5-Stage Pipelined RISC-V Processor

Cheng-Yan, Du Pang Cheng, Chen Jui Yang, Hsu

Department of Electrical Engineering, National Taiwan University

{b05901156, b05901037, b05901022}@ntu.edu.tw

https://github.com/MRdudu156/5-stages-pipelined-RISCV-CPU.git

Abstract

In our work, we implemented a 5-stage pipelined RISC-V CPU based on hardware description language Verilog, and designed our own caches to bridge our processor and external memories. Additionally, we implemented branch prediction and allowed compression instructions based on RISC-V C standard extension to improve performance in our structure. We experimented over two types of branch prediction structure and compared four kinds of instruction caches. Overall results show that practicing branch prediction by two level adaptive predictor and by 2-way associative half word cache using LRU placement scheme outperformed other experiments. As for baseline structure, synthesized results show that our processor can reach gate-level simulation time of 5835.75 ns on Fibonacci n=16 case, with area 277399.2874μm^2.

Index terms: *pipelined RISC-V, branch prediction, compressed instructions*

1. Baseline Architecture

As an initial architecture, we designed a structure which splits the whole computation into 5 stages. By cutting the stages, we can reduce the critical path significantly. However, not every instruction can be written back to RISC-V's register within 5 cycles. If an instruction does a computation based on the result of the previous instruction, "hazard" would occur. To handle the most common hazard, which is the case mentioned above, we need to add a forwarding unit to this structure to make sure the results of the previous computation could be sent to the following one right in time.

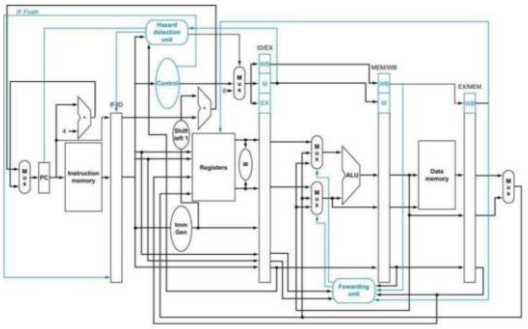

figure 1.1 basic structure of the RISC-V

The given structure computes the branch instruction in ID stage. Although the whole circuit is constructed intuitively, we would face two serious critical path in such case. To handle this problem, we came up with two different ways to solve the problem.

In general, processor communicates with memory directly requires a lot of stall cycle every time it interacts with memory. Thus, the cache is needed to bridge between our processor and memory. We implement two kinds of cache, which are direct-map and 2-way cache, to see which has better performance.

What's more, the cycle time of the gate-level simulation that our circuit could pass is way larger than the synthesize cycle we give. After surveying on this problem, we found that it was the structure in cache that leads to the problem. Therefore, we optimize the cache with some technique.

1.1 Branch in ID/EX stage.

The critical path in the case of branch in ID is very long. The first part of the path is that forwards data from ALU in EX stage to ID stage. The second one is from branch to next PC address since it needs to calculate the address after the data is read from the register. Then PC

address will choose the real address through a series of MUX. The first part can be reduced by considering the situation of forwarding from EX to ID as a hazard. Thus, by forwarding data from MEM and WB can handle the critical path by giving a bubble whenever an EX-to-ID hazard happened. However, the second path cannot be handled easily in this structure. Therefore, we came up with an idea to branch the address at EX stage.

The structure of branching in EX is made to reduce the critical path. Since it calculates the path by the data read from the flip-flop, we predict that it's acceptable cycle time should be lower than branch in ID stage.

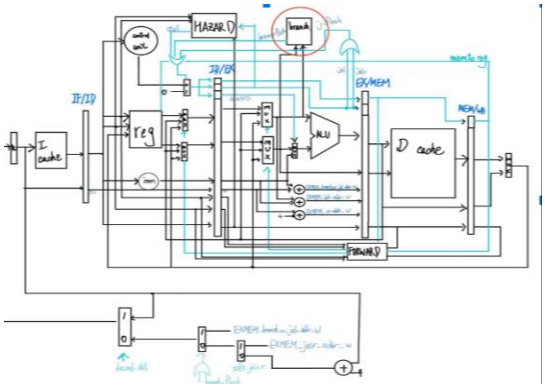

figure 1.2 structure of branch in EX stage

1.2 Cache optimization

Our initial cache calculates the data right after receiving the signal from external memory. Nevertheless, the given memory sends signal on negedge, which restricts our cache to handle data within a half cycle. Thus, the structure that DC Compiler synthesized will somehow be broken since the asynchronous problem. To ease the restriction, we add a state of "buffer" to stall data to the next cycle. Therefore, the cache could have a complete cycle to process the data from memory.

