**Question #1** (4 points)

Let’s consider the exceptions.

You are requested to:

1. Provide the definition of exception and give an opinion regarding the difficulty of their handling between pipelined and unpipelined processors.
2. Cite at least 3 examples of possible sources of exceptions and the definition of restartable machine.
3. Provide a detailed description regarding the classification of exceptions.
4. Give the definition of precise exceptions.

Write your answer here.

1-EXCEPTIONS ARE UNEXPECTED EVENT WHICH CHANGE THE PROGRAM FLOW.

IN THE PIPLEINE IT\S HARD TO MANAGE BUT IN UNPIPLINED ITS EASY B/C WE RUN 1 INSTRUCION AT A TIME

2-

3-

**Question 2** (4 points)

Let's consider a superscalar MIPS64 architecture implementing dynamic scheduling, speculation, and multiple issues and composed of the following units:

* An issue unit able to process 2 instructions per clock period; in the case of a branch instruction, only one instruction is issued per clock period
* A commit unit able to process 1 instruction per clock period
* The following functional units (for each unit the number of clock periods to complete one instruction is reported):
  + 1 unit for memory access:1 clock period
  + 1 unit for integer arithmetic instructions: 1 clock period
  + 1 unit for branch instructions: 1 clock period
  + 1 unit for FP multiplication (pipelined): 5 clock periods
  + 1 unit for FP division (unpipelined): 6 clock periods
  + 1 unit for other FP instructions (pipelined): 3 clock periods
  + 2 Common Data Bus.
* Let's also assume that:
  + Branch predictions are always correct
  + All memory accesses never trigger a cache miss.

You should use the following table to describe the behavior of the processor during the execution of the first 2 iterations of the cycle, computing the total number of required clock cycles.

*Tip: For the EXE stage, it is recommended to add a lowercase letter to help distinguish between the different functional units available.*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **# Iteration** | **Instruction** | **ISSUE x2** | **EXE** | **MEM** | **CDB x2** | **COMMIT** | **Notes** |
| 1 | l.d f1,v1(r1) | 1 | 2i | 3 | 4 | 5 |  |
| 1 | l.d f2,v2(r1) | 1 | 3i | 4 | 5 | 6 |  |
| 1 | l.d f3,v3(r1) | 2 | 4i | 5 | 6 | 7 |  |
| 1 | div f4, f1, f2 | 2 | 11d |  | 12 | 13 | f1,f2 are ready at 5 |
| 1 | add.d f5, f5, f3 | 3 | 9a |  | 13 | 14 | f3 ready at 6 |
| 1 | mul f6, f4, f5 | 3 | 18m |  | 19 | 20 | Data at 13 |
| 1 | s.d f6,v4(r1) | 4 | 20 | 21 | 22 | 23 |  |
| 1 | daddui r1,r1,8 | 4 | 24 |  | 25 | 26 |  |
| 1 | daddi r2,r2,-1 | 5 | 25 |  | 26 | 27 |  |
| 1 | bnez r2,loop | 6 | 27 | 28 |  | 29 |  |
| 2 | l.d f1,v1(r1) | 7 | 28 | 29 | 30 | 31 |  |
| 2 | l.d f2,v2(r1) | 7 | 29 | 30 | 31 | 32 |  |
| 2 | l.d f3,v3(r1) | 8 | 30 | 31 | 32 | 33 |  |
| 2 | div f4, f1, f2 | 8 | 37 |  | 38 | 39 | F1/f2 ready at 31 |
| 2 | add.d f5, f5, f3 | 9 | 38 |  | 39 | 40 | F3 ready at 32 |
| 2 | mul f6, f4, f5 | 9 | 44 |  | 45 | 46 | F4/f5 ready at 39 |
| 2 | s.d f6,v4(r1) | 10 | 21 | 32 |  | 47 |  |
| 2 | daddui r1,r1,8 | 10 | 22 |  |  |  |  |
| 2 | daddi r2,r2,-1 | 11 |  |  |  |  |  |
| 2 | bnez r2,loop | 12 |  |  |  |  |  |

**Question 3** (6 points)

A 8x8 matrix MATR of bytes stores A-Z ascii characters only (all in upper case; English alphabet). Write an 8086 assembly program which extracts the characters where i=j (main diagonal) and counts how many occurrences of these characters are found in all the matrix. In other words, the program needs to fill in the two arrays KODE and OCCURRENCES according to the following rule:

KODE (k) = MATR (k,k) (i.e., the ascii code of the character stored in that position).

OCCURRENCES (k) = number of times the character KODE (k) is found inside the matrix MATR

Please observe/comply with the following

* It is mandatory to cut the matrix by rows.
* In your solution, please provide the declaration of all the arrays and the code, together with a short description of the algorithm used and significant comments to the code and instructions.
* It is guaranteed that MATR only stores A-Z ascii English alphabet characters (all in upper case) and that the characters on the main diagonal are all different.
* As this is an assembly program, please do NOT design an algorithm which is suitable to a high-level language approach, but strongly focus on the cut by rows of the matrix and its related properties. (= refer to its array implementation and “do not use” the original i and j).
* ANY (EVEN PARTIAL) BRUTE FORCE APPROACH IS NOT ACCEPTABLE. Any high-level-language-like approach is discouraged; please look at the array implementation!
* Hint: to devise a suitable algorithm, take as an example a smaller matrix (e.g. 4x4), “write it” when cut by rows, and identify the property of elements on the main diagonal.

