**Question #1** (4 points)

Let’s consider the multicycle operations.

You are requested to:

1. Discuss about the difficulties involved in executing floating-point instructions;
2. Give the definition of Latency and Initiation Interval;
3. Discuss about the MIPS R-4000 pipeline;
4. Give the definition of Branch Delay Slot and Load Delay Slot.

Write your answer here.

**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 2 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): 4 clock periods
  + 1 unit for FP division (unpipelined): 12 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 loop, 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.*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Loop #** | **Instruction** | **ISSUE x2** | **EXE** | **MEM** | **CDB x2** | **COMMIT x2** | **Notes** |
| 1 | l.d f1, v1(r1) | 1 | 2m | 3 | 4 | 5 |  |
| 1 | l.d f2, v2(r1) | 1 | 3m | 4 | 5 | 6 |  |
| 1 | mul.d f5, f1, f2 | 2 | 4mul |  | 8 | 9 |  |
| 1 | l.d f3, v3(r1) | 2 | 5m | 6 | 8 | 9 |  |
| 1 | l.d f4, v4(r1) | 3 | 6m | 7 | 9 | 10 |  |
| 1 | mul.d f6, f3, f4 | 3 | 10 mul |  | 14 | 15 |  |
| 1 | sub.d f7, f5, f6 | 4 | 11 |  | 14 | 15 |  |
| 1 | add.d f7, f7, f8 | 4 | 15 |  | 18 | 19 |  |
| 1 | s.d f7, v5(r1) | 5 | 19m |  |  | 20 |  |
| 1 | daddui r1, r1, 8 | 5 | 6i |  | 7 | 20 |  |
| 1 | daddi r2, r2, -1 | 6 | 7i |  | 9 | 21 |  |
| 1 | bnez r2, loop | 7 | 8j |  |  | 21 |  |
| 2 | l.d f1, v1(r1) | 8 | 9m | 10 | 11 | 22 |  |
| 2 | l.d f2, v2(r1) | 8 | 10m | 11 | 12 | 22 |  |
| 2 | mul.d f5, f1, f2 | 9 | 14mul |  | 15 | 23 |  |
| 2 | l.d f3, v3(r1) | 9 | 11m | 12 | 13 | 23 |  |
| 2 | l.d f4, v4(r1) | 10 | 12m | 13 | 14 | 24 |  |
| 2 | mul.d f6, f3, f4 | 10 | 16mul |  | 20 | 24 |  |
| 2 | sub.d f7, f5, f6 | 11 | 21 |  | 24 | 25 |  |
| 2 | add.d f7, f7, f8 | 11 | 25 |  | 28 | 29 |  |
| 2 | s.d f7, v5(r1) | 12 | 13m |  |  | 29 |  |
| 2 | daddui r1, r1, 8 | 12 | 13i |  | 14 | 30 |  |
| 2 | daddi r2, r2, -1 | 13 | 14i |  | 15 | 30 |  |
| 2 | bnez r2, loop | 14 | 15j |  |  | 31 |  |

Small variations with respect to this solution could be accepted as correct in a case by case manner.

**Question 3** (6 points)

Let us consider a series of values

ai = ai-1 - ai-2 + ai-3

with a0=0, a1=1 and a2=2

Write an 8086 program which computes the first 20 elements ai of the series and stores into the array A [0..19] where each cell is on 8 bits. Please consider that the values ai are all non-negative and that no overflow will occur.

As a part of the program, please also count and write into the 3 variables num\_0, num\_1 and num\_2 the number of 0’s, 1’s and 2’s, respectively, as stored in the array A.

Please **clearly & concisely** explain your proposed code. Failure to provide an explanation and relevant comments to the most important instructions/parts of the program, will result in severe score penalties.

The algorithm is straightforward and self-included in the formula for computing the elements of the series. Complex implementations, i.e., not following step by step the computation of the series, will be evaluated zero points, no matter if they are “possibly/likely” working.

Please note that during the exam NO questions will be responded on this exercise. It is assumed that each student at the master level can understand and implement the computation of series.

**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** (9 points)

Write the depthFirstSearch subroutine in ARM assembly to generate a maze according to the randomized depth-first search algorithm. The maze will be stored in a NUM\_ROW \* NUM\_COL matrix of bytes with the following encoding for the inner cells (i.e., cells not in the border):

* Bit 0: 1 if the cell has been already visited (i.e., processed) by the algorithm, 0 otherwise
* Bit 1: 1 if there is a passage between the cell and the neighbor at its right, 0 if there is a wall
* Bit 2: 1 if there is a passage between the cell and the neighbor at its bottom, 0 if there is wall
* Bit 3: 1 if there is a passage between the cell and the neighbor at its left, 0 if there is a wall
* Bit 4: 1 if there is a passage between the cell and the neighbor at its top, 0 if there is a wall
* Bit 5, 6, and 7: they are fixed at 0

Instead, the cells along the border have all bits set to 1 (they are not processed by the algorithm).

