IPLC Simulator Write-up

Project completed by: Liliana Campuzano, Nick Curtis, Annie Pa, and Karina Sinha

Project Divisions

Nick and Annie worked on the cache functions while Karina and Liliana worked on the pipeline functions.

Summary of Implementation

Our cache line struct includes:

- An array of integers to represent the valid bits
- An array of integers to represent the tags
- An integer to hold information on associativity
- An array of integers for selecting which item in the cache will be replaced

We begin by creating our cache's valid bit, tag, and replace arrays using malloc. We initialize the valid bit and tag arrays to 0, and the replace array with just the index of whatever number it is. We then initialize the pipeline to 0 and NOP instructions.

Then we implement our LRU_replace_on_miss function. This function takes the index and tag of where we want to replace and uses those values to change the values in the cache. We do a similar method of implementing the LRU_update_on_hit. First we loop through the cache until we find the index of what we want ot update, then we update the LRU.

The pop_bits function uses a bitmask to do its job, and we included a function to print a 32-bit binary string.

We floor divide first in the trap_address function to find the bit association. We then set up a bit mask of 1s the size of the cache index and find the index bits of the address, then we remove the unnecessary bits. Then we set up a bit mask to find the tag and repeat the removal of unnecessary bits. We check to see if the tag we get is already in our tag array, and if it is, we stop and take it. If not, we continue on to call the LRU_update_on_hit function. If it's a miss, we call the LRU_replace_on_miss function.

Now for the pipeline functions.

In the push_pipeline_stage function, we set up integers to check for stalls and misses. We then go on to check whether or not the prediction was correct or not. Then, we check for stalls if the lw is loading branch or rtype. Then we check for hits or misses. We then check for SW memory accesses and data misses and add delay cycles if needed. We increment pipe_cycles by 1, and then we push the stages through.

We finish off the pipeline functions by basing them off of the one we were given. For all of them, we push_pipeline_stage and then proceed to assign itypes, stages, and instruction addresses depending on which function we are in.

Performance Evaluation:

Which configuration yielded the best performance?

In this section, we will evaluate our data found and decide which configuration of cache size, block size, level of association, and branch prediction (taken or not taken) yields the best performance.

Before we begin, here is a quick definition of performance:

$$Performance = \frac{1}{Execution\ Time}$$

$$Performance_A > Performance_B$$

$$\frac{1}{Execution\ Time_A} > \frac{1}{Execution\ Time_B}$$

$$Execution\ Time_B > Execution\ Time_A$$

Conclusion:

Lower Execution time = Better Performance

Execution time, however, is difficult to measure because the computer clock cycles work differently than an analog clock. This is because computers often shared; A processor may work on multiple programs at a time in which case the system may try to increase throughput (work done per time) rather than decrease the elapsed time of a single program. In this case, we will be using CPU time to determine our performance because CPU time is the actual time the CPU spends computing for a specific task.

To compute this, we will use the following formula:

$$CPU\ time = \frac{Instruction\ count * CPI}{Clock\ rate}$$

We will be looking for the lowest CPU time to determine the best performance. Our clock rate and Instruction count will be the same in all tests, so we can pretend they're not there which also means we can pretend this following equation is valid:

$$CPU\ time = CPI$$

So in the end we are just looking for the lowest CPIs we can find.

This is a chart of the tests performed in order.