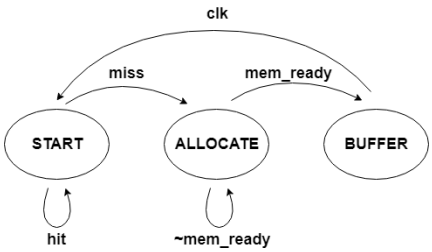

figure. 1.2.1 FSM of cache after adding buffer

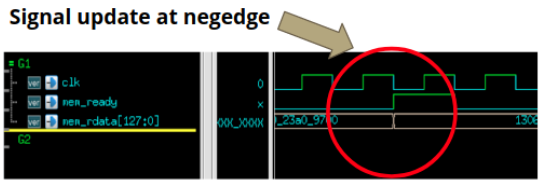

figure 1.2.2 waveform of interaction between cache and memory

2. Extension - Branch Prediction

Branch instructions are not detectable in our default structure, which means RISC-V CPU will load the next instruction at PC+4 instead of the branch target, and will cause a two-cycle-penalty if we execute the branch instruction in EX stage. To reduce such penalty, a "Branch Prediction Unit" (BPU) is proposed.

With BPU, we can identify and execute the branch instructions in IF stage. That is, the target address needs to be calculated and the prediction needs to be made in IF stage. The prediction will be verified in ID stage, in order to fix miss-prediction and maintain the correctness. The penalty of miss-prediction is two cycles, because we need to flush the wrong instructions in IF stage and ID stage. The block diagram of our BPU design is shown in figure 2.1.

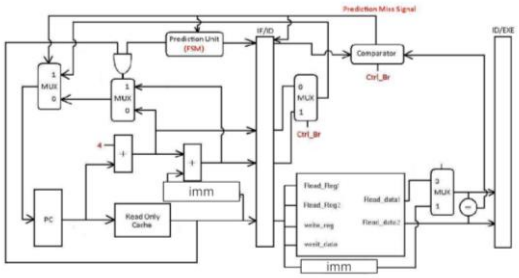

figure 2.1 block diagram of BPU

In our work, we implemented two kinds of BPU: (1) one-level predictor with 2-bit saturating counter, (2) two-level adaptive predictor.

2.1 One-level predictor with 2-bit saturating counter

A 2-bit saturating counter is a finite state machine with four states: (1) strong-taken, (2) weak-taken, (3) strong-not-taken, (4) weak-not-taken. The finite state machine is updated by whether the branches are taken or not.

When we evaluate the branch as taken, the state changes toward strong taken; otherwise, the state changes toward strong-not-taken. The feature of 2-bit saturating counter is that when it is in the strong-state, it will not change prediction until it missed twice. The 2-bit saturating counter performs well when branches always being taken or not taken. However, it has poor performance in handling inter-branches. In the worst case, it will keep making wrong prediction because the state is repetitively switching between weak-taken and weak-not-taken.

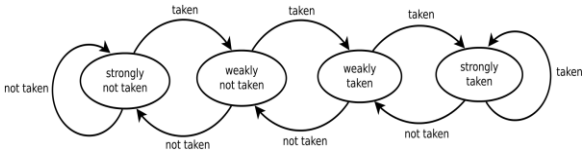

fig. 2.2 2-bit saturating counter

2.2 Two-level adaptive predictor

A two-level adaptive predictor remembers the history of the last n occurrences of the branch and uses one saturating counter for each of the possible 2^n history patterns. This method is illustrated in figure 2.2.1. With the help of pattern history table, two-level adaptive predictor works more efficiently than the 2-bit saturating counter in the case that branch patterns are regularly recurring. In our work, we implemented the last mentioned BPU with n = 2 and used a 2-bit saturating counter for four possible history patterns.

