branch prediction is either incremented or decremented depending on whether the branch was taken or not.

2. Analysis

The following is an analysis of our advanced features.

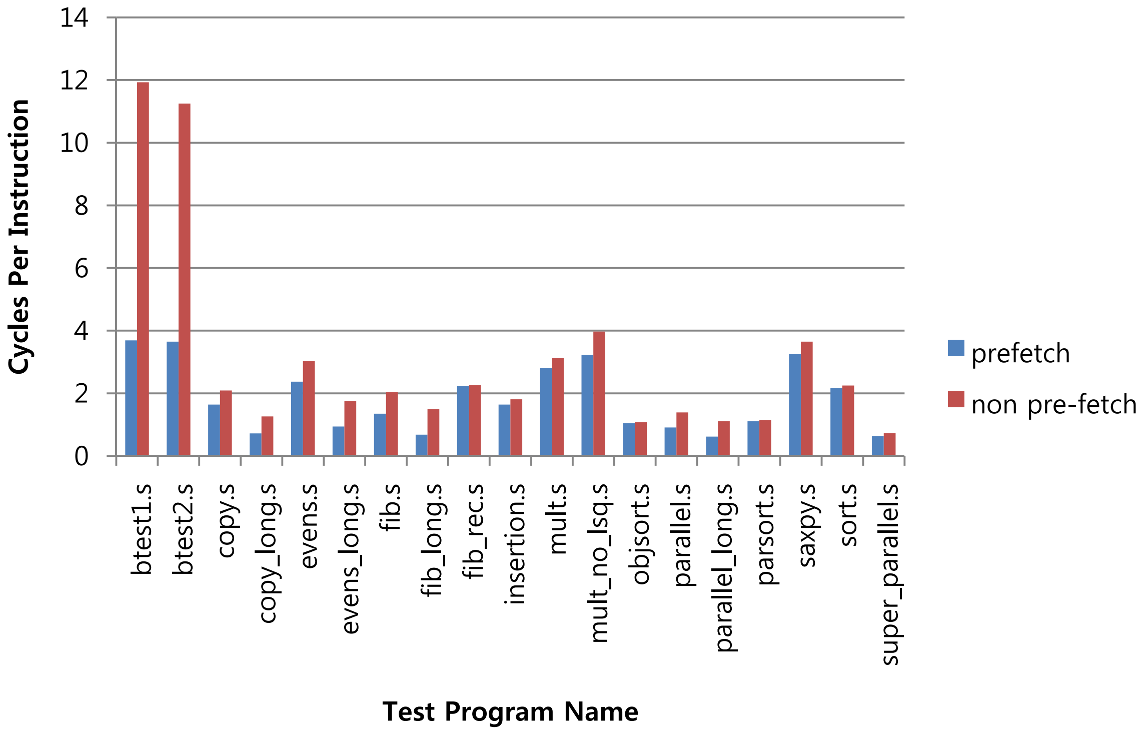
2.1 BTB Analysis



Graph 2.1: Branch misprediction percentage rate with respect to program.

The BTB and branch predictor that was implemented was tested over the 18 test programs provided to us and super_parallel.s. The percentage of mispredictions out of all predictions for each program is shown above. The graph above shows the effectiveness of our branch predictor and BTB. The two programs that seemed least affected, btest1.s and btest2.s, both have a misprediction rate of about 67%. However, both those programs were specifically designed to do poorly with branch prediction. The programs evens_long.s and even.s also seemed not to do too well. This is because it had relatively few branches, and the predictor did not have enough time to warm up to an adequate level. The branch predictor seemed to do the best in objsort.s. This is because objsort.s has a large number of branches that are highly predictable. Almost every branch in objsort.s is part of a loop that has a substantial number of iterations. In addition, objsort.s executes for a large number of cycles with plenty of branches, allowing the branch predictor and BTB adequate time to warm up.

