Two-way Superscalar Out-of-order Processor with Simultaneous Multithreading (SMT)

EECS 470 Final Project Report

Group 6: Yusuf Azman, Bradley England, Hong Moon, Gautam Rajagopal, Pietro Vitale
Department of EECS, University of Michigan, Ann Arbor
{stylo, bradengl, hsmfxw, rgautam, ptvitale}@umich.edu

Abstract -- This paper presents the design and implementation of a 2-way superscalar, out-of-order processor with simultaneous multithreading (SMT). The processor was implemented on Alpha 64-bit instruction set architecture, based on MIPS R10000 architecture. Advanced features include SMT, 2-way set associative non-blocking I-cache with prefetching, 4-way set associative non-blocking D-cache, and a split load buffer and store queue. Effects of various parameters, such as the table sizes of modules and prefetching depth, have been analyzed and tested to achieve optimal performance of the processor.

I. Introduction

The following report is on the design of our fully synthesizable, two-way superscalar processor with out of order execution which supports simultaneous multithreading (SMT). The core architecture of the processor is MIPS R10000 architecture, and the processor is based on Alpha 64-bit instruction set architecture (ISA), implemented with SystemVerilog. This processor is capable of running two threads from a reduced Alpha instruction set at a time when in SMT mode and, when it is not, it is capable of running one thread in two-way superscalar. Our processor is able to execute instructions out of order with the use of a reorder buffer (ROB), reservation station (RS), physical register file (PRF), register alias table (RAT), and a retirement register alias table (RRAT) in order to hide memory latency. We also make use of a 2-way set associative instruction cache (I-cache) with prefetching and a non-blocking 4-way set associative data cache (D-cache). To also increase performance, our processor uses a local history predictor with a branch history table and pattern history table along with a branch target buffer (BTB) to limit the number of branch mispredictions. Finally, our processor has one 8-stage pipelined multiplication unit, two arithmetic logic units (ALU), and one branch address calculator in the execution stage. Our processor meets the minimal design requirements in addition to possessing several advanced features such as a split load-store queue (LSQ) design, with a load buffer (LB) and a store queue (SQ) in addition to those mentioned above. . In this report, performance is generally discussed in terms of cycles per instruction (CPI). The rest of the report will discuss in greater detail and analyze the design of our processor.

II. Design

Our team created many different components to meet and exceed the design requirements spelled out in the project assignment. While designing these components the group took into consideration how to provide the most performance while limiting the complexity of implementation. A general overview of our complete processor with the individual components can be seen below in Figure 1.

