**Question #1** (4 points)

Let’s consider the Multicycle operations.

You are requested to:

1. Discuss the main differences between executing an integer and floating point operation
2. Draw an example of a pipeline capable of executing integer operations
3. Draw an example of a pipeline capable of executing floating point operations
4. Give the definition of Latency and Initiation Interval

**Write your answer here.**

**Question #2** (4 points)

Let's consider a MIPS architecture using a *Branch History Table* (BHT) composed of *8 1-bit entries*. Let's assume that this architecture executes the following code, which, taking a *vec* vector, calculates the number of even and odd characters it contains with respect to the relative index of the ASCII table (for example, the character '*a*' in the ASCII table is 0x*61* (97 in decimal) and is therefore odd, the character '*b*' is 0x*62* (98 in decimal) and is therefore even, and so on) and then writes the result into the variables *res0* and *res1*. The calculation of the remainder is performed using the following formula: *a mod n = a – (a / n) \* n*. For every instruction, the hexadecimal address of the memory cell storing the instruction is reported.

Assuming that when the execution of the code fragment the BHT is full of null values (corresponding to the prediction Not Taken) you are asked to compute:

* The number of mispredicted branches during the execution of the code
* The BHT content when the execution finishes (using the table reported on the next page).

For all computations, it is suggested the usage of the table on the next page. Write in the highlighted cells whether the result of the prediction of the current branch and the real behavior (result) of the software is *Taken* (T) or *Not* *Taken* (NT).

*Hint: To calculate the BHT entry corresponding to each branch instruction, remember that you should exclude the last two bits from the instruction address as they are always equal to 0.*