2.2 Pre-fetch Analysis



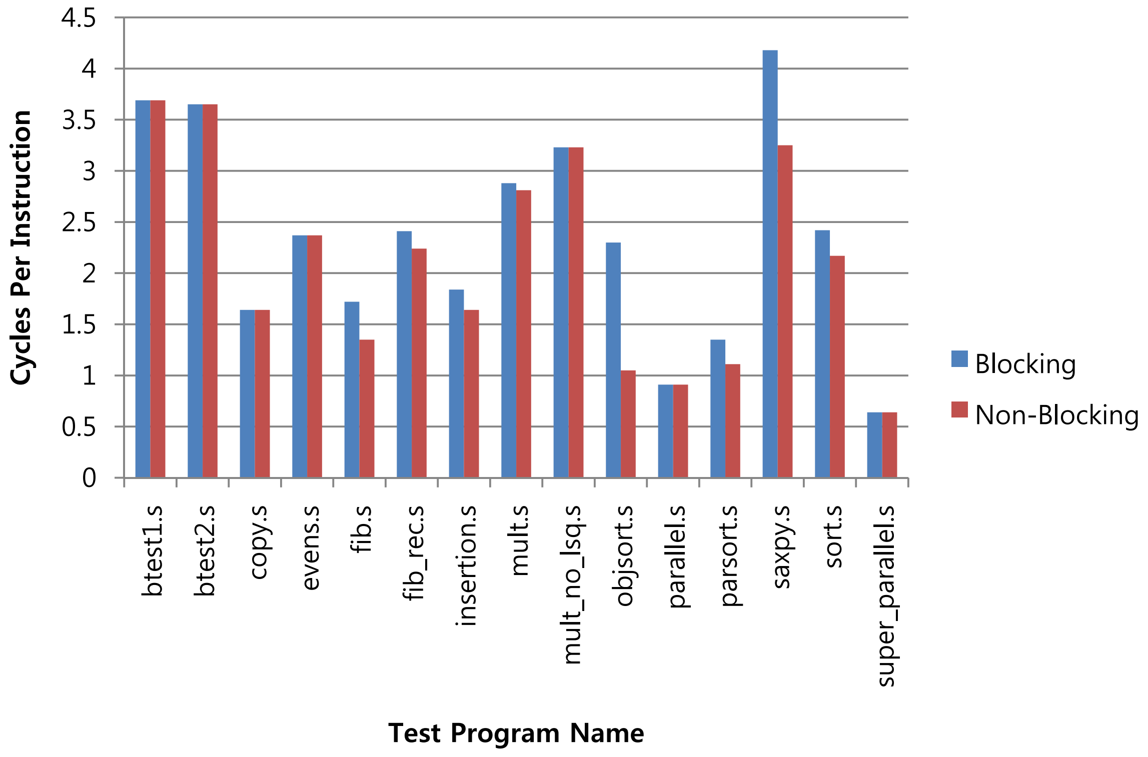
Graph 2.2: CPI after execution for various programs with and without pre-fetching.

Graph 2.2 shows the effects of pre-fetching alone for various programs. The most significant benefit was for btest1.s and btest2.s. Both of those programs had a large number of branch mispredicts. After each mispredict, pre-fetching would restart at the new PC address. With pre-fetching, the initial memory latency time is reduced to a one-time latency. Since btest1.s and btest2.s had so many branch mispredicts, the number of post-mispredict initial memory latencies is high. However, with pre-fetching, the number of initial memory latencies is reduced drastically.

Though other programs did not share as significant of a drop in CPI as btest1.s and btest2.s, all programs did benefit from it. This is because pre-fetching helps reduce initial compulsory misses in the instruction cache. In addition, pre-fetching makes the cost of branch misprediction much less. Since every program has at least one branch, every program could benefit.

The most significant benefit of pre-fetching, however, is to keep the memory busy during memory latency. Before pre-fetching, memory would essentially process only one memory transaction at a time. During the time it took for that memory transaction to process, both the cache and memory would stay idle. Pre-fetching makes use of the memory latency by sending additional memory loads. As such, the latencies for each load can be superimposed, making the average latency for all the loads much lower.