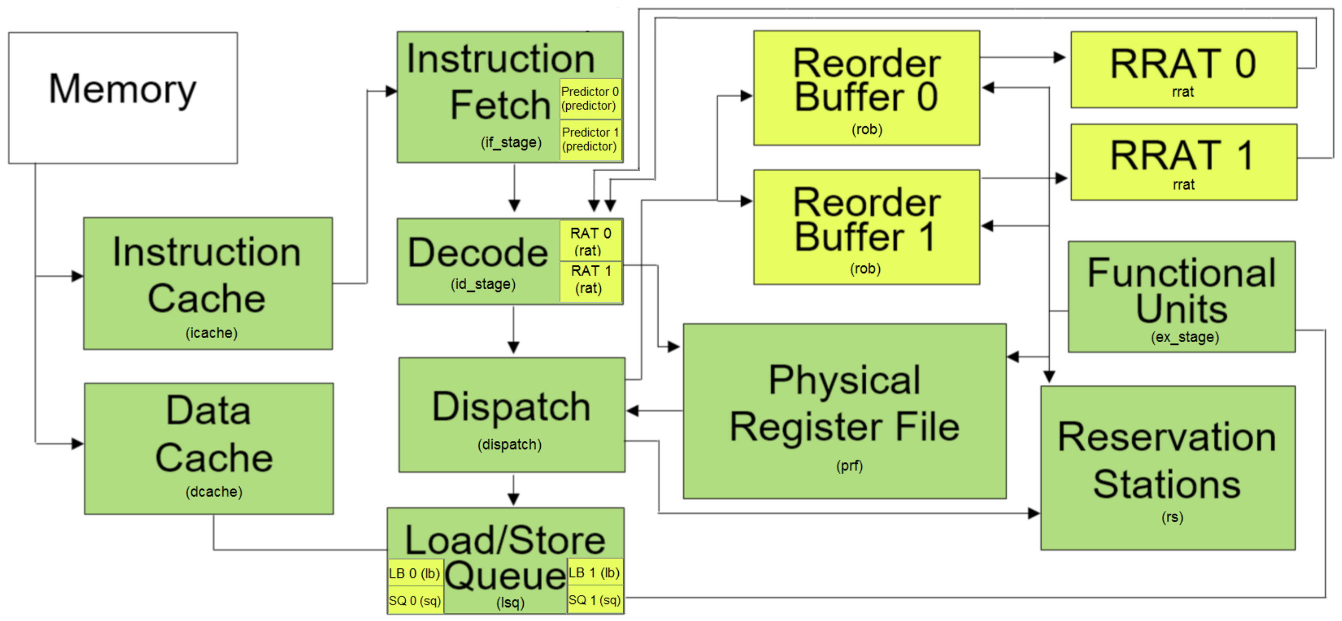

Figure 1: Multithreaded instruction flow and layout of the Processor with Verilog names in parentheses

The following sections go into greater detail discussing each component's features and use. The sections discuss, in order, from when an instruction is fetched from memory to when an instruction is committed.

A. I-Cache

The processor has a dual-ported, two-way associative I-cache with 16 cache sets, each containing two 8-byte cache-lines. The I-cache is capable of prefetching instructions up to eight cycles in advance to help hide memory latency when fetching instruction from memory. This cache uses a true LRU replacement policy because it offers good performance with minimal complexity. Once the instruction is pulled from memory, it is then passed on to the IF stage.

B. Instruction Fetch

The IF stage is able to fetch two instructions from the I-cache at a time. To limit the number of branch mispredictions, the IF stage contains the following features.

- A local history predictor with a branch history table with 3 bits of history indexed in with 5 PC bits. The branch history table then indexes into a pattern history table with each entry containing a 2-bit predictor initialized to weakly not taken.
- A direct-mapped BTB that is capable of holding up to 128 branch PCs and their target addresses.

The IF stage also contains logic to determine when to start fetching in SMT mode, and tells the rest of the processor when it is in this mode. The instructions are passed from the IF stage to the ID stage.

C. Instruction Decode

Our processor has an instruction decode stage with two decoders to allow for 2-way superscalar and SMT. There are also two RATs in this stage, used to rename architectural registers to physical registers. The RATs also keep track of the free physical registers available for renaming. Once instructions are decoded, they are passed on to the dispatch stage.

D. Dispatch

There is also a dispatch stage used to send all the instructions to RS, LSQ, and ROB. This stage also communicates with the other modules and stalls the rest of the processor if any of these modules are full and cannot hold any more instructions.

E. Reservation Stations

The group decided to have a generic RS with 16 entries to store instructions waiting to be executed. While in SMT mode, the RS is capable of selectively flushing instructions from each thread upon a branch misprediction. This module also listens to the CDBs to update any entries whose source operands are dependent on either of the values being broadcasted. Entries whose source operands are ready are selected by priority selectors based on the functional units that they will be issued to. Wake up and select are pipelined and therefore take at least two cycles.

F. Load-Store Queue

There are two split LSQs, one for each thread, each containing a load buffer (LB) and a store queue (SQ). The SQ and LB have dedicated adders for calculating the effective addresses of instructions with memory operations so the instructions do not have to be sent off to the execution stage. The LSQ is used to allow stores and loads to execute out of order, which is very helpful in reducing the effects of memory latency. Both the load buffer and store queue have 8 entries in them, and they listen to both of the CDBs to get the data and calculate the effective memory address. Once a store resolves, it is sent off to the D-cache. Once the D-cache accepts the write request, the tag of the store is broadcasted over the CDB, and then the store can be committed by its ROB.

G. D-Cache

The processor has a 4-way associative, non-blocking D-cache with 8 cache sets, each containing four 8-byte cache-lines. The D-cache uses a pseudo-LRU replacement policy. It also was implemented with a write-through policy, and a write-allocate policy on a miss. The D-cache also has two request buffers, one for each thread, to make it non-blocking. Both of these features are also meant to reduce the effects of memory latency on load and stores to enhance the overall performance.

