First name, Last name, ID: .....

Question 1 (4 points)

Let consider the Branch Prediction Mechanism based on the Branch History Table (BHT).
You are requested to

- Describe the architecture of a BHT
- Describe the behavior of a BHT, detailing when the processor accesses it, how, and which information it gets from it
- Assuming that the processor uses 32 bit addresses, each instruction is 4 byte wide, and the BHT is composed of 8 entries, identify the final content of the BHT if
- The BHT is initially empty (i.e., full of 0s, meaning NotTaken)
- The following instructions are executed in sequence (for each instruction the corresponding address is reported)
- `0x00040250  l.d r2,v1(r1)`
- `0x00040254  bez r2,l1` (branch taken, jumping to 0x00008000)
- `0x00008000  andi r2, r2, #1`
- `0x00008004  bnez r2, l2` (branch taken, jumping to 0x00004000)
- `0x00004000  addi r5, r5, #1`
- `0x00004004  bez r5, l3` (branch taken, jumping to 0x0000A010)
- `0x0000A010  add r1, r2, r3`
- `0x0000A014  bez r1, l4` (branch taken, jumping to 0x0000A200)
- `0x0000A200  add r4, r5 r0`
- `0x0000A204  bez r4, l1` (branch taken, jumping to 0x00008000)

  Determine also which of the branch instructions were correctly predicted, and which were mispredicted.

Write your answer here.

1-Branch history table is hardware unit it is on the decode stage. It has a N bit memory which by logN takes the least portion of the address.

2-After the issue unit the BHT access the value with memory of M to address portion of logN and depending on that portion checks the table as the same instruction is implemented or no. If the program just started BHT can take the table values randomly or by static branch prediction.

3-

Question 2 (4 points)

Let consider a MIPS64 architecture including the following functional units (for each unit the number of clock periods to complete one instruction is reported):

- Integer ALU: 1 clock period

- Data memory: 1 clock period
- FP arithmetic unit: 2 clock periods (pipelined)
- FP multiplier unit: 4 clock periods (pipelined)
- FP divider unit: 8 clock periods (unpipelined)

You should also assume that

- The branch delay slot corresponds to 1 clock cycle, and the branch delay slot is not enabled
- Data forwarding is enabled
- The EXE phase can be completed out-of-order.

You should consider the following code fragment and, filling the following tables, determine the pipeline behavior in each clock period, as well as the total number of clock periods required to execute the fragment, assuming that f5 is always different than 0. The values of the constants k1 and k2 are written in f10 and f11 before the beginning of the code fragment.

```
; ********************* MIPS64 ***********************
;  for (i = 0; i < 10; i++) {
;       tmp = v1[i]*k1 + v2[i]*k2;
        if (tmp!=0)
           v3[i]=2*tmp;
;  }
```

.data	Comments	Clock cycles
v1: .double“10 values”		
v2: .double “10 values”		
v3: .double “10 values”		
.text		
main: daddui r1,r0,0	r1← pointer	5
daddui r2,r0,10	r2 <= 10	1
loop: l.d f1,v1(r1)	f1 <= v1[i]	1
l.d f2,v2(r1)	f2 <= v2[i]	1
mul.d f3, f1, f10	f3 <= v1[i]*k1	4
mul.d f4, f2, f11	f4 <= v2[i]*k2	1
add.d f5, f3, f4	f5 <= v1[i]*k1 + v2[i]*k2	2
beq f5, l1	if(f5==0) goto l1	1
add.d f5, f5, f5	f5 <= f5+f5	2
l1: s.d f5,v3(r1)	v3[i] <= f5	1
daddui r1,r1,8	r1 <= r1 + 8	1
daddi r2,r2,-1	r2 <= r2 - 1	1
bnez r2,loop		2
halt		1
total		18*10+6=186

main: daddui r1,r0,0	F	D	E	M	W																								
daddui r2,r0,10		F	D	E	M	W																							
loop: l.d f1,v1(r1)			F	D	E	M	W																						

