**Question #1** (4 points)

Let’s consider the static branch prediction techniques.

You are requested to:

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

Write your answer here.

Q1-Static branch prediction are techniques based on compiler using the preliminary analysis of the code. Techniques are:

Q1.1-predict branch as always taken

Q1.2-Predict based on the branch direction: Forward branches are often untaken and backward branches are often taken.

Q1.3-predict based on profile information coming from earlier runs. We provide some input stimula for the program—then we run the program limited number of times using these inputs—we gain statistics based on these runs—we use these statistics to predict branches

**Advantage:** static techniques are simple to implement and NOT expensive

**Disadvantage:** They have a variable rate of misprediction and that’s why they aren’t trustworthy for branch prediction.

Q2-The best one is Profile information which has a smaller rate of misprediction.

Q3- I wrote on top

-predict branch as always taken

-Predict based on the branch direction: Forward branches are often untaken and backward branches are often taken.

S-predict based on profile information coming from earlier runs. We provide some input stimula for the program—then we run the program limited number of times using these inputs—we gain statistics based on these runs—we use these statistics to predict branches

Q4-Instead of static techniques, we use Dynamic solutions that are implemented by Hardware and are based on the branch instruction address and some of them are: Branch History Table,2bit prediction-n bit prediction and Branch Target Buffer

**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 *0x0000370c* |
| 2 | beq f1, f2, lab1 | located at the address *0x00003710*; the branch is taken, and the branch target address is *0x00003724* |
| 3 | add.d f2, f4, f5 | located at the address *0x00003724* |
| 4 | bne f2, f3, lab2 | located at the address *0x00003728*; the branch is taken, and the branch target address is *0x0000373c* |
| 5 | mul.d f7, f7, f8 | located at the address *0x0000373c* |
| 6 | sub r1, r1, 1 | located at the address *0x00003740* |
| 7 | beqz r1, term | located at the address *0x00003744*; the branch is not taken |
| 8 | jr r2 | located at the address *0x00003748*; the branch is taken, and the branch target address is *0x0000370c* |
| 9 | div.d f1, f2, f3 | located at the address *0x0000370c* |
| 10 | beq f1, f2, lab1 | located at the address *0x00003710*; the branch is taken, and the branch target address is *0x00003724* |
| 11 | add.d f2, f4, f5 | located at the address *0x00003724* |
| 12 | bne f2, f3, lab2 | located at the address *0x00003728*; the branch is not taken |
| 13 | sub.d f6, f6, f10 | located at the address *0x0000372c* |
| 14 | beqz f6, lab3 | located at the address *0x00003730*; the branch is taken, and the branch target address is *0x00003744* |
| 15 | beqz r1, term | located at the address *0x00003744*; the branch is taken, and the branch target address is *0x00003750* |
| 16 | halt | located at the address *0x00003750* |

The use of the calculator is forbidden. Before 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 | *0x0000370c* | 2\_0000 0000 0000 0000 0011 0111 0000 1100 | No entry- only branches |
| beq f1, f2, lab1 | *0x00003710* | 2\_0000 0000 0000 0000 0011 0111 0001 0000 | 4 |
| add.d f2, f4, f5 | *0x00003724* | 2\_0000 0000 0000 0000 0011 0111 0010 0100 | No entry- only branches |
| bne f2, f3, lab2 | *0x00003728* | 2\_0000 0000 0000 0000 0011 0111 0010 1000 | 2 |
| mul.d f7, f7, f8 | *0x0000373c* | 2\_0000 0000 0000 0000 0011 0111 0011 1100 | No entry- only branches |
| sub r1, r1, 1 | *0x00003740* | 2\_0000 0000 0000 0000 0011 0111 0100 0000 | No entry- only branches |
| beqz r1, term | *0x00003744* | 2\_0000 0000 0000 0000 0011 0111 0100 0100 | 1 |
| jr r2 | *0x00003748* | 2\_0000 0000 0000 0000 0011 0111 0100 1000 | 2 |
| sub.d f6, f6, f10 | *0x0000372c* | 2\_0000 0000 0000 0000 0011 0111 0010 1100 | No entry- only branches |
| beqz f6, lab3 | *0x00003730* | 2\_0000 0000 0000 0000 0011 0111 0011 0000 | 4 |
| halt | *0x00003750* | 2\_0000 0000 0000 0000 0011 0111 0101 0000 | No entry- only branches |

