EECS 470 Final Project Report

Group No: 11 (Team: Minion)
Animesh Jain, Akanksha Jain, Ryan Mammina, Jasjit Singh, Zhuoran Fan
Department of Computer Science and Engineering
University of Michigan, Ann Arbor
{anijain, akanjain, rmammina, jasjit, fanzhuo}@umich.edu

Abstract—In this paper, we are presenting a 2-way superscalar out-of-order processor with R10K scheme designed, implemented, integrated and verified by our group as the course final project. It includes all basic functioning features for correctness and also our optimizations in order to improve performance.

I. INTRODUCTION

The course project is a 2-way superscalar out-of-order processor implemented in MIPS R10000 scheme. The motivation for choosing R10000 is less overhead of copying values because of the presence of a physical register file. The ideology of our team while deciding the features was to learn as much as possible while still keeping the target achievable. To learn about the complexities of increasing the width of the processor, we decided on 2-way superscalar. To compensate for the lower IPC because of small width, we implemented Early Branch Resolution, Stores to Load forwarding (Unified Load Store Queue) and Non-blocking caches. As suggested in the class, we kept a bimodal branch predictor because of the small program length which doesn’t give enough time for aggressive branch predictors. Since unconditional branches are always taken, we decided to handle them in Decode stage. We also experimented with associativity of the caches and Prefetcher to choose the one best suited to our needs.

We also used the provided Visual debugger infrastructure and built our own comprehensive visual debugger for our processor. We recommend the visual debugger for all the groups. It reduced our debugging time to just 3-4 days. For verification, we tried our best to test the individual units before integration. This is also the reason why verifying the integration did not come out to be a highly demanding task for us.

II. DESIGN

Below is the overview high-level diagram of our R10K scheme 2-way superscalar processor. The different stages and components of each stage are described below. The detailed analysis will be provided in the following section.

A. Fetch stage (IF)

The size of the cache line of the provided cache is 8 bytes (2 instructions). Since we are designing 2-way superscalar processor, we did not have to modify the fetch logic much as we were already fetching 2 instructions in a cycle. This stage has a 2-bit bimodal branch predictor, branch target buffer and an intelligent prefetcher attached to it. A non-blocking Instruction cache (Icache) was used to reap the advantages of prefetching. Further, a 4-entry fully associative victim cache was added, but it improved performance of only 2 programs.

Figure I High Level Processor Overview
a) Non-Blocking Instruction cache and Intelligent Prefetcher

We designed a Non-Blocking Direct Mapped Cache to prefetch the instructions when there is a miss in the IF stage. There are 4 miss status handle register (MSHR) entries in the non-blocking Icache as we don’t prefetch more than 4 instructions at a time. Initially, a prefetcher was implemented and used to prefetch the next 2-3 instructions whenever there was a miss in Icache. As we analyzed the data, we saw that this kind of prefetching was benefitting the small programs but slowing/not improving the big programs at all. So there was a drive to have prefetching only in the start. So we decided to stop the prefetching once the prefetcher prefetches 32 instructions. This way it can prefetch all the instructions for a small program. We analyzed the CPI numbers for prefetching 16 and 64 instructions but the best results were best with 32 prefetches. The prefetcher was given the lowest priority in case of stores/loads to the memory.

b) Bimodal Branch Predictor and Branch Target Buffer (BTB)

We have 2 banks each of bimodal branch predictor and Branch Target buffers. The reason for having just 512 entries was that the programs which we are targeting are smaller than 512 instructions. In fact, there is just one program mergesort (decaf program) which shows aliasing because that program has more than 1000 instructions. Moreover, branch target buffer is fully tagged, so there are no chances of going functionally incorrect path.

