National University of Computer and Emerging Sciences, Lahore Campus

Course:	COAL	Course Code:	EE2003
Program:	BS(CS,DS)	Semester:	Fall 2022
Duration:	3 Hours	Total Marks:	90
Paper Date:	16-12-2022	Page(s):	11
Section:	All	Roll No.	
Exam:	Final	Your Section:	

Instruction/Notes: **This is an open notes/book exam. Sharing notes and calculators is NOT ALLOWED.** All the answers should be written in provided space on this paper. Rough sheets can be used but will not be collected and checked. In case of any ambiguity, make reasonable assumptions. Questions during exams are not allowed.

Question 1 [Pipelining] [CLO 6] [10 Marks]:

Important Instruction: Following question is ONLY for section BSCS-A, BSCS-B, BSCS-F and BSCS-G

For the code segment given below, fill-in the following pipeline diagram. Clearly show the stall AND/OR forwarding where required. In case of forwarding, clearly draw the arrow and mention the name of operand that needs forwarding. Assume you have optimized pipelined MIPS Architecture (with all the hazards control implementation) as we have studied in class.

```
sub $1, $2, $3
or $4, $5, $6
lw $6, 100 ($4)
and $7, $6, $8
```


Important Instruction: Following question is for ALL THE SECTIONS EXCEPT section BSCS-A, BSCS-B, BSCS-F and BSCS-G

Let us consider the following decomposition of the instruction processing

Fetch Instruction (FI): Read the next expected instruction into a buffer.

Decode Instruction (DI): Determine the opcode and the operand specifiers.

Fetch Operands (FO): Calculate the effective address of each source operand and fetch each operand from the memory. Operand in registers need not to be fetched.

Execute Instruction (EI): Perform the indicated operation and store the result if any, in the specified destination operand location.

Write Operand (WO): Store the result in memory.

Following is a set of instructions. Their implementation through pipelining has some data hazards. You have to solve those hazards by using stalling method.

Set of instructions

I1: mov bx, 0
I2: mov word [n1], ax
I3: add word [n1], bx
I4: add word [n1], 01
I5: mov cx, 0
I6: mov word [n2], cx
I7: add bx, [n1]
I8: add bx, word[n2]

Do it with Stalling Method (without Data Forwarding)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
I1	FI	DI	FO	EI	WO															
I2		FI	DI	FO	EI	WO														
I3			FI	DI	ST	ST	FO	EI	WO											
I4				FI	DI	ST	ST	ST	ST	FO	EI	WO								
I5					FI	DI	ST	FO	EI	WO										
I6						FI	DI	ST	ST	ST	FO	EI	WO							
I7							FI	DI	ST	ST	ST	ST	FO	EI	WO					
I8								FI	DI	ST	ST	ST	ST	ST	ST	FO	EI	WO		

Question 2 [Cache] [CLO 6] [5+5 = 10 Marks]:

Consider a sequence of memory address references given below. In the sequence, each word address is provided in both the decimal and binary formats. Below each address, the relative time at which these references occur is also listed. Memory contents and addresses are shown in the second table.

Memory		
Decimal Address	**Binary Address**	**Data**
0	00 00 00 00	2
120	01 11 10 00	100
248	11 11 10 00	7
170	10 10 10 10	50
187	10 11 10 11	52
51	00 11 00 11	80
15	00 00 11 11	41
174	10 10 11 10	32
150	10 01 01 10	77
9	00 00 10 01	5
4	00 00 01 00	9
253	11 11 11 01	2
1	00 00 00 01	3
7	00 00 01 11	65
6	00 00 0 1 10	90
2	00 00 00 10	55

Memory Access Sequence		
Time	**Address Decimal**	**Address Binary**
1	6	00 00 0 1 10
2	0	00 00 00 00
3	15	00 00 11 11
4	120	01 11 10 00
5	253	11 11 11 01
6	1	00 00 00 01
7	248	11 11 10 00
8	9	00 00 10 01
9	4	00 00 01 00
10	51	00 11 00 11
11	2	00 00 00 10
12	1	00 00 00 01

Now consider two different 8-word caches shown below. Assume that each of the caches was used independently to facilitate memory access for the sequence above. For each cache type, assume that the cache is initially empty. Assume that the least-recently used (LRU) scheme is used where appropriate. Also, when inserting an element into the cache, if there are multiple empty slots for one index, you should insert the new element into the left-most slot (first available slot)

Part (A) [5 Marks]: Use the **direct-mapped cache** to facilitate memory access for the memory sequence above. You should fill in the binary form of the Tag values. **Show the final contents of the cache in the table below.**

Note: V means Valid OR Value Bit.