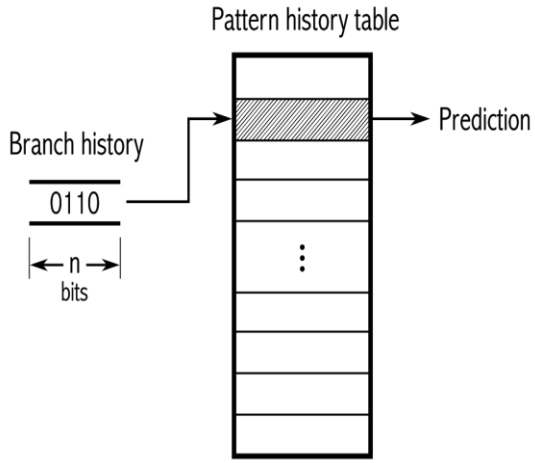

fig. 2.2.1: Two-level adaptive branch predictor. Every entry in the pattern history table represents a 2-bit saturating counter of the type shown in fig. 2.2

3. Extension - C Extension

Based on RISC-V C standard extension, our hardware supports 16 basic 16-bit compressed instructions as followed:

C.ADD	C.ANDI	C.LW	C.J
C.MV	C.SLLI	C.SW	C.JAL
C.ADDI	C.SRLI	C.BEQZ	C.JR
C.NOP	C.SRAI	C.BNEZ	C.JALR

The decompression function can be done via at most 4 layers of MUX. Caution that only C.SW and C.LW are with zero-extension immediate, other instructions are with sign-extension immediate.

During usage, placement of 16-bit and 32-bit instructions are not constrained, thus addition to PC may differ (i.e. PC+4 for 32-bit instructions and PC+2 for 16-bit instructions). Our implementation on C extension also involved with re-design of instruction cache, to further optimize instruction fetching.

In our standard cache, we have 8 blocks with 4 words each, totally 32 32-bit instructions capacity. The instruction cache is customized to read-only for the redundancy of writing back to instruction memory. Loading 16-bit and 32-bit instructions in the standard cache, we have 4 cases to consider: (1) 32-bit instruction starting from a complete word, (2) 16-bit instructions starting from a complete word, (3) 32-bit instruction starting from half of a word, (4) 16-bit instruction starting from half of a word. Among the 4 cases, (1) (2) (4) can be directly done, (3) needs information of the next word for a complete instruction. Aiming at the (3) case, we considered three attempts to solve the issue: NOP insertion, instruction pre-loading, and half word cache.

3.1 NOP insertion and Instruction Pre-Loading

Since (3) case has an incomplete instruction fetch, the simplest way to handle is to insert a NOP to CPU for complete instruction loading,

and buffer the previous half instruction. Nevertheless, such method is too cycle expensive if consecutive (3) cases occur.

To compensate the disadvantage of NOP insertion, we added an instruction pre-loading mechanism. When (2) or (3) (consider if it is already completely fetched) case occurred, we can simultaneously observe the next half word and examine if it is a 16 or 32-bit instruction. If it is a 16-bit instruction, proceed normally; if it is a 32-bit instruction, buffer the next half word and send a PC+4 signal to pre-load the rest 16-bit of the instruction. When the 32-bit instruction is successfully loaded, simply send a PC+2 signal to compensate the borrowed PC on previous step. Caution that the PC addition we mentioned here is only send to cache in the next cycle, other PC signal transmission and PC addition on other stages should be done normally.

However, for cases with branch or jump to half of a word, we still need to insert a NOP to CPU for complete reading, which cost still remains possibly high.

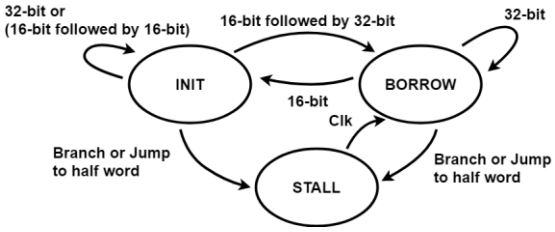

fig. 3.1.1 FSM of instruction pre-loading mechanism

These methods are not efficient enough, but can be done almost without changing the standard cache structure. Benefit of the scheme is that additional area may be small. To further improve the mechanism, we need to redesign the cache for avoiding the NOP.

