CS 382 Fall 2023
Computer Architecture and Organization

Homework 4

Anna Hauk

I pledge my honor that I have abided by the Stevens Honor System.

Requirement (READ FIRST!)

You have to **type** the solutions. **Handwritten homework will not be graded and will receive zero credit.** You can annotate on this document directly (you might need to know how to insert a picture into a PDF file); or you can submit a separate PDF, but with solutions clearly marked with question numbers.

1 (15 points)

Consider three different processors P1, P2, and P3 executing the same instruction set. P1 has a 3.2 GHz clock rate and a CPI of 1.5. P2 has a 2.0 GHz clock rate and a CPI of 1.0. P3 has a 4.0 GHz clock rate and has a CPI of 2.3.

$CR = clock\ cycles/sec = \frac{1}{clock\ period}$ $C = Clock\ period = time\ spent\ on\ each\ cycle = \frac{1}{CR}$

$CPU\ time = \frac{I \times CPI}{CR} = I \times CPI \times C$ $CPI = Clock\ cycles\ per\ instruction$ $CPU\ performace = \frac{1}{CPU\ time}$

$\frac{Instruct}{sec} = \frac{1}{CPI}$

Units

$CR = \frac{cycle}{sec}$ $C = \frac{sec}{cycle}$ $CPI = \frac{cycle}{instruction}$ $CPU\ time = \frac{I \times cycle \times sec}{I \times cycle}$

1.1 (6 points)

Calculate the performance of each processor expressed in instructions per second. $giga = 10^9$

$P_1 IPS = CR \times \frac{1}{CPI} = \frac{1}{CPU\ time} = \frac{cycle \times instruct}{sec \times cycle}$ $P_1 perf = 3.2 \times \frac{1}{1.5} \times 10^9 = 2.13 \times 10^9 \frac{instructions}{sec}$

$P_2 IPS = 2 \times 1 \times 10^9 = 2 \times 10^9 \frac{instructions}{sec}$

$P_3 IPS = 4 \times \frac{1}{2.3} = 1.739 \times 10^9 \frac{instructions}{sec}$

1.2 (6 points)

If each of these processors executes a program in 10 seconds, calculate the number of cycles and the number of instructions.

# of instructions (from 1) multiply by cpi

$I_{P1} = 2.13 \times 10^9 \times 10 = 2.13 \times 10^{10}$ instructions

$Cycles_{P1} = CPI \times instructions = 1.5 \times 2.13 \times 10^{10} = 3.195 \times 10^{10} cycles$

$I_{P2} = 2 \times 10^9 \times 10 = 2 \times 10^{10}$ instructions

$Cycles_{P2} = CPI \times instructions = 1 \times 2 \times 10^{10} = 2 \times 10^{10} cycles$

$I_{P3} = 1.739 \times 10^9 \times 10 = 1.739 \times 10^{10}$ instructions

$Cycles_{P3} = CPI \times instructions = 2.3 \times 1.739 \times 10^{10} = 4 \times 10^{10} cycles$

1.3 (3 points)

We are trying to reduce the execution time of P2 by 30%, but this leads to an increase of 20% in the CPI. What should the clock rate be to obtain this reduction in execution time?

$CR_{new} = I_{P2} \times CPI_{p2} \times 1.2 \times (\frac{1}{0.7CPU_{P2}}) = \frac{(2\times10^{10}\times1\times1.2)}{(0.7\times10)} = 3.43\ GHz$

2 (10 points)

Consider two different implementations of the same instruction set architecture. The instructions can be divided into four classes according to their CPI (classes A, B, C, and D). P1 with a clock rate of 2.5 GHz and CPIs of 1, 2, 3, and 3 for the corresponding classes, and P2 with a clock rate of 3 GHz and CPIs of 2, 2, 2, and 2. Given a program with a dynamic instruction count of 1.0e6 instructions divided into classes as follows: 30% class A, 20% class B, 30% class C, and 20% class D:

2.1 (4 points)

What is the total CPI for each implementation?