H. Execution

Our execution stage has two ALUs, one 8-staged pipelined multiplication unit, and one branch address calculator. The group decided to only use one multiplication unit because multiplies are usually rare in a thread. Since the execution stage can execute two instructions at a time, two CDBs are needed to broadcast to the LSQ, RS, physical register file, and both ROBs. The results that broadcast on the CDBs are arbitrated based on operation latency.

I. Reorder Buffer

Finally, there are two ROBs, one for each thread, with 32 entries meant to store instructions in order so instructions can be committed in order. Each ROB is capable of committing up to two instructions at a time. Each ROB module also communicates to its own RRAT every time an instruction commits. Thus in the event of a branch misprediction, the processor is capable of restoring the RAT to the correct state by copying the RRAT to RAT. Also, each ROB contains logic to determine if a mispredicted branch occurred. When a mispredicted branch hits the head of the ROB, the ROB flushes itself entirely and tells other modules to flush incorrect instructions.

III. Effects of Advanced Features and Performance

A. Simultaneous Multithreading Design

The most significant feature the group implemented is simultaneous multithreading (SMT) with a 2-way superscalar pipeline and out-of-order execution. Enabling SMT required extensive design changes, which are outlined in this section.

The team implemented SMT by having an IF stage that is able to either fetch two instructions from the same thread when in single threaded mode, or one instruction each from two different threads while running in multithreaded mode. The IF stage also assigns each instruction a thread ID (either 0 or 1), so the modules are able to keep track of each thread later on in the processor's pipeline. The team also decided to have two branch predictors, one for each thread, to prevent one threads' branch predictions from clashing. The IF stage gets data from each of the ROBs to update the branch predictors of their respective threads correctly and to handle branch mispredict correctly from both threads. Along with the IF stages, the team had to make some design decisions in the instruction decode stage.

To handle SMT mode, the only major change that needed to be done to the instruction decoders was to provide a way to decode fork instructions, so that the IF stage able to jump to the second thread and start executing it. Also in the ID stage there are two RATs, one for each thread, to keep track of the register renaming in both threads. The RAT also talks to two retirement register alias tables, which provides US a way to recover both threads independently when a branch mispredict happens.

From there the instructions are passed to a dispatch stage that correctly places the instructions into the load store queue, reservation station, and both reorder buffers. The group decided to instantiate 2 ROBs, one for each thread, because this was the least complex way to keep track of program order of two threads. Selectively deleting entries in a queue, such as the ROB, would be very complex since a queue must maintain order. The RS holds instructions waiting to execute from both threads and it is capable of selectively deleting entries from one thread because it is a buffer so order does not matter, which means selectively deleting entries is much easier than doing so in a queue. The processor was designed with only one RS for SMT mode since the RS can be shared efficiently without noticeably reducing CPI for either thread.

The LSQ contains two store queues and two load buffers because this was the most effective way to handle entries on a per thread basis when a branch misprediction happens. The LSQ communicates to our D-cache, which has one request buffer for each thread for selective flushing based on which thread has a branch mispredict. While in multithreaded mode, the thread that gets priority to the single-ported D-cache changes based on if a thread makes a successful request. The LSQ also contained one special purpose register per thread which change based on special load-link and store-conditional instructions. When a load-link instruction successfully makes a request to memory the special register associated with the same thread would have the memory address of the load-link saved to it, as well as a 'loaded' bit. When a store conditional would make a request, it first checks if its associated special register's address is the same as the store's address, and if the loaded bit was set to 1. If those conditions are met, the store will go to memory and sent over a CDB with a result of 1, indicating that the store was successful. If the conditions are not met, the store does not go to memory, and instead a 0 is broadcasted over a CDB, indicating a store failure. If a store-conditional is successful, it sets any special registers that have the same memory address as the store's memory address to 0, including its own. Any store-conditional dependent on a special register set to 0 in this way will then be unsuccessful until another load-link instruction occurs.