3.2 Half word cache

Rethinking the 4 cases we have mentioned, one of the affecting factors is whether the address located on a complete word or middle of a word. To deconstruct the boundaries, we changed our cache from 8 blocks with 4 words each to 8 blocks with 8 half words each. In other words, we stored 64 16-bit half words instead of 32 32-bit words (see fig. 3.2.1).

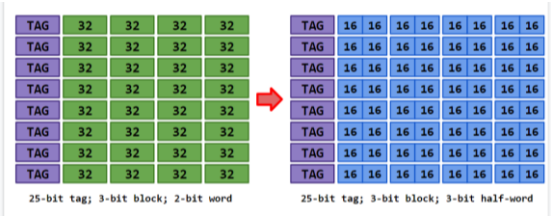

fig. 3.2.1 Half word cache

Using half word cache, we simply get 2 consecutive half words to either get a complete 32-bit instruction or only use the previous one as a 16-bit instruction. The detection of 16 or 32 bit can be easily done by checking OP code of the first half word. If the instruction is 16-bit, it will be fed into decompression unit for translation immediately, and send to the output of our read-only cache.

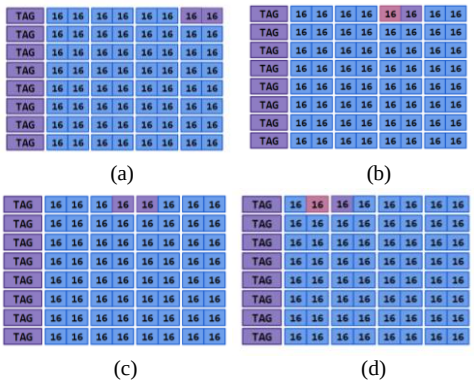

fig. 3.2.2 (a)~(b) are the 4 cases presented in half word format. Purple half words are instructions needed for current cases, and pink half words are the remaining half of a 16-bit instruction. All (1) ~ (4) cases can be easily solved since there is no more boundaries between PC+2 and PC+4 addresses anymore.

From Fig. 3.2.2, we can observe the flexible nature of half word cache can reach any address without buffering old half words, which can even benefit in avoiding branch or jump fetching errors. The only remaining problem is that if the current PC located at the last half word of a block, the latter half word might raise a tag missing error. For solution, we added an additional error detection to simultaneously check the validity of the first and second half word.

Beyond the benefit of compressed instructions, we further changed our direct mapping half word cache into a 2-way associative half word cache. As block- placement scheme, we chose LRU rule to rewrite old blocks. That is, we will only replace sub-blocks that are not

most recently used. It turns out that it saves even more miss cycles and improved simulation time with only a slight increase in area.

4. Experiment and Analysis

4.1 Baseline

To fully understand the difference between ID branch and EX branch structures and other changes, we analyzed their cycle times and area in combination with two different types of cache. The testing data in our experiment is based on (1) Fibonacci loop with different number N and (2) bubble sort. First, to know the variation of cycle time in simulation between branching in different stages, we analyzed the simulation time of both structures with the same direct mapping instruction cache. Second, we recorded the area change between branch in ID and EX stage in order to get the full AT performance of both structure. Last, we lay eyes on hit rate between two kinds of our cache. As a result, we can pick the cache that suits most to our testbench.

4.1.1 AT analysis

Initially, we assume that the structure of branch in ID should have better performance due to the fact that it will only waste a single cycle (i.e. flush IF stage for correctness) when needed. However, the experiment shows that branch in EX stage performs slightly better. The reason is that branch in ID stage requires a bubble every time when the branch instruction is used in the given testbench, which means the structure still waste 2 cycles as the branch in ID did. Even if the branch is not taken, the structure still needs to give a bubble away. On the other hand, branch in EX only needs to flush whenever the branch is taken. The meaningless loss in ID stage then leads to the slight difference of the experiment result than what we have predicted.

branch stage	simulation time
ID	20835 ns
EX	20255 ns

fig. 4.1.1 simulation time between two structure
(Tb cycle = 10, N = 16)

