Group3 0xF4 EECS470 Final Project Report

Jingru Hou, **Mathew Whittlesey**, **Shiyu Wu**, **Tianyu Qiao**, **Yujia Xie**
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor
{hjingru, tmwhitt, shiyuwu, qiaotian, yujiaxie}@umich.edu

***Abstract*—In this paper, we are presenting the MIPS R10000 2-way superscalar processor which our group has developed over this semester. We first discussed, wrote the top level module and designed the top level diagram together. Then we assigned each team member to work on different submodules, and after two weeks, we integrated different stages. After milestone two, we designed and implemented the memory system with LSQ, and non-blocking 2-way set associative write-back data cache. Besides that, we implemented several different branch predictors and compared the accuracy of them. The last two weeks were spent on optimization to minimize clock period and CPI. We reconstructed several modules to shorten the critical path. In this paper, we will elaborate on our project design, testing procedures, and optional features to improve our processors performance.**

I. Introduction

We were tasked with creating an out-of-order pipelined processor using SystemVerilog for EECS 470 final project. Specifically, we were required to implement an instruction and data cache, dynamic branch prediction including a branch target buffer (BTB) and a predictor, into an out-of-order processor with a design of our choosing. We chose to implement a 2-way superscalar MIPS R10K style processor with many additional features, such as a tournament predictor and a dependency-based reservation station. Our processor operates on a subset of the Alpha64 instruction set architecture outlined by the EECS 470 staff. This report details the design of the system, its performance against benchmarks, and our testing strategies to ensure the correctness of our processor.

II. Design

The high level architectural diagram of our design is shown in Fig 1.

The following is an in-depth explanation of each stage of our processor.

A. Fetch Stage

Since we were 2-way superscalar, our fetch stage outputs two instructions. If either instruction is ready, it will be sent to the instruction buffer, as it is possible that we discard the second instruction. The next PCs are determined by the branch predictor.

1) Icache: We implemented a non-blocking direct mapped Icache with prefetching capabilities. Direct mapping was used since instructions often execute sequentially, however if there are many branches in the code this could cause a higher miss rate. First, the cache requests the addresses from memory that the fetch stage is asking for. Once the request is sent, the next in-order addresses are also requested from memory. Once either the next prefetch address is a cache hit or 20 prefetches have occured, prefetching will stop. If the fetch stage asks for an address that is a cache miss but has already been requested by prefetching, the cache will simply wait for the response. Additionally, a victim cache is used alongside the instruction cache. Our victim cache is direct-mapped and contains four entries. If the victim cache has an entry that is being requested, it will swap data with the main cache. If the main cache is evicting an instruction, the victim cache will store it.

2) Branch prediction: The high level structure of the branch prediction is shown in Fig.2.

The main components involve Predecoder, Direction Predictor (DIRP), Branch Target Buffer (BTB), Return Address Stack (RAS), and Branch Prediction Controller.

Given current PCs and fetched instructions, the instructions will first go through the pre-decoder and be classified into 8 different categories {`COND_DIRECT_BR_INST`, `UNCOND_BR_INST`, `UNCOND_BSR_INST`, `UNCOND_RET_INST`, `UNCOND_JMP_INST`, `UNCOND_JSR_INST`, `UNCOND_JSR_CO_INST`, `NON_BR_INST`}. Different policies will be used to handle instructions in different categories, which are summarized below.

i) `COND_DIRECT_BR_INST`: Use DIRP to predict the direction. If the branch predicted taken, directly calcu-