$CPI_{P1}(\frac{clocks}{instruct}) = \frac{\Sigma(CPI*instructions)}{1\times10^6 instructions} = \frac{(1\times0.3\times10^6)+(2\times0.2\times10^6)+(3\times0.3\times10^6)+(3\times0.2\times10^6)}{1\times10^6} = 2.2CPI$

$CPI_{P2}(\frac{clocks}{instruct}) = \frac{\Sigma(CPI*instructions)}{1\times10^6 instructions} = \frac{(2\times0.3\times10^6)+(2\times0.2\times10^6)+(2\times0.3\times10^6)+(2\times0.2\times10^6)}{1\times10^6} = 2CPI$

2.2 (4 points)

Calculate the clock cycles required in both cases.

$CPU\ time_{P1} = I \times CPI \times \frac{1}{Clock\ Rate} = 10^6 \times 2.2 = 2.2 \times 10^6 clock\ cycles$

$CPU\ time_{P2} = I \times CPI \times \frac{1}{Clock\ Rate} = 10^6 \times 2 = 2 \times 10^6 clock\ cycles$

2.3 (2 points)

Which is faster: P1 or P2?

$CPU\ time_{P1} = I \times CPI \times \frac{1}{Clock\ Rate} = 10^6 \times 2.2 \times (\frac{1}{2.5\times10^9}) = 0.00088\ sec = 880\ microsec$

$$CPU\ time_{P2} = I \times CPI \times \frac{1}{Clock\ Rate} = 10^6 \times 2 \times (\frac{1}{3\times10^9}) = 0.000667\ sec = 667\ microsec$$

Processor 2 takes less time to execute the 1 million instruction program therefore it is faster.

3 (15 points)

In this exercise we look at memory locality properties of matrix computation. The following code is written in C, where *elements within the same row of a matrix are stored contiguously.* You can assume that undefined symbols are primitive types, *i.e.,* `int`, `double` *etc*.

```
for( int j = 1; j < n-1; j++) {
   for(int i = 1; i < m-1; i++) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] + A[j-1][i] + A[j+1][i]);
      err = max(err, fabs(Anew[j][i] - A[j][i]);
   }
}
```

```
A[j][i+1] + A[j][i-1] + A[j-1][i] + A[j+1][i] makes a cross
[- X -]
[X - X]
[- X -]
```

3.1 (5 points)

Which variables exhibit temporal locality?

Variables i, j and err and Anew[j][i] exhibit temporal locality. Anew[j][i] because it is referenced twice in succession during one loop iteration.

3.2 (5 points)

Which variables exhibit spatial locality? Locality is affected by both the reference order and data layout.

Variables Anew, A exhibit spatial locality.

3.3 (5 points) Would the program be slower or faster if the outer loop iterated over `i` and the inner loop iterated over `j`? Why?

The program would be slower if the outer loop iterated over I and the inner loop iterated over j because by having the inner loop iterate over j, both the Anew array and the A array would be accessing different rows firstly and a fixed column. In the C language, columns in the same row of a 2 dimensional array are stored contiguously whereas entries at the same column make a jump of M locations in address space. So, by iterating over the rows first, you would be making many jumps around in memory locations to read the source array locations in the arithmetic expression as well as jumping around in the A_{new} array locations to store the result.

2 d array is a long list with columns stored next to each other in the C programming language

4 (30 points)

Below is a list of 8-bit memory address references, given as **word** addresses. Answer the following questions based on **Section 4.2.3.1** of the textbook.

```
0x43, 0xc4, 0x2b, 0x42, 0xc5, 0x28,
0xbe, 0x05, 0x92, 0x2a, 0xba, 0xbd
```

$C = cache\ capatcity\ (bytes),\ S = sets,\ B = Bytes,\ E = lines \quad s, t, b = bits\ for\ set, tag, offset$
$C = S \times B \times E \quad E = 1 = direct - mapped\ cache \quad m = address$
$s = log_2 S \quad b = log_2 B \quad t = m - s - b$

$x - bit\ addresses \ -> \ can\ store\ 2^x\ bytes\ of\ data$

4.1 (15 points)

For each of these references, identify the binary word address, the tag, and the index given a direct-mapped cache with 16 one-word blocks. Also list whether each reference is a hit or a miss, assuming the cache is initially empty.

