**Question #1** (4 points)

Let’s consider the dynamic branch prediction techniques.

You are requested to:

1. Describe in detail what dynamic branch prediction techniques consist of and their general advantages and disadvantages;
2. Discuss the best dynamic branch prediction technique. Why is the best and how it works;
3. Discuss the remaining dynamic branch prediction technique in detail;
4. Discuss alternatives to dynamic branch prediction techniques.

Write your answer here.

**Question 2** (4 points)

Let's consider a generic processor that executes 32-bit instructions, uses 32-bit addresses, and performs branch predictions via a Branch Target Buffer (BTB) composed of 8 entries.

Assuming that the BTB only stores branch taken predictions and starts from a known state (shown at the bottom of the page), report the content of the BTB after the execution of the following instructions:

|  |  |  |
| --- | --- | --- |
| 1 | div.d f1, f2, f3 | located at the address *0x00048000* |
| 2 | beq f1, f2, lab1 | located at the address *0x00048004*; the branch is taken, and the branch target address is *0x00048010* |
| 3 | add.d f2, f4, f5 | located at the address *0x00048010* |
| 4 | bne f2, f3, lab2 | located at the address *0x00048014*; the branch is taken, and the branch target address is *0x00048030* |
| 5 | mul.d f7, f7, f8 | located at the address *0x00048030* |
| 6 | sub r1, r1, 1 | located at the address *0x00048034* |
| 7 | beqz r1, term | located at the address *0x00048038*; the branch is not taken |
| 8 | jr r2 | located at the address *0x0004803c*; the branch is taken, and the branch target address is *0x00048000* |
| 9 | div.d f1, f2, f3 | located at the address *0x00048000* |
| 10 | beq f1, f2, lab1 | located at the address *0x00048004*; the branch is taken, and the branch target address is *0x00048010* |
| 11 | add.d f2, f4, f5 | located at the address *0x00048010* |
| 12 | bne f2, f3, lab2 | located at the address *0x00048014*; the branch is not taken |
| 13 | sub.d f6, f6, f10 | located at the address *0x00048018* |
| 14 | beqz f6, lab3 | located at the address *0x0004801c*; the branch is taken, and the branch target address is *0x00048038* |
| 15 | beqz r1, term | located at the address *0x00048038*; the branch is taken, and the branch target address is *0x00048040* |
| 16 | halt | located at the address *0x00048040* |

The use of the calculator is forbidden. starting to fill the BTB itself, it is necessary to fill in the contents of this table so as to understand which entry of the BTB each instruction will refer to.

*Hint: To calculate the BTB 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.*

|  |  |  |  |
| --- | --- | --- | --- |
| Instruction | Address in Hex | Address in Binary | Entry No. |
| div.d f1, f2, f3 |  |  |  |
| beq f1, f2, lab1 |  |  |  |
| add.d f2, f4, f5 |  |  |  |
| bne f2, f3, lab2 |  |  |  |
| mul.d f7, f7, f8 |  |  |  |
| sub r1, r1, 1 |  |  |  |
| beqz r1, term |  |  |  |
| jr r2 |  |  |  |
| sub.d f6, f6, f10 |  |  |  |
| beqz f6, lab3 |  |  |  |
| halt |  |  |  |

Then, describe the state of the BTB in each step and, upon completion, report the total number of correct and incorrect predictions.

*Hint: “Smart” cut and paste is accepted/recommended.*

1. BTB initial content (***must not be changed***)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 | 0x00000000 | 0x00000000 |  | 4 | 0x00047f8c | 0x00047fa0 |
| 1 | 0x00047fe0 | 0x00048000 |  | 5 | 0x00047f58 | 0x00047f80 |
| 2 | 0x00047fa4 | 0x00047fb8 |  | 6 | 0x00000000 | 0x00000000 |
| 3 | 0x00000000 | 0x00000000 |  | 7 | 0x00000000 | 0x00000000 |

1. BTB content after ***div.d f1, f2, f3*** execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. BTB content after ***beq f1, f2, lab1*** execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. BTB content after ***add.d f2, f4, f5*** execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. BTB content after ***bne f2, f3, lab2*** execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. BTB content after ***mul.d f7, f7, f8*** execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. BTB content after ***sub r1, r1, 1*** execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. BTB content after ***beqz r1, term*** execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. BTB content after ***jr r2*** execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. BTB content after ***div.d f1, f2, f3*** execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. BTB content after ***beq f1, f2, lab1*** execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. BTB content after ***add.d f2, f4, f5*** execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. BTB content after ***bne f2, f3, lab2*** execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. BTB content after ***sub.d f6, f6, f10*** execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. BTB content after ***beqz f6, lab3*** execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. BTB content after ***beqz r1, term*** execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. BTB content after ***halt*** execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

|  |  |
| --- | --- |
| Total number of correct predictions: \_\_\_\_ | Total number of incorrect predictions: \_\_\_\_ |

**Question 3** (6 points)

A PREMISE: this 8086 exercise is much “shorter” than in other calls, to compensate the slightly longer ARM part. It is requested to solve it by extensively writing your textual response as well as a few-lines-long working program. The program, if less than 15 lines long, does not need to be compiled, but the source-only solution is sufficient.

