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

Question 1 (4 points)

The Tomasulo architecture for superscalar processors with dynamic scheduling and speculation often uses a Reorder Buffer (ROB)
You are requested to

- Explain what the ROB is and how it works
- Describe its architecture
- Detail when data are written in the ROB, when an entry is marked as busy and when it is released
- List the advantages stemming from the use of the ROB.

Write your answer here.

1-The ReOrder Buffer is the part where speculated answers are reserved.
2-The Reorder Buffer has 4 part
-Instruction: Where we imply the instruction type
-Value: The result of operation
-Destination: The register number or for load/store operation it is address
-Busy: Shows is the slot is free or not
3- Reorder buffer is allocated when the instruction fetch and the slot is become busy. The ROB is written when the result carried out by Functional unit and carried by the CDB. The values is read when the instruction required its result. It is de allocated when the instruction become the oldest and commited.
4-Advantage of the ROB is removing th WAW and WAR hazards by after the Out of order execution of the Execution unit by In order execvution on its inside

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: 5 clock periods (pipelined)
- FP divider unit: 7 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. 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 < 100; i++) {
;       v4[i] = ((v1[i]*k1) + (v2[i]*k2))/v3[i];
;  }
```

.data	Comments	Clock cycles
v1: .double “100 values”		
v2: .double “100 values”		
v3: .double “100 values”		
v4: .double “100 values”		
.text		
main: daddui r1,r0, 0	r1← pointer	
daddui r2,r0, 100	r2 <= 100	
loop: l.d f1,v1(r1)	f1 <= v1[i]	
mul.d f2, f1, f10	f2 <= v1[i]*k1	
l.d f3,v2(r1)	f3 <= v2[i]	
mul.d f4, f3, f11	f4 <= v2[i]*k2	
add.d f5,f2,f4	f5 <= v1[i]*k1 + v2[i]*k2	
l.d f6,v3(r1)	f6 <= v3[i]	
div.d f7,f5,f6	f7 <= (v1[i]*k1+v2[i]*k2)/v3[i]	
s.d f7,v4(r1)	v4[i] <= f7	
daddui r1,r1,8	r1 <= r1 + 8	
daddi r2,r2,-1	r2 <= r2 - 1	
bnez r2,loop		
halt		
total		2706

main: daddui r1,r0,0	F	D	E	M	W																									
daddui r2,r0,100		F	D	E	M	W																								
loop: l.d f1,v1(r1)			F	D	E	M	W																							
mul.d f2, f1, f10				F	D		E	E	E	E	E	M	W																	
l.d f3,v2(r1)					F		D	E	M	W																				
mul.d f4, f3, f11							F	D		E	E	E	E	E	M	W														
add.d f5,f2,f4								F		D					E	E	M	W												
l.d f6,v3(r1)										F					D	E		M	W											
div.d f7,f5,f6															F	D			E	E	E	E	E	E	E	M	W			
s.d f7,v4(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	
halt																													F	

Question 3 (6 points)

The variables DIVIDEND DW and DIVISOR DB store two unsigned integer numbers in binary. Write an 8086 assembly program to compute the result of the division DIVIDEND/DIVISOR on 8 integer bits and on 8 fractional bits, i.e., in fixed point on 16 bits, under the assumption that the division does not generate an overflow. The result has to be stored to the variable RESULT, whose size and definition are left to the student.

Up to 2 points more can be awarded, if the program also successfully and in full tackles the case DIVIDEND/DIVISOR possibly generating an overflow (indeed in this case the definition of RESULT should be different from the one above).

BRUTE FORCE APPROACHES ARE NOT ACCEPTABLE.

Comments are mandatory, as well a full and detailed explanation of the algorithm used.

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

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

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

Question 4 (8 points)

In the fixed-point representation, a fixed number of digits is used to represent the fractional part of a number. Negative values are expressed in two's complement representation.