What's more, branch in EX stage also performs better on area. We suppose the cause is that if we compute the next PC address in EX stage, DC Compiler may merge the adder with other adders that are needed. Because the longest path, which is the path that ends at ALU unit, is almost parallel with the data flow that computes the next PC address.

branch stage	area
ID	290358.9372
EX	277399.2874

figure 4.1.2 area of branching in different stage
with the same direct-mapped cache

Different types of cache also affect the performance. In the given testbench (Fibonacci number = 16), the type of direct mapping performs the best. Different Fibonacci number will also alter the performance. The reason will be analyzed deeper in the next subtitle.

In short, the best performance on the given testbench comes with the structure of branch in EX stage matching with direct mapping cache. The area is 277399.2874 μm^2, with simulation time of 5835.75 ns. Under the synthesize cycle of 2.7 ns, our chip can pass gate-level simulation on 2.71 ns.

4.1.2 Cache

As mentioned above, the direct mapping cache has the best AT performance. In the case of Fibonacci number N = 16, the direct mapping cache performs the best. The difference is minor and stays the same until N = 28. The reason that the difference stays the same is caused by instructions in the beginning. Since the 2-way cache is only rewriting with FIFO rule, it accidentally replaced the different blocks that we want to throw back. This minor effect causes direct mapping cache to save about 10 cycle comparing with 2-way cache.

Fibonacci	n = 16	n = 20	n = 30
dm	20.835 us	30.455 us	***63.875 us***
2way	20.755 us	30.375 us	***63.625 us***

fig. 4.1.2 performance of different cache with different N in the structure of branch in ID stage

However, when N becomes 28 ~ 31, direct mapping cache stalls more than cases when N < 28. The reason behind is that the testbench will store the test to an "output-TestPort 0xff" before it stores the data in to the real address, which is the same as the index of Fibonacci series. Thus, when the N is bigger than 28, the direct map cache needs to stall every time it switches from 0xff to its real address. The 2-way cache then performs better.

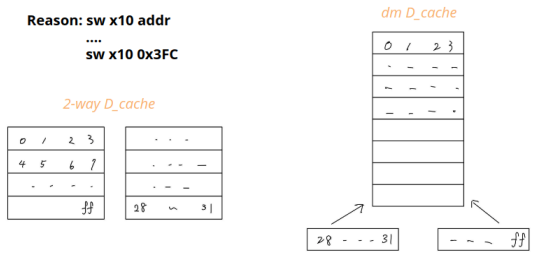

figure 4.1.3 2-way vs. dm cache in given test

To reduce the area of our whole chip, we make the instruction cache a read-only cache.

I_cache type	area
dm	304104.4808
read_only_dm	277399.2874

figure 4.1.4 area difference with different I cache in the structure of branch in EX stage

By removing "write back" state of our instruction cache, we observe that the area drops a significant rate of 8.8% without affecting the simulation time at all. Thus, the chip we design contains the structure of branch in EX stage, a read-only instruction cache and a direct mapping data cache.

4.2 Branch prediction

We have experimented with two different schemes of BPU: (1)2-bit saturating counter, (2)2-level adaptive predictor and record their execution time in different branch patterns. Then we calculated the AT performance under two different structure with Fibonacci series and Bubble sort testing data.

4.2.1 Performance analysis for Branch Prediction Unit

In order to analyze the performance of RISC-V after adding BPU, we used two kinds of testing data for experiment, one has inter-branch and the other one does not. For the experiment on no-inter-branch data, we set the value of total number of branches to 100 which is the sum of branch-taken and branch-not-taken then record the execution time for different combinations. Figure 4.2.1 shows that as the number of branch-not-taken decreases the execution time for RISC-V with BPU also decreases. When the number of branch-not-taken is ten, the execution time of the circuit with BPU is 43% less than the circuit which executes branch in EX stage and does not have the BPU. We can see that the more correct prediction for branch-taken, the more cycle time (2-cycle for each branch-taken) will be saved by BPU. Also, the performance of two different BPU is the same in this case. The reason is that this testing data does not have inter-branch cases, so it can't show the advantage of 2-level adaptive predictor over 2-bit saturating counter in prediction.