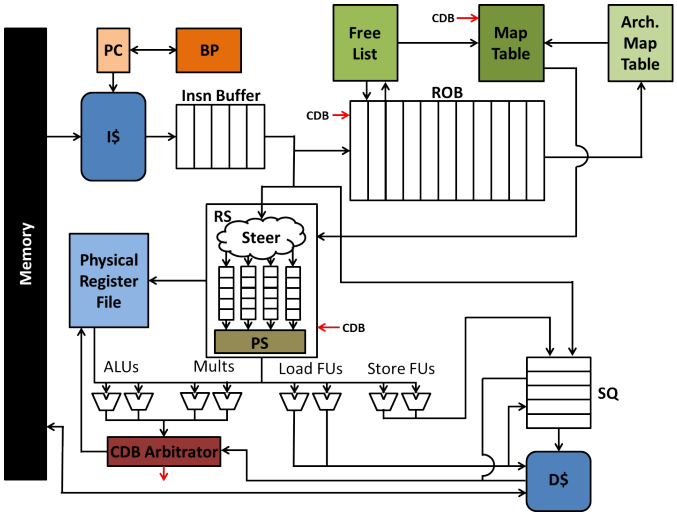

Fig. 1. Block diagram of our 2-way superscalar MIPS R10K OoO processor

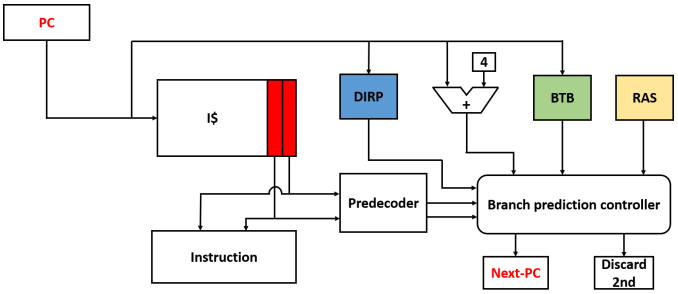

Fig. 2. Block diagram of our Branch Predictor

late the target branch PC using fetched instructions and current PCs, instead of querying BTB. Else, use PC+4.

ii) `UNCOND_BR_INST`: Directly calculate the target branch PC using fetched instructions and current PCs.

iii) `UNCOND_BSR_INST`: Similar to `UNCOND_BR_INST`, further push PC+4 into RAS.

iv) `UNCOND_RET_INST/UNCOND_JMP_INST`: If RAS not empty, pop Top of Stack(TOS) PC of RAS. Else if BTB hit, use target address from BTB. Else, use PC+4.

v) `UNCOND_JSR_INST`: Push PC+4 onto RAS. If BTB hit, use target address from BTB. Else, use PC+4.

vi) `UNCOND_JSR_CO_INST`: First pop TOS of RAS, then push PC+4 onto RAS. If RAS not empty, use TOS of RAS. Else if BTB hit, use target address from BTB. Else, use PC+4.

Besides that, we also implemented three different DIRPs:

i) Bimodal predictor with 2-bit saturation counter

ii) Gshare predictor with 2-bit saturation counter

iii) Tournament predictor combing Bimodal and Gshare predictor

We found that the size of DIRP and the initialization of the counter will affect the prediction accuracies, which will be covered in more details in Analysis part.

For RAS, in order to handle the program involving very long recursion calls, when a new PC is pushed into a full stack, the bottom item in the stack will be dropped. When a misprediction happens, some PCs in the stack are speculatively pushed into and should not exist. To simplify our design, we decide to pop all the items from RAS upon a misprediction.

Since we implemented the 2-way superscalar feature for our project, we use the simplest way to handle the issue of multiple branch predictions. When the first instruction is predicted to take the branch, the second instruction will be discarded and thus will not be sent into the instruction buffer.

3) Instruction Buffer: The instruction buffer decouples the memory system and the OoO pipeline and allows instructions to be fetched even if they are not ready to dispatch. This helps when cache misses occur and the fetch stage is waiting on memory, allowing instructions to continue to be dispatched. The instruction buffer holds 32 instructions, along with their next PC and branch prediction information.

B. Dispatch Stage

1) Reorder Buffer: Our Reorder Buffer (ROB) contains 32 entries, storing the instruction, registers used, and the branch prediction information. Entries are allocated to the tail of the ROB from the instruction buffer, and up to two entries can be retired from the head each cycle.