The performance improvements while running in multithreaded mode are significant. Table 1 shows the number of cycles and CPI for object sort, the longest of the provided test programs, and fib_rec, the second longest. In addition, the table shows the total number of cycles running object sort then fib_rec in single threaded mode, and running object sort and fib_rec together in multithreaded mode. For the entire program, the number of cycles the processor takes to complete in multithreaded mode is 46,303. Running the two programs back-to-back using only 1 thread takes 58,469 cycles. The performance improvement for the total length of time is is 20.8%. However, since object sort is much longer than fib_rec, the actual performance improvement of multithreaded mode should be considered to be the overall CPI while both threads are actually running together. Discovery visual environment (DVE) shows that object sort and fib_rec together run for 37,511 cycles, and in those cycles 27,924 instructions are committed. This implies that our CPI while there was overlap between the programs is 1.34332. Thus, the performance improvement in terms of CPI for multithreading with our processor is 25.0%.

	Object Sort	**Fib_rec**	**Non-SMT**	**SMT**
Cycles	37948	20521	58469	46303
Instructions	19704	12942	32646	32647
CPI	1.92590	1.58561	1.79100	1.41829

Table 1: Cycles and CPI for the programs object sort and fib_rec in single threaded mode and utilizing SMT.

B. Branch Predictor

As seen in Table 2 the predictor often got a prediction rate of around 70 to 90 percent. This section will analyze why we got the prediction rates we did and what we could have done to get a better prediction rate.

Program	Prediction rate
prime	70.0450%
objsort	96.8973%
fib_rec	74.0900%
parsort	92.5134%
Test suite average	87.8733%

Table 2: Branch Prediction Rate of Programs

As seen in Table 2 the predictor often predicts with an accuracy of 70 to 90 percent. This section will analyze why the processor gets the prediction rates it does and what the group could have done to get a better prediction rate.

Prime, which calculates prime numbers, gets a lower prediction rate because the program is short and the predictor was not able to reach a steady state. The rate could have been improved by having a smaller number of history bits, but this could have led to more aliasing in the pattern history table. In the end, using three history bits was decided to be optimal when also factoring in larger programs. Another program with a low prediction rate was fib_rec. This program had many function calls, which predictors have a hard time predicting since the same function can be called in different parts of the program. This could be solved by having a return address stack, allowing for better prediction rates.

C. 2-way Set Associative I-cache

This section describes the differences in performance between a direct-mapped and 2-way set associative I-cache. Table 3 compares the CPI for Direct-mapped I-cache and the CPI of 2-way set associative I-cache in non-SMT mode for select programs is below. Table 4 shows the CPI for Direct-mapped I-cache and that of 2-way set associative I-cache in SMT mode.

Program	CPI (Direct-mapped)	CPI (2-way)
bubble_sort	2.592330	2.631765
merge_sort	2.121048	2.140892
objsort	2.161845	1.925903
fib_rec	1.585613	1.585613
parsort	0.843058	0.845759
Average for test suite	2.43337	2.443171

Table 3: CPI of Direct-mapped I-cache vs. CPI of 2-way Set Associative I-cache in Non- SMT mode

Program	CPI (Direct-mapped)	CPI (2-way)
objsort_fib_rec_smt	1.713634	1.418293

Table 4: CPI of Direct mapped I-cache vs. CPI of 2-way Set Associative I-cache in SMT mode

Usually set associative caches give better overall performance as opposed to direct mapped, but the CPI has not been really affected by changing the cache into a 2-way associative one. Switching from a direct mapped I-cache to a 2-way set associative I-cache slightly decreased CPI for the test suite, but it helped SMT performance by over 17%. Hence, it made sense to adopt the 2-way set associative I-cache as the final design for the processor.

D. Prefetching I-cache

Prefetching is an integral part of our I-cache. The performance with and without prefetching is found below in Table 5.