The following program is supposed to return 0 in AX. However, in some cases the final value in AX could be different from 0. Please **clearly & concisely** explain when it is different from zero and please rewrite it (with the same framework/model) in a fixed form such that at the end AX always stores 0. (Indeed, AX at the beginning stores a value in unsigned binary). Failure to provide an explanation will result in a severe score penalty.

PUSH BX

MOV BX, AX

ADD BX, AX

SHR BX,1

SUB AX, BX

POP BX

**You can write your code here or you can save it in a file 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)

Write the mazeSolver subroutine in ARM assembly language to find the exits of a maze.

The maze is saved as a NUM\_ROW \* NUM\_COL matrix of bytes. The character ‘\*’ indicates a wall, and the space indicates a passage. The exits of the maze are in the outer border only. In particular, the exits in the top border are indicated with ‘n’, in the right border with ‘e’, in the bottom border with ‘s’, and in the left border with ‘w’. Note that a border may have no exits or it may have more than one exit. It is guaranteed that the cells in the borders are either exits or walls.

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

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| \* | n | \* | \* | \* | \* | \* | \* | \* |
| \* |  |  |  | \* |  | \* |  | \* |
| \* |  | \* | \* | \* | \* | \* |  | \* |
| \* |  | \* |  | \* |  |  |  | \* |
| \* |  | \* |  | \* | \* | \* |  | \* |
| \* |  |  |  | \* |  |  |  | \* |
| \* | \* | \* | \* | \* |  | \* |  | \* |
| \* |  |  |  |  |  | \* |  | \* |
| \* | \* | \* | \* | \* | \* | \* | s | \* |

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

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

For each cell of the maze, the subroutine finds the shortest path to reach an exit, giving the directions 'n' (North), 'e' (East), 's' (South), 'w' (West). The algorithm followed by the procedure consists of two phases.

Phase 1) For each cell *x* containing a space:

* if the neighboring cell at the top is a lowercase letter, then the new content of *x* is 'N'
* else if the neighboring cell at the right is a lowercase letter, then the new content of *x* is 'E'
* else if the neighboring cell at the bottom is a lowercase letter, then the new content of *x* is 'S'
* else if the neighboring cell at the left is a lowercase letter, then the new content of *x* is 'W'

Phase 2) For each cell with an uppercase letter, the content of the cell is replaced with the corresponding lowercase letter.

The two phases are repeated as long as the content of the *maze* matrix changes.

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

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| \* | n | \* | \* | \* | \* | \* | \* | \* |  |  |  | \* | n | \* | \* | \* | \* | \* | \* | \* |  |  |  | \* | n | \* | \* | \* | \* | \* | \* | \* |
| \* | n |  |  | \* |  | \* |  | \* |  |  |  | \* | n | w |  | \* |  | \* |  | \* |  |  |  | \* | n | w | w | \* |  | \* |  | \* |
| \* |  | \* | \* | \* | \* | \* |  | \* |  |  |  | \* | n | \* | \* | \* | \* | \* |  | \* |  |  |  | \* | n | \* | \* | \* | \* | \* |  | \* |
| \* |  | \* |  | \* |  |  |  | \* |  |  |  | \* |  | \* |  | \* |  |  |  | \* |  |  |  | \* | n | \* |  | \* |  |  |  | \* |
| \* |  | \* |  | \* | \* | \* |  | \* |  |  |  | \* |  | \* |  | \* | \* | \* |  | \* |  |  |  | \* |  | \* |  | \* | \* | \* |  | \* |
| \* |  |  |  | \* |  |  |  | \* |  |  |  | \* |  |  |  | \* |  |  |  | \* |  |  |  | \* |  |  |  | \* |  |  | s | \* |
| \* | \* | \* | \* | \* |  | \* |  | \* |  |  |  | \* | \* | \* | \* | \* |  | \* | s | \* |  |  |  | \* | \* | \* | \* | \* |  | \* | s | \* |
| \* |  |  |  |  |  | \* | s | \* |  |  |  | \* |  |  |  |  |  | \* | s | \* |  |  |  | \* |  |  |  |  |  | \* | s | \* |
| \* | \* | \* | \* | \* | \* | \* | s | \* |  |  |  | \* | \* | \* | \* | \* | \* | \* | s | \* |  |  |  | \* | \* | \* | \* | \* | \* | \* | s | \* |