$word = 32\ bits \quad S = 16,\ B = 1,\ E = 1 \quad s = log_2 16 = 4,\ b = log_2 1 = 0, t = 8 - 4 - 0 = 4$

Word address → every 4 bytes has unique address. Each set holds 32 bits.

Which leaves 4 bits for t.

Address (hex)	Address (binary)	Tag	Index	Cache
0x43	0b01000011	0b0100	0b0011	miss
0xc4	0b11000100	0b1100	0b0100	miss
0x2b	0b00101011	0b0010	0b1011	miss
0x42	0b01000010	0b0100	0b0010	miss
0xc5	0b11000101	0b1100	0b0101	miss
0x28	0b00101000	0b0010	0b1000	miss
0xbe	0b10111110	0b1011	0b1110	miss
0x05	0b00000101	0b0000	0b0101	miss
0x92	0b10010010	0b1001	0b0010	miss
0x2a	0b00101010	0b0010	0b1010	miss
0xba	0b10111010	0b1011	0b1010	miss
0xbd	0b10111101	0b1011	0b1101	miss

4.2 (15 points)

For each of these references, identify the binary word address, the tag, the index, and the offset given a direct mapped cache with two-word blocks and a total size of eight blocks. Also list if each reference is a hit or a miss, assuming the cache is initially empty.

$m = 8\,bits,\ S = 8,\ s = log_2 8 = 3,\ B = 2,\ b = log_2 2 = 1,\ t = 8 - 3 - 1 = 4$

Hit when TAGS ARE EQUAL, INDICIES ARE EQUAL, OFFSETS 0 VS 1

Address (hex)	Address (binary)	Tag	Index	Offset	Cache
0x43	0b01000011	0b0100	0b001	0b1	miss
0xc4	0b11000100	0b1100	0b010	0b0	miss
0x2b	0b00101011	0b0010	0b101	0b1	miss
0x42	0b01000010	0b0100	0b001	0b0	**hit**
0xc5	0b11000101	0b1100	0b010	0b1	**hit**
0x28	0b00101000	0b0010	0b100	0b0	miss
0xbe	0b10111110	0b1011	0b111	0b0	miss
0x05	0b00000101	0b0000	0b010	0b1	miss
0x92	0b10010010	0b1001	0b001	0b0	miss
0x2a	0b00101010	0b0010	0b101	0b0	**hit**
0xba	0b10111010	0b1011	0b101	0b0	miss
0xbd	0b10111101	0b1011	0b110	0b1	miss

word[0]	Set index[1]	set
0x42/0x92	0x43	1
0xc4	0xc5/0x05	2
		3
0x28		4
0x2a/0xba	0x2b	5
	0xbd	6
0xbe		7
		8

5 (20 points)

Consider a **byte addressing** architecture with 64-bit memory addresses. (different here because word addresses before so offset not used but now we're byte addressing, need offset)

5.1 (5 points)

Which bits of the address would be used in the tag, index and offset in a direct-mapped cache with 512 1-word blocks?

$direct\ mapped = E = 1 \qquad word = 4\ bytes,\ m = 64 \qquad B = 4,\ b = log_2 4 = 2$

$S = 512,\ s = log_2 512 = 9 \qquad t = 64 - 2 - 9 = 53\ bits$

T-Bits (Tag)	S-Bits (Index)	B-Bits (Offset)
Bits 63-11	Bits 10-2	Bits 1-0

$C = 512 \times 1 \times 4 = 2028 = 2^{11}$

5.2 (5 points)

Which bits of the address would be used in the tag, index and offset in a direct-mapped cache with 64 8-word blocks?

