NAME: _____

(50 marks total)

This is an individual assignment. You may collaborate with others from your class, but your answers must demonstrate significant individual effort differentiating it from others' work.

1. The **MIC-2** is about to execute an IJVM "**ILOAD**" instruction. Memory and the registers contain the following information (all addresses and data values are shown in hexadecimal, and although they are 32 bits long, the high order 16 bits are all zeros and are not shown):

Stack area (words)

5E63	**5032**
5E62	**4F34**
5E61	**000C**
5E60	**007F**
5E5F	**3033**
5E5E	**C078**
5E5D	**0551**
5E5C	**0B81**
5E5B	**7A42**
5E5A	**0005**
5E59	**0000**
5E58	**002C**

Register values

LV	**5E51**
PC	**1805**
CPP	**2F00**
SP	**5E61**

The JVM bytecodes at the current PC location in the method area look like this:

15	**09**	**59**	**60**	**36**	**0C**	**00**	**00**	**A7**	**FF**	**00**
1805	1806	1807	1808	1809	180A	180B	180C	180D	180E	180F

a) On the reverse side of this page, **trace the execution of the IJVM program** through the ILOAD and following instructions, up to and including **the first NOP (hex "00") instruction at address 180B**. **Show the hexadecimal contents of the MIC-2 registers after each microinstruction is executed.** ***Note*** - you may not need to use all of the lines provided. (5 marks)

b) **How many clock cycles** does it take to execute the complete sequence of IJVM instructions from ILOAD through to and including NOP? (1 mark)

c) **What** is the **value** of **local variable 8**? (1 mark)

d) **How big is the local variable frame** for the current procedure? (1 mark)

e) **What will the PC register contain** after the instruction at address **180D** is executed? (2 marks)

Show the **microinstructions** that are executed in each clock cycle in the gray space below, and show the **hexadecimal** contents of the MIC-2 registers at the **end** of each clock cycle after the microinstruction is executed. The last micro-instruction for the previous IJVM instruction (not shown on previous page) is shown so you can see how the system got to "ILOAD1". You only need to show register values that have **changed** since for each microinstruction:

	PC	MBR1U	MBR2U	SP	TOS	H	MAR	MDR
	C43B	15	1510	3016	5032	0000	0000	0000
(prev) ......goto (MBR1)								
	C43C	10	1010	3016	5032	0000	0000	0000
iload1 MAR = LV + MBR1U; rd								

2. **Write MIC-2 microcode** to implement the following (hypothetical) IJVM instructions:

   a) **IADD1**
   Interger Add 1 - Replaces the value on the top of the stack with a value one greater (ie, if the word at the top of the stack contained the value "6", this instruction would change it to "7"). (3 marks)

   b) **INEG**
   Integer Negate – reverses the sign of the value on the top of the stack. For example, if the value at the top of the stack was "8", this instruction would replace it with the value "-8". (3 marks)

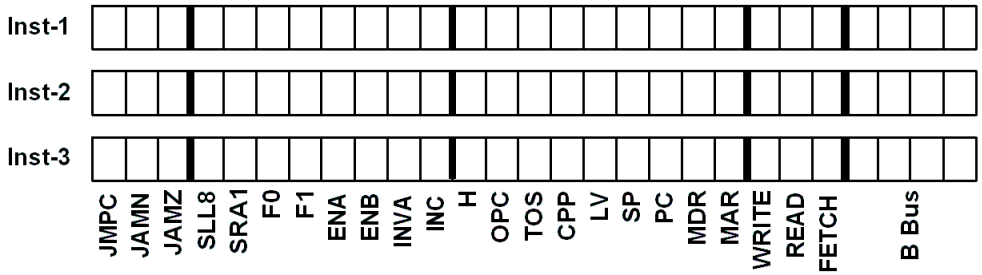
   c) **SGOTO**
   Short goto – performs the same operation as a normal GOTO instruction, but uses an 8-bit signed offset instead of a 16-bit one. (4 marks)