Iteration 4 Iteration 5 Iteration 11

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| \* | n | \* | \* | \* | \* | \* | \* | \* |  |  |  | \* | n | \* | \* | \* | \* | \* | \* | \* |  |  |  | \* | n | \* | \* | \* | \* | \* | \* | \* |
| \* | n | w | w | \* |  | \* |  | \* |  |  |  | \* | n | w | w | \* |  | \* |  | \* |  |  |  | \* | n | w | w | \* |  | \* | s | \* |
| \* | n | \* | \* | \* | \* | \* |  | \* |  |  |  | \* | n | \* | \* | \* | \* | \* |  | \* |  |  |  | \* | n | \* | \* | \* | \* | \* | s | \* |
| \* | n | \* |  | \* |  |  |  | \* |  |  |  | \* | n | \* |  | \* |  |  | s | \* |  |  |  | \* | n | \* | s | \* | e | e | s | \* |
| \* | n | \* |  | \* | \* | \* | s | \* |  |  |  | \* | n | \* |  | \* | \* | \* | s | \* | … | | | \* | n | \* | s | \* | \* | \* | s | \* |
| \* |  |  |  | \* |  | e | s | \* |  |  |  | \* | n |  |  | \* | e | e | s | \* |  |  |  | \* | n | w | w | \* | e | e | s | \* |
| \* | \* | \* | \* | \* |  | \* | s | \* |  |  |  | \* | \* | \* | \* | \* |  | \* | s | \* |  |  |  | \* | \* | \* | \* | \* | n | \* | s | \* |
| \* |  |  |  |  |  | \* | s | \* |  |  |  | \* |  |  |  |  |  | \* | s | \* |  |  |  | \* | e | e | e | e | n | \* | s | \* |
| \* | \* | \* | \* | \* | \* | \* | s | \* |  |  |  | \* | \* | \* | \* | \* | \* | \* | s | \* |  |  |  | \* | \* | \* | \* | \* | \* | \* | s | \* |

The procedure returns the length of the longest path, i.e., the number of iterations executed. In the example, the return value is 11.

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)

We want to generate a maze randomly and then call the subroutine developed in the previous exercise to find the shortest path towards the exits.

A NUM\_ROWS \* NUM\_COLUMNS matrix of chars is declared in C. NUM\_ROWS and NUM\_COLUMNS are defined as constants, for example NUM\_ROWS = 10 and NUM\_COLUMNS = 8.

The value '\*' is assigned to the four cells in the corners. For each remaining cell, a pseudo-random value in the range [0, 100] is generated:

* for every cell in the upper border, if the pseudo-random value is lower than 90, the cell is initialized with '\*', otherwise with 'n'.
* for every cell in the right border, if the pseudo-random value is lower than 90, the cell is initialized with '\*', otherwise with 'e'.
* for every cell in the lower border, if the pseudo-random value is lower than 90, the cell is initialized with '\*', otherwise with 's'.
* for every cell in the left border, if the pseudo-random value is lower than 90, the cell is initialized with '\*', otherwise with 'w'.
* for each other cell, if the pseudo-random value is lower than 60, the cell is initialized with ' ', otherwise with '\*'.

The sequence of pseudo-random numbers is obtained by means of the following linear congruential generator: value\_new = (value\_old \* 18) % 101, where % indicates the modulo operation.

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

1. In the main function, start timer 0, without generating interrupts and stopping the timer.
2. When the user presses the button Key2, we read the current value of timer 0. This is a truly random number and we use it as a seed to compute the first pseudo-random number. Then, each subsequent pseudo-random number is obtained using the previous pseudo-random number.  
   Example: if timer 0 is 300 and NUM\_COLUMNS = 8, then 6 pseudo-random numbers are needed for generating the first row of the matrix (because the corners are fixed). The sequence of pseudo-random numbers is:
   * value1 = (300 \* 18) % 101 = 47
   * value2 = (47 \* 18) % 101 = 38
   * value3 = (38 \* 18) % 101 = 78
   * value4 = (78 \* 18) % 101 = 91
   * value5 = (91 \* 18) % 101 = 22
   * value6 = (22 \* 18) % 101 = 93

The first row of the matrix is \*\*\*\*n\*n\*

1. Finally, call the mazeSolver subroutine.

Steps 2 and 3 have to be implemented inside the handler of button Key2.