	l.d f2,v2(r1)				F	D	E	M	W																		
	mul.d f3, f1, f10					F	D	E	E	E	E	M	W														
	mul.d f4, f2, f11						F	D	E	E	E	E	M	W													
	add.d f5, f3, f4							F	D				E	E	M	W											
	beq f5, l1								F				D		E	M	W										
	add.d f5, f5, f5												F		D	E	E	M	W								
l1:	s.d f5,v3(r1)														F	D	E		M	W							
	daddui r1,r1,8															F	D		E	M	W						
	daddi r2,r2,-1																F		D	E	M	W					
	bnez r2,loop																		F	D		E	M	W			
	halt																			F		D	E	M	W		

Question 3 (8 points)

In the fixed-point representation, a fixed number of digits is used to represent the fractional part of a number. Example of fixed-point number with 8 fractional digits: 101.10011101. The corresponding decimal value is $1 * 2^2 + 0 * 2^1 + 1 * 2^0 + 1 * 2^{-1} + 0 * 2^{-2} + 0 * 2^{-3} + 1 * 2^{-4} + 1 * 2^{-5} + 1 * 2^{-6} + 0 * 2^{-7} + 1 * 2^{-8} = 4 + 0 + 1 + 0.5 + 0 + 0 + 0.0625 + 0.03125 + 0.015625 + 0 + 0.00390625 = 5.61328125$

Write the `computeG` subroutine in ARM assembly language, which computes the following function: $g(x, y) = x(3 - y * x^2) / 2$.
The subroutine receives in input:

- x: a 32-bit value which represents an unsigned fixed-point number
- n: the number of fractional digits in x
- y: a 32-bit value which represents another unsigned fixed-point number
- m: the number of fractional digits in y

The subroutine returns $g(x, y)$, with a number of fractional digits equal to $\max(n, m)$.

Before multiplying two operands (in every multiplication), you have to remove not significant digits (i.e., the trailing zeros after the decimal point) in both operands, and update their number of fractional digits consequently. Then, the result of the multiplication between two fixed-point numbers is computed as the integer product of their underlying integers. The number of fractional digits of the result is the sum of the number of fractional digits of the operands.

In order to subtract two fixed-point numbers, they must have the same number of fractional digits. If the number of fractional digits is different, the fixed-point number with few fractional digits must be shifted left by a proper number of positions.

It is suggested to implement the division by 2 with a right shift.

Example: $x = 0.10$, $n = 2$, $y = 101.00100$, $m = 5$.
The '.' separating integer and fractional digits is shown only for the sake of clarity.

After the removal of not significant digits, $x = 0.1$, $n' = 1$
$x^2 = 0.01$, with $p = n' + n' = 2$ fractional digits

After the removal of not significant digits, $y = 101.001$, $m' = 3$
$y * x^2 = 1.01001$, with $q = p + m' = 5$ fractional digits

The value 3 with $q = 5$ fractional digits is 11.00000
$3 - y * x^2 = 1.10111$

$x(3 - y * x^2) = 0.1 * 1.10111 = 0.110111$ with $r = q + n' = 6$ fractional digits

$x(3 - y * x^2) / 2 = 0.011011$ with $r = 6$ fractional digits

$\max(n, m) = 5$, so the subroutine returns 0.01101

Important notes:

- **Create a new project with Keil inside the "ARM" directory and write your code there.**
- The assembly subroutine must comply with the ARM Architecture Procedure Call Standard (AAPCS) standard (about parameter passing, returned value, callee-saved registers).
- Click on the following links to open web pages with the ARM instruction set

http://www.keil.com/support/man/docs/armasm

https://developer.arm.com/documentation/ddi0337/e/Introduction/Instruction-set-summary?lang=en

- You can convert fixed-point numbers from/to base 2 and 10 at this link:

https://www.exploringbinary.com/binary-converter/

Question 4 (5 points)

The subroutine developed in the previous exercise can be called iteratively to compute the inverse of the square root of a number y. In this exercise, we want to compute the inverse of the square root of $y = 3$.