Index	Cache		
	TAG	DATA	VALUE BIT
0	11111	7	0
1	00000	3	0
2	00000	55	0
3	00110	80	0
4	00000	9	0
5	11111	2	0
6	00000	90	0
7	00001	41	0

Hit Rate: _______0__________________

Miss Rate: ________1________________

Part (B) [5 Marks]: Use the **2-way set associative cache** to facilitate memory access for the memory sequence above. You should fill in the binary form of the Tag values. **Show the final contents of the cache in the table below.**

Index	2-way set associative Cache					
	TAG	DATA	VALUE BIT	TAG	Data	Value Bit
0	111110	7	0	000001	9	0
1	000010	5	0	000000	3	0
2	000001	90	0	000000	55	0
3	000011	41	0	001100	80	0

Hit Rate: _______1/12___________________

Miss Rate: ________11/12________________

Question 3 [Performance] [CLO 6] [10 Marks]: It takes 15µs to complete one instruction in a non-pipelined processor. We were able to convert the circuit to a 6 stage pipeline processor. Stage 1 to 6 take 2µs, 1.5µs, 3µs, 4µs, 1.5µs, 3µs resp. Time to move from one pipe stage to another is 2µs. (**Note for Section BSCS-A, BSCS-B, BSCS-F and BSCS-G:** Assume the transition time, to move from one pipe stage to another, is zero.)

Calculate the following values for pipeline and non-pipelined processor (Write the answer inthe given table)

Value	Non-Pipeline	Pipeline
Clock Cycle	15µs	6µs
Clock Speed	1/15 x 10^{-6} Hz = 0.067 x 10^{6} Hz	1/6 x 10^{-6} Hz = 0.17 x 10^{6} Hz
Latency	15µs	36µs
Throughput for 46 instructions	46/46x15 x 10^{-6} instructions/s = 0.067 x 10^{6}	46/51 x 6 x 10^{-6}

		Instructions/s = 0.15 x 10^6
Throughput for 1 instruction	1/15 x 10^{-6} instructions/s 0.067 x 10^6	1/36 x 10^{-6} instructions/s 0.03 x10^6
Speedup of pipeline processor for 1 instruction	15/36 = 0.42	
Speedup of pipeline processor for 75 instructions	15x 75/80 x 6 = 2.34	

Question 4 [Short Questions] [CLO 1,2,3,4,5] [5x6 = 30 Marks]:

a. The following program is trying to add the first three numbers in the array num1 and store the sum in the fourth index of the num1 array. However, after running the program, the final sum generated is incorrect. **Identify mistakes in the program and write the correct code in the box on the right side.**

```
 ; a program to add three numbers
[org 0x0100]
mov ax, [num1]
mov bx, [num1+1]
add ax, bx
mov bx, [num1+2]
add ax, bx
mov [num1+3], ax
mov ax, 0x4c00
int 0x21
num1: dw 5, 10, 15, 0
```

```
; Write Correct Code here

[org 0x0100]
mov ax, [num1]
mov bx, [num1+2]
add ax, bx
mov bx, [num1+4]
add ax, bx
mov [num1+6], ax
mov ax, 0x4c00
int 0x21
num1: dw 5, 10, 15, 0
```