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 | 0x00003654 | 0x00003670 |
| 1 | 0x00000000 | 0x00000000 |  | 5 | 0x00000000 | 0x00000000 |
| 2 | 0x0000364c | 0x0000365c |  | 6 | 0x00003634 | 0x00003644 |
| 3 | 0x0000367c | 0x0000370c |  | 7 | 0x00000000 | 0x00000000 |

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 | 0x00000000 | 0x00000000 |  | 4 | 0x00003654 | 0x00003670 |
| 1 | 0x00000000 | 0x00000000 |  | 5 | 0x00000000 | 0x00000000 |
| 2 | 0x0000364c | 0x0000365c |  | 6 | 0x00003634 | 0x00003644 |
| 3 | 0x0000367c | 0x0000370c |  | 7 | 0x00000000 | 0x00000000 |

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 | 0x00000000 | 0x00000000 |  | 4 | *0x00003710* | *0x00003724* |
| 1 | 0x00000000 | 0x00000000 |  | 5 | 0x00000000 | 0x00000000 |
| 2 | 0x0000364c | 0x0000365c |  | 6 | 0x00003634 | 0x00003644 |
| 3 | 0x0000367c | 0x0000370c |  | 7 | 0x00000000 | 0x00000000 |

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 | 0x00000000 | 0x00000000 |  | 4 | *0x00003710* | *0x00003724* |
| 1 | 0x00000000 | 0x00000000 |  | 5 | 0x00000000 | 0x00000000 |
| 2 | 0x0000364c | 0x0000365c |  | 6 | 0x00003634 | 0x00003644 |
| 3 | 0x0000367c | 0x0000370c |  | 7 | 0x00000000 | 0x00000000 |

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 | 0x00000000 | 0x00000000 |  | 4 | *0x00003710* | *0x00003724* |
| 1 | 0x00000000 | 0x00000000 |  | 5 | 0x00000000 | 0x00000000 |
| 2 | *0x00003728* | *0x0000373c* |  | 6 | 0x00003634 | 0x00003644 |
| 3 | 0x0000367c | 0x0000370c |  | 7 | 0x00000000 | 0x00000000 |

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 | 0x00000000 | 0x00000000 |  | 4 | *0x00003710* | *0x00003724* |
| 1 | 0x00000000 | 0x00000000 |  | 5 | 0x00000000 | 0x00000000 |
| 2 | *0x00003728* | *0x0000373c* |  | 6 | 0x00003634 | 0x00003644 |
| 3 | 0x0000367c | 0x0000370c |  | 7 | 0x00000000 | 0x00000000 |

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 | 0x00000000 | 0x00000000 |  | 4 | *0x00003710* | *0x00003724* |
| 1 | 0x00000000 | 0x00000000 |  | 5 | 0x00000000 | 0x00000000 |
| 2 | *0x00003728* | *0x0000373c* |  | 6 | 0x00003634 | 0x00003644 |
| 3 | 0x0000367c | 0x0000370c |  | 7 | 0x00000000 | 0x00000000 |

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 | 0x00000000 | 0x00000000 |  | 4 | *0x00003710* | *0x00003724* |
| 1 | *0x00003744* | 0x00000000 |  | 5 | 0x00000000 | 0x00000000 |
| 2 | *0x00003728* | *0x0000373c* |  | 6 | 0x00003634 | 0x00003644 |
| 3 | 0x0000367c | 0x0000370c |  | 7 | 0x00000000 | 0x00000000 |