2) Dependency-based Reservation Station: We implemented a dependency-based reservation station[1] to optimize our design in term of both CPI and clock period. This design greatly simplifies the issue selection logic when multiple instructions are ready at the same time. Depending on the testbench, our CPI either goes up or goes down. A more detailed performance analysis will be described in section IV.

The dependency-based reservation station consists of 4 First In First Out(FIFO) queues and each queue can contain up to 4 instructions. Only the instruction at the head of the queue will be issued when they are ready. A 4-2 priority encoder is used for selecting the issued instructions. When an instruction is dispatched, it can go into either one of the four FIFOs based on its source register producer. This is implemented by adding a few new hardware in our design and a steering cloud before the instruction enters the reservation station.

First, we will need a new column in the map table to keep track of the **producer instruction**. This producer update logic is similar to updating T and Told, but now the ROB index is updated instead. Also, we will need a new structure called the **reservation table** inside the reservation station. It is indexed by the physical registers and has a one-bit field for each physical register to record the ready status. When an instruction enters the reservation station, the ready bit of the product physical register in reservation table is cleared. Similarly, when a tag is broadcasted from the Common Data Bus(CDB), the ready bit is set.

The reservation table greatly reduces the cost of the Common Data Bus(CDB) in term of the circuit delay. A conventional centralized reservation station requires the tag to be broadcasted to 16 different instructions if we keep the same windows size, which is a very high fan-out net and can potentially become the bottleneck of the processor. In our design, only the instructions at the head of the FIFO queue will need to be broadcasted, and the later instructions will look up for the register ready status in the reservation table. That means a new delay to look up the table will be added to the issue logic . It is worth mentioning that looking up the reservation table ready is actually a relatively cheaper approach comparing to broadcasting. In our design, we have 64 physical registers, which means our reservation station will only be 8-bytes large. Four instructions looking up an 8-bytes structure, two bits for each is actually very cheap.

On the other hand, the simplified issuing logic means we have less scheduling flexibility. This can potentially affect the CPI for specific test benches. The solution is to add an **instruction steering cloud**. With the instruction steering cloud, we are able to achieve a relatively close CPI performance comparing to the conventional approach. The steering cloud is a pure combinational module responsible for steering the input instructions. The function of the steering logic can be described as the Algorithm 1 shown below.

From algorithm 1, we can learn that the instructions in the same dependency chain will go into the same FIFO. This means instructions that are not at the head of the FIFO

```
for each dispatched instruction I do
    if the operand A source register is not ready, and the
     instruction producing that register is at the tail of
     the FIFO queue F_a and F_a is not full then
        assign I to F_a
    else if the operand B source register is not ready
     then
        do similar steering
    else if there is at least one empty FIFO then
        assign I to one of them
    else
        Stall
    end
end
```

Algorithm 1: Algorithm for steering logic

will never be ready to issue since the instruction before them always blocks them by data dependency. Through the steering logic, we can further exploit the reservation station potential. Theoretically, this approach will achieve similar CPI to the conventional approach if all the instructions in the issue window form less than 4 dependency chain. In our real experiment, we achieve a no-worse CPI comparing to the conventional approach.

To wrap up, we were able to achieve both a better clock period and a better CPI after using the dependency-based reservation station. For the detailed analysis of performance, please refer to part IV.

3) Instruction Decoder: We used the same decoder from project 3.

4) Free List: Normal functionality.

5) Map Table: Normal functionality. The only special case is to handle instructions with non destination register, such as Store, Nop. For these instructions, we will set the old tag Told sent to the ROB and RS to be the same as the new tag T obtained from the free list.

6) Architectural Map Table: Normal functionality.

7) Physical Register File: To support 2-way superscalar, we implemented a PRF with 4 read ports and 2 write ports, which allows internal value forwarding.