2.3 Non-blocking Analysis



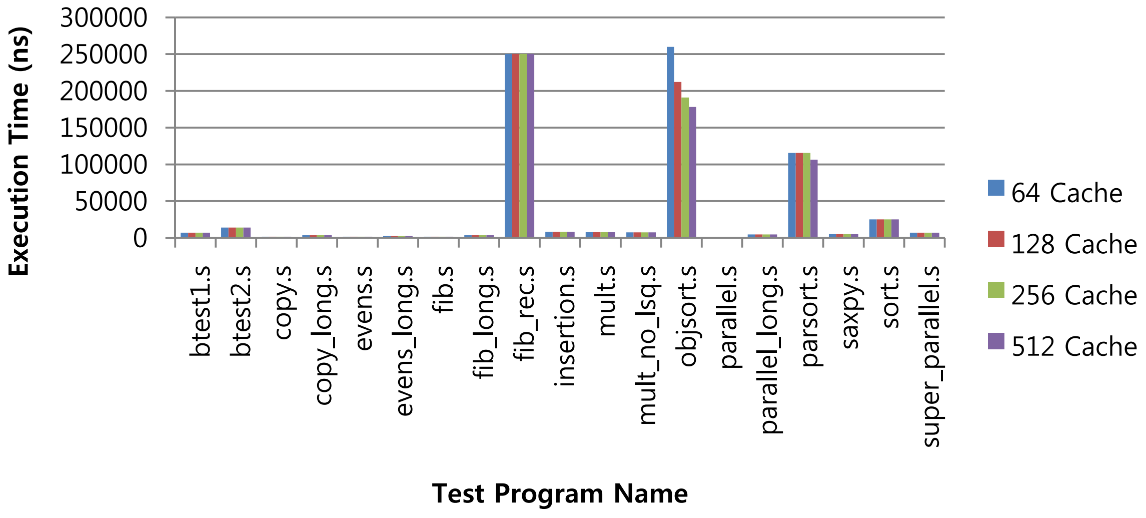
Graph 2.3: CPI after execution for various programs with a blocking and non-block data cache.

We tested the effects of the non-blocking data cache. The non-blocking data cache was coupled with a LSQ that could send loads every cycle. In addition, the non-blocking cache was much bigger in size; 512 lines as opposed to 128 lines. Regardless, the effects of the non-blocking aspect alone can still be surmised from the graph above.

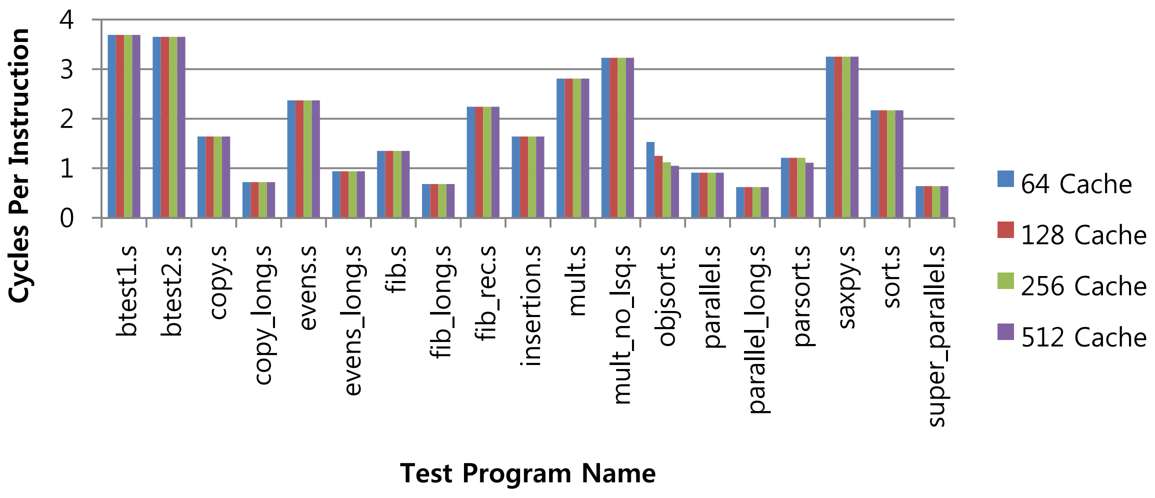
The non-blocking data cache did not seem to improve the CPI of btest1.s, btest2.s, copy.s, mult_no_lsq.s, parallel.s, and super_parallel.s. This is because any improvements that occur are due to subsequent loads. The programs btest1.s, btest2.s, mult_no_lsq.s, parallel.s, and super_parallel.s did not have any loads at all, only stores. The program copy.s had loads, but they followed a store of the same location. So, almost every load to the data cache in copy.s was a cache hit. This was true regardless of whether the data cache was blocking or non-blocking. Therefore, the non-blocking data cache did not improve the CPI for copy.s.

The non-blocking data cache did improve performance for all other programs, however. The greatest improvement is in objsort.s. This is because objsort.s has a large number of instructions executed and a large number of loads. The loads don’t interfere with each other because of the non-blocking cache. Saxpy.s also benefitted greatly. This is because almost all the loads in saxpy.s are cache misses. Usually, these misses must come back from memory before any other loads can be processed. However, with the non-blocking cache, the loads could be sent one after another, meaning the cache miss latency could be utilized to send more loads.

