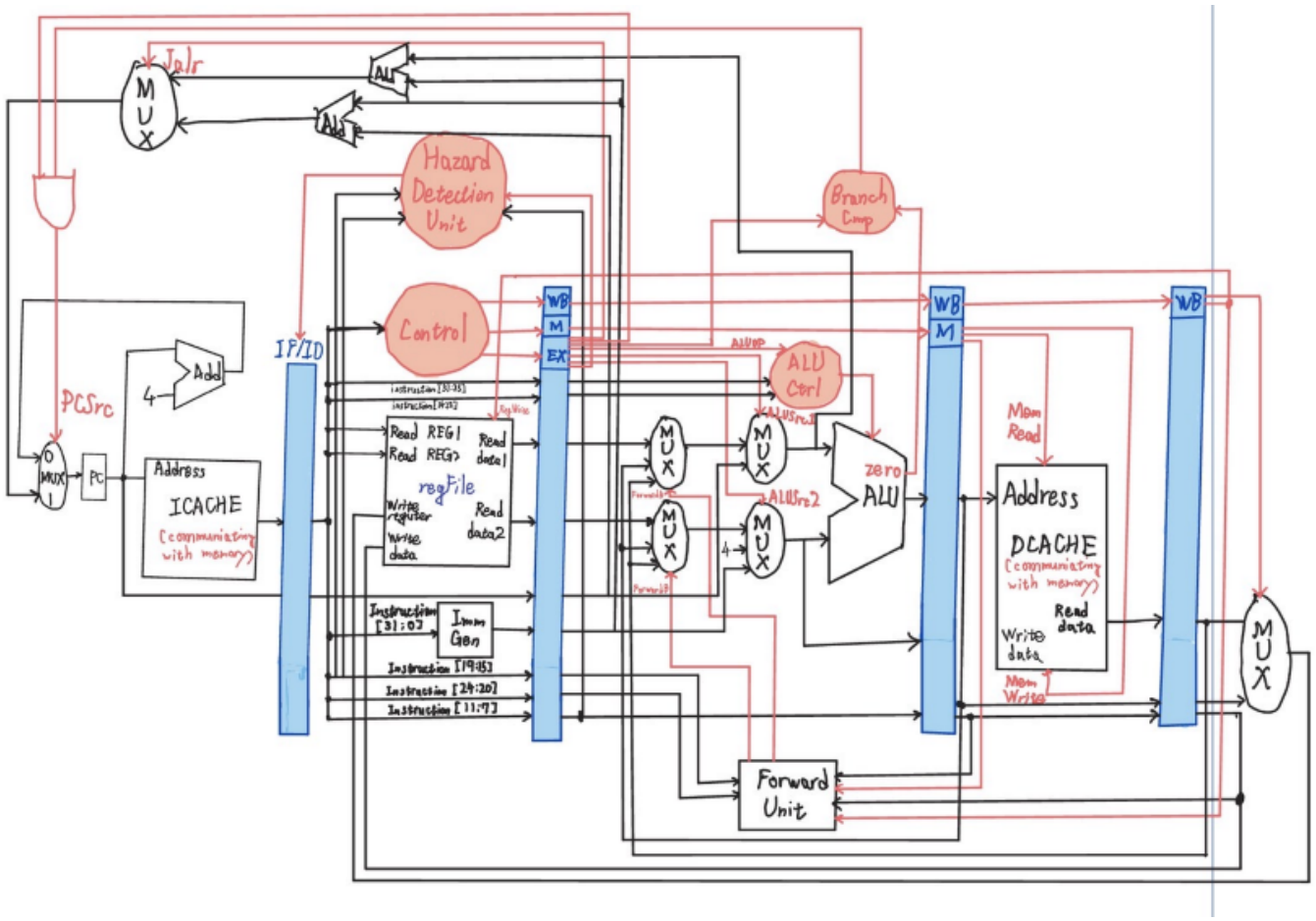
Digital System Design
Final Project Report

Group 7 田祐行 吳庭毅 林沛翰

CPU Architecture

We spent a lot of time drawing the entire CPU architecture, but it definitely saved us a lot of time.