C. Execute Stage

The execute stage consists of eight Functional Units: 2 ALUs for normal arithmetic operations except multiplication, 2 four-stage pipelined multipliers for multiplication, 2 Load functional units for load instructions, 2 Store functional units for store instructions. Branch instructions are calculated using the normal ALUs. The instructions are issued from the heads of the RS FIFO queues. It is worth mentioning that our normal arithmetic operation can be finished in zero cycle due to the simplified broadcasting and issuing logic in reservation station. The reservation station and our zero-cycle execution works well with each other and greatly improves our CPI. For more detailed performance analysis, please refer to part IV.

1) Store Queue: The size of our store queue is 7. There are actually 8 entries, but we only allow at most 7 stores in the queue. This makes it much easier for the empty logic. For all dispatched instructions, the SQ always gives out its tail, which is used for the age logic. The SQ also takes information from the execute stage and uses that for the store data and address. We have store to load forwarding in our SQ. For load instructions, it would search through the queue until the oldest unknown store to see whether there is a load hit.

2) Dcache: We implemented a non-blocking 2-way cache. Two way associativity is doing better than direct mapping for D-cache due to the temporal locality of cache. The strategy for setting the LRU bits is the key part to improve the performance and we decide to give load instructions the priority, then store instructions, and finally, the data coming back from memory. For the dcache controller which handles the multiple outstanding cache misses, we use the MSHR and we integrate this part into the pipeline module. Besides, if a load instruction comes and have a same address as the a previous load instruction which is in processing, we will follow the data comes back from memory to both load instructions directly.

D. Complete Stage

We have a 2-wide Common Data Bus (CDB). Since we have 8 different functional units in the execution stage, we add a 8-2 priority selector to select two completed instructions for broadcasting.

III. Testing / Verification

The following explains the process we did testing and verification for our processor.

For testing, we have a random testcase generator which will generate assembly codes consisting of load, store, mult and branch instructions with different probabilities. Most branch instructions are forward branches to avoid infinite loops. Besides the random testcases, we also wrote testcases for different algorithm to test specific functionalities and compiled them using Decaf470 compiler, such as the program of Hofstader sequence to test RAS, a matrix multiplication program to test ALUs and multipliers.

We also have unit testbenches for each module, though they may not work at this point since we have changed the interface when connecting modules together. However, when we were trying to implement the modules the first time, we tested each module with edge cases to make sure it functions correctly.

We have a bash script which runs all the testcases and compares the writeback value and memory value with the results we obtain using project 3 files. We use it to verify the correctness of our project and do the regression test. The test script will print the CPI for each test case. It is able to test the correctness and CPI of synthesis results as well. Besides that, the script could also automatically save the synthesis results after the synthesis process with a unique hash tag. This greatly simplifies our testing and verification process.

IV. Analysis

The following provides some analysis on the overall performance of our processor and how certain main features we have implemented affect the performance. Our team provided analysis on overall CPI, branch prediction result and its effect, dependency-based reservation station and its effect.

A. Overall CPI Analysis

We test our processor on the given public testcases and the average CPI we achieve is 1.46 along with 8ns for clock period.

The following chart summarizes the performance on different public test programs.

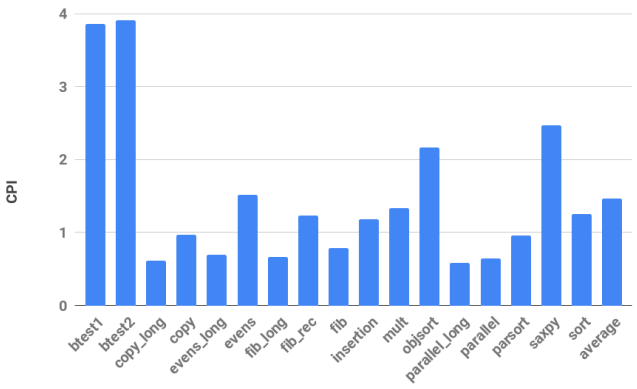

Fig. 3. CPI for different public test programs

B. Branch Prediction Analysis

As mentioned above, we implemented three different DIRPs: i) Bimodal predictor with 2-bit saturation counter, ii) Gshare predictor with 2-bit saturation counter, iii) Tournament predictor. The prediction accuracy of each predictor on non-trivial testcases is shown below.