|  |  |  |  |
| --- | --- | --- | --- |
| vec: | . asciiz | "coding" | # input vector with the termination character |
| rese: | .space | 1 | # counter of even elements |
| reso: | .space | 1 | # counter of odd elements |
| … |  |  |  |
| 0x0000 |  | daddui r1, r0, 2 | # initialize the divisor |
| 0x0004 |  | daddi r2, r0, -1 | # initialize the pointer |
| 0x0008 |  | daddui r5, r0, 0 | # initialize the counter of even elements |
| 0x000c |  | daddui r6, r0, 0 | # initialize the counter of odd elements |
| 0x0010 | cyc: | daddui r2, r2, 1 | # increment the pointer |
| 0x0014 |  | lb r3, vec(r2) | # load an element from vec |
| 0x0018 |  | beqz r3, term | # check if the loaded char is the termination one |
| 0x001c |  | daddu r4, r3, r0 | # create a copy of the previously loaded char |
| 0x0020 |  | ddivu r3, r3, r1 | # calculate the remainder - *step 1: a / n* |
| 0x0024 |  | dmulu r3, r3, r1 | # calculate the remainder - *step 2: (a / n) \* n* |
| 0x0028 |  | dsubu r4, r4, r3 | # calculate the remainder - *step 3: a – (a / n) \* n* |
| 0x002c |  | bnez r4, nxt | # check if the remainder is not equal to zero (odd char) |
| 0x0030 |  | daddui r5, r5, 1 | # increment the counter of even elements |
| 0x0034 |  | j cyc | # next cycle |
| 0x0038 | nxt: | beqz r4, cyc | # check if the remainder is equal to zero (even char) |
| 0x003c |  | daddui r6, r6, 1 | # increment the counter of odd elements |
| 0x0040 |  | j cyc | # next cycle |
| 0x0044 | term: | sb r5, rese(r0) | # store the result |
| 0x0048 |  | sb r6, reso(r0) | # store the result |
| 0x004c | halt |  | # termination of the program |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Address | Code | | BHT | Iteration #1 | | Iteration #2 | | Iteration #3 | | Iteration #4 | |
|  |  | | entry # | prediction | result | prediction | result | prediction | result | prediction | result |
| 0x0000 |  | daddui r1, r0, 2 |  |  |  |  |  |  |  |  |  |
| 0x0004 |  | daddi r2, r0, -1 |  |  |  |  |  |  |  |  |  |
| 0x0008 |  | daddui r5, r0, 0 |  |  |  |  |  |  |  |  |  |
| 0x000c |  | daddui r6, r0, 0 |  |  |  |  |  |  |  |  |  |
| 0x0010 | cyc: | daddui r2, r2, 1 |  |  |  |  |  |  |  |  |  |
| 0x0014 |  | lb r3, vec(r2) |  |  |  |  |  |  |  |  |  |
| 0x0018 |  | beqz r3, term | 6 | Nt | Nt | Nt | Nt | Nt | Nt | Nt | Nt |
| 0x001c |  | daddu r4, r3, r0 |  |  |  |  |  |  |  |  |  |
| 0x0020 |  | ddivu r3, r3, r1 |  |  |  |  |  |  |  |  |  |
| 0x0024 |  | dmulu r3, r3, r1 |  |  |  |  |  |  |  |  |  |
| 0x0028 |  | dsubu r4, r4, r3 |  |  |  |  |  |  |  |  |  |
| 0x002c |  | bnez r4, nxt | 3 | Nt | T | T | T | T | Nt | Nt | T |
| 0x0030 |  | daddui r5, r5, 1 |  |  |  |  |  |  |  |  |  |
| 0x0034 |  | j cyc |  |  |  |  |  |  |  |  |  |
| 0x0038 | nxt: | beqz r4, cyc | 6 | Nt | Nt | Nt | Nt |  |  | Nt | Nt |
| 0x003c |  | daddui r6, r6, 1 |  |  |  |  |  |  |  |  |  |
| 0x0040 |  | j cyc |  |  |  |  |  |  |  |  |  |
| 0x0044 | term: | sb r5, rese(r0) |  |  |  |  |  |  |  |  |  |
| 0x0048 |  | sb r6, reso(r0) |  |  |  |  |  |  |  |  |  |
| 0x004c | halt |  |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Address | Code | | BHT | Iteration #5 | | Iteration #6 | | Iteration #7 | | Iteration #8 | |
|  |  | | entry # | prediction | result | prediction | result | prediction | result | prediction | result |
| 0x0000 |  | daddui r1, r0, 2 |  |  |  |  |  |  |  |  |  |
| 0x0004 |  | daddi r2, r0, -1 |  |  |  |  |  |  |  |  |  |
| 0x0008 |  | daddui r5, r0, 0 |  |  |  |  |  |  |  |  |  |
| 0x000c |  | daddui r6, r0, 0 |  |  |  |  |  |  |  |  |  |
| 0x0010 | cyc: | daddui r2, r2, 1 |  |  |  |  |  |  |  |  |  |
| 0x0014 |  | lb r3, vec(r2) |  |  |  |  |  |  |  |  |  |
| 0x0018 |  | beqz r3, term | 6 | Nt | Nt | nt | nt | nt | t |  |  |
| 0x001c |  | daddu r4, r3, r0 |  |  |  |  |  |  |  |  |  |
| 0x0020 |  | ddivu r3, r3, r1 |  |  |  |  |  |  |  |  |  |
| 0x0024 |  | dmulu r3, r3, r1 |  |  |  |  |  |  |  |  |  |
| 0x0028 |  | dsubu r4, r4, r3 |  |  |  |  |  |  |  |  |  |
| 0x002c |  | bnez r4, nxt | 3 | T | Nt | nt | T |  |  |  |  |
| 0x0030 |  | daddui r5, r5, 1 |  |  |  |  |  |  |  |  |  |
| 0x0034 |  | j cyc |  |  |  |  |  |  |  |  |  |
| 0x0038 | nxt: | beqz r4, cyc | 6 |  |  | nt | nt |  |  |  |  |
| 0x003c |  | daddui r6, r6, 1 |  |  |  |  |  |  |  |  |  |
| 0x0040 |  | j cyc |  |  |  |  |  |  |  |  |  |
| 0x0044 | term: | sb r5, rese(r0) |  |  |  |  |  |  |  |  |  |
| 0x0048 |  | sb r6, reso(r0) |  |  |  |  |  |  |  |  |  |
| 0x004c | halt |  |  |  |  |  |  |  |  |  |  |

The number of mispredicted branches during the execution of the code is: \_\_6\_\_\_\_\_

**BHT - Final content**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Entry 0 | 0 |  | Entry 4 | 0 |
| Entry 1 | 0 |  | Entry 5 | 0 |
| Entry 2 | 0 |  | Entry 6 | 1 |
| Entry 3 | 1 |  | Entry 7 | 0 |

**Question #3** (6 points)

A 8 x 8 matrix MATR of bytes stores letters a-z both in lower and upper cases, with 2 and only 2 instances of the letter “a” either in lower or upper case(s). The matrix is cut by rows and implemented by the array FIELD. Write an 8086 assembly program which, once found the positions inside FIELD of the two “a” letters (either in lower or upper case(s)) returns the following information:

1. The two “a” letters are on the same row of MATR
2. The two “a” letters are on the same column of MATR
3. Neither #1 nor #2

In addition, the program should return how many letters are in upper and how many are in lower case.