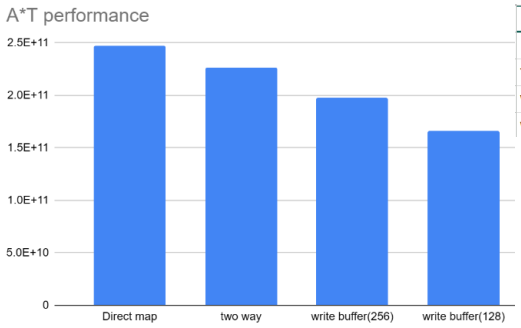
Optimization

Cache

A. Cache Architecture and writing policy

For the optimization of cache, we compare different cache architectures and writing policies under the same conditions such as branch policy, synthesis sdc constraints and other parts of cpu in order to evaluate their impact on CPU performance. From this analysis, we will select the best architecture to optimize other parts of the CPU. Four types of cache are compared in terms of AT performance:

1. dm + write_through
2. Two way + write_through
3. dm + write back + 256bits write buffer(128bits buffer for read and write respectively)
4. dm + write back + 128bits write buffer(read and write share same buffer)



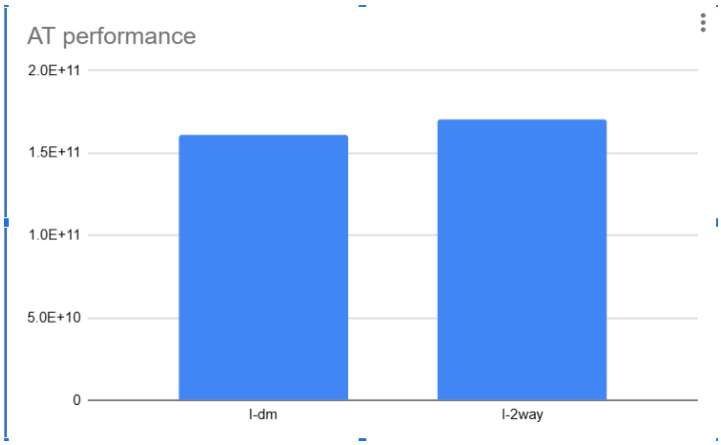
	Area(um^2)	Total time	A*T
Direct map	230384	1072151	247006435984
two way	271597	832618	226136550946
write buffer(256)	319377	619671	197908664967
write buffer(128)	321613	516392	166078380296

Remark: above architectures are simulated on Qsort.

Based on the figure above, it can be observed that when using write back+write buffer(128), although the area increases by 40%, it saves 51.8% of the total time simulation in comparison to the “direct map” architecture because we allow cpu not to stall when we write data to memory . As a result, the overall AT performance can be significantly improved.

B. D_Cache and I_Cache

In part A, both D_Cache and I_Cache are configured as read/write caches. However, I_Cache can be implemented as a read-only cache. Therefore, for the I_Cache component, we compare the performance of direct-mapped (dm) and two-way set-associative (two-way) caches to determine which one is more suitable as the I_Cache.

