Spring 2022 *Name*: Ivan Zelenkov, ID: 2600950
CSCI 3301 Homework # 2

Due: **Monday, May 3, 2022 (11.59 PM), via Moodle.**

The rules:

- All work must be your own. You are not to work in teams on this assignment.
- Submit as a single file (via Moodle) containing a PDF file. Email me (ayn@cs.uno.edu) assignment only if Moodle is not working.
- You may use the textbook and lecture notes, but do NOT search the Internet for solutions.
- If you use the textbook or slides, do NOT copy paste. Write the answer by your own sentence.

Total Marks: 100

Q1-) (21pt) Explain following questions SHORTLY.

a-) (7pt) Briefly discuss performance issues of non-pipeline processors. Why do we need pipeline processors?

Non-pipelined processors have performance issues compared to pipelined ones. A non-pipelined processor executes one instruction at a time, it is not feasible to vary the period for different instructions and overall execution takes more time which increases the total time of clock cycles. A pipelined processor does not have to spend time during the stall because many instructions can be executed at the same time.

b-) (7pt) Explain memory hierarchy. (Chapter 5 - Slide 3)

The central idea of a memory hierarchy is that for each *L*, the faster and smaller storage device at level *L* serves as a cache for the larger and slower storage device at level *L+1*. In general, the highest level *L0* are a small number of fast CPU registers that the CPU can access in a single clock cycle. Next are smaller moderate-size SRAM-based cache machine memories that can be accessed in a few CPU clock cycles. These are followed by a large DRAM-based main memory that can be accessed in tens of hundreds of clock cycles. Next are slow but enormous local disks. Finally, some systems even include an additional level of disks on remote servers that can be accessed over a network, for example, AFS (Andrew File System) or NFS (Network File System).

c-) (7pt) Explain replacement policies on Cache.

Replacement policies are algorithms that hardware can utilize to manage data in the cache on a computer. We have direct mapped, set associative, fully associative, LRU, Random algorithms. *Direct mapped* is where a block can be placed in exactly one location. A *set-associative* cache has a fixed number of locations where each block can be placed. A *fully associative* is where a block in memory can be associated with any entry in the cache. *LRU (least recently used)* is the block that is replaced by the block that has not been used for the longest time. A *random* algorithm finds a block in the cache randomly.

Q2-) (18pt) Show if the following codes require stalls. Is it possible to avoid stalls by rearranging the instructions? If yes, show the rearranged code and explain it. Assume that forwarding is allowed.
Note 1: Show all the steps. Show initial pipeline stages and final pipeline stages if you rearrange the code. DO NOT just give the rearranged code.
Note 2: If the code doesn't require any stalls, you need to show that as well.

a-)
lw $t0, 4($s0)
add $t0, $t0, $t1
sub $t0, $t0, $t2
lw $t1, 4($s0)

a) Initial code's pipeline stage

lw $t0, 4($s0)	IF	ID	EX	MEM	WB				
add $t0, $t0, $t1	X	IF	X	ID	EX	MEM	WB		
sub $t0, $t0, $t2	X	X	X	IF	ID	EX	MEM	WB	
lw $t1, 4($s0)	X	X	X	X	IF	ID	EX	MEM	WB

We have only one stall in add instruction. There is no possible improved version because for example subtraction between lw and add would violate the entire logic of add instruction. sub would overwrite $t0 which will cause a different result in add instruction. Also, we cannot put the last lw between the first lw and add because that would overwrite register $t1 which is used in add instruction.

b-)
sw $t0, 4($s0)
lw $t0, 8($s0)
add $t1, $t0, $t3
lw $t2, 4($s2)
sw $t2, 8($s2)

b) Initial code's pipeline stage											
sw $t0, 4($s0)	IF	ID	EX	MEM	WB						
lw $t0, 8($s0)	X	IF	ID	EX	MEM	WB					
add $t1, $t0, $t3	X	X	IF	X	ID	EX	MEM	WB			
lw $t2, 4($s2)	X	X	X	X	IF	ID	EX	MEM	WB		
sw $t2, 8($s2)	X	X	X	X	X	IF	X	ID	EX	MEM	WB
Improved version											
sw $t0, 4($s0)	IF	ID	EX	MEM	WB						
lw $t0, 8($s0)	X	IF	ID	EX	MEM	WB					
lw $t2, 4($s2)	X	X	IF	ID	EX	MEM	WB				
add $t1, $t0, $t3	X	X	X	IF	ID	EX	MEM	WB			
sw $t2, 8($s2)	X	X	X	X	IF	ID	EX	MEM	WB		

We have two stalls in the initial pipeline stage, and it is improved by moving the second lw between the first lw and add. The result is that there are no stalls.