Program	CPI (No Prefetch)	CPI (Prefetch)
bubble_sort	7.879351	2.631765
merge_sort	5.050866	2.140892
objsort	1.937779	1.925903
fib_rec	1.599985	1.585613
parsort	0.873942	0.845759
Average for test suite	5.307587	2.443171

Table 5: Effect of I-cache Prefetching on CPI in Non-SMT mode

Prefetching is very useful when the code size is very big, as shown in Table 5. This is because there will be more instructions going through the I-cache. With the limitation on the size of the I-cache, it is crucial that future instructions are brought into the cache before they are needed, thus minimizing the cycle penalty for an I-cache miss. When there is no prefetching, each instruction will only go into the cache when it is used. The majority of the time is spent waiting for the instruction to come from memory. It is easily seen with programs that have large code sizes, such as bubble_sort and merge_sort. However, short programs, such as parsort, all the instructions can fit into the I-cache, thus resulting in little to no delay for instructions from the memory.

E. Out-of-Order LSQ

When designing the LSQ, there were two choices: an in-order LSQ or an out-of-order LSQ. With an in-order LSQ, loads can only access memory when they are not dependent by order on any store in the store queue. Below is the CPI for select programs with both types of LSQs.

Program	CPI (In-Order LSQ)	CPI (Out-of-Order LSQ)
bubble_sort	2.0901697	2.631765
merge_sort	2.299143	2.140892
objsort	2.045727	1.925903
fib_rec	1.707850	1.585613
parsort	1.295876	0.845759
Average for test suite	2.669944	2.443171

Table 6: Effect of I-cache Prefetching on CPI in Non-SMT mode

In order to gather data for the in-order LSQ, loads were changed to only make a request to the D-cache when it was not dependent by order on any of the stores in the store queue. This way, the loads would make in-order memory accesses. From Table 6 it is clear that out of order memory accesses help performance by hiding the memory latency of loads.

IV. Analysis on Sizes and Performance

A. ROB Size

ROB Size	Non-SMT Average CPI
8	2.92607
16	2.52582
32	2.44317
64	2.50304

Table 7: Effect of ROB Size on CPI in Non-SMT Mode

Table 7 shows that increasing the ROB size gives better CPI values, but making it too big actually hurts performance. By having more ROB entries, a mispredicted branch takes longer to reach the head of the ROB, and you will have to flush more entries to get back on the right path. Additionally, the I-cache will be prefetching from the wrong path, thus polluting it with incorrect instructions. This also leads to more I-cache misses, further reducing performance. By having the ROB size as 32 it is big enough that a sufficient number of instructions can be in flight at once but not so big that branch mispredictions hurt CPI.

B. RS Size

RS Size	Non-SMT Average CPI
8	2.49183
16	2.44317
32	2.42954
64	2.42954

Table 8: Effect of RS Size on CPI in Non-SMT Mode

It can be seen above in Table 8 that increasing the RS size leads to performance gains, but it levels off after a certain size (64 in this case). Part of this depends on the size of the ROB. If the ROB is smaller than the RS, then it makes no difference when increasing the RS. Since instructions have to be dispatched to both the RS and the ROB, if the ROB fills up, then no more instructions can be dispatched. We used a ROB size of 32 and an RS size of 16. This is reflected in the table. Increasing the RS from 16 to 32 decreased CPI, but increasing it further yields no improvement.

C. Prefetching Depth

Prefetching Depth	Non-SMT Average CPI	SMT CPI: Objsort/Fib_rec
1	5.007857	1.442154
2	3.275839	1.418293
3	2.779141	1.425736
4	2.565569	1.427666
5	2.470061	1.451435
6	2.423684	1.462707
7	2.405299	1.466260
8	2.367869	1.457132
9	2.359707	1.460594
10	2.353696	1.456520
11	2.350444	1.453671
12	2.362332	1.460900

Table 9: Effect of Prefetching Depth on Average CPI in Non-SMT and SMT Modes

As you can see in Table 9, prefetching helps reduce the CPI considerably. Since memory only has a single port, and priority is given to the D-cache, prefetching is a necessity. If the D-cache is always accessing memory, and the next instruction to be fetched is a cache miss, the IF stage must stall before it receives more instructions. However, if the I-cache prefetches while the D-cache is not accessing memory, the number of I-cache misses is reduced. This is due to spatial locality. If the IF stage fetches a certain instruction from a given PC, there is a high chance that it will request other instructions near that PC. Table 9 shows that for SMT, the ideal prefetch depth is 2. Since instructions from both programs have to be in the I-cache, it is important not to prefetch too much from one program. By prefetching only 2 cache lines from each thread, we reduce the number of instructions that are evicted from the I-cache.