2.4 Cache Analysis



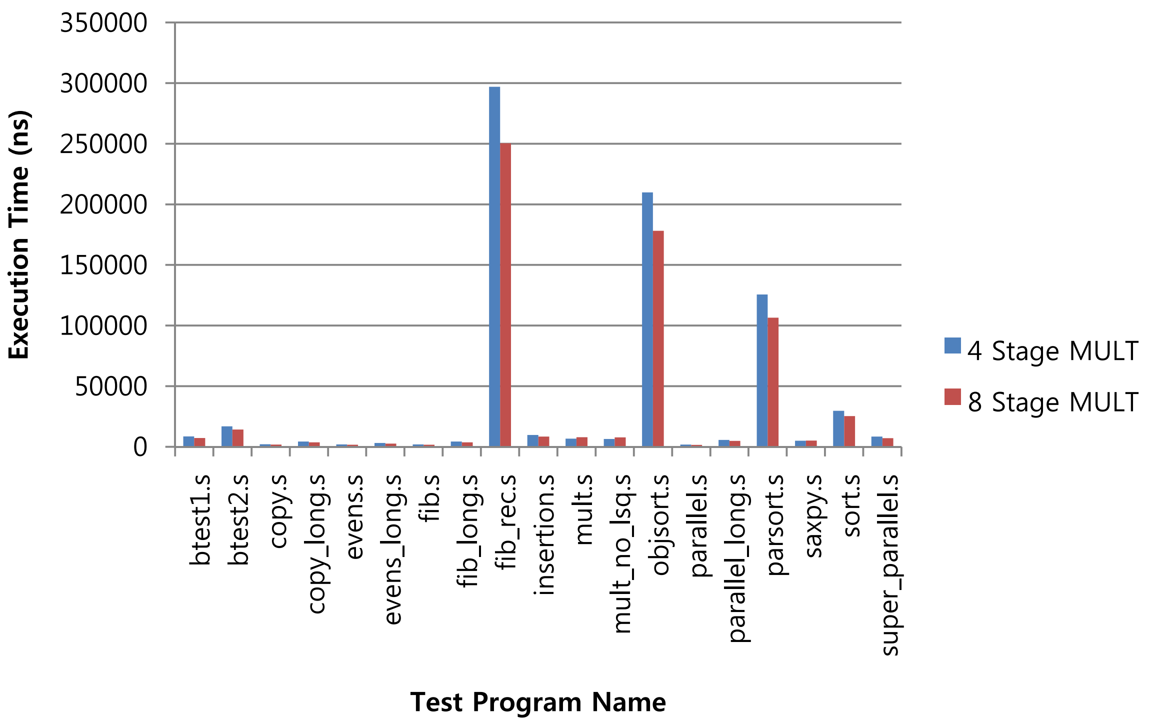
Graph 2.4.1: Total program execution time for different cache sizes.



Graph 2.4.2: Program CPIs for different cache sizes.

The cache sizes were changed in both the instruction cache and the data cache. The sizes indicated on the graphs represent the size of both the instruction cache and the data cache. From both graphs 2.4.1 and 2.4.2, it is obvious that cache size did not have a significant impact on the programs executed. Creating a bigger cache only seemed to help for two programs – objsort.s and parsort.s. Even then, parsort.s did not receive any benefit until the cache size became 512 lines. Both these programs had a significant number of loads and executed a large number of instructions. As such, the programs behaved in such a way that entries in the cache would be eventually kicked out. Increasing the cache size helped in this case. It seems that other programs did not really fill up the cache to the extent that there were capacity misses, or did so in a way that increasing the cache size did not increase the number of cache hits. Other programs might have received some benefit if we had implemented some associativity instead of increasing cache size only.

2.5 Multiplier Analysis



Graph 2.5: Total program execution time for each program using a 4 stage and 8 stage multiplier.

Switching from a 4 stage multiplier to an 8 stage multiplier helped our clock period a lot, improving it from 10.3ns to 8.6ns. The better clock period helped total execution time for almost every program. However, the 8 stage multiplier takes twice as long as the 4 stage multiplier and so affects some programs adversely. Specifically, mult.s, mult_no_lsq.s, and saxpy.s took longer to execute with the 8 stage multiplier. All three of these programs have a large number of multiplies. In addition, in these three programs, there are several instructions that are dependent on the multiplications. Since the multiplication now takes 8 cycles as opposed to 4, the CPI increased greatly, thus hurting the total execution time.