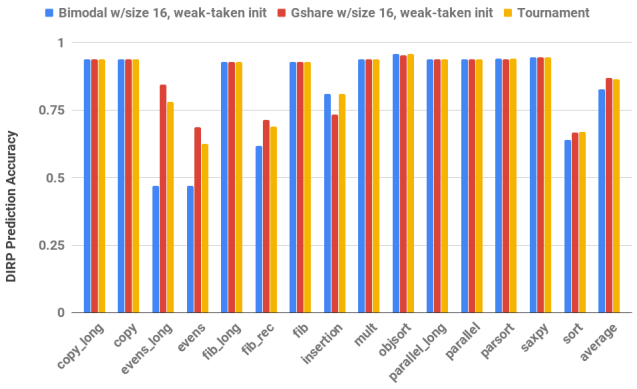

Fig. 4. Prediction accuracy for each DIRP

From the chart above, we could see that the performances of all three predictors on the public testcases are relatively good. All three could achieve accuracies above 80%, with Bimodal 82.6%, Gshare 86.8%, and Tournament 86.4%.

Since Tournament predictor is a meta predictor which utilizes both local Bimodal and global Gshare predictor, its performance is more balanced across all the testcases. For example, it performs relatively well on both `even_long` and `insertion`, in which either Bimodal or Gshare would perform slightly poorly.

Another important factor we found related to the performance of DIRP is the initial value of 2-bit saturation counter. The following chart summarizes the performance of Gshare with size 16 using different initial values.

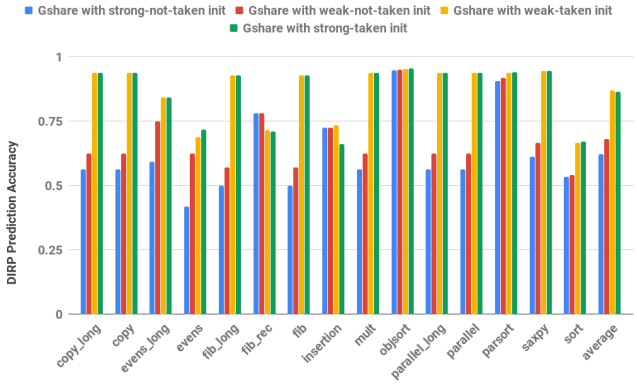

Fig. 5. Prediction accuracy for Gshare with different initial values

From the result, we could see that Gshare with `Weak-taken` or `Strong-taken` gives much better performance than initializing with `Weak-not-taken` or `Strong-not-taken`. We believe that the main reason is that in the most test programs, the conditional branch will be taken considering the For loop and While loop in users' programs. Besides that, since most test programs are not quite long, it may take too much time for the predictor with `Weak-not-taken` or `Strong-not-taken` to warm up, and the prediction accuracy has already been hurt.

Another influencing factor of DIRP is its size. Fig. 6 demonstrates the effect of size on prediction accuracy.

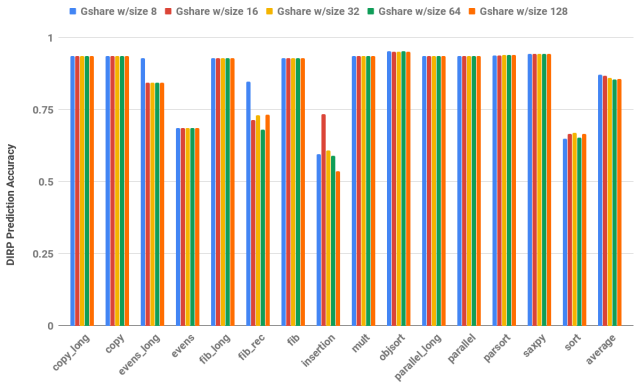

Fig. 6. Prediction accuracy for Gshare with different size

Intuitively, Gshare predictor with larger size is likely to give a better performance since it exploits the global correlation more thoroughly. However, this is not the case in our analysis. The reason could be similar to the reason for initial values. Given the fact that our test programs are not large enough, large predictor size may take too long to warm up, and eventually hurt the performance.