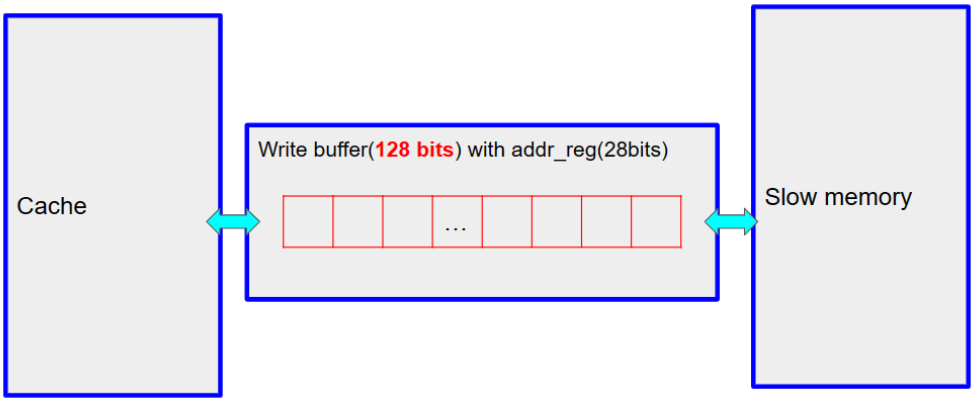


Remark: above architectures are simulated on Qsort

Based on the comparison in the above figure, it is observed that replacing the I_Cache with a read-only cache results in a reduction in area and an increase in AT performance. Additionally, through analysis of the graph, it can be seen that a direct-mapped (dm) cache performs better as the I_Cache. This is because the I_Cache exhibits the characteristic of sequentially reading instructions in a predictable manner, thus minimizing conflict misses in a direct-mapped cache.

C. How to solve the setup time violation of cache

The figure above shows the modified architecture with a write buffer serving as a communication bridge between the cache and memory. However, the slow memory



provides data at the negative edge, while the write buffer reads the data at the positive edge. As a result, this circuit has only half a cycle to run, and it cannot finish computations in time(half cycle). Consequently, a setup time violation occurs at the flip flop that receives memory data in the write buffer.

D. What we choose

Based on the above analysis, we have decided to utilize the following cache architecture as the baseline for optimization:

D_cache: dm + write back + 128bits write buffer
I_cache: dm

Critical Path

We have designed 4 different versions of baseline architecture:

(1) Jump, branch comparator and determine branch at ID stage
(2) Jump, branch comparator at ID stage, but determine branch at EX stage
(3) Jump at ID stage, branch comparator and determine branch at EX stage
(4) Jump, branch comparator and determine branch at EX stage

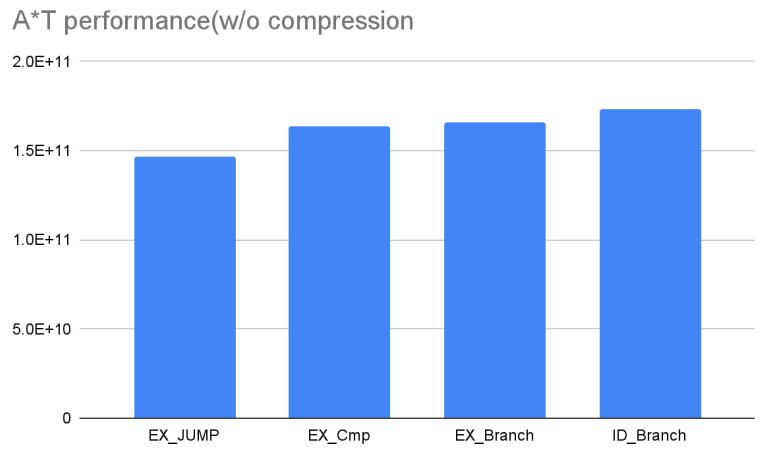
Although (1) has only one cycle penalty when jump and branch prediction is wrong, the critical path is longer beginning from MEM stage forwards reg data to branch comparator, making

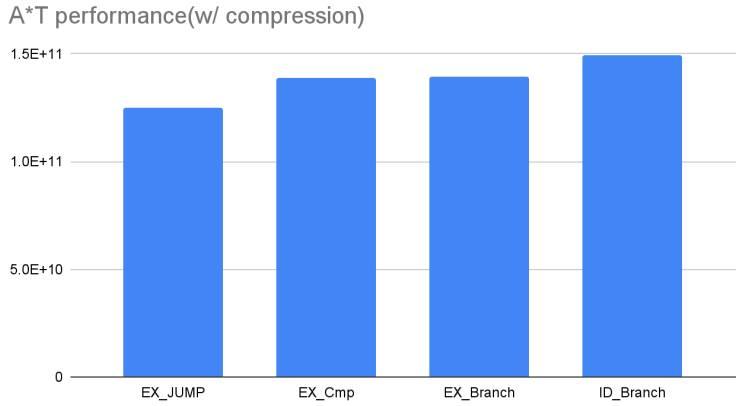
branch decision, and changing the next PC. It results that we can not optimize our simulation time because of constrained cycle time, and it also has a larger area.