B. Decode and Renaming stage (IDR)

In this stage, there are 2 parallel decoders to decode the 2 instructions coming from the IF stage pipeline registers. It decides what to dispatch to ROB, Reservation stations and Load Store Queue. It has the following basic blocks

a) Register Alias Table

This is implemented in the form of a vector table. It has 4 read ports and 2 write ports.

b) Free list for Physical Registers

There are a total of 64 physical registers. The free list is implemented in the form of 2 parallel circular FIFOs. If an instruction requires a destination register, a physical register is read from the circular queue. Whenever an instruction commits, the previous physical register corresponding to the Architected register which was committed and will be written into this physical register. Physical register 63 holds the same place as Architected register 31.

c) Handling Branches in IDR stage

One of the extra features which we implemented in Decode stage is to handle some portions of branches here. Since unconditional branches are always TAKEN, it makes sense to get the target address as soon as possible and jump to that address. Sending these instructions to RS will waste some of the cycles. The implementation with regard to branches is as follows

- Unconditional branches - If the source operands are ready in IDR stage, they are read and the target address is sent to the IF stage. If the source operands are not ready, IF stage is stalled and the unconditional branch is sent to RS. Through this implementation, we get the advantage which a Return Address Stack (RAS) would have given us (though there is a one cycle delay, RAS would have given us the target address in IF stage).
- Conditional branches - We calculate the target address of the conditional branches here. We already had adders in the IDR stage for unconditional branches, we used the same adders for conditional branches.

d) Early branch resolution (EBR)

We implemented a branch stack to recover the architectural state as soon as a branch is mispredicted. The drive for choosing this feature was that there is no point in fetching if a branch is mispredicted and then flush the instructions later when the branch hits the head of the ROB. Changing the PC at the time of misprediction will try to expose ILP in the right direction. To implement Early Branch resolution, we copied following values in the BRAT

- ROB tail pointer - the place where branch has just arrived
- Circular free list read pointers - When the branch is mispredicted, free some of the registers which are used by the instructions after the branch. This can be easily accomplished by changing the read pointer of the free lists.
- RAT – Restore so that we have the right architected register to physical register mapping.
- LSQ tail pointer - The LSQ pointer was shifted to the place at which the branch arrived.

We did not restore the valid bit table. Moreover, there was no need of restoring the valid bit table if we free up the registers in Circular free list read pointers. It makes debugging hard but visual debugger alleviated that problem.

To flush the instructions in the RS, pipeline registers and multiplier, we used Branch Masking. This branch mask is sent along with the instructions. Whenever a branch mispredicts, a branch mask is broadcasted on the CDB and all the instructions depending on that branch are flushed. If the branch is predicted correctly, then the branch mask is updated to reflect the new dependency on the branch.

C. Reservation Stations (RS)

There are 2 banks of reservation stations. Each reservation station has 8 entries. We analyzed the effect of increasing the number of entries at the end of project, but there was no improvement at all. Reservation stations were becoming full only for programs mult_no_lsq, mult. And even after increasing the number of reservation stations, the instructions were not getting issued as all of them depended on the multiplication instructions. So we kept the total number of entries as 8. It made the issuing and dispatching very simple. However, the tradeoff was that even if one of the Reservation station is full we had to stall the IDR and IF stage. Each RS has a dedicated functional unit.
a) Priority selector for Issuing

The order of priority from high to low is branches, multiplications and ALU instructions. We do not have any heuristics to justify this selection. But we chose this with the ideology that since branches change the flow of the program, we should execute them earlier. This will work in conjunction with early branch resolution and we will not be fetching instructions which are not supposed to be fetched in case of misprediction. And since multiplication takes 8 cycles to furnish the result, we issue multiplications first so that the instructions dependent on that do not need to wait longer.

b) Forwarding values from CDB for back to back instructions

There is a pipeline register between the Issue logic and the functional units. When an instruction is issued, Physical register file (PRF) is read and at the same time the value of the broadcasted physical register is also considered. If the source destination physical register index matches that of the broadcasted destination physical register index, then the broadcasted value is forwarded to the functional units. Otherwise the value read from the physical register is passed on.