Another interesting fact we found was when we increased the size of BTB from 32 to 128, the performance on `Objsort` is much better, which is shown in Fig. 7.

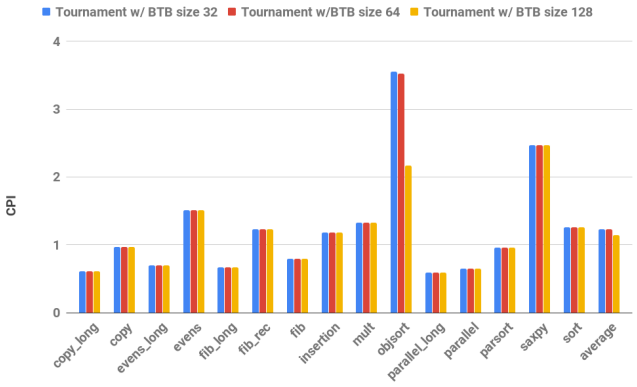

Fig. 7. CPI with different BTB size

Notice that besides `Objsort`, the performance is almost the same with respect to BTB size. As for `Objsort`, we believe the main reason for improvement is that given the policy of `JSR` instructions mentioned above, it resolves the conflict between multiple `JSR` instructions. For example, the `JSR` on line 22 and `JSR` on line 54 in `Objsort` may have conflict in BTB if the size is too small. Increasing BTB size to 128 could well solve this problem.

Overall, the following chart summarizes the improvement brought by branch prediction.

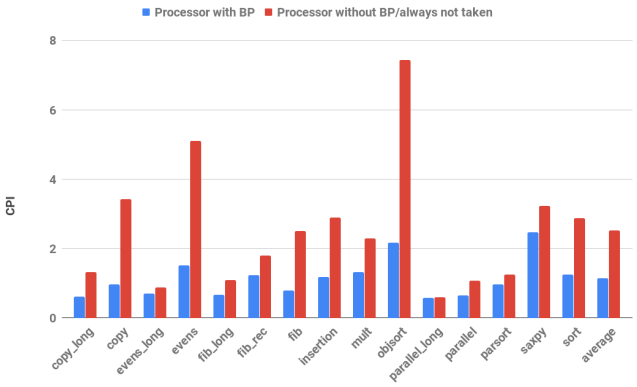

Fig. 8. CPI with BP vs. CPI without BP (always not taken)

We could see that using branch predictor, our performance will improve by 121.43%, with respect to the baseline processor which uses always-not-taken policy. This illustrates the effectiveness of our branch predictor.

C. Dependency-based Reservation Station Analysis

The following chart demonstrates the comparison of CPI between normal centralized reservation station vs. dependency-based reservation station.

One thing we find interesting is that when we increase the depth of steering RS form 4 to 8, the performance becomes

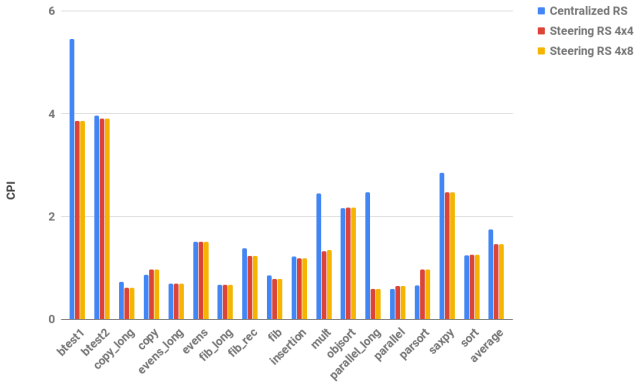

Fig. 9. CPI for centralized RS vs. dependency-based RS

worse. We think that the main reason is that some garbage instructions after halt come into the RS and issuing of illegal instructions will block the issuing of valid instruction.