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 the MATR only stores letters a-z both in lower and upper cases, and that there are two and only two occurrences of the “a” letter either in lower or upper case(s).
* 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 FIELD and “do not use” the original i and j).
* ANY 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 same column.

Examples:

Matrix MATR

s b c D E F g H

E I O **a** k n b v lowercase = 33

K y t r e d x h

L L L L L L i i uppercase = 31

G g g g G G G Y

S Q R **A** U I i o the two “a” letters are on the

W r y u i o p k same column of MATR

E B N U e e s w

Matrix MATR

s b c D E F g H

E I O d k n b v lowercase = 33

K y t r e d x h

L L L L L L i i uppercase = 31

G g g g G G G Y

**A** Q R **A** U I i o the two “a” letters are on the

W r y u i o p k same row of MATR

E B N U e e s w

**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)

The aliquot sum *s(n)* of a natural numbers *n* is the sum of all proper divisors of *n*, that is, all divisors of *n* other than *n* itself. For example, the aliquot sum of 28 is: s(28) = 1 + 2 + 4 + 7 + 14 = 28.

A natural number equal to its aliquot sum is called a *perfect number*. For example, 28 is a perfect number.

Two different natural numbers related in such a way that the aliquot sum of each is equal to the other number are called *amicable numbers*. For example, 220 and 284 are amicable numbers:

s(220) = 1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 = 284

s(284) = 1 + 2 + 4 + 71 + 142 = 220

Numbers whose aliquot sums form a periodic sequence are called *sociable numbers*. They are generalizations of the concepts of perfect numbers and amicable numbers. For example:

s(12496) = 1 + 2 + 4 + 8 + 11 + 16 + 22 + 44 + 71 + 88 + 142 + 176 + 284 + 568 + 781 + 1136 + 1562 + 3124 + 6248 = 14288

s(14288) = 1 + 2 + 4 + 8 + 16 + 19 + 38 + 47 + 76 + 94 + 152 + 188 + 304 + 376 + 752 + 893 + 1786 + 3572 + 7144 = 15472

s(15472) = 1 + 2 + 4 + 8 + 16 + 967 + 1934 + 3868 + 7736 = 14536

s(14536) = 1 + 2 + 4 + 8 + 23 + 46 + 79 + 92 + 158 + 184 + 316 + 632 + 1817 + 3634 + 7268 = 14264

s(14264) = 1 + 2 + 4 + 8 + 1783 + 3566 + 7132 = 12496.

You have to write the isSociable subroutine in ARM assembly language, which receives a natural number *n* in input, and repetitively computes the sequence of aliquot sums, until one of the following conditions occurs:

1. the last computed aliquot sum is equal to the input parameter (i.e., the number is perfect, amicable, or sociable)
2. the last computed aliquot sum is equal to 1 (i.e., the number is neither perfect, amicable, or sociable)
3. 8 terms in the sequence have been computed (i.e., the number is neither perfect or amicable and probably is not sociable).

In the first case, the subroutine returns the length of the sequence. In the last two cases, it returns 0. For example:

* with *n* = 28, the subroutine computes s(28) = 28 and immediately stops returning 1
* with *n* = 220, the subroutine computes s(220) = 284, then s(284) = 220 and returns 2
* with *n* = 12496, the subroutine computes 5 aliquot sums and finally it returns 5
* with *n* = 100, it computes s(100) = 117, s(117) = 65, s(65) = 19, s(19) = 1. It stops returning 0.

In order to compute the aliquot sum of *n*, you have to use the following algorithm:

1. initialize *sum* = 1, *a* = 2
2. while(true):
3. if *n* is divisible by *a*:
4. *b* = *n* / *a*
5. if *a* < *b*:
6. *sum* = *sum* + *a* + *b*
7. else:
8. if *a* == *b*:
9. *sum* = *sum* + *a*
10. exit while and stop
11. *a* = *a* + 1

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

**Question 5** (5 points)

Extend the project developed in the previous question as follows.

Declare an array of unsigned integers with the following values: {8128, 5564, 5400, 14264, 1305184, 1598470, 4938136}.

Initialize timer 1 to raise an interrupt every 2 seconds. In the interrupt handler, each time, a new value of the array is read (i.e., the first time the value 8128 is read; then 5564 and so on. After 7 interrupt calls, the 8th value is 8128 again). Then, the isSociable subroutine is called passing the current value of the array. The output of the subroutine is used to switch on the corresponding led (all other leds are switched off):value 1 -> led4; value 2 -> led 5; value 3 -> led6; … ; value 8 -> led 11.

If the subroutine returns 0, all leds are switched off.

**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.