b. **Write a piece of code to check if a number ‘num’ is a power of two or not. If the number is power of two, set the PowerOfTwo flag to 1. You are only allowed to use shifting and logical instructions. You are not allowed to use DIV instruction.**

```
[org 0x0100]
jmp start
PowerOfTwo :db 0
num: dw 0

start:
mov ax, [num]
push ax
call CheckPowerOfTwo
terminate:
mov ax,0x4c00
int 0x21
```

```
CheckPowerOfTwo:
push bp
mov bp,sp
push ax
push bx
mov ax,[bp+4]
;write your code here

mov bx,ax
sub bx,1
AND ax,bx
jnz l2

l1:
mov byte[PowerOfTwo],1






l2:
pop bx
pop ax
pop bp
ret 2
```

c. Following program has a function add1 that takes 2 numbers from stack and if sum of these two numbers is greater than 0, it returns their sum through stack otherwise it returns 0 through stack. Code has some logical errors. **Highlight the errors and correct those errors so that you can pop the correct answer in the dx register. You can add or modify existing lines but you cannot remove any line.**

```
jmp start
add1:
push bp
mov bp, sp
sub sp,2
push ax
mov ax, [bp+2]
mov [bp-2], ax
```

```
; Write only updated or new lines. Don’t re-write full code.
add1:
jmp start
push bp
mov bp, sp
sub sp,2
push ax
```

mov ax, [bp+4] add [bp+8],ax cmp word [bp+8],0 ja end mov [bp+8], 0 end: pop bp **ret 2** **start:** sub sp, 2 push 8 push 5 **call add1** pop dx	mov ax, [bp+4] mov [bp+8], ax mov ax, [bp+6] add [bp+8],ax cmp word [bp+8],0 jg end mov [bp+8], 0 end: pop ax mov sp, bp pop bp ret 4 start: sub sp, 2 push 8 push 5 call add1 pop dx

d. Answer the following questions:

(i) Which interrupt will be hooked after execution of following code?

____**44h**________________

```
[org 0x100]
;;;; myISR is written here
xor ax,ax
mov es, ax
mov [es: 0x110 ] , myISR
mov [es : 0x112] , CS
mov ax, 4c00h
Int 21h
```

(ii) What is the total size (in bytes) of the interrupt vector table?

______**1024 bytes**________________

The first 22 words (in hex) of the 0th segment of the physical memory are shown in the following table (starting from data 0120).

0120
0140
0280
123F
124A
A198
BCD6
78D2
197B
CD79
E106
56AB
9851
0CDA
6502
AB69

(iii) What is the segment and offset of the interrupt service routine corresponding to interrupt 1?

Offset: _____**0x8002**______________ **Segment:** ___**0x3F12**_______________

(iv) What is the segment and offset of the interrupt service routine

corresponding to interrupt 5?

Offset: _0x06E1_________________ Segment: __0xAB56_______________

(v) What is the segment and offset of the interrupt service routine corresponding to interrupt 10h?

Offset: _____not shown_____________ Segment: __not shown______________

F156
49D8
12E5
9857
146B
98A2

e. Write a piece of code that disables the timer interrupt in the PIC mask register.

```
[org 0x0100]
in al, 0x21
or al, 1
out 0x21, al
mov ax, 0x4c00
int 0x21
```

Question 5 **[CLO 1,2,3,4,5] [30 Marks]**: You are required to implement **Notepad** Application according to the functionality as described below:

1- Notepad application will start/load with following specifications:

a. Notepad will have two partitions in the display memory. Upper half (1st 12 rows) will be editor window while 2nd half (last 12 rows) will be read-only window. 13th row will be boundary line like “========”. Both the editor and read-only windows will be space with white background initially. **[4 marks]**

b. A blinking cursor ‘|’ (attribute: black on white background) will be on top-left cell of the Editor Window (1st half). For now, ignore the default cursor due to time constraint. **[4 marks]**