Another interesting thing is that dependency-based reservation station not only reduces our clock period (From 8.5ns to 7.8ns) due to the simplification of issuing logic, but also improves the CPI by 19.76%. One potential reason could be that for centralized RS, the issue policy may not select the best instruction to execute. For example, RS may issue the instructions after a branch instruction first. But if that branch is mispredicted, these instructions are useless and it will eventually hurt the performance. On the other side, for dependency-based RS, since its issuing logic is simpler, it could potentially prevent these bad situations.

Despite the comparison between centralized reservation station and dependency-based reservation station, we also conducted some experiment to carry out the best depth(FIFO length) and width(FIFO number) of the dependency-based reservation station. we experiment with different settings including W4*D4, W4*D8, W2*D8, W8*D2 , and W8*D4. A quick conclusion is that W4*D4 achieve the best CPI-clock period balance. To keep the report tidy, we only represent a few interesting subset of data here, which includes W4*D4, W4*D8, W2*D8 and W8*D2.

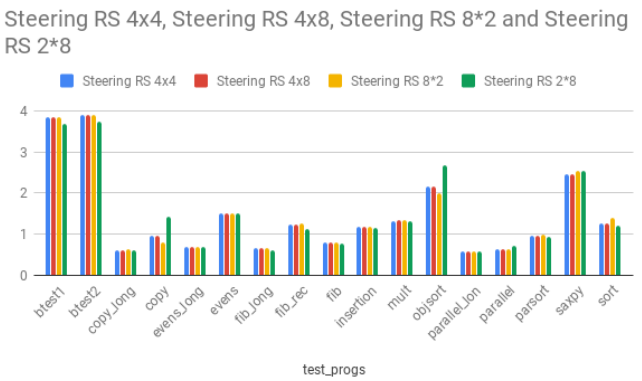

Fig. 10. CPI for different RS parameters

First of all, when comparing the W4*D4 and W4*D8, we

discovered that the depth of the reservation station can not affect the CPI on all test bench, this is due to the fact that our ROB is only 32 large. The W4*D8 reservation station can contain up to 32 instructions and that is the same as the ROB size. That makes it meaningless to increase the depth to 8.

For the same windows size 16, including W4*D4,W8*D2,W2*D8. The W4*D4 is also the most balanced one. It achieves a decent CPI on all test benches. For the W8*D2 version, which is quite good as well, but due to the fact of the increased issuing width, we were not able to reach the same clock period and we choose not to use it. For the W2*D8 version, the standard deviation is too high, so we also give up that one.

In conclusion, the W4*D4 is the most reliable version in terms of both CPI and clock period and is included in out final submission.

D. Functional Units Number Analysis

Fig. 11 shows the CPI performance regarding to different numbers of functional units. On average, reducing the number of ALUs to 1 is an obviously worse choice, and it helps only a little to increase the number of ALUs to 4. The increasement in the number of ALUs reduces the situations that the issues get stuck due to structural hazards in ALUs. The reduction of multipliers doesnt cause much penalty because the multiplications appear not so frequently in most of the programs.