It is interesting to note that even where performance was hurt, it was not hurt significantly. The lower clock period resulting from the 8 stage multiplier seemed to offset the increase in CPI. For example, referring to table B5 in Appendix B, the total execution time for saxpy.s only went up by 170 ns when switching to an 8 stage multiplier. The increases in execution times for mult.s and mult_no_lsq.s are more significant, but it is reasonable to expect since both these programs are primarily multiplication type programs.

All other programs benefitted from the better clock period resulting from the 8 stage multiplier, however. Multiplies are usually sparse in programs, and that is reflected in the program execution times above.

2.6 Stalling Analysis

Test Name	Total Cycle	Instruction Count	CPI	Total cycles	Total cycles	Total cycles
				RS full	LDQ full	STQ full
btest1.s	842	228	3.69	0	0	0
btest2.s	1658	454	3.65	0	0	31
copy.s	214	130	1.64	0	0	92
copy_long.s	429	590	0.72	0	0	0
evens.s	195	82	2.37	0	0	0
evens_long.s	318	302	0.94	0	0	0
fib.s	199	147	1.35	0	0	31
fib_long.s	428	626	0.68	0	0	0
fib_rec.s	29111	12941	2.24	0	0	2076
insertion.s	982	598	1.64	0	0	107
mult.s	916	325	2.81	0	0	595
mult_no_lsq.s	899	278	3.23	601	0	0
objsort.s	20719	19704	1.05	272	183	4
parallel.s	178	194	0.91	0	0	0
parallel_long.s	566	910	0.62	0	0	0
parsort.s	12383	11106	1.11	4442	0	0
saxpy.s	602	185	3.25	0	0	368
sort.s	2940	1349	2.17	0	0	255
super_parallel.s	825	1288	0.64	0	0	0

Table 2.6: The total number of stalls from the RS and LSQ for each program.

From the table above, it seems stalling in the pipeline is not as efficient as it could be. For example, the RS stalls for 601 cycles out of 899 cycles for mult_no_lsq.s, and 4442 cycles out of 12383 cycles for parsort.s. The reason the RS could be stalling so much is due to the way we determine whether there should be a stall. For the RS, a stall signal is sent anytime the two priorities for choosing ALU entries are equal, or anytime the priorities for choosing a multiplication entry are equal. So, the RS stalls if there are 0 or 1 entries free for ALU functions, or there are 0 or 1 entries free for multiplication instructions. The reason this is not ideal is because even if the multiplication entries are full, the RS should be able to accept ALU functions, and vice-versa. However, this is not the case. Instead, in our case, if the multiplication entries are full, no new instructions will be issued to the RS. Similarly, the load queue (LDQ) also stalls anytime there are only 1 or 0 entries available.

The store queue (SDQ) seems to stall a lot for fib_rec.s. The SDQ currently stalls whenever its next tail pointer equals its current head pointer. The logic for stalling seems appropriate, but is not sufficient enough to lower the amount of stalling. The best solution in this case would be to either hoist stores earlier (which adds much complexity), or increase the size of the store queue.

3. Testing Strategy

Our testing strategy essentially begins with smaller modules and works upwards towards the final pipeline. Each small module was tested and synthesized individually. Each module, when tested, used a separate testbench that used registers for inputs and measured the outputs using wires. After the individual modules were tested in this fashion, we grouped a few modules and tested the group. We then periodically added modules into this group and re-tested it until the module group became the entire pipeline. At this stage, we started using the test programs provided in the test_progs folder to test the whole pipeline.

We used several tools to find bugs. The simplest was to add display statements to the code to dump the value of registers into an output. This proved especially useful when trying to determine a value inside a register array, like the PRF or ROB. In addition, we modified our testbench.v file to display the cycle count in our writeback.out file. That way, when there was a difference in register values, we knew the cycle at which it happened. We also used the waveform viewer that came with the VCS compiler. When used in conjunction with the cycle count information provided in the writeback.out file, the waveform viewer helped us pinpoint exact outputs or inputs that were incorrect.

After we passed all the test programs given, we modified the virtual cycle time to values between 7ns and 100ns and re-tested everything.

4. What works, what doesn’t work: Bugs and Fixes

At submission time, all the features of our processor worked as planned. However, there were several issues that we encountered along the way. The following are important bugs and issues we found, and our fixes for them.