$8 \times 4 = B = 32\ bytes,\ b = log_2 32 = 5 \qquad S = 64,\ s = log_2 64 = 6 \qquad t = 64 - 5 - 6 = 53\ bits$

T-Bits (Tag)	S-Bits (Index)	B-Bits (Offset)
Bits 63-11	Bits 10-5	Bits 4-0

$C = 64 \times 32 \times 1 = 2048 = 2^{10}$

5.3 (5 points)

What is the ratio of bits used for storing data to total bits stored in the cache in each of the above cases?

$\frac{bits\ for\ storing}{total\ bits\ stored}$

In 5.1:

$valid\ bit\ =\ 1,\ t = 53\ bits,\ 1\ word\ blocks = 4bytes = 32bits$

$S = 512 \qquad \#sets(valid\ bit\ +\ tag\ bit\ +\ data\ stored\ in\ blocks)\colon \#sets(data\ stored\ in\ blocks)$

$\frac{512(1+53+32)}{512(32)} = \frac{44032}{16384} = 2.6875$

In 5.2:

$valid\ bit\ = 1,\ t = 53\ bits,\ 8\ word\ blocks\ = 8 \times 4 \times 8 = 256,\ S = 64$

$\frac{64(1+53+256)}{64(256)} = 1.209$

5.4 (5 points)

Which bits of the address would be used in the tag, index and offset in a two-way set associative cache with 1-word blocks and a total capacity of 512 words.

$E = 2,\ B = 4, b = log_2 4 = 2,\ C = 512 * 4 = 2048\ bytes$

$C = E \times B \times S = 2 \times 4 \times S \qquad S = 64,\ s = log_2 64 = 8$

So, the least two significant bits would be used for the offset. The next significant 8 bits would be used for the index. That leaves the 54 most significant bits of the address for the tag bits.

6 (10 points)

Given a byte-addressed memory wit-bit addresses, and an attached four-way set associative cache with a total of 16 one-word blocks, make a table showing the final state of the cache after accessing the following memory addresses. The table should include the tags and data that will be stored in cache at the end of all insertions. Data should be shown using the corresponding addresses in main memory. Use the least-recently-used rule to evict from cache as needed.

The sequence of addresses accessed is:

`0x143, 0xc4a, 0x22b, 0x42f, 0x492, 0x2a2, 0x3ba, 0xb2d`

$m = 12,\ word = 4\ bytes,\ E = 4,\ C = \ 16 \times 4 = 64,\ B = 4, b = log_2 4 = 2,\ S = \frac{C}{B \times E} = 4,\ s = log_2 4 = 2$

$t = 12 - 2 - 2 = 8\ bits$

Address (hex)	Address (binary)	Tag	Index	Offset
~~0x143~~	0b000101000011	0x14	0b00	0b11
~~0xc4a~~	0b110001001010	0xc4	0b10	0b10
~~0x22b~~	0b001000101011	0x22	0b10	0b11
~~0x42f~~	0b010000101111	0x42	0b11	0b11
~~0x492~~	0b010010010010	0x49	0b00	0b10
~~0x2a2~~	0b001010100010	0x2a	0b00	0b10
~~0x3ba~~	0b001110111010	0x3b	0b10	0b10
0xb2d	0b101100101101	0xb2	0b11	0b01

	Bytes						Bytes						Bytes						Bytes			
V	0	1	2	3	Tag	V	0	1	2	3	Tag	V	0	1	2	3	Tag	V	0	1	2	3
1	M[0x140]	M[0x141]	M[0x142]	M[0x143]																		
1																						
1																						

	Bytes						Bytes						Bytes						Bytes			
V	0	1	2	3	Tag	V	0	1	2	3	Tag	V	0	1	2	3	Tag	V	0	1	2	3
1	M[0x140]	M[0x141]	M[0x142]	M[0x143]																		
1	M[0xc48]	M[0xc49]	M[0xc4a]	M[0xc4b]																		