Example:

Matrix MATR

C D A F K K J M

B B B D H G R E

O O P U Y R E F

W W W W F R Y Z

T T T T T T T T

D E A H T U I O

R E R T S W E T

B T U O K Z X D

KODE = C B P W T U E D

OCCURRENCES = 1 4 1 5 12 3 5 4

**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)

Given any positive integer *a*, the operation of digitaddition generates a new positive integer *b* > *a* by adding to *a* the sum of the digits of *a*. For example, if *a* = 47, the sum of the digits of *a* is 4 + 7 = 11, and then *b* = *a* + 11 = 58.

The process can be repeated endlessly, forming a digitaddition series: 47, 58, 71, 79, 95, ....

Define an area of memory that will store the digitaddition series. Each element of the series is a word, and the area of memory must be large enough to store 50 elements.

Then, in the Reset\_Handler, assign a value of your choice to the first element of the series.

You are requested to write two subroutines in ARM assembly language:

1. The digitSum subroutine receives in input a positive integer *a* and returns the value *b*, computed by adding to *a* the sum of its digits.  
   Important: *a* can have any number of digits (the upper limit is due to its representation as word).  
   Hint: You can obtain the digits of a number *a* (from the least significant to the most significant) by repeatedly dividing *a* by 10 and taking the remainder.
2. The digitaddition subroutine receives in input:
   * the address of an area of memory (where only the first value is initialized)
   * the number *N* of elements of the digitaddition series to be computed (including the first one)

The subroutine reads the first value saved in the area of memory. Then, it computes (and writes in that area of memory) all the subsequent elements of the digitaddition series. In the computation of each element, it calls the digitSum subroutine.  
The digitaddition subroutine returns the sum of all the digits in the digitaddition series.  
For example, it the initial element stored in the area of memory is 47 and *N* = 5, the subroutine stores in memory the next elements 58, 71, 79, 95. Finally, it returns 4 + 7 + 5 + 8 + 7 + 1 + 7 + 9 + 9 + 5 = 62.

If there is an overflow in the computation of an element, the subroutine immediately ends and returns 0.

Important notes:

1. **Create a new project with Keil inside the “ARM” directory and write your code there. The “ARM” directory contains some subdirectories that you can add to your project if you need them.**
2. The assembly subroutine must comply with the ARM Architecture Procedure Call Standard (AAPCS) standard (in terms of parameter passing, returned value, callee-saved registers).
3. Click on the following links to open web pages with the ARM instruction set

https://developer.arm.com/documentation/dui0473/m/preface

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

**Question 5** (5 points)

The sum of all the digits in a digitaddition series can be computed with a simple formula: you have to subtract the first number of the series from the last one and add the sum of the digits of the last number. For example, considering the series 47, 58, 71, 79, 95, the sum of all its digits is 95 – 47 + 9 + 5 = 62. We want to check this formula in our program.

Add the following functionalities to the project created in the previous exercise:

1. The user can specify a binary value *K* through buttons Key1 and Key2. Key1 inserts a new digit equal to 0, while key2 inserts a new digit equal to 1. For example, if the user presses Key2, Key1, Key1, Key2, Key2, Key1, the final value is 100110
2. Declare an uninitialized array of int, with size 10.
3. When the user presses INT0:
   * set the first element of the array to *K*
   * call the digitaddition subroutine passing the address of the array and 10 (as the length of the series). The subroutine writes the other elements of the array and returns the sum of all the digits.
   * call the digitSum subroutine passing the last element of the array in order to obtain the sum of its digits.
   * If the value returned by the digitaddition subroutine is equal to the value returned by the digitSum subroutine plus the last element of the series minus the first element, then switch on led 4 and switch off led 5. If it is different, switch on led 5 and switch off led 4.

**Notes about the leds.** The pins of leds 4-11 are P2.7 – P2.0. The function LED\_init (included in the provided template) initializes the pins as GPIO Port 2.0 (LPC\_GPIO2). You have to switch on the required leds by means of the following accessible registers:

* FIODIR: Fast GPIO Port Direction control register. This register individually controls the direction of each port pin.
* FIOMASK: Fast Mask register for port. Writes, sets, clears, and reads to port (done via writes to FIOPIN, FIOSET, and FIOCLR, and reads of FIOPIN) alter or return only the bits enabled by zeros in this register.
* FIOPIN: Fast Port Pin value register using FIOMASK. The current state of digital port pins can be read from this register. The value read is masked by ANDing with inverted FIOMASK. Writing to this register places corresponding values in all bits enabled by zeros in FIOMASK.
* FIOSET: Fast Port Output Set register using FIOMASK. This register controls the state of output pins. Writing 1s produces highs at the corresponding port pins. Writing 0s has no effect. Reading this register returns the current contents of the port output register. Only bits enabled by 0 in FIOMASK can be altered.
* FIOCLR: Fast Port Output Clear register using FIOMASK. This register controls the state of output pins. Writing 1s produces lows at the corresponding port pins. Writing 0s has no effect. Only bits enabled by 0 in FIOMASK can be altered.