4.1 Ships Passing in the Night

‘Ships passing in the night’ was a big problem for us. We added logic to fix the ‘ships’ problem in the LSQ, the ROB, and the RS. However, there was an additional location that we had not anticipated ships to be a problem in. When an instruction leaves the rename/issue stage, the registers are renamed, and all receive values from the PRF. This instruction is then stored in a pipeline register. In the case of a stall, these values are prevented from leaving their pipeline register and entering the out-of-order core. This presented a problem in a very specific case. When there was an instruction stuck in the pipeline register whose register values are updated by the CDB, the new values would not enter the out-of-order core. Usually, we would check for ‘ships’ when something is being issued into the core. So we expected that if values change while an instruction was entering the out-of-order core (OOC), the modules that grab that instruction would simply grab the values from the CDB instead. However, in this case, the instructions were being held and not allowed to enter the OOC, but the values still changed. So, we needed a sort of pipeline register ‘ships’ implementation. We solved the situation by adding logic to the pipeline register.

4.2 Critical Path and Clock Cycle

When we first synthesized our pipeline, our clock period was around 8ns. We thought this ideal, and so never bothered with our critical path. However, after adding some more complexity to our pipeline to improve our CPI, our clocked period jumped drastically to 10.3ns. We expected the problem to be with some of the new modules we added. In particular, we believed the non-blocking cache was the cause of the issue. No amount of code changing, however, seemed to fix the problem. When we re-examined the synthesis outputs, we realized that the critical path could be in our multiplication unit. We altered the multiplier to use 8 stages instead of 4, and our clock period was once again around 8ns.

5. Team Member Contributions

All team members contributed significantly to the overall design. Exact contributions are given below:

Vikas Khurana (33%):	Caches, Execution Units, Rename/Issue, CDB, Fetch (33%), Decode, Test & Debug (33%)
Niranjan Ramadas (33%):	LSQ, RS (50%), ROB (50%), Pipeline Module, Fetch (33%), Test & Debug (33%)
Sunwoo Kim (33%):	RS (50%), ROB (50%), BTB, Out-of-order Core Module, Fetch (33%), Test & Debug (33%)

6. Appendicies

6.1 Appendix A – Module Name List

In folder /verilog:

- btb.v : Branch target buffer and branch predictor
- CDB.v : Common data bus
- dcache.v : Non-blocking data cache
- ex_alu.v : ALU functional unit
- ex_mult.v : Multiplication functional unit
- icache.v : Instruction cache with prefetching
- id_stage.v : Decode stage
- if_stage.v : Fetch stage
- ld_entry.v : Load entry
- lsq.v : Load store queue
- ooc.v : Out-of-order core
- pe.v : Priority encoder
- pipeline.v : Pipeline
- prf.v : Physical register file
- ps.v : 16 bit priority selector
- ps_96.v : 96 bit priority selector
- ps_rev_96.v : 96 bit reverse priority selector
- rename_issue.v : Rename and issue
- revps.v : 16 bit reverse priority selector
- ROB.v : Re-order buffer
- rs.v : Reservation station
- rs_entry_alu.v : RS entry module for ALU operations
- rs_entry_alu.v : RS entry module for Multiplication operations

In folder /testbench:

- testbench.v : Testbench to run the simulation
- cachemem.v : Instruction cache memory
- dcachemem.v : Data cache memory
- mem.v : Unified memory

6.2 Appendix B – Performance Results

B1. Final CPI and Excution Time

Test Name	CPI	Total Execution Time (ns)
btest1.s	3.69	7232
btest2.s	3.65	14250
copy.s	1.64	1831
copy_long.s	0.72	3680
evens.s	2.37	1668
evens_long.s	0.94	2588
fib.s	1.35	1702
fib_long.s	0.68	3672
fib_rec.s	2.24	250346
insertion.s	1.64	8436
mult.s	2.81	7869
mult_no_lsq.s	3.23	7722
objsort.s	1.05	178174
parallel.s	0.91	1522
parallel_long.s	0.62	4859
parsort.s	1.11	106485
saxpy.s	3.25	5168
sort.s	2.17	25275
super_parallel.s	0.64	7086