					Cache Performance				Pipeline Performance				
Test #	Cache Size (index)	Blocksize	Level of Association	Taken/Not-taken	Num of Cache Accesses	Num of Cache Misses	Num of Cache Hits	Cache Miss Rate	Total Cycles	Total Instructions	Total Branch Instructions	Total Correct Branch Predictions	CPI
1	1	1	1	0	35863	35863	0	1.000000	364569	34752	7044	0	10.490590
2	1	1	1	1	35863	35863	0	1.000000	357525	34752	7044	7044	10.287897
3	2	1	1	0	35863	17190	18673	0.479324	192162	34752	7044	4350	5.529523
4	2	1	1	1	35863	17190	18673	0.479324	193818	34752	7044	2694	5.577175
5	2	2	4	0	35863	24450	11413	0.681761	256298	34752	7044	5554	7.375058
6	2	2	4	1	35863	24450	11413	0.681761	260362	34752	7044	1490	7.492000
7	3	3	1	0	35863	4074	31789	0.113599	73210	34752	7044	5358	2.106641
8	3	3	1	1	35863	4074	31789	0.113599	76882	34752	7044	1686	2.212304
9	2	1	2	0	35863	17190	18673	0.479324	192162	34752	7044	4350	5.529523
10	2	1	2	1	35863	17190	18673	0.479324	193818	34752	7044	2694	5.577150
11	3	3	2	0	35863	4074	31789	0.113599	73210	34752	7044	5358	2.106641
12	3	3	2	1	35863	4074	31789	0.113599	76882	34752	7044	1686	2.212304
13	3	3	3	0	35863	9635	26228	0.268661	122963	34752	7044	5554	3.538300
14	3	3	3	1	35863	9635	26228	0.268661	127027	34752	7044	1490	3.655243
15	1	1	4	0	35863	30303	5560	0.844966	314529	34752	7044	0	9.050673
16	1	1	4	1	35863	30303	5560	0.844966	307485	34752	7044	7044	8.847980
17	4	1	4	0	35863	35766	97	0.997295	363696	34752	7044	0	10.465470
18	4	1	4	1	35863	35766	97	0.997295	356652	34752	7044	7044	10.262776
19	4	3	2	0	35863	2055	33808	0.057301	54817	34752	7044	5605	1.577377
20	1	3	1	0	35863	7750	28113	0.216100	106788	34752	7044	4764	3.072859
21	4	6	2	0	35863	1055	34808	0.029418	45714	34752	7044	5711	1.315435
22	4	9	2	0	35863	1055	34808	0.029418	45714	34752	7044	5711	1.315435
23	4	9	1	0	35863	1055	34808	0.029418	45714	34752	7044	5711	1.315435
24	4	6	1	1	35863	1055	34808	0.029418	50092	34752	7044	1333	1.441413
25	4	5	1	0	35863	2055	33808	0.057301	54817	34752	7044	5605	1.577377
26	5	6	1	0	35863	759	35104	0.021164	43175	34752	7044	5711	1.242375
27	6	4	1	0	35863	770	35093	0.021471	43232	34752	7044	5729	1.244015
28	7	1	1	0	35863	1390	34473	0.038759	48704	34752	7044	5730	1.401473
29	5	6	0	0	35863	35863	0	1.000000	364569	34752	7044	0	10.490590
30	5	6	1	1	35863	759	35104	0.021164	47553	34752	7044	1333	1.368353
31	5	4	1	0	35863	1283	34580	0.035775	47745	34752	7044	5729	1.373878

Editor's Note: I understand that we were only asked to test 18 different configurations. I did more because I was determined to find the best configuration possible in this simulation.

After sorting, we can see that the lowest CPI we can find is 1.242375 which stems from the following combination:

Cache size (index): 5

Blocksize: 6

Level of Association: 1

Taken/Not-taken: 0 (Not-taken)