- In a data area, allocate space for:

- *numIteration*: a 32-bit variable
- *arrayG*: an array of 10 elements; each element is a word (32 bits)

Inizialize *numIteration* to 0 and the first element of *arrayF* to 2^{-4}, to be represented as a fixed-point value with 10 fractional digits. The other elements are not initialized.

- In the Reset_Handler, configure the SYSTICK timer in order to generate an interrupt every 2^{20} clock cycles. Then, enter in an infinite loop.

The SYSTICK timer is configured by means of the following registers:
- Control and Status Register: size 32 bits, address 0xE000E010
- Reload Value Register: size 24 bits, address 0xE000E014
- Current Value Register: 24 bits, address 0xE000E018

The meaning of the bits in the Control and Status Register is as follows:
- Bit 16 (read-only): it is read as 1 if the counter reaches 0 since last time this register is read; it is cleared to 0 when read or when the current counter value is cleared
- Bit 2 (read/write): if 1, the processor free running clock is used; if 0, an external reference clock is used
- Bit 1 (read/write): if 1, an interrupt is generated when the timer reaches 0; if 0, the interrupt is not generated
- Bit 0 (read/write): if 1, SYSTICK timer is enabled; if 0, SYSTICK timer is disabled.

The Reload Value Register stores the value to reload when the timer reaches 0.
The Current Value Register stores the current value of the timer. Writing any number clears its content.

- In the handler of the SYSTICK timer, read the value of *numIteration* stored in memory. It counts the number of times the `computeG` subroutine has been called so far. Get the value *k* of the element of *arrayG* at position *numIteration*. Call the `computeG` subroutine with the following parameters: *x* = *k*, *n* = 10, *y* = 3, *m* = 0.
Increment *numIteration* by 1 and save the new result in memory.
Get the value returned by `computeG` and save it in *arrayG* at position *numIteration* (after the increment, this is the first empty element of the array).
Note that the `computeG` subroutine has to be called 10 times (at 10 different SYSTICK timer interrupts) because this is the lenght of *arrayG*. At the end, the last element of *arrayG* stores the best approximation of the inverse of the square root of 3.

Question 5 (6 points)

Given a 4 x 3 matrix of bytes SOURCE representing two's complement numbers on 8 bits each one, write an 8086 assembly program which changes the sign of each element and transposes the matrix, i.e. reverses the columns and rows from SOURCE to DESTINATION, i.e.:

DESTINATION (i, j) = - SOURCE (j,i)

- SOURCE has 4 rows and 3 columns and the transposed DESTINATION has 3 rows and 4 columns

- Both SOURCE and DESTINATION are "cut" by columns to be mapped to the 8086 data memory

- It is known in advance that the absolute value of each element of SOURCE does not exceed 120
- BRUTE FORCE APPROACH (i.e., without "loops") IS NOT ACCEPTABLE
- In your solution, please provide the declaration of SOURCE and DESTINATION, the code, and significant comments to the code and instructions. The class program does not include input output, while the home code should include input of SOURCE and printing of DESTINATION
- HINT: cut by columns the two matrices SOURCE (a..l) and DESTINATION found below and depict the corresponding 8086 data memory mapping organizations; then find an algorithm to switch from the 8086 data memory mapping of SOURCE to that of DESTINATION
- The single-loop-smart-algorithm is eligible to get up to one extra point (do not ask what is this algorithm)

Write your code in a file saved in the 8086 folder.

Example (where a, ..., l represent 2's complement numbers in the range -120...+120)

SOURCE

a	e	i
b	f	j
c	g	k
d	h	l

the following matrix DESTINATION is computed

-a	-b	-c	-d
-e	-f	-g	-h
-i	-j	-k	-l

**

Initial matrix SOURCE

-1	5	9
-2	-6	0
3	7	-3
-4	-8	1

the following matrix DESTINATION is computed

1	2	-3	4
-5	6	-7	8
-9	0	3	-1

Click on the following link to open a web page with the 8086 instruction set:

http://www.jegerlehner.ch/intel/IntelCodeTable.pdf