	Bytes						Bytes						Bytes						Bytes			
V	0	1	2	3	Tag	V	0	1	2	3	Tag	V	0	1	2	3	Tag	V	0	1	2	3
1	M[0x140]	M[0x141]	M[0x142]	M[0x143]																		
1	M[0xc48]	M[0xc49]	M[0xc4a]	M[0xc4b]	0x22	1	M[0x228]	M[0x229]	M[0x22a]	M[0x22b]												

	Bytes						Bytes						Bytes						Bytes			
V	0	1	2	3	Tag	V	0	1	2	3	Tag	V	0	1	2	3	Tag	V	0	1	2	3
1	M[0x140]	M[0x141]	M[0x142]	M[0x143]																		
1	M[0xc48]	M[0xc49]	M[0xc4a]	M[0xc4b]	0x22	1	M[0x228]	M[0x229]	M[0x22a]	M[0x22b]												
1	M[0x42c]	M[0x42d]	M[0x42e]	M[0x42f]																		

	Bytes						Bytes						Bytes						Bytes			
V	0	1	2	3	Tag	V	0	1	2	3	Tag	V	0	1	2	3	Tag	V	0	1	2	3
1	M[0x140]	M[0x141]	M[0x142]	M[0x143]	0x49	1	M[0x490]	M[0x491]	M[0x492]	M[0x493]												
1	M[0xc48]	M[0xc49]	M[0xc4a]	M[0xc4b]	0x22	1	M[0x228]	M[0x229]	M[0x22a]	M[0x22b]												
1	M[0x42c]	M[0x42d]	M[0x42e]	M[0x42f]																		

	Bytes						Bytes						Bytes						Bytes			
V	0	1	2	3	Tag	V	0	1	2	3	Tag	V	0	1	2	3	Tag	V	0	1	2	3
1	M[0x140]	M[0x141]	M[0x142]	M[0x143]	0x49	1	M[0x490]	M[0x491]	M[0x492]	M[0x493]	0x2a	1	M[0x2a0]	M[0x2a1]	M[0x2a2]	M[0x2a3]						
1	M[0xc48]	M[0xc49]	M[0xc4a]	M[0xc4b]	0x22	1	M[0x228]	M[0x229]	M[0x22a]	M[0x22b]												
1	M[0x42c]	M[0x42d]	M[0x42e]	M[0x42f]																		

	Bytes						Bytes						Bytes						Bytes			
V	0	1	2	3	Tag	V	0	1	2	3	Tag	V	0	1	2	3	Tag	V	0	1	2	3
1	M[0x140]	M[0x141]	M[0x142]	M[0x143]	0x49	1	M[0x490]	M[0x491]	M[0x492]	M[0x493]	0x2a	1	M[0x2a0]	M[0x2a1]	M[0x2a2]	M[0x2a3]						
1	M[0xc48]	M[0xc49]	M[0xc4a]	M[0xc4b]	0x22	1	M[0x228]	M[0x229]	M[0x22a]	M[0x22b]	0x3b	1	M[0x3b8]	M[0x3b9]	M[0x3ba]	M[0x3bb]						
1	M[0x42c]	M[0x42d]	M[0x42e]	M[0x42f]																		

	Bytes						Bytes						Bytes						Bytes			
V	0	1	2	3	Tag	V	0	1	2	3	Tag	V	0	1	2	3	Tag	V	0	1	2	3
1	M[0x140]	M[0x141]	M[0x142]	M[0x143]	0x49	1	M[0x490]	M[0x491]	M[0x492]	M[0x493]	0x2a	1	M[0x2a0]	M[0x2a1]	M[0x2a2]	M[0x2a3]						
1	M[0xc48]	M[0xc49]	M[0xc4a]	M[0xc4b]	0x22	1	M[0x228]	M[0x229]	M[0x22a]	M[0x22b]	0x3b	1	M[0x3b8]	M[0x3b9]	M[0x3ba]	M[0x3bb]						
1	M[0x42c]	M[0x42d]	M[0x42e]	M[0x42f]	0xb2	1	M[0xb2c]	M[0xb2d]	M[0xb2e]	M[0xb2f]												