B2. Cache Size Analysis

Test Name	64 line caches		128 line caches		256 line caches		512 line caches	
btest1.s	3.69	7232	3.69	7232	3.69	7232	3.69	7232
btest2.s	3.65	14250	3.65	14250	3.65	14250	3.65	14250
copy.s	1.64	1831	1.64	1831	1.64	1831	1.64	1831
copy_long.s	0.72	3680	0.72	3680	0.72	3680	0.72	3680
evens.s	2.37	1668	2.37	1668	2.37	1668	2.37	1668
evens_long.s	0.94	2588	0.94	2588	0.94	2588	0.94	2588
fib.s	1.35	1702	1.35	1702	1.35	1702	1.35	1702
fib_long.s	0.68	3672	0.68	3672	0.68	3672	0.68	3672
fib_rec.s	2.24	250346	2.24	250346	2.24	250346	2.24	250346
insertion.s	1.64	8436	1.64	8436	1.64	8436	1.64	8436
mult.s	2.81	7869	2.81	7869	2.81	7869	2.81	7869
mult_no_lsq.s	3.23	7722	3.23	7722	3.23	7722	3.23	7722
objsort.s	1.53	259806	1.25	212170	1.12	191040	1.05	178174
parallel.s	0.91	1522	0.91	1522	0.91	1522	0.91	1522
parallel_long.s	0.62	4859	0.62	4859	0.62	4859	0.62	4859
parsort.s	1.21	115584	1.21	115584	1.21	115584	1.11	106485
saxpy.s	3.25	5168	3.25	5168	3.25	5168	3.25	5168
sort.s	2.17	25275	2.17	25275	2.17	25275	2.17	25275
super_parallel.s	0.64	7086	0.64	7086	0.64	7086	0.64	7086

B3. Blocking vs. Non-Blocking Cache with 512 cache size

Test Name	Blocking	Non-Blocking
btest1.s	3.69	3.69
btest2.s	3.65	3.65
copy.s	1.64	1.64
Copy_long.s	1.64	1.64
evens.s	2.37	2.37
evens_long.s	2.37	2.37
fib.s	1.72	1.35
fib_long.s	1.72	1.35
fib_rec.s	2.41	2.24
insertion.s	1.84	1.64
mult.s	2.88	2.81
mult_no_lsq.s	3.23	3.23
objsort.s	2.3	1.05
parallel.s	0.91	0.91
parsort.s	1.35	1.11
saxpy.s	4.18	3.25
sort.s	2.42	2.17
super_parallel.s	0.64	0.64

B4. Prefetch vs. Non-Prefetch

Test Name	Pre-fetch	Non Pre-fetch
btest1.s	3.69	11.93
btest2.s	3.65	11.25
copy.s	1.64	2.09
copy_long.s	0.72	1.26
evens.s	2.37	3.03
evens_long.s	0.94	1.76
fib.s	1.35	2.04
fib_long.s	0.68	1.5
fib_rec.s	2.24	2.26
insertion.s	1.64	1.81
mult.s	2.81	3.13
mult_no_lsq.s	3.23	3.97
objsort.s	1.05	1.08
parallel.s	0.91	1.39
parallel_long.s	0.62	1.11
parsort.s	1.11	1.15
saxpy.s	3.25	3.65
sort.s	2.17	2.25
super_parallel.s	0.64	0.73

B5. 4 Stage vs. 8 Stage MULT

Test Name	4 Stage MULT	8 Stage MULT
btest1.s	8537	7232
btest2.s	16840	14250
copy.s	2070	1831
copy_long.s	4304	3680
evens.s	1958	1668
evens_long.s	3111	2588
fib.s	1999	1702
fib_long.s	4406	3672
fib_rec.s	296901	250346
insertion.s	9751	8436
mult.s	6681	7869
mult_no_lsq.s	6528	7722
objsort.s	209834	178174
parallel.s	1785	1522
parallel_long.s	5681	4859
parsort.s	125613	106485
saxpy.s	4998	5168
sort.s	29661	25275
super_parallel.s	8384	7086

B6. Branch Prediction

Test Name	Branch Misprediction Rate
btest1.s	67 %
btest2.s	67 %
copy.s	13 %
copy_long.s	13 %
evens.s	58 %
evens_long.s	58 %
fib.s	15 %
fib_long.s	15 %
fib_rec.s	39 %
insertion.s	19 %
mult.s	10 %
mult_no_lsq.s	13 %
objsort.s	4 %
parallel.s	13 %
parallel_long.s	13 %
parsort.s	11 %
saxpy.s	16 %
sort.s	44 %
super_parallel.s	1 %