(2) delay branching to EX stage, it slightly shortens the critical path, but still too long and has two cycle penalty when branch prediction is wrong, so the performance is not so good.

(3) delay branch comparison and branching to EX stage, it lower the long branching path, but the critical path changes to the jump path, starting from MEM stage forwards reg data to ID jump determination, and changing the next PC. Although the area and critical path is slightly lower, the critical path is still too large.

The performance of the 4 different architectures are shown below:





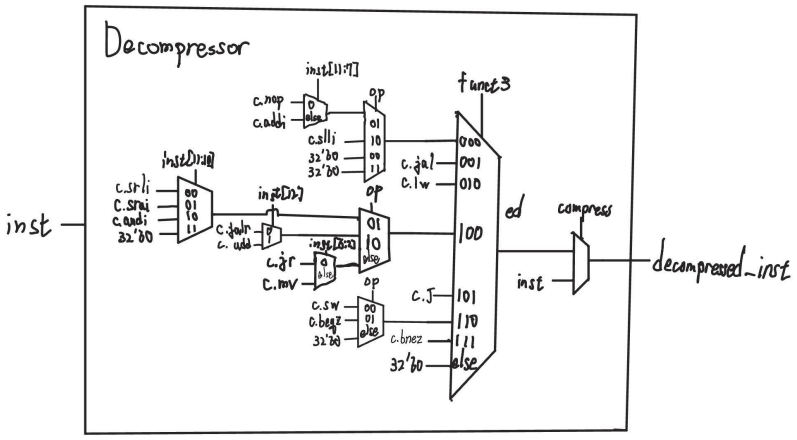
From above observation and analysis, we choose (4) architecture at last, although when jump and branch prediction is wrong, it flushes IF and ID stage, causing two cycle penalty, its critical path is the lowest as low as 3.0ns or so. Because it doesn't need to forward reg data for branch and jump, the critical path changes to reading register data from RegFile at ID stage. This version also has the lowest area, so the A*T performance is the best.

Extension - Compression

To support 16 bits compression instructions, we have to solve 3 problems:

1. Extract Information in C instructions
2. PC Increment Selection
3. Address Alignment

We solve the first problem by adding a decompressor, it takes the instructions from memory as input, then converts them to 32 bits instructions. Here is the architecture of decompressor:

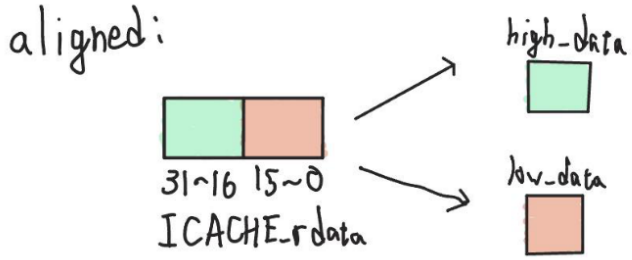


The second problem is easy and can be solved by first determining whether the current instruction is compressed or not, and making the next PC +2 or +4.