D. Functional units

There are 2 functional units each one dedicated to one of the RS. Each functional unit has 3 components
- ALU
- 8 stage pipelined multiplier
- Logic for calculating branch direction and whether the prediction for branch is correct

Since there are 2 Common data bus (CDB) and there are 3 units trying to take control of it, there is an arbitration logic for grabbing CDB which is described later. The destination tag is broadcasted on CDB. However as described earlier, the value is also sent to forwarding logic (apart from the PRF) to handle back to back instructions.

E. Load Store Queue (LSQ)

Loads and Stores are dispatched to a unified circular LSQ containing a total of 8 entries with associated ROB tag. At the same time, they are dispatched to two dedicated RS banks each containing 4 entries with all the renamed operands along with the immediate value. There are two head and two tail pointers for retiring and dispatching of two instructions respectively at a time. As soon as the operands become ready, effective memory address is calculated by the dedicated adder allocated to each RS bank and it is sent to the associated entry of Load/Store Queue along with data in case of Stores.

Once the address is available for loads in LSQ, the oldest load is sent to non-blocking data cache for fetching the data.

This way we send the load to the cache and don’t wait for it to reach the head of the LSQ. If load to cache results in a hit, data is supplied to the load else MSHR tag is provided to the load which will wait for the data to be supplied from memory. However, the load is not set to be ready till all the stores before load has been resolved. If any resolved store before load is found to have matching address, then the data is forwarded from the store to load and the load request is not sent to memory. When all the stores are resolved before load and data is supplied either from pending store in LSQ or from memory the ready bit for load is set to 1 and the load is broadcasted to available CDB. In case of stores, when the address and data for store becomes ready and store hits the head of ROB, the store is retired from LSQ, writing data to memory and head pointer is incremented. Loads retire as soon as they hit head of LSQ as well as ROB and head pointer incremented.

Store to Load Forwarding is done by comparing each memory address of the load with each store address before it in LSQ. For each load/store entry in LSQ branch mask associated with branches on which load/store is dependent on is kept in order to flush the load/store instructions on misprediction of branches. In case the RS for LSQ or LSQ becomes full and any load or store is ready for dispatch structural hazard occurs and stall is generated in IF and ID stages.

To take advantage of the unified load store queue, we implemented a non-blocking D-cache. It also had 4 miss status handle register (MSHR) entries. We have a write through cache, the store writes into the cache in the same cycle but is written into the memory in the next cycle. This was done to break a long combinational path.

F. CDB arbitration logic

There are 3 units - 2 functional units and 1 LSQ trying to grab CDB. We implemented a CDB arbitration logic to take care of this structural hazard. The arbitration logic is kept very simple. Whenever LSQ wants to grab a CDB, it sends a request to the arbiter. Arbiter gives LSQ the lowest priority. It keeps monitoring when a CDB is going to be free in next cycle. And as soon as it gets the slot, it permits the LSQ to grab the CDB.

G. Reorder Buffer (ROB)

The ROB is implemented as a 32 entry circular queue with two head and two tail pointers so that at a time two instructions can dispatch and retire. Tail and Head pointers are incremented on dispatch and retire respectively. An instructions retires only when it reaches head of the ROB and is executed. Branch Mask in kept in each ROB entry in order to flush instructions from ROB on branch misprediction (EBR). Tail pointer is restored to the entry just before the entry of mispredicted branch. ROB contains information about the execution status of the instruction. On retirement from ROB, previous physical register mapped to the logical destination register is returned to the free list and the valid bit table entry is changed to 0 for that physical register. ROB sends retire information to LSQ for loads and stores when they reach the head of ROB. Status counter is maintained to count number of free entries in ROB. If ROB becomes full, IF and ID stages are stalled to prevent dispatch of further instructions.
H. Physical Register File (PRF) and Valid-Bit Table

Both PRF and Valid Bit tables were implemented in a vector table manner. PRF is read during IDR stage for handling unconditional branches and during the issuing of instructions from RS to functional units. It is written when the functional units complete their execution. The Valid bit table tells whether the physical register is valid/invalid. The physical register is written invalid in IDR stage and is written valid when the functional unit finishes execution. It is read during the IDR stage to know whether the source operands are ready or not.