For example, the initial matrix with NUM\_ROW = 6 and NUM\_COL = 5 is:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 11111111 | 11111111 | 11111111 | 11111111 | 11111111 |  | 0 | 1 | 2 | 3 | 4 |
| 11111111 | 0 | 1 | 0 | 11111111 |  | 5 | 6 | 7 | 8 | 9 |
| 11111111 | 0 | 0 | 0 | 11111111 |  | A | B | C | D | E |
| 11111111 | 0 | 0 | 0 | 11111111 |  | F | 10 | 11 | 12 | 13 |
| 11111111 | 0 | 0 | 0 | 11111111 |  | 14 | 15 | 16 | 17 | 18 |
| 11111111 | 11111111 | 11111111 | 11111111 | 11111111 |  | 19 | 1A | 1B | 1C | 1D |

In the table, the lines between each cell represent a wall (i.e., no passage exists between each couple of neighboring cells). The cell with value 1 is the entrance of the maze. For the sake of clarity, the hexadecimal numeration of the cells according to the row-major order is reported on the right.

The depthFirstSearch subroutine receives the following parameters (in the order indicated):

* address of the maze matrix
* number of rows NUM\_ROW
* number of columns NUM\_COL
* index of the cell at the starting point (in the example it is 7)

The algorithm for the maze generation finds all the neighbors of the current cell not yet visited:

* If there is at least one unvisited neighbor, a passage between the current cell and a chosen neighbor is created (changing the corresponding bits in both cells). The index of the current cell is pushed on the stack. The neighbor is marked as visited (by setting its bit 0) and becomes the current cell for the next iteration of the algorithm.
* If there are not unvisited neighbors, the content of the current cell does not change. Pop a value from the stack: it will become the current cell for the next iteration of the algorithm.

The algorithm ends when the stack is empty (meaning that all cells have been visited).

In details, you have to implement the search of visited or unvisited neighbors in the following way:

* Set r0 = 0 if the neighboring cell at the right has not been visited, 1 otherwise
* Set r1 = 0 if the neighboring cell at the bottom has not been visited, 1 otherwise
* Set r2 = 0 if the neighboring cell at the left has not been visited, 1 otherwise
* Set r3 = 0 if the neighboring cell at the top has not been visited, 1 otherwise

Then, depthFirstSearch calls the chooseNeighbor subroutine, which returns

* 1 if r0 = 0
* 2 if r0 = 1 and r1 = 0
* 3 if r0 = r1 = 1 and r2 = 0
* 4 if r0 = r1 = r2 = 1 and r3 = 0
* 0 if r0 = r1 = r2 = r3 = 1 (meaning that all neighbors have been visited)

For example, if chooseNeighbor returns 2, depthFirstSearch creates a passage between the current cell and its neighbor at the bottom: bit 2 of current cell and bit 4 of the neighbor are set.

Example of execution with the 6x5 matrix shown at the beginning.

**Iteration 1**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11111111 | 11111111 | 11111111 | 11111111 | 11111111 |
| 11111111 | 0 | 11 | 1001 | 11111111 |
| 11111111 | 0 | 0 | 0 | 11111111 |
| 11111111 | 0 | 0 | 0 | 11111111 |
| 11111111 | 0 | 0 | 0 | 11111111 |
| 11111111 | 11111111 | 11111111 | 11111111 | 11111111 |

Index of current cell is 7.

r0 = 0, r1 = 0, r2 = 0, r3 = 1.

chooseNeighbor returns 1.

A passage is carved between cells 7 and 8.

The new current cell is 8.

Stack now contains 7.

**Iteration 2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11111111 | 11111111 | 11111111 | 11111111 | 11111111 |
| 11111111 | 0 | 11 | 1101 | 11111111 |
| 11111111 | 0 | 0 | 10001 | 11111111 |
| 11111111 | 0 | 0 | 0 | 11111111 |
| 11111111 | 0 | 0 | 0 | 11111111 |
| 11111111 | 11111111 | 11111111 | 11111111 | 11111111 |

Index of current cell is 8.

r0 = 1, r1 = 0, r2 = 0, r3 = 1.

chooseNeighbor returns 2.

A passage is carved between cells 8 and D.

The new current cell is D.