The third problem is complex and we think of a strategy to solve it. Here is our strategy:

1. ICACHE_addr = PC[31:2] or PC[31:2] + 1, depending on PC aligned or unaligned
2. Use 2 registers high_data and low_data to store low/high part of ICACHE_rdata
3. Determine the fetched instruction is compressed or not from high_data register
4. Determine the original instruction, put it into Decompressor

In this way, we can make sure that we fetch correct instruction corresponding to current PC, we use the following pictures to explain:

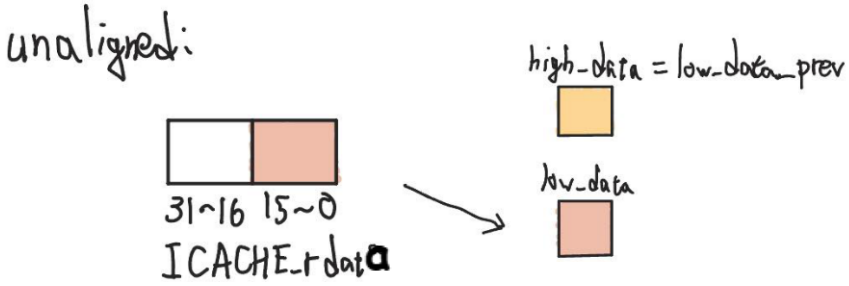


If not compressed

=> inst = {low_data[7:0], low_data[15:8], high_data[7:0], high_data[15:8] }

If compressed

=> inst = {l6'b0, high_data[7:0], high_data[15:8] }



If not compressed

=> inst = {ICACHE_rdata[23:16], ICACHE_rdata[31:24], high_data[7:0], high_data[15:8]}

If compressed

=> inst = {16'b0, high_data[7:0], high_data[15:8]}

We may encounter some situations including:

(1) PC is aligned, and fetch 32 bits instruction

(2) PC is aligned, and fetch 16 bits instruction

(3) PC is unaligned, and fetch 32 bits instruction

(4) PC is unaligned, and fetch 16 bits instruction

The (1) is easy and will remain this situation at next PC + 4

The (2) is also easy, make ICACHE_addr is PC[31:2], low_data and high_data store the low/high part of ICACHE_rdata. Because the fetched data is little-endian, the lower 16 bits of an instruction is stored by high_data now, we can use it to determine if the current instruction is compressed or not. If it is, we only use high_data in this cycle, and store low_data value so that it can be used in the next cycle.
After (2), the PC is added by 2 and becomes unaligned, then we may encounter (3) or (4).
If (4) happens, ICACHE_addr = PC[31:2] , low_data still store the low part of ICACHE_rdata, but make high_data = low_data_previous so that the high_data is now the low part of previous ICACHE_addr.
If (3) happens, the 32 bits instruction is not at a 4-byte address, so we only get half of it from the high part of the current ICACHE_rdata. But because we have already store its another half in (2) as low_data, we can get it from high_data = low_data_previous, and combine them to be a complete 32 bits instruction.

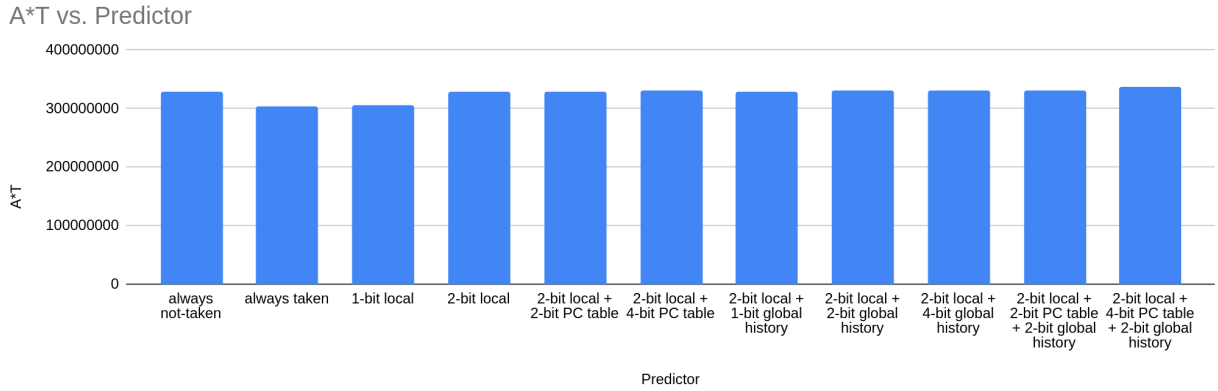
However, except for (1)~(4), we also encounter the branch and jump problems, especially when the branch/jump PC is unaligned and a 32 bits instruction is not at the 4-byte boundary, i.e. (4) is encountered after branch/jump. Because in our strategy, we should store the half of 32 bits instruction at the previous ICACHE_addr. But in this situation, we cannot do it in this way, we have to stall the PC for one cycle and look forward to get half of the 32 bits instruction. We use a flag to determine whether we should look forward, and set the flag in the situation that branch/jump is encountered and the next PC is unaligned. Then, in the next cycle, we stall the PC, insert a nop, make ICACHE_addr = PC[31:2](namely, PC-2,), and store the low part in ICACHE_rdata. Therefore, we can combine the complete 16 bits or 32 bits instruction in the next cycle as we solve (3) and (4).