Q3-) (20pt)
Assume that your hardware uses branch prediction which always assumes that branch not taken. Show the pipeline of the following code. Forwarding is allowed.
Note 1: Show both cases assuming the values in the registers $t0 == $t1 and $t0 != $t1.
Note 2: Use 5 stages for each instruction

```
            add $t0, $t1, $t2
            lw $t0, 0($s0)
            beq $t0, $t1, Label
            sll $t1, $t6, 2
            j Exit
Label:      lw $t0, 4($s0)
            srl $t1, $t0, 4
Exit:
```

1) $t0 == $t1													
add $t0, $t1, $t2	IF	ID	EX	MEM	WB								
lw $t0, 0($s0)	X	IF	ID	EX	MEM	WB							
beq $t0, $t1, Label	X	X	IF	X	ID	EX	MEM	WB					
sll $t1, $t6, 2	X	X	X	X	IF	ID	X	X					
j Exit	X	X	X	X	X	IF	X	X					
Label: lw $t0, 4($s0)	X	X	X	X	X	X	IF	ID	EX	MEM	WB		
srl $t1, $t0, 4	X	X	X	X	X	X	X	IF	X	ID	EX	MEM	WB
Exit:													
2) $t0 != $t1													
add $t0, $t1, $t2	IF	ID	EX	MEM	WB								
lw $t0, 0($s0)	X	IF	ID	EX	MEM	WB							
beq $t0, $t1, Label	X	X	IF	X	ID	EX	MEM	WB					
sll $t1, $t6, 2	X	X	X	X	IF	ID	EX	MEM	WB				
j Exit	X	X	X	X	X	IF	ID	EX	MEM	WB			
Label: lw $t0, 4($s0)													
srl $t1, $t0, 4													
Exit:													

Q4-) (20pt)

a-) (10pt) **(ENCODING)** Calculate the parity of a byte with the value 0x7C.

Note: Remember that the question says, "byte". If the value is less than 8-bits, insert 0 in the beginning and make the value 8-bits. e.g., 101101 => 00101101

0x7C = 01111100

We want to store 8-bit 01111100, therefore we need a 12-bit hamming code to store 8 bits of data.

p1 p2 0 p3 1 1 1 p4 1 1 0 0

For p1 check bits: 1 3 5 7 9 11 → p1-0-1-1-1-0.

We have an odd number of 1s (3). Therefore, p1 = 1 which gives us

1 p2 0 p3 1 1 1 p4 1 1 0 0

For p2 check bits: 2 3 6 7 10 11 → -p2 0--11--10-.

We have an odd number of 1s (3). Therefore, p2 = 1 which gives us

1 1 0 p3 1 1 1 p4 1 1 0 0

For p3 check bits: 4 5 6 7 12 → ---p3 111----0.

We have an odd number of 1s (3). Therefore, p3 = 1 which gives us

1 1 0 1 1 1 1 p4 1 1 0 0

For p4 check bits: 8 9 10 11 12 → -------p4 1100
We have an even number of 1s (2). Therefore, p4 = 0 which gives us
1 1 0 1 1 1 1 0 1 1 0 0

Result: 1 1 0 1 1 1 1 0 1 1 0 0 = 0xDEC.

b-) (10pt) (DECODING) Consider a SEC code that protects 8-bit words with 4 parity bits. If we read the value 0x375, is there an error? If so, correct the error.

0x375 = 001101110101

p1 p2 d1 **p3** d2 d3 d4 **p4** d5 d6 d7 d8
0 0 1 1 0 1 1 1 0 1 0 1

For p1 check bits: 1 3 5 7 9 11 → 0-1-0-1-0-0.
- We have an even number of 1s (2) → ***NO ERROR***

For p2 check bits: 2 3 6 7 10 11 → -01--11--10-.
- We have an even number of 1s (4) → ***NO ERROR***

For p3 check bits: 4 5 6 7 12 → ---1011----1.
- We have an even number of 1s (4) → ***NO ERROR***

For p4 check bits: 8 9 10 11 12 → -------10101
- We have an odd number of 1s (3) → ***ERROR***

p4 has an odd parity. Therefore, we must correct bit in $2^3 = 8th$ position.

0 0 1 1 0 1 1 **1** 0 1 0 1 → 0 0 1 1 0 1 1 **0** 0 1 0 1 → *0x365*.

Result: 0 0 1 1 0 1 1 0 0 1 0 1 = 0x365.

Q5-) (21pt)
Assume that you have a 4-blocks, 1 word/block, direct mapped cache. Show step by step cache snapshot for the given binary address of references below.
Hint: Chapter 5 – Slide28 - 34
Hint: Use mod 4 as your hash function.

10110
11011
10110
10010

$Cache\ size = 2^x = 2^2 = 4, where\ x = 2. It\ means\ that\ the\ last\ 2\ bits\ will\ be\ an\ index.$

1) Binary address 101 10, miss, cache block 10

Index	V	Tag	Data
00	N		
01	N		
10	**Y**	**101**	**Mem[10110]**
11	N		

2) Binary address 110 11, miss, cache block 11

Index	V	Tag	Data
00	N		
01	N		
10	**Y**	**101**	**Mem[10110]**
11	**Y**	**110**	**Mem[11011]**

3) Binary address 101 10, hit, cache block 10

Index	V	Tag	Data
00	N		
01	N		
10	**Y**	**101**	**Mem[10110]**
11	**Y**	**110**	**Mem[11011]**

4) Binary address 100 10, miss, cache block 10

Index	V	Tag	Data
00	N		
01	N		
10	**Y**	**100**	**Mem[10010]**
11	**Y**	**110**	**Mem[11011]**