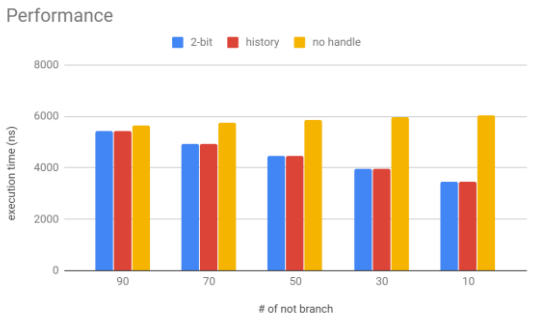

figure 4.2.1 execution time of no-inter-branch test data for different BPU structure

In order to test the performance on inter-branch, we used the testing data provided by TAs which will execute Fibonacci series and Bubble sort. Figure 4.2.2 illustrates the cycle time BPU saved compared with the circuit

without BPU, and N indicates the length of Fibonacci series (the bigger N, the more inter-branch). The saving cycle time of 2-level adaptive predictor is almost 2 times better than the 2-bit saturating counter.

	no handle	2-bit	save cycle(%)
N = 8	7245	6895	4.83%
N = 16	20045	18615	7.13%
N = 32	68485	62975	8.05%

(a) 2-bit saturating counter

	no handle	2-level	save cycle(%)
N = 8	7245	6635	8.42%
N = 16	20045	17355	13.42%
N = 32	68485	57795	15.61%

(b) 2-level adaptive predictor

figure 4.2.2 execution time of inter-branch test data for different BPU structure

The reason why 2-level predictor performs better is that it has branch history pattern table. For example, when the branch pattern is 001001001...001(0 for not-taken, 1 for taken), for 2-bit saturating counter, it will change state toward strong-not-taken and make wrong- prediction every time when branch needs to be taken. In contrast, for 2-level adaptive predictor, it will remember the relationship of states and history pattern (pattern 00 will saturate in strong-taken, the state for pattern 01 will saturate in strong-not-taken, the state for pattern 10 will saturate in strong-not-taken) so it will always do the right prediction in this case. Thus, 2-level adaptive predictor predicts better when branch patterns is periodic.

4.2.2 AT Analysis for BPU

Figure 4.2.3 illustrates the area of different BPU structures. Note that the synthesis cycle is 5 ns in our experiment. To support branch prediction, 2-bit saturating counter costs 7.05% more area and 2-level adaptive predictor costs 7.55% more area compared to the circuit which executes branch in EX stage. Figure 4.2.4 illustrates the value of area * (total simulation cycles). The test data in this experiment is Fibonacci series and Bubble sort with N = 8, 16, 32. When N = 32, we can see that 2-level adaptive predictor improves AT by 9.24% while 2-bit saturating counter improves only 1.56%. In general, the circuit with 2-level adaptive predictor has the best AT performance in our experiment and it is worth exchanging area for better AT performance with BPU structure.

	Area(5ns)	
2-bit	260730.8212	7.05%
2-level	261932.5804	7.55%
no handle	243554.831	x

figure 4.2.3 area of different BPU structure

	2-bit	2-level	no handle
N = 8	101.88%	98.49%	100.00%
N =16	99.42%	93.11%	100.00%
N = 32	98.44%	90.76%	100.00%

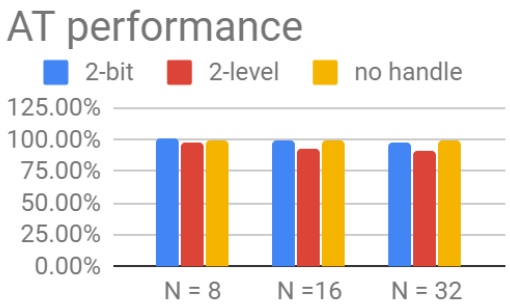

figure 4.2.4 AT performance of different BPU structure

4.3 C Extension

Over the C extension issue, we experimented variations of caches to optimize the performance and observe improvements of using compressed instructions. The instructions are set to execute 0x1234*0xABCD=0x0C374FA4, which the compressed instructions are generated

for the same purpose as the original instructions. The compressed instructions are with 18 32-bit instructions and 28 16-bit instructions in our testing case.