III. PERFORMANCE SUMMARY

We have verified our processor on the provided test suite and some programs from the decaf compiler. We have achieved 100% pass rate on these programs. We did timing closure of the design at 7.6 ns. The average CPI is 1.8241315. If we consider only the big programs and neglect small functionality checking programs then the average CPI is 1.952348. The program by program CPI is presented in the following table.

<table>
<thead>
<tr>
<th>Program name</th>
<th>Instructions</th>
<th>CPI</th>
<th>Cycle time (ns)</th>
<th>Tcpu (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>objsort</td>
<td>19705</td>
<td>4.203</td>
<td>7.6</td>
<td>629454.74</td>
</tr>
<tr>
<td>mergesort</td>
<td>32203</td>
<td>2.349</td>
<td>7.6</td>
<td>575069.22</td>
</tr>
<tr>
<td>fib_rcc</td>
<td>12943</td>
<td>1.326</td>
<td>7.6</td>
<td>130469.20</td>
</tr>
<tr>
<td>parsort</td>
<td>11107</td>
<td>1.115</td>
<td>7.6</td>
<td>94171.62</td>
</tr>
<tr>
<td>factorial</td>
<td>4454</td>
<td>2.231</td>
<td>7.6</td>
<td>75551.59</td>
</tr>
<tr>
<td>bittest2</td>
<td>456</td>
<td>4.688</td>
<td>7.6</td>
<td>16248.80</td>
</tr>
<tr>
<td>prime</td>
<td>1767</td>
<td>0.991</td>
<td>7.6</td>
<td>13315.20</td>
</tr>
<tr>
<td>sort</td>
<td>1350</td>
<td>1.245</td>
<td>7.6</td>
<td>12783.20</td>
</tr>
<tr>
<td>bittest1</td>
<td>230</td>
<td>5.434</td>
<td>7.6</td>
<td>9500.00</td>
</tr>
<tr>
<td>mult_no_lsq</td>
<td>279</td>
<td>2.551</td>
<td>7.6</td>
<td>5411.20</td>
</tr>
<tr>
<td>mult</td>
<td>326</td>
<td>2.101</td>
<td>7.6</td>
<td>5206.00</td>
</tr>
<tr>
<td>insertion</td>
<td>599</td>
<td>1.035</td>
<td>7.6</td>
<td>4711.99</td>
</tr>
<tr>
<td>parallel_long</td>
<td>911</td>
<td>0.656</td>
<td>7.6</td>
<td>4544.80</td>
</tr>
<tr>
<td>fib_long</td>
<td>628</td>
<td>0.769</td>
<td>7.6</td>
<td>3670.80</td>
</tr>
<tr>
<td>copy_long</td>
<td>591</td>
<td>0.746</td>
<td>7.6</td>
<td>3351.60</td>
</tr>
<tr>
<td>saxpy</td>
<td>186</td>
<td>1.715</td>
<td>7.6</td>
<td>2424.40</td>
</tr>
<tr>
<td>evens_long</td>
<td>319</td>
<td>0.862</td>
<td>7.6</td>
<td>2090.00</td>
</tr>
<tr>
<td>fib</td>
<td>148</td>
<td>1.080</td>
<td>7.6</td>
<td>1193.70</td>
</tr>
<tr>
<td>parallel</td>
<td>195</td>
<td>0.779</td>
<td>7.6</td>
<td>1185.20</td>
</tr>
<tr>
<td>copy</td>
<td>131</td>
<td>1.152</td>
<td>7.6</td>
<td>1147.60</td>
</tr>
<tr>
<td>evens</td>
<td>83</td>
<td>1.289</td>
<td>7.6</td>
<td>813.20</td>
</tr>
</tbody>
</table>

IV. TEST STRATEGY

A. Visual Debugger

The visual debugger was a valuable tool in debugging and analyzing our processor. By laying out the visual processor as we did in our high level diagram, we saved a lot of time in narrowing down issues in our processor. After finding were the problem occurred, we then used DVE to find the specific issue.

