**Question #1** (4 points)

Let’s consider the Dynamic Scheduling.

You are requested to:

1. Give the definition of Dynamic Scheduling;
2. Describe what Reservation Stations are;
3. Describe in detail each of the fields that make up each of the Reservation Stations entries;
4. Give the definition of Dynamic Disambiguation.

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: it counts the number of values in the vector *vec* that are equal to or multiples of 2 and 3 and then writes the results into the variables *mul2* and *mul3*, respectively. The calculation of the modulus, an operation used here to understand whether a value is a multiple of another, is done using the Barrett reduction, which can be described as follows:

*a mod n = a – [a/n]n*

In the code the module is calculated twice: the first time to check if the previously loaded value is a multiple of 2, the second time to check if it is a multiple of 3. In general, *a* is the value loaded from *vec* and *n* is first 2 and then 3. **Please note that some numbers may be multiples of both values.**

For every instruction, the hexadecimal address of the memory cell storing the instruction is reported.

Assuming that before executing 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 third table).

For all computations, it is suggested to use the two tables 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). Then, report the results on the third table.

*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.*

|  |  |  |  |
| --- | --- | --- | --- |
| ADDR | CODE | | |
|  | *.data* |  |  |
|  | vec: | .byte 3, 4, 6, 15, 2, 8, 12, 9 | # input vector |
|  | mul2: | .space 1 | # number of values equal to or multiples of 2 |
|  | mul3: | .space 1 | # number of values equal to or multiples of 3 |
|  | *.text* |  |  |
| 0x0000 |  | daddui r1, r0, 2 | # initialize the first value of n (2) |
| 0x0004 |  | daddui r2, r0, 3 | # initialize the second value of n (3) |
| 0x0008 |  | daddui r3, r0, 8 | # initialize the value used as a comparator |
| 0x000c |  | daddui r4, r0, 0 | # initialize the pointer |
| 0x0010 |  | daddui r5, r0, 0 | # initialize the counter of values equal to or multiples of 2 |
| 0x0014 |  | daddui r6, r0, 0 | # initialize the counter of values equal to or multiples of 3 |
| 0x0018 | cyc: | beq r3, r4, term | # condition for exiting the cycle |
| 0x001c |  | lbu r7, vec(r4) | # load an element from *vec* |
| 0x0020 |  | ddiv r8, r7, r1 | # Barrett reduction (modulus calculation), *n = 2* |
| 0x0024 |  | dmulu r8, r8, r1 | # Barrett reduction (modulus calculation), *n = 2* |
| 0x0028 |  | dsubu r8, r7, r8 | # Barrett reduction (modulus calculation), *n = 2* |
| 0x002c |  | bnez r8, m3 | # jump to *m3* if the modulus is not equal to zero |
| 0x0030 |  | daddui r5, r5, 1 | # increment the counter of values equal to or multiples of 2 |
| 0x0034 | m3: | ddiv r8, r7, r2 | # Barrett reduction (modulus calculation), *n = 3* |
| 0x0038 |  | dmulu r8, r8, r2 | # Barrett reduction (modulus calculation), *n = 3* |
| 0x003c |  | dsubu r8, r7, r8 | # Barrett reduction (modulus calculation), *n = 3* |
| 0x0040 |  | bnez r8, nxt | # jump to *nxt* if the modulus is not equal to zero |
| 0x0044 |  | daddui r6, r6, 1 | # increment the counter of values equal to or multiples of 3 |
| 0x0048 | nxt: | daddui r4, r4, 1 | # increment the pointer |
| 0x004c |  | j cyc | # next cycle |
| 0x0050 | term: | sb r5, mul2(r0) | # store the number of values equal to or multiples of 2 |
| 0x0054 |  | sb r6, mul3(r0) | # store the number of values equal to or multiples of 3 |