2- Editor window can be edited with following specifications (DO NOT take characters from user using software interrupts)

a. If user enters any key, that character or digit will be displayed at the position of cursor and cursor will move by one cell towards right. If cursor was at last cell of a row, it will move to 1st cell of next row within editor boundary. Due to time constraint, we assume that user will not write at bottom-right cell of Editor Window and beyond. You do not need to check this boundary condition. Assume that user will press only characters and numeric keys. Also assume that you are already given a function ScanCodeToAsciiConverter that reads scancode from AL and saves corresponding ascii in AH (You do not need to push or pop any parameter or return value and you do not need to re-write this converter, just use it wisely). **[6 marks]**

b. Make sure that your application doesn’t write anything on Editor Window on key release. **[2 marks]**

c. Within Editor Window, user can move the cursor up, down, left or right by pressing Up, Down, Left, Right ARROW KEYS respectively. Due to time constraint we will implement only DOWN ARROW KEY (i.e. scancode 0x50). If user presses DOWN ARROW key, the cursor moves to same column of next row. Assume that user will not cross the Editor Window’s boundary; you do not need to implement this boundary check. Make sure you properly handle previous position of cursor. This cursor movement is allowed even if there is no text written in editor window. [6 marks]

3- After every minute, Read-only window updates/refreshes itself with the latest content available on Editor Window i.e. after every minute, paste the content of Editor Window on Read-only Window (You do not need to remove cursor from read-only window). [8 marks]

Important Instructions:

- Credit will be given on code efficiency, so use string instructions where required.
- You may use the functions given in book examples. Give proper reference and use them wisely. Function calls should exactly support the required functionality.

```
; Write your code here
; Data Declarations (if required)
; and Start/Main Functionality here
[org 0x100]
jmp start

offset: dw 0
count: dw 0

start:
call clrscr
call mkBoundary
mov ax, 0xB800
mov es, ax
mov di, 0
mov ah, 0xF0
mov al, ‘|’
mov word [es:di], ax
xor ax, ax
mov es, ax ; point es to IVT base
cli ; disable interrupts
mov word [es:9*4], kbisr ; store offset at n*4
mov [es:9*4+2], cs ; store segment at n*4+2
mov word [es:8*4], timer ; store offset at n*4
mov [es:8*4+2], cs ; store segment at n*4+
sti
again: jmp again


clrscr: push es
push ax
push cx
push di
mov ax, 0xb800
mov es, ax ; point es to video base
xor di, di ; point di to top left column
mov ax, 0x7020 ; space char in black on white
mov cx, 2000 ; number of screen locations
cld ; auto increment mode
rep stosw ; clear the whole screen
```

```
;Write Timer Code here (if required)
mkBoundary :
push es
push ax
push bx
push cx
push di

mov ax, 0xB800
mov es, ax
mov ax, 80
mov bx, 12
mul bx
shl ax, 1
mov di, ax
mov cx, 80
mov ah, 0x70
mov al, ‘=’ ; black = character on white
cld
rep stosw
pop di
pop cx
pop bx
pop ax
pop es
ret

kbisr:

push es
push ax
push di
mov ax, 0xB800
mov es, ax

in al, 0x60
cmp al, 0x50  ; down arrow is pressed
jne next
add word [cs:offset], 160
mov di, [cs:offset]
mov ah, 0xF0
```

```
pop di
pop cx
pop ax
pop es
ret


timer:
push ax
add word [cs:count], 1
cmp word [cs:count], 1080
jne end
mov word [cs:count], 0
call copyToBottom
end:
pop ax
mov al, 0x20
out 0x20, al
iret


copyTo Bottom:

push es
push ax
push bx
push ds
push si
push di
push cx

mov ax, 0xB800
mov es, ax
push es
pop ds
mov si, 0
mov ax, 80
mov bx, 13
mul bx
shl ax, 1
mov di, ax
mov cx, 960
cld
rep movsw

pop cx
pop di
pop si
pop ds
pop bx
pop ax
pop es

ret
```

```
mov al, ‘|’
mov  [es:di],  ax// blinking black | on white
jmp end

next:
test al, 0x80
jnz end

call ScanCodeToAsciiConverter
mov al, ah
mov ah, 0x70
mov di, [cs:offset]
mov [es:di], ax
add di, 2
add word [cs:offset], 2
mov ah, 0xF0
mov al, ‘|’
mov [es:di], ax

end:
pop di
pop ax
pop es
mov al, 0x20
out 0x20, al
iret
```