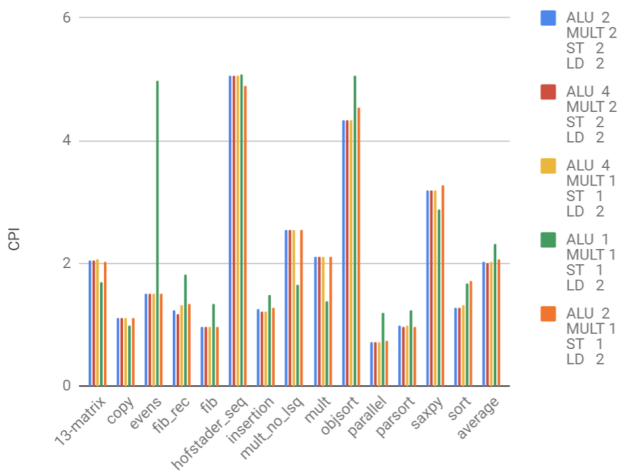

Fig. 11. CPI for different numbers of functional units

V. Group Dynamics

Jingru Hou (18%): Pipeline, ROB, Architecture map table, Store queue

Mathew Whittlesey (17%): Pipeline, Icache, Prefetch

Shiyu Wu (20%): Pipeline, Branch prediction, Victim cache, Reservation station

Tianyu Qiao (25%): Except the Branch Prediction and Prefetch.

Yujia Xie (20%): Pipeline, Map table, Dcache and ROB

VI. Conclusion

We implemented a MIPS R10K style 2-way superscalar out-of-order processor in SystemVerilog. Our processor uses a subset of the Alpha64 ISA, and incorporates a number of additional features discussed in this report. Our processor synthesizes with a clock period of 8ns, and achieves an average CPI of 1.46 across the public test cases provided for us. We utilized a random test case generator along with unit tests in order to rigorously validate the functionality of our processor.

VII. Acknowledgement

We are greatly thankful to Prof. Ronald Dreslinski, teaching assistants Jielun Tan, James Connolly, and Yichen Yang. We can not make it without the help from all of you.

VIII. Appendix

A. Feature Table

The following table summarizes the features we have implemented.

Feature	Included	Comments
MIPS R10K OoO	Yes	MIPS R10K Style Out of Order execution
2-way Superscalar	Yes	Two issues and two retires per cycle
Bimodal predictor w/BTB	Yes	Use 2-bit saturation counter
Gshare predictor	Yes	Use 2-bit saturation counter
Tournament predictor	Yes	Combine Gshare and Bimodal predictor
Return Address Stack	Yes	Drop the bottom item if full
Dependency-based Reservation Station	Yes	4 FIFO buffers with steering logic
More complex issue policy	No	No improvement for CPI
Multi-bank RS	No	No improvement for clock period
Store/Load forwading	Yes	Single store queue
Multiple outstanding load missed	Yes	
Next-line prefetching for I$	Yes	With prefetch size 20
Directly-mapped victim I$	Yes	
2-way associative D$	Yes	
Regression test	Yes	Run testcaes and compare with project3
Random testcase generator	Yes	Mainly for testing ALU and LSQ

TABLE I
List of our implemented features, along with any unique facts about them

B. Specification Table

The following table summarizes the parameters and specifications of our final submitted design.

Parameter	Value
ROB size	32
RS size	4FIFO queues, 4 instructions/FIFO queue
SQ size	7
Branch predictor	Tournament predictor
BTB size	128
Gshare predictor size	16
Bimodal predictor size	16
RAS size	16
Freelist size	32
# architectural register	32
# physical register	64
# ALU	2
# multiplier	2
# stage of multiplier	4
# load functional unit	2
# store functional unit	2
Width of CDB	2
Instruction buffer size	32
Prefetch size	20
Icache	256 bytes in total with 32 8-byte blocks
Victim icache	32 bytes in total with 4 8-byte blocks
Dcache	256 bytes in total with 32 8-byte blocks

TABLE II
PARAMETERS AND SPECIFICATIONS OF VARIOUS PROCESSOR STRUCTURES

REFERENCES

[1] Palacharla, Subbarao, Norman P. Jouppi, and James E. Smith. Complexity-effective superscalar processors. Vol. 25. No. 2. ACM, 1997.