B. Testing Script

Automating the testing was helpful as it helped reduce time in performing the same mundane tasks of testing. Further, begging able to write an analysis report after all the test cases in a text file and in an excel formatted sheet, was helpful for quick graphing and analyzing our processor performance.

V. ANALYSIS

The following section shows the detailed analysis of various aspects of our processor.

A. IPC Improvements

We analyzed the improvements of enabling branch prediction, prefetcher and store to load forwarding. Base performance does not have any branch prediction, prefetcher and stores to load forwarding enabled.

![IPC Improvement with features enabling.](image)

Figure 2. IPC Improvement with features enabling.

Following are the inferences which we reached from this graph:

- Programs like mult and parallel don’t see much improvement after enabling branch prediction which follows our intuition as these programs have a single loop and branch is taken most of the time.
- There are few programs which have stores to load forwarding. Mergesort and factorial, decaf compiled programs, have benefitted a lot from stores to load forwarding. These programs have many stores and loads and test the LSQ very thoroughly. These programs exposed a lot of small corner case bugs which were not caught through the provided test suite.
- As we thought about the prefetcher, it helps the small programs a lot as it brings the whole program into the cache. However, program like objsort don’t show much improvement. The reason behind this can be twofold
  - The prefetcher is kicking the needed instructions out
  - The program flow is so erratic (a lot of branches) that prefetching is not able to prefetch the right instructions.

B. Prefetcher Analysis

Before finalizing the prefetcher we tried to analyze 5 different kinds of prefetch with direct mapped and 4-ways set associative cache (5*2 combinations).
The five variants of prefetchers we analyzed are:

- No prefetch
- Next-Line Prefetch - Prefetch only next line when there is a miss in Icache for IF stage.
- Burst Prefetch - Keep prefetching when you get an Icache miss in IF stage until the instruction that caused the miss returns from the memory. Since we had only 4 MSHR entries in non-blocking Icache, so burst prefetch can maximum fetch 4 instructions.
- Intelligent Prefetcher - It prefetches only 32 instructions and then stop prefetching after that. It prefetches only next line.
- Intelligent Burst Prefetcher - It prefetches only 32 instructions and then stop prefetching. It follows the burst prefetching for those 32 instructions.

The graph compares these 5 variants when used with direct mapped cache and 4-way set associative cache. It can be seen the 4-way set associative cache with Burst prefetcher is the best combination. We used 7 longest programs (excluding btest1 and btest2) to perform this comparison.

We tried implementing 4-way set associative cache but we encountered a combinational loop in the synthesized netlist which we did not have time to debug.

So we had to go back to Direct Mapped cache and we chose Intel Burst prefetcher 32 as it showed the best performance.

C. Branch Predictor Analysis

This graph shows the analysis for branch predictor. It shows that the 2-bit Bimodal branch predictor show notable improvements overall.

There are 2 interesting points to note from this graph:

- Programs btest1 and btest2 do not show any improvement with the branch predictor. It is because 50% branches are TAKEN and rest are NOT TAKEN and every branch is visited just once. Thus, both the predictors show the same misprediction rate.
- For fib_rec program, the misprediction rate increases. When we enable the predictor, it goes towards different path and encounters more number of branches and most of these extra branches mispredicts. Even the CPI degrades for fib_rec when we enable the branch predictor. This is apparent from the following table.