| 0x0058 |  | halt | # termination of the program |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Address | Code | | BHT | Iteration #1 | | Iteration #2 | | Iteration #3 | | Iteration #4 | | Iteration #5 | |
|  |  | | entry # | pred | result | pred | result | pred | result | pred | result | pred | result |
| 0x0000 |  | daddui r1, r0, 2 |  |  |  |  |  |  |  |  |  |  |  |
| 0x0004 |  | daddui r2, r0, 3 |  |  |  |  |  |  |  |  |  |  |  |
| 0x0008 |  | daddui r3, r0, 8 |  |  |  |  |  |  |  |  |  |  |  |
| 0x000c |  | daddui r4, r0, 0 |  |  |  |  |  |  |  |  |  |  |  |
| 0x0010 |  | daddui r5, r0, 0 |  |  |  |  |  |  |  |  |  |  |  |
| 0x0014 |  | daddui r6, r0, 0 |  |  |  |  |  |  |  |  |  |  |  |
| 0x0018 | cyc: | beq r3, r4, term |  |  |  |  |  |  |  |  |  |  |  |
| 0x001c |  | lbu r7, vec(r4) |  |  |  |  |  |  |  |  |  |  |  |
| 0x0020 |  | ddiv r8, r7, r1 |  |  |  |  |  |  |  |  |  |  |  |
| 0x0024 |  | dmulu r8, r8, r1 |  |  |  |  |  |  |  |  |  |  |  |
| 0x0028 |  | dsubu r8, r7, r8 |  |  |  |  |  |  |  |  |  |  |  |
| 0x002c |  | bnez r8, m3 |  |  |  |  |  |  |  |  |  |  |  |
| 0x0030 |  | daddui r5, r5, 1 |  |  |  |  |  |  |  |  |  |  |  |
| 0x0034 | m3: | ddiv r8, r7, r2 |  |  |  |  |  |  |  |  |  |  |  |
| 0x0038 |  | dmulu r8, r8, r2 |  |  |  |  |  |  |  |  |  |  |  |
| 0x003c |  | dsubu r8, r7, r8 |  |  |  |  |  |  |  |  |  |  |  |
| 0x0040 |  | bnez r8, nxt |  |  |  |  |  |  |  |  |  |  |  |
| 0x0044 |  | daddui r6, r6, 1 |  |  |  |  |  |  |  |  |  |  |  |
| 0x0048 | nxt: | daddui r4, r4, 1 |  |  |  |  |  |  |  |  |  |  |  |
| 0x004c |  | j cyc |  |  |  |  |  |  |  |  |  |  |  |
| 0x0050 | term: | sb r5, mul2(r2) |  |  |  |  |  |  |  |  |  |  |  |
| 0x0054 |  | sb r6, mul3(r2) |  |  |  |  |  |  |  |  |  |  |  |
| 0x0058 |  | halt |  |  |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Address | Code | | BHT | Iteration #6 | | Iteration #7 | | Iteration #8 | | Iteration #9 | | Iteration #10 | |
|  |  | | entry # | pred | result | pred | result | pred | result | pred | result | pred | result |
| 0x0000 |  | daddui r1, r0, 2 |  |  |  |  |  |  |  |  |  |  |  |
| 0x0004 |  | daddui r2, r0, 3 |  |  |  |  |  |  |  |  |  |  |  |
| 0x0008 |  | daddui r3, r0, 8 |  |  |  |  |  |  |  |  |  |  |  |
| 0x000c |  | daddui r4, r0, 0 |  |  |  |  |  |  |  |  |  |  |  |
| 0x0010 |  | daddui r5, r0, 0 |  |  |  |  |  |  |  |  |  |  |  |
| 0x0014 |  | daddui r6, r0, 0 |  |  |  |  |  |  |  |  |  |  |  |
| 0x0018 | cyc: | beq r3, r4, term |  |  |  |  |  |  |  |  |  |  |  |
| 0x001c |  | lbu r7, vec(r4) |  |  |  |  |  |  |  |  |  |  |  |
| 0x0020 |  | ddiv r8, r7, r1 |  |  |  |  |  |  |  |  |  |  |  |
| 0x0024 |  | dmulu r8, r8, r1 |  |  |  |  |  |  |  |  |  |  |  |
| 0x0028 |  | dsubu r8, r7, r8 |  |  |  |  |  |  |  |  |  |  |  |
| 0x002c |  | bnez r8, m3 |  |  |  |  |  |  |  |  |  |  |  |
| 0x0030 |  | daddui r5, r5, 1 |  |  |  |  |  |  |  |  |  |  |  |
| 0x0034 | m3: | ddiv r8, r7, r2 |  |  |  |  |  |  |  |  |  |  |  |
| 0x0038 |  | dmulu r8, r8, r2 |  |  |  |  |  |  |  |  |  |  |  |
| 0x003c |  | dsubu r8, r7, r8 |  |  |  |  |  |  |  |  |  |  |  |
| 0x0040 |  | bnez r8, nxt |  |  |  |  |  |  |  |  |  |  |  |
| 0x0044 |  | daddui r6, r6, 1 |  |  |  |  |  |  |  |  |  |  |  |
| 0x0048 | nxt: | daddui r4, r4, 1 |  |  |  |  |  |  |  |  |  |  |  |
| 0x004c |  | j cyc |  |  |  |  |  |  |  |  |  |  |  |
| 0x0050 | term: | sb r5, mul2(r2) |  |  |  |  |  |  |  |  |  |  |  |
| 0x0054 |  | sb r6, mul3(r2) |  |  |  |  |  |  |  |  |  |  |  |
| 0x0058 |  | halt |  |  |  |  |  |  |  |  |  |  |  |