Stack now contains 8, 7.

**Iteration 3**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11111111 | 11111111 | 11111111 | 11111111 | 11111111 |
| 11111111 | 0 | 11 | 1101 | 11111111 |
| 11111111 | 0 | 0 | 10101 | 11111111 |
| 11111111 | 0 | 0 | 10001 | 11111111 |
| 11111111 | 0 | 0 | 0 | 11111111 |
| 11111111 | 11111111 | 11111111 | 11111111 | 11111111 |

Index of current cell is D.

r0 = 1, r1 = 0, r2 = 0, r3 = 1.

chooseNeighbor returns 2.

A passage is carved between cells D and 12.

The new current cell is 12.

Stack now contains D, 8, 7.

**Iteration 4**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11111111 | 11111111 | 11111111 | 11111111 | 11111111 |
| 11111111 | 0 | 11 | 1101 | 11111111 |
| 11111111 | 0 | 0 | 10101 | 11111111 |
| 11111111 | 0 | 0 | 10101 | 11111111 |
| 11111111 | 0 | 0 | 10001 | 11111111 |
| 11111111 | 11111111 | 11111111 | 11111111 | 11111111 |

Index of current cell is 12.

r0 = 1, r1 = 0, r2 = 0, r3 = 1.

chooseNeighbor returns 2.

A passage is carved between cells 12 and 17.

The new current cell is 17.

Stack now contains 12, D, 8, 7.

**Iteration 5**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11111111 | 11111111 | 11111111 | 11111111 | 11111111 |
| 11111111 | 0 | 11 | 1101 | 11111111 |
| 11111111 | 0 | 0 | 10101 | 11111111 |
| 11111111 | 0 | 0 | 10101 | 11111111 |
| 11111111 | 0 | 11 | 11001 | 11111111 |
| 11111111 | 11111111 | 11111111 | 11111111 | 11111111 |

Index of current cell is 17.

r0 = 1, r1 = 1, r2 = 0, r3 = 1.

chooseNeighbor returns 3.

A passage is carved between cells 17 and 16.

The new current cell is 16.

Stack now contains 17, 12, D, 8, 7.

…

**Iteration 11**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11111111 | 11111111 | 11111111 | 11111111 | 11111111 |
| 11111111 | 101 | 11 | 1101 | 11111111 |
| 11111111 | 10011 | 1101 | 10101 | 11111111 |
| 11111111 | 111 | 11001 | 10101 | 11111111 |
| 11111111 | 10011 | 1011 | 11001 | 11111111 |
| 11111111 | 11111111 | 11111111 | 11111111 | 11111111 |

After iteration 11, the current cell is 6.

Stack contains B, C, 11, 10, 15, 16, 17, 12, D, 8, 7.

Other 11 iterations follow, popping a value from the stack until the stack is empty, because no neighbors remain.

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. **It also contains the startup\_LPC17xx.s file with the Reset\_Handler procedure and the declaration of the memory areas.**
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** (4 points)

The maze generated in the previous exercise is always the same because the chooseNeighbor subroutine is deterministic. We want to write a chooseRandomNeighbor subroutine that chooses a random neighbor exploiting the systick timer. First, in the Reset\_Handler, set the Reload Value Register of the systick timer to 0xFFFFF and configure the systick timer in order not to generate an interrupt when the timer counter reaches 0. Then, start the timer.

The chooseRandomNeighbor subroutine receives the same parameters as chooseNeighbor

* r0 = 0 if the neighboring cell at the right has not been visited, 1 otherwise
* r1 = 0 if the neighboring cell at the bottom has not been visited, 1 otherwise
* r2 = 0 if the neighboring cell at the left has not been visited, 1 otherwise
* r3 = 0 if the neighboring cell at the top has not been visited, 1 otherwise

First, the subroutine pushes some values on the stack:

* if r0 = 0, it pushes 1
* if r1 = 0, it pushes 2
* if r2 = 0, it pushes 3
* if r3 = 0, it pushes 4

If no values are pushed, the subroutine immediately returns 0. Otherwise, it counts how many registers, among the input parameters, are 0. Then, it computes the module between the systick timer counter and the number of registers equal to 0. For example, if r0 = 0, r1 = 0, r2 = 0, r3 = 1, it computes the module with 3. The module indicates the index of the element of the stack to retrieve, which is the return value. It is important to pop all elements from the stack before concluding the subroutine to leave the stack balanced.

**Note about the systick timer.**

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

In the provided startup\_LPC17xx.s file, the addresses are defined as constants.

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 use
* 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, the systick timer is enabled; if 0, the 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.