<table>
<thead>
<tr>
<th>TABLE II</th>
<th>BRANCH PREDICTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib_rec</td>
<td>Not taken</td>
</tr>
<tr>
<td>Number of branches</td>
<td>2440</td>
</tr>
<tr>
<td>Number of mispredicts</td>
<td>611</td>
</tr>
<tr>
<td>Misprediction rate (in %)</td>
<td>25.0409836066</td>
</tr>
<tr>
<td>CPI</td>
<td>1.289</td>
</tr>
</tbody>
</table>

D. Load Store Queue Analysis

The following graph shows the number of stores to load forwarding for different test cases. There are only a few test cases where such forwards are actually happening. Mergesort and factorial, decaf compiled programs, show significant number of such forward. This is in congruence with the improvement shown in Figure.

E. Clock Period Analysis

Initially, we were able to do timing closure at 11.5 ns due to a long critical path from IDR->RS->ROB->LSQ->non-blocking cache. We broke the path by placing registers after functional units but the clock period reduced to only 10.7 ns and CPI degraded more than the improvement in clock period. Instead, we broke the path by placing registers after ROB and we were able to do timing closure at 7.6 ns with clock period easily covering up the slight degradation in CPI.
F. Prefetch analysis from the viewpoint of Icache miss rate

The main intent of adding a prefetch is to get rid of compulsory misses. To get a rough idea of how good our prefetcher is working, we analyzed the number of Icache misses for each program. Some programs like factorial, mergesort show significant improvement following the intuition. However, small programs like evens_long and copy_long show increase in Icache miss rate even though the CPI numbers are improving. After analyzing we found that in the case when prefetcher is enabled, the program is fetching more instructions. Most of these instructions are the instructions after halt and are not needed at all. Since the percentage of Icache miss rate is very small for these programs, even getting another miss on these unneeded instructions gives wrong indication of increment in miss rate.

G. Victim Cache analysis

We added a 4 entry fully associative victim cache. This cache showed improvement only for long programs like objsort and mergesort. It is reasonable as victim cache will only be accessed when the program is long and there is a jump to a location which was previously accessed and was kicked out due to another fetch. The improvement for objsort was that the CPI went from 4.2 to 4.1 (this was not the final design and hence the final CPI number for objsort in section 4 differs from 4.1).

H. Memory Latency analysis

Figure 7 shows average CPI dependence on varying memory latency. If memory latency is increased, loads will receive data after a larger delay and hence instructions waiting for the result of load cannot execute and this leads to CPI degradation. Also, in case when we do not have loads in the program, still we have delay in fetching instruction from I-cache, hence we may have to wait for instruction and this leads to execution of fewer instructions with in a given time and hence CPI degrades as it can be seen for mult_no_lsq. As we have added prefetching of instructions in our design, it might give us some advantage rather than not having it and waiting for instructions to come from memory.

VI. GROUP DYNAMICS

Animesh Jain (20%) - Decoding, RS, BRAT, Non-blocking cache, Integration, Debugging
Akanksha Jain (20%) - ROB, LSQ, Reservation Stations, Bimodal Branch predictor, Debugging
Ryan Mammina (20%) - RAT, LSQ, BRAT, Visual debugger, Automation scripts, Integration
Jasjit Singh (20%) - ROB, BTB, Non-blocking caches, Victim caches, Prefetcher, IF stage, Integration
Zhuoran Fan (20%) - PRF, Valid Bit table, Prefetcher, Functional units, RS Issue logic

VII. CONCLUSION

We have successfully implemented a 2 way superscalar Out of order processor. The project taught us the complexities of an out of order processor and it developed our thinking of how to approach a problem in computer architecture. Since we decided for a 2-way superscalar processor, we strived hard to optimize our design and get the clock cycle down to compensate for low IPC. We were able to close the design at 7.6 ns with an average CPI of 1.824. The Visual Debugger reduced the effort of debugging to a great extent. We think that we have been able to achieve the target we set initially of 100% correctness and a decent clock period. Our ideology of learning as much as possible was achieved as we implemented Load Store Queue with Stores to Load forwarding, Early Branch Resolution and Non-blocking caches.