4.3.1 Half word cache

We experimented 4 types of instruction caches in our study: (1) read-only + direct mapping, (2) read-only + 2 way associative, (3) half word read-only + direct mapping, (4) half word read-only + 2 way associative. Note that our 2-way associative cache are using LRU rule for rewriting blocks, which turns out to be more reasonable. All experiments are done using the same RISC-V structure (except for slight changes to suit compressed instructions) and the same direct mapping data cache. For synthesizing constraints, we set a 5ns simulation cycle for all experiments to simplify control variables. All experiments are able to pass gate-level simulation with a simulation cycle of 5 ns, except for direct mapping half word cache which needs 5.2 ns to avoid simulation time violations.

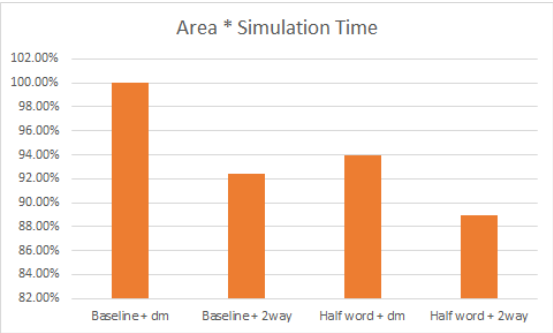

fig. 4.3.1 AT analysis of different instruction caches

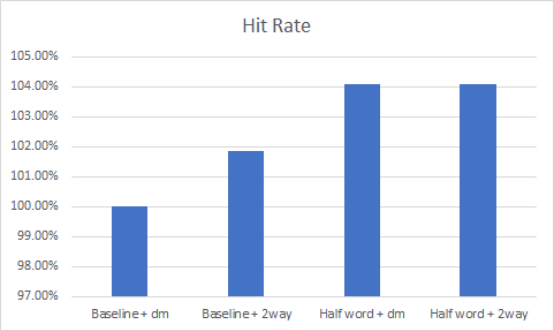

fig. 4.3.2 hit rate of different instruction caches

	Baseline + dm	Baseline + 2way	Half word + dm	Half word + 2way
Area ratio	100.00%	99.57%	106.11%	105.65%
Best cycle ratio	100.00%	92.64%	88.26%	83.84%
Best simulation time ratio	100.00%	92.78%	88.56%	84.15%
Best hit ratio ratio	100.00%	101.88%	104.10%	104.10%
Best AT ratio	100.00%	92.38%	93.97%	88.91%

fig. 4.3.3 Overall comparison of different instruction caches (for cycle ratio of half word + dm, we added a 5.2/5 cycle penalty for extra simulation cycle)

Fig. 4.3.3 shows the results of our 4 experiments. To support compressed instructions, half word costs 6% more area(approximately 16000 μ^2) for decompression unit. However, half word cache using compressed instructions can save approximately 10% simulation time. The reducing in time results from the high information density nature of compressed instructions, which with the same fetched information from external memory, compressed instruction can contain at most 4 times more instructions compared with original ones. With higher information density, we can reduce the miss rate for a ratio of up to 16%. The result can be think as exchanging information fetching time of cache with information fetching time of external memory, which is reasonable since cache can be further synthesized. Note that such tradeoff is successful only by using compressed instructions, and might worsen the result if not enough instructions are compressed. 2-way associative cache with LRU replacement scheme also outperform other experiments because the recursive instructions by the given testbench.

2-way associative half word cache outperforms other experiments with 11% of AT improvement compared with baseline structure, and 3.8% improvement with baseline 2-way associative structure.

5. Conclusion

In our work, we experimented 2 types of branch schemes for baseline architecture and obtained best result from EX branch structure, which outperformed ID branch structure with 4.5% in area and 2.8% in simulation time. As for extension, we implemented branch prediction in our RISC-V CPU structure and improved 9.24% of AT by adding 2-level adaptive predictor to our baseline architecture. We also implemented compression extension in our work, and improved 11% of AT by adding 2way read-only cache with compressed instructions.

6. Reference

[1] The RISC-V Instruction Set Manual
https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

[2] Computer Organization and Design
RISC-V edition