Example of fixed-point number with 8 integer digits and 8 fractional digits: 10000101.10011101. The corresponding decimal value is $-1 * 2^7 + 1 * 2^2 + 1 * 2^0 + 1 * 2^{-1} + 1 * 2^{-4} + 1 * 2^{-5} + 1 * 2^{-6} + 1 * 2^{-8} = -128 + 4 + 1 + 0.5 + 0.0625 + 0.03125 + 0.015625 + 0.00390625 = -122.38671875$

Write the `squareRoot` subroutine in ARM assembly language, which returns the square root of a fixed-point number lower than 1. The subroutine receives in input:

- a 32-bit value X
- the number of fractional digits k

The subroutine computes the square root according to the following algorithm:

- Initialization: r = X, Q = 0
- for i = 1 to k:
- r = 2 * r
- if $r \geq Q + 2^{-i}$:
- $r = r - 2 * Q - 2^{-i}$
- $Q = Q + 2^{-i}$
- else:

- if $r < -(Q + 2^{-i})$:
- $r = r + 2 * Q - 2^{-i}$
- $Q = Q - 2^{-i}$
- Return Q

Solutions without multiplications and divisions are preferred.

Example: X = 0.100110, k = 6
The '.' separating integer and fractional digits is shown only for the sake of clarity.
Initialization: r = 0.100110, Q = 0.000000
Iteration i = 1
$r = 2 * 0.100110 = 1.001100$
$Q + 2^{-i} = 0.000000 + 0.100000 = 0.100000$
$r \geq Q + 2^{-i} \Rightarrow r = 1.001100 - 2 * 0.000000 - 0.100000 = 0.101100$
$Q = 0.000000 + 0.100000 = 0.100000$
Iteration i = 2
$r = 2 * 0.101100 = 1.011000$
$Q + 2^{-i} = 0.100000 + 0.010000 = 0.110000$
$r \geq Q + 2^{-i} \Rightarrow r = 1.011000 - 2 * 0.100000 - 0.010000 = 0.001000$
$Q = 0.100000 + 0.010000 = 0.110000$
Iteration i = 3
$r = 2 * 0.001000 = 0.010000$
$Q + 2^{-i} = 0.110000 + 0.001000 = 0.111000$
$-(Q + 2^{-i}) = 111111111111111111111111.001000$
$Q + 2^{-i} \leq r < -(Q + 2^{-i})$ => no action
Iteration i = 4
$r = 2 * 0.010000 = 0.100000$
$Q + 2^{-i} = 0.110000 + 0.000100 = 0.110100$
$-(Q + 2^{-i}) = 111111111111111111111111.001100$
$Q + 2^{-i} \leq r < -(Q + 2^{-i})$ => no action
Iteration i = 5
$r = 2 * 0.100000 = 1.000000$
$Q + 2^{-i} = 0.110000 + 0.000010 = 0.110010$
$r \geq Q + 2^{-i} \Rightarrow r = 1.000000 - 2 * 0.110000 - 0.000010 =$
$= 111111111111111111111111.011110$
$Q = 0.110000 - 0.000010 = 0.110010$
Iteration i = 6
$r = 2 * 111111111111111111111111.011110 =$
$111111111111111111111110.111100$
$-(Q + 2^{-i}) = -(0.110010 + 0.000001) =$
$111111111111111111111111.001101$
$r < -(Q + 2^{-i}) \Rightarrow r = 111111111111111111111110.111100 + 2 * 0.110010 - 0.000001 =$
$= 0.011111$
$Q = 0.110010 - 0.000001 = 0.110001$
The subroutine returns Q = 0110001.

Important notes:

• **Create a new project with Keil inside the "template" directory and write your code there. The "template" directory contains the subdirectories "led" and "button" that you can add to your project if you need them.**

• 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 positive fixed-point numbers from/to base 2 and 10 at this link:

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

Question 5 (5 points)

1) Write the `startSysTickTimer` subroutine in ARM assembly language, which configures the SYSTICK timer. The subroutine is called when button Key1 is pressed.

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 details, the `startSysTickTimer` subroutine configures the SYSTICK timer as follows:

- set the Reload Value Register to 0xFFFFFF
- clear the Current Value Register
- start the SYSTICK timer without generating interrupts.

2) Write the `stopSysTickTimer` subroutine in ARM assembly language, which stops the SYSTICK timer and returns the value of the Current Value Register.

When button Key2 is pressed, call the `stopSysTickTimer` subroutine. Then, call the `squareRoot` subroutine passing the value of the Current Value Register as first parameter and 24 as second parameter. You have to store the result in a global variable.