Anna Tang
3180300155

Computer Organization: Assignment 5

7.5a) For a processor that runs data intensive applications with many load and store operations, the **write-back policy** is the most suitable. Because the processor is processing a significant number of operations, a write-through policy would take too much time and the write-buffer may not even be able to keep up. Write-back would be the most ideal since we can assume that the processor can generate writes as fast as they are handled by the cache.

b) For safety critical systems, **write-through** is the way to go. The write-through policy ensures consistency over performance, by always simultaneously writing data into both memory and cache. A buffer can also be used to allow the processor to continue with executing operations, because it stores the data waiting to be written into memory. In the event of an inconsistency, it is possible to refer to the data in the memory and rewrite it into the cache.

7.9) Direct-Mapped Cache

Index = Block Address mod 16

Reference Address	Index	Binary Index	Block Address	Data	H/M
2	2	0010	0	~~16, 64,~~ 48	**M**
3	3	0011	1		**M**
11	11	1011	2	2	**M**
16	0	0000	3	~~3, 19,~~ 3	**M**
21	5	0101	4	4	**M**
13	13	1101	5	21	**M**
64	0	0000	6	~~22,~~ 6	**M**
48	0	0000	7		**M**
19	3	0011	8		**M**
11	**11**	**1011**	**9**		**H**
3	3	0011	10		**M**
22	6	0110	11	~~11, 27,~~ 11	**M**
4	4	0100	12		**M**
27	11	1011	13	13	**M**
6	6	0110	14		**M**
11	11	1011	15		**M**

7.10) Direct-Mapped Cache w/ 4-Block Words

Block Offset = Block Address mod 4

Index = floor[(Block Address mod 16) / 4]

Reference Address	Index	Block Offset	H/M
2	0	2	**M**
3	**0**	**3**	**H**
11	2	3	**M**
16	0	0	**M**
21	1	1	**M**
13	3	1	**M**
64	0	0	**M**
48	0	0	**M**
19	0	3	**M**
11	**2**	**3**	**H**
3	0	3	**M**
22	**1**	**2**	**H**
4	1	0	**M**
27	2	3	**M**
6	**1**	**2**	**H**
11	2	3	**M**

Block Address	0	1	2	3
0	48%16 = 0		2	3
1	4	21%16 = 5	6	
2				11
3		13		

7.11)

```c
#include <stdio.h>
#include <time.h>

int main() {
    int arr[1000][1000];
    clock_t start, stop;
    double d = 0.0;

    //row major order
    for (int i = 0; i < 1000; i++) {
        for (int j = 0; j < 1000; j++) {
            arr[i][j] *= 2;
        }
    }

    stop = clock();
```

```
    d = (double)(stop - start) / CLOCKS_PER_SEC;
    printf("The run-time of row major order is %lf\n", d);

    //column major order
     for (int j = 0; j < 1000; j++) {
        for (int i = 0; i < 1000; i++) {
            arr[i][j] *= 2;
        }
    }

    stop = clock();
    d = (double)(stop - start) / CLOCKS_PER_SEC;
    printf("The run-time of column major order is %lf\n", d);

    return 0;
}
```

Output:

```
The run-time of row major order is 0.011132

The run-time of column major order is 0.020264
```

This shows that row major order is more efficient than column major order in the C programming language. It has a higher level of temporal and spatial locality, because of the loops and data accesses from sequential addresses.

7.12) Compute the total number of bits required to implement the cache

16 kB cache, 256 blocks with 16 words/block, 32-bit address assumed.

Total # of Bits = 2^n * (Block Size + Tag Size + Valid Field Size)

256 = 2^8 -> n = 8

2^m words -> m = 4

Tag Size = 32 – (n + m + 2) = 32 – (8 – 4 + 2) = 18

Total # of Bits = 2^8 * [(16 * 32) + 18 + 1]

= 135 966 bits

7.14)

Block Size = 16 Words, (b) Width = 4 Words, (c) # of Banks = 4

Memory latency is 10 memory clock cycles, transfer time is 1 memory bus clock cycle. Find miss penalties.

miss penalty = cycles to send address + cycles for each memory access + cycles to send a word

(a) miss penalty = 1 + (16 * 10) + (16 * 1) = 177 cycles
(b) miss penalty = 1 + (16/4 * 10) + (16/4 * 1) = 45 cycles
(c) miss penalty = 1 + (16/4 * 10) + (16 * 1) = 57 cycles
*cost is still paid to transmit EACH word!

7.20)

Cache Block = Address mod 4; system supports interleaving either 4 reads and 4 writes

Address	Cache Block	Conflict
3	3	No
9	1	No
17	1	Yes, 9
2	2	No
51	3	No
37	1	Yes, 17
13	1	Yes, 37
4	0	No
8	0	Yes, 4
41	1	No
67	3	No
10	2	No

7.32)

Cache miss penalty = 6 + Block Size (Words), CPI = 2, ½ of the instructions contain a data reference
Memory Stall Cycles = Read-Stall Cycles + Write-Stall Cycles

	Miss Rates	Miss Penalty	Read-Stall Cycles	Write-Stall Cycles	Memory Stall Cycles (Cache Misses)
Cache 1: Direct-Mapped w/ 1-Word Blocks	Instructions: 4% Data: 6%	6 + 1 = 7 cycles	0.04 * 7 reads/program = 0.28 reads/program	0.06 * 7 * 0.5 = 0.21 writes/program	0.28 + 0.21 = 0.49
Cache 2: Direct-Mapped w/ 4-Word Blocks	Instructions: 2% Data: 4%	6 + 4 = 10 cycles	0.02 * 10 reads/program = 0.2 reads/program	0.04* 10 * 0.5 = 0.2 writes/program	0.2 + 0.2 = 0.4
Cache 3: 2-way Set Associative w/ 4-Word Blocks	Instructions: 2% Data: 3%	6 + 4 = 10 cycles	0.02 * 10 * reads/program = 0.2 reads/program	0.03 * 10 * 0.5 = 0.15 writes/program	0.2 * 0.15 = 0.35

Cache 1's processor spends the most clock cycles on cache misses.

7.33)

CPU Time = Instruction Count (I) * CPI * Cycle Time, CPI = Memory Stall Cycles + CPI Perfect

If measured CPI of cache 1 is 2, then to find the base CPIs of each cache we do

CPI Perfect = CPI – Memory Stall Cycles -> CPI Perfect = 2 – 0.49 = 1.51

	Cycle Time	CPI	CPU Time
Cache 1: Direct-Mapped w/ 1-Word Blocks	420 ps	2	I * 2 * 420ps = 840 * 10 ^ -12 * I
Cache 2: Direct-Mapped w/ 4-Word Blocks	420 ps	0.4 + 1.51 = 1.91	I * 1.91 * 420ps = 802 * 10 ^ -12 * I
Cache 3: 2-way Set Associative w/ 4-Word Blocks	310 ps	0.35 + 1.51 = 1.86	I * 1.86 * 310 = 577 * 10 ^ -12 * I

Cache 1 is the slowest, and cache 3 is the fastest by the comparison of their CPU times.