Extension - Branch Prediction

Different branch prediction strategies have been examined based on the BrPred and Q_sort testbench. Since the performance highly depends on the characteristics of the test program, all prediction strategies are compared in terms of AT performance. The one with the best performance is chosen for the final results.

BrPred Test Case

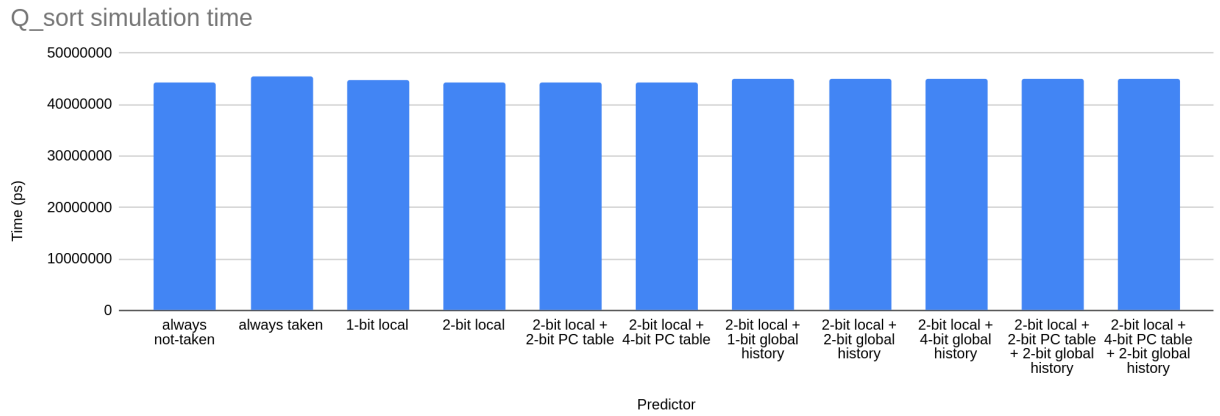
The chart below is the AT score of each prediction strategy. The “always taken” and 1-bit predictor result in the best performance, thus the 1-bit predictor is used for this test case.



When comparing the AT scores of different strategies, the same cycle time (10 ns) is used for synthesis. Also, “compile” option is used rather than “compile_ultra” for a fair comparison.

Q_sort Test Case

The chart below is the RTL simulation time of each prediction strategy. The two predictors, “always not taken” and “2-bit predictor + 2-bit PC table” have the best performance. Taking the area into consideration, the “always not taken” strategy is used for this test case. It can also be seen that for this case, the impact of the branch predictor on the overall performance is negligible. Possible reasons are that incorrect predictions in our architecture result in only one cycle wasted, and that taken predictions sometimes have extra penalties related to compressed instruction alignment.



Q_sort final architecture

From above observation and analysis, we decide to run Qsort testcase using

Branch strategy: always no taken

D_cache: dm + write back + 128bits write buffer

I_cache: dm

Instruction: Compression

With

Area: 270511 um^2

Total simulation time: 461440 ns

A*T: 270511 * 461440 = 1.248*E11