3. **Convert the following MAL instructions into MIC-1 microinstructions** (1 and 0 bits), ignoring the "Next Address" portion of the microinstruction word. (3 marks)

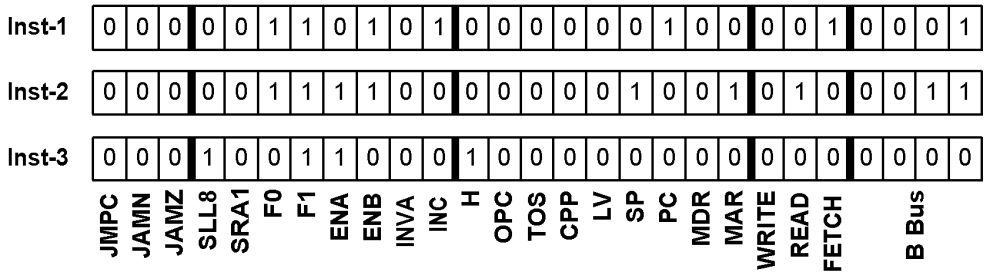
Inst-1 **MAR = SP = SP + 1; rd**

Inst-2 **MDR = TOS = MBRU; wr; goto Main1**

Inst-3 **Z = OPC – H; if (Z) goto T; else goto F**



4. **Rewrite the following MIC-1 microinstructions using MAL notation:** (3 marks)



Inst-1

Inst-2

Inst-3

5. The following MIC-1 microinstructions implement an IJVM instruction that's not described in the textbook:

Inst-1 **MAR = SP = SP – 1; rd**
Inst-2
Inst-3 **Z = MDR; if (Z) goto Inst-6; else goto Inst-4**
Inst-4 **TOS = TOS >> 1**
Inst 5 **MDR = MDR – 1; goto Inst-3**
Inst-6 **MDR = TOS; wr; goto Main**

Describe what the instruction **does** at the **IJVM level** (ie, describe it in a manner similar to the description of the IADD instruction on page 27 of the Week 7 presentation material) (3 marks)

6. Which of the following microinstructions are **invalid** for the **MIC-1**? It doesn't matter if the instructions "make sense", as long as they can be turned into a microinstruction word that does the equivalent work. If invalid, explain why. (4 marks)

Instruction	Valid?	If not, why not?
SP = MAR = OPC – H; rd		
MDR= H + MBR2; fetch; goto (MBR)		
MAR = SP; goto (MBR OR 0x1000)		
MAR = LV + MBR; fetch		
MDR = TOS = MDR + H + 1; wr		
PC = PC + 1; fetch; goto (MDR)		
MDR = H – TOS; rd		

7. **Draw a picture** of the **microinstruction word** that would be required for the **MIC-2 microarchitecture**, labeling the various part and identifying the purposes of the bits. (2 marks)

8. Write microinstructions to implement the IJVM **GOTO** instruction using the pipelined MIC-3 architecture. Show your answer using the chart below (***note*** – you may not need to use every line of the chart).
(Hint – a good place to start is to check the textbook for the MIC-2 microcode for this instruction) (4 marks)

Cy	Load ALU Inputs	Perform ALU Operation	Store ALU Outputs
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			

9. A system that can address **1 MB of memory** uses a **two-way set associative** cache that has **256 cache lines** with **16 bytes** stored in each line. The cache uses an LRU replacement algorithm.

   a) **Draw a diagram of the memory address,** and label how the cache uses each part of it (1 mark)

   b) The system accesses the following hexadecimal memory addresses:

First 8 addresses accessed	Next 8 addresses accessed	Last 8 addresses accessed
522c3	6a2c8	0fbcf
6a2cf	a1679	94bc5
2e5ad	ba5ae	74679
315f2	74677	522c5
7abcf	17672	742c1
675f4	74672	94bc3
315f2	522c3	ba5a5
375a3	f65f4	1e5a0

   Assuming that the cache was empty before the accesses took place, **show which cache lines contain data and what the tags are for each entry** after all of the accesses have taken place. (4 marks)