D. LB and SQ Sizes

LB and SQ Size	Average CPI
4	2.979493
8	2.405299
16	2.350862
32	2.350737

Table 10: Effect of LB and SQ Sizes on Average CPI

Increasing the size of the LB and SQ decreases the CPI significantly, as Table 10 shows. Having them be bigger allows for more memory instructions to be in flight at a time. It also allows more loads to access memory sooner, thus hiding their memory latency. Unfortunately, increasing the size of the SQ means the LB has to listen to more SQ entries, meaning there are more comparators. We found that making the LB too big increased its clock period substantially. The group felt that decreasing the clock period was more important than increasing the size of the LB.

E. Branch History Bits and BTB Size

Branch History Bits	Average CPI
2	2.459482
3	2.440279
4	2.442063
5	2.437276
6	2.441820
7	2.440945

Table 11: Effect of Branch History Bits on Average CPI

The effect on CPI from changing the number of branch history bits can be seen above. There are improvements from going from 2 to 5 bits of history, but after that, CPI slightly worsens. The number of branch history bits is directly related to how long it takes the predictor to warm up. In larger programs where more branches are committed, having more history bits can capture more types of branch patterns. In shorter programs, having too many bits can lead to more mispredictions, since all the predictors are initialized to weakly not taken.

BTB Size	Average CPI
8	2.857703
16	2.806856
32	2.471788
64	2.465793
128	2.440279

Table 12: Effect of BTB Size on Average CPI

The BTB holds the target addresses of all taken branches. It is indexed by using the lower order bits of the PC. Table 12 shows that increasing the size of the BTB improves performance since there will be potentially fewer mispredictions. Since the majority of the test programs did not contain thousands of unique branches, it did not make sense to make the BTB set associative. By using enough bits of the PC, the chance of aliasing is very low.

F. D Cache Associativity

Program	CPI (Direct-mapped)	CPI (4-way)
bubble_sort	3.134194	2.631765
merge_sort	2.217968	2.140892
objsort	1.981324	1.925903
fib_rec	1.577345	1.585613
parsort	0.866469	0.845759
Average for test suite	2.622998	2.443171

Table 13: Direct-mapped Non-blocking D-cache vs 4-way set Associative Non-blocking D-cache in Non-SMT mode

The 4-way set associative non-blocking D-cache showed about 6.9% improvement in average CPI compared to a direct-mapped non-blocking D-cache in Non-SMT mode. Table 13 compares the CPI of direct-mapped D-cache and the CPI of 4-way set associative D-cache. The program that showed the biggest improvement in CPI from the set associative D-cache was bubble_sort, which might have received the most benefit due to the spatial locality of frequent contiguous array accesses in the program. The CPIs for most of the programs have been improved, but the CPI for fib_rec was slightly worse, perhaps because there is not much temporal locality in the pattern of memory accesses in fib_rec.

VI. GROUP CONTRIBUTION

The ROB was done by all group members for Milestone 1. All group members came up with unit test cases for the modules they contributed, respectively. Yusuf and Bradley spent significantly more effort in integration and debugging in addition to their design contributions. The table below shows the modules that each member worked on throughout the project.

Name	Contribution	Percentage Contribution
Yusuf Azman	PRF, RAT, RRAT, I-cache, Integration, debugging system	23%
Bradley England	Dispatch, CDB arbitration, Ex stage, LSQ, SMT debugging	19%
Hong Moon	RS, Ex stage, D-cache, I-cache	20%
Gautam Rajagopal	RS, branch predictor, LB, IF stage	19%
Pietro Vitale	branch predictor, IF stage, SQ, ID stage	19%

VII. CONCLUSION

In conclusion, a 2-way superscalar processor with out of order execution and simultaneous multithreading is beneficial. From the data in Table 1, it can be seen that running objsort and fib_rec in parallel is faster than the sum of running both separately. By exploiting thread-level parallelism, latencies in one thread can be exploited by executing instructions from another thread. Sharing the execution units and the reservation stations between two programs allows for the maximum utilization of the processor’s resources.