					Cache Performance				Pipeline Performance				
Test #	Cache Size (index)	Blocksize	Level of Association	Taken/Not-taken	Num of Cache Accesses	Num of Cache Misses	Num of Cache Hits	Cache Miss Rate	Total Cycles	Total Instructions	Total Branch Instructions	Total Correct Branch Predictions	CPI
26	5	6	1	0	35863	759	35104	0.021164	43175	34752	7044	5711	1.242375
27	6	4	1	0	35863	770	35093	0.021471	43232	34752	7044	5729	1.244015
21	4	6	2	0	35863	1055	34808	0.029418	45714	34752	7044	5711	1.315435
22	4	9	2	0	35863	1055	34808	0.029418	45714	34752	7044	5711	1.315435
23	4	9	1	0	35863	1055	34808	0.029418	45714	34752	7044	5711	1.315435
30	5	6	1	1	35863	759	35104	0.021164	47553	34752	7044	1333	1.368353
31	5	4	1	0	35863	1283	34580	0.035775	47745	34752	7044	5729	1.373878
28	7	1	1	0	35863	1390	34473	0.038759	48704	34752	7044	5730	1.401473
24	4	6	1	1	35863	1055	34808	0.029418	50092	34752	7044	1333	1.441413
19	4	3	2	0	35863	2055	33808	0.057301	54817	34752	7044	5605	1.577377
25	4	5	1	0	35863	2055	33808	0.057301	54817	34752	7044	5605	1.577377
7	3	3	1	0	35863	4074	31789	0.113599	73210	34752	7044	5358	2.106641
11	3	3	2	0	35863	4074	31789	0.113599	73210	34752	7044	5358	2.106641
8	3	3	1	1	35863	4074	31789	0.113599	76882	34752	7044	1686	2.212304
12	3	3	2	1	35863	4074	31789	0.113599	76882	34752	7044	1686	2.212304
20	1	3	1	0	35863	7750	28113	0.216100	106788	34752	7044	4764	3.072859
13	3	3	3	0	35863	9635	26228	0.268661	122963	34752	7044	5554	3.538300
14	3	3	3	1	35863	9635	26228	0.268661	127027	34752	7044	1490	3.655243
3	2	1	1	0	35863	17190	18673	0.479324	192162	34752	7044	4350	5.529523
9	2	1	2	0	35863	17190	18673	0.479324	192162	34752	7044	4350	5.529523
10	2	1	2	1	35863	17190	18673	0.479324	193818	34752	7044	2694	5.577150
4	2	1	1	1	35863	17190	18673	0.479324	193818	34752	7044	2694	5.577175
5	2	2	4	0	35863	24450	11413	0.681761	256298	34752	7044	5554	7.375058
6	2	2	4	1	35863	24450	11413	0.681761	260362	34752	7044	1490	7.492000
16	1	1	4	1	35863	30303	5560	0.844966	307485	34752	7044	7044	8.847980
15	1	1	4	0	35863	30303	5560	0.844966	314529	34752	7044	0	9.050673
18	4	1	4	1	35863	35766	97	0.997295	356652	34752	7044	7044	10.262776
2	1	1	1	1	35863	35863	0	1.000000	357525	34752	7044	7044	10.287897
17	4	1	4	0	35863	35766	97	0.997295	363696	34752	7044	0	10.465470
1	1	1	1	0	35863	35863	0	1.000000	364569	34752	7044	0	10.490590
29	5	6	0	0	35863	35863	0	1.000000	364569	34752	7044	0	10.490590

The following explains how we got to the above calculations.

After starting with the first 18 tests, it became apparent that, on average, the CPIs for branch predictions of NOT-TAKEN were less than those of branch predictions of TAKEN (See figure below).

Taken/Not-taken	CPI
0	10.490590
1	10.287897
0	5.529523
1	5.577175
0	7.375058
1	7.492000
0	2.106641
1	2.212304
0	5.529523
1	5.577150
0	2.106641
1	2.212304
0	3.538300
1	3.655243
0	9.050673
1	8.847980

10.490590	10.2879
5.529523	5.577175
7.375058	7.492000
2.106641	2.212304
5.529523	5.577150
2.106641	2.212304
3.538300	3.655243
9.050673	8.847980
0 5.715869	1 5.732757

Averages ^

So it was decided to continue testing with an emphasis on predicting NOT TAKEN.

In the beginning, it also became apparent that a block size of 3 was yielding the best CPI. However, a random decision to try out bigger numbers. Restrictions in size led to the decision that 4, 6, or 9 would be the best choice for block size (More apparent if you look at the chart chronologically aka chart 1).

Blocksize	CPI
1	1.401473
1	5.529523
1	5.529523
1	5.577150
1	5.577175
1	8.847980
1	9.050673
1	10.262776
1	10.287897
1	10.465470
1	10.490590
2	7.375058
2	7.492000
3	1.577377
3	2.106641
3	2.106641
3	2.212304
3	2.212304
3	3.072859
3	3.538300
3	3.655243
4	1.244015
4	1.373878
5	1.577377
6	1.242375
6	1.315435
6	1.368353
6	1.441413
6	10.490590
9	1.315435
9	1.315435

1	2	3
1.401473	7.375058	1.577377
5.529523	7.492000	2.106641
5.529523	7.433529	2.106641
5.577150		2.212304
5.577175		2.212304
8.847980		3.072859
9.050673		3.538300
10.262776		3.655243
10.2879		2.560208625
10.465470		
10.490590		
7.547294		

Green = Averages

4	5	6	
1.244015	1.577377	1.242375	1.242375
1.373878	1.577377	1.315435	1.315435
1.308947		1.368353	1.368353
		1.441413	1.441413
		10.490590	
		3.1716332	1.375067

9
1.315435
1.315435
1.315435

After multiple tests and double checking that the data didn’t lead to somewhere else, it was decided that the combination of 5 as the cache size, 6 as the block size, 1 as the level of association, and 0 as the branch prediction was the best choice as it yielded the smallest CPI (see chart 2).