Cache Line	Tags

   c) **Write an "*"** beside each memory access shown in part "b" which causes an **actual memory read** (2 marks)

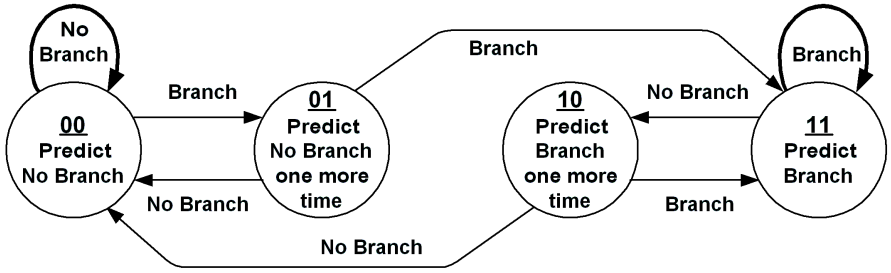
10. Designers are trying to decide how to design the branch prediction logic for a new generation of CPUs. They're using the following set of instructions as a test case to evaluate the behaviour of the branch prediction logic. The test case instructions convert a string of text from mixed case to all lowercase characters:

```
        TEXT  DB    'c:\A52C37~.tmp'  ;Note – quotes not part of the text

              MOV   CX,14    ;Number of bytes to process
              MOV   SI,#TEXT ;Point to start of string of bytes to process

        A10:  MOV   AL,[SI]  ;Read next byte into AL register
              CMP   AL,'A'   ;Is the byte less than uppercase 'A'?
Jump 1:       JL    A20      ; Yes – skip down to instruction at A20
              CMP   AL,'Z'   ;Is the byte greater than uppercase 'Z'?
Jump 2:       JG    A20      ; Yes – jump down to instruction at A20
              ADD   AL,20H   ;Convert to lowercase by adding hex 20
              MOV   [SI],AL  ; and put back into the string in memory

        A20:  ADD   SI,1     ;Point to the next byte in the string
              SUB   CX,1     ;Subtract 1 from CX
Jump 3:       JNZ   A10      ; and jump back up to A10 if not the last
```

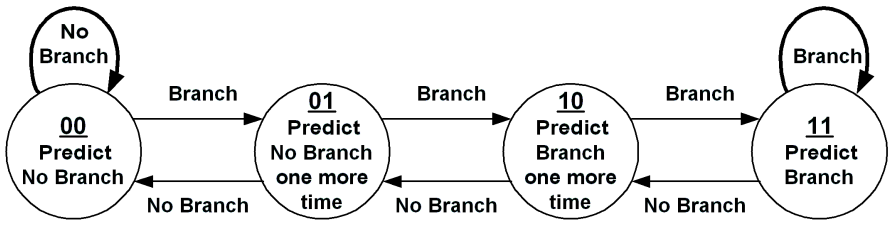
Characters from the "TEXT" string declared in the first line are put into the AL register one by one, and the *Jump1* and *Jump2* instructions branch or not depending on whether the character is less than lowercase "a" or greater than lowercase "z". The final *Jump3* instruction repeats the loop once for each character in the string.

The CPU stores two bits for **each of the three conditional jump instructions** in order to predict whether or not the instruction will actually jump. The designers are evaluating two different finite state machines:

Machine "A"



Machine "B"



Assume that the branch prediction bits for all three jump instructions start out as "**00**". **For both finite state machines, determine how many times each conditional branch will be *mis*predicted, and what the final state of the branch prediction bits will be after all of the instructions have executed.** The ASCII code table on page 129 of the textbook may be useful since it shows how the characters compare (ie, 'A' < 'a'). (6 marks)

	Finite State Machine "A"		Finite State Machine "B"	
Jump Instruction	**No. of mispredictions**	**Final prediction bits state**	**No. of mispredictions**	**Final prediction bits state**
Jump 1				
Jump 2				
Jump 3				