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

**BHT - Final content**

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

**Question 3** (6 points)

A PREMISE: this 8086 exercise is focused on “brute force” approach, i.e., processing a (very) large number of cases/operations. Please focus on the “brute force” approach and **DO NOT TRY to find a FULLY OPTIMIZED algorithm**. Minor optimizations will be acceptable (but will not lead to increases in the score), but please keep in mind that a non-brute-force approach (WHICH IS NOT REQUESTED) could result in a much more difficult algorithm and implementation, i.e., well above what is requested in today’s exam.

Let us assume that to have two **UNSIGNED** values A and B, both greater than zero with A>=B, stored in DX and BL, respectively. By using a brute force approach, please compute the value of the **quotient** of the **division A/B** and have it finally stored in **AX** (note: it is guaranteed that there is no overflow as AX is on 16 bits).

**Kindly note that, for this exercise, all registers are available, but remember not to overwrite the inputs!**

Hint: the brute force approach is based on the basic definition of division, i.e.,

**how many times the value B can be taken from the value A, without (first) obtaining a negative value (after B has been taken)**

Please **clearly & concisely** explain your algorithm. Please assume that A and B are already known inside your program. Failure to provide an explanation and relevant comments to the most important instructions/parts of the program, will result in severe score penalties.

**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)

A maze is saved in a NUM\_ROW \* NUM\_COL matrix of bytes. The character ‘\*’ indicates a wall, and the space indicates a passage. It is guaranteed that all the borders of the maze are walls. The entrance and exit of the maze are indicated with ‘S’. The maze is *perfect*, i.e., there exists only one unique *solution* (i.e., a path from the entrance to the exit).

Example: a maze matrix with NUM\_ROW = 9 and NUM\_COL = 8 is shown below:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| \* | \* | \* | \* | \* | \* | \* | \* |
| \* | S |  |  |  | \* | S | \* |
| \* |  | \* | \* | \* | \* |  | \* |
| \* |  |  |  |  |  |  | \* |
| \* | \* |  | \* |  | \* | \* | \* |
| \* |  |  | \* |  |  |  | \* |
| \* | \* |  | \* |  | \* | \* | \* |
| \* |  |  | \* |  |  |  | \* |
| \* | \* | \* | \* | \* | \* | \* | \* |

Write the deadEndFilling subroutine in ARM assembly language to solve the maze by means of the dead-end filling algorithm. The subroutine receives the following parameters (in the order indicated):

* number of rows
* number of columns
* address of the *maze* matrix

The subroutine scans the maze, and fills in each dead end. In details, for each cell *x* containing a space, if the cell is surrounded by three walls (up, down, left or right), then it is a dead end and becomes a wall (i.e., the content of the cell *x* becomes “\*”). The maze is scanned as long as cells are changed into walls. At the end, the cells that are not wall indicate the unique path.

**Example:** Iteration 1 Iteration 2 Iteration 3

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| \* | \* | \* | \* | \* | \* | \* | \* |  |  |  |  | \* | \* | \* | \* | \* | \* | \* | \* |  |  |  |  | \* | \* | \* | \* | \* | \* | \* | \* |
| \* | S |  |  | \* | \* | S | \* |  |  |  |  | \* | S |  | \* | \* | \* | S | \* |  |  |  |  | \* | S | \* | \* | \* | \* | S | \* |
| \* |  | \* | \* | \* | \* |  | \* |  |  |  |  | \* |  | \* | \* | \* | \* |  | \* |  |  |  |  | \* |  | \* | \* | \* | \* |  | \* |
| \* |  |  |  |  |  |  | \* |  |  |  |  | \* |  |  |  |  |  |  | \* |  |  |  |  | \* |  |  |  |  |  |  | \* |
| \* | \* |  | \* |  | \* | \* | \* |  |  |  |  | \* | \* |  | \* |  | \* | \* | \* |  |  |  |  | \* | \* |  | \* |  | \* | \* | \* |
| \* | \* |  | \* |  |  | \* | \* |  |  |  |  | \* | \* |  | \* |  | \* | \* | \* |  |  |  |  | \* | \* | \* | \* |  | \* | \* | \* |
| \* | \* |  | \* |  | \* | \* | \* |  |  |  |  | \* | \* | \* | \* |  | \* | \* | \* |  |  |  |  | \* | \* | \* | \* |  | \* | \* | \* |
| \* | \* | \* | \* |  |  | \* | \* |  |  |  |  | \* | \* | \* | \* |  | \* | \* | \* |  |  |  |  | \* | \* | \* | \* | \* | \* | \* | \* |
| \* | \* | \* | \* | \* | \* | \* | \* |  |  |  |  | \* | \* | \* | \* | \* | \* | \* | \* |  |  |  |  | \* | \* | \* | \* | \* | \* | \* | \* |

Iteration 4 Iteration 5 Iteration 6

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| \* | \* | \* | \* | \* | \* | \* | \* |  |  |  |  | \* | \* | \* | \* | \* | \* | \* | \* |  |  |  |  | \* | \* | \* | \* | \* | \* | \* | \* |
| \* | S | \* | \* | \* | \* | S | \* |  |  |  |  | \* | S | \* | \* | \* | \* | S | \* |  |  |  |  | \* | S | \* | \* | \* | \* | S | \* |
| \* |  | \* | \* | \* | \* |  | \* |  |  |  |  | \* |  | \* | \* | \* | \* |  | \* |  |  |  |  | \* |  | \* | \* | \* | \* |  | \* |
| \* |  |  |  |  |  |  | \* |  |  |  |  | \* |  |  |  |  |  |  | \* |  |  |  |  | \* |  |  |  |  |  |  | \* |
| \* | \* | \* | \* |  | \* | \* | \* |  |  |  |  | \* | \* | \* | \* |  | \* | \* | \* |  |  |  |  | \* | \* | \* | \* | \* | \* | \* | \* |
| \* | \* | \* | \* |  | \* | \* | \* |  |  |  |  | \* | \* | \* | \* | \* | \* | \* | \* |  |  |  |  | \* | \* | \* | \* | \* | \* | \* | \* |
| \* | \* | \* | \* | \* | \* | \* | \* |  |  |  |  | \* | \* | \* | \* | \* | \* | \* | \* |  |  |  |  | \* | \* | \* | \* | \* | \* | \* | \* |
| \* | \* | \* | \* | \* | \* | \* | \* |  |  |  |  | \* | \* | \* | \* | \* | \* | \* | \* |  |  |  |  | \* | \* | \* | \* | \* | \* | \* | \* |
| \* | \* | \* | \* | \* | \* | \* | \* |  |  |  |  | \* | \* | \* | \* | \* | \* | \* | \* |  |  |  |  | \* | \* | \* | \* | \* | \* | \* | \* |

In the 7-th iteration, no walls are added, so the subroutine ends.

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** (5 points)

The following matrix of chars is declared in C:

char maze[9][8] = {"\*\*\*\*\*\*\*\*", "\*S \*S\*", "\* \*\*\*\* \*", "\* \*", "\*\* \* \*\*\*", "\* \* \*", "\*\* \* \*\*\*", "\* \* \*", "\*\*\*\*\*\*\*\*"};

Pass this matrix to the deadEndFilling subroutine to solve the maze.

Then, show the sequence of movements to arrive to the exit using the leds. Initially, the current cell is either the entrance or the exit (you can choose the one you prefer, both are represented with ‘S’). You look for the only neighboring cell *x* containing a space:

* if *x* is at the right of the current cell, switch on led 8
* if *x* is at the bottom of the current cell, switch on led 9
* if *x* is at the left of the current cell, switch on led 10
* if *x* is at the top of the current cell, switch on led 11

Then, the content of *x* becomes ‘X’ and the loop continues as long as there is a neighbor with a space. When no movements are allowed, it means that you arrived at the other end of the maze.

In the example, if we start at the cell (1, 1), the sequence of leds are: led 9, led 9, led 8, led 8, led 8, led 8 led 8, led 11.

Each led remains lit for 0.5 s. Then, you have to wait 0.5 s before switching on the next led. (Suggestion: you need a timer generate interrupts every 0.5 s.)

**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.