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 | 0x00000000 | 0x00000000 |  | 4 | *0x00003710* | *0x00003724* |
| 1 | *0x00003744* | 0x00000000 |  | 5 | 0x00000000 | 0x00000000 |
| 2 | *0x00003748* | *0x0000370c* |  | 6 | 0x00003634 | 0x00003644 |
| 3 | 0x0000367c | 0x0000370c |  | 7 | 0x00000000 | 0x00000000 |

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 | 0x00000000 | 0x00000000 |  | 4 | *0x00003710* | *0x00003724* |
| 1 | *0x00003744* | 0x00000000 |  | 5 | 0x00000000 | 0x00000000 |
| 2 | *0x00003748* | *0x0000370c* |  | 6 | 0x00003634 | 0x00003644 |
| 3 | 0x0000367c | 0x0000370c |  | 7 | 0x00000000 | 0x00000000 |

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 | 0x00000000 | 0x00000000 |  | 4 | *0x00003710* | *0x00003724* |
| 1 | *0x00003744* | 0x00000000 |  | 5 | 0x00000000 | 0x00000000 |
| 2 | *0x00003748* | *0x0000370c* |  | 6 | 0x00003634 | 0x00003644 |
| 3 | 0x0000367c | 0x0000370c |  | 7 | 0x00000000 | 0x00000000 |

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 | 0x00000000 | 0x00000000 |  | 4 | *0x00003710* | *0x00003724* |
| 1 | *0x00003744* | 0x00000000 |  | 5 | 0x00000000 | 0x00000000 |
| 2 | *0x00003748* | *0x0000370c* |  | 6 | 0x00003634 | 0x00003644 |
| 3 | 0x0000367c | 0x0000370c |  | 7 | 0x00000000 | 0x00000000 |

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 | 0x00000000 | 0x00000000 |  | 4 | *0x00003710* | *0x00003724* |
| 1 | *0x00003744* | 0x00000000 |  | 5 | 0x00000000 | 0x00000000 |
| 2 | *0x00003728* | *0x0000372c* |  | 6 | 0x00003634 | 0x00003644 |
| 3 | 0x0000367c | 0x0000370c |  | 7 | 0x00000000 | 0x00000000 |

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 | 0x00000000 | 0x00000000 |  | 4 | *0x00003710* | *0x00003724* |
| 1 | *0x00003744* | 0x00000000 |  | 5 | 0x00000000 | 0x00000000 |
| 2 | *0x00003728* | *0x0000372c* |  | 6 | 0x00003634 | 0x00003644 |
| 3 | 0x0000367c | 0x0000370c |  | 7 | 0x00000000 | 0x00000000 |

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 | 0x00000000 | 0x00000000 |  | 4 | *0x00003730* | *0x00003744* |
| 1 | *0x00003744* | 0x00000000 |  | 5 | 0x00000000 | 0x00000000 |
| 2 | *0x00003728* | *0x0000372c* |  | 6 | 0x00003634 | 0x00003644 |
| 3 | 0x0000367c | 0x0000370c |  | 7 | 0x00000000 | 0x00000000 |

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 | 0x00000000 | 0x00000000 |  | 4 | *0x00003730* | *0x00003744* |
| 1 | *0x00003744* | *0x00003750* |  | 5 | 0x00000000 | 0x00000000 |
| 2 | *0x00003728* | *0x0000372c* |  | 6 | 0x00003634 | 0x00003644 |
| 3 | 0x0000367c | 0x0000370c |  | 7 | 0x00000000 | 0x00000000 |

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Entry No. | Address | Target |  | Entry No. | Address | Target |
| 0 | 0x00000000 | 0x00000000 |  | 4 | *0x00003730* | *0x00003744* |
| 1 | *0x00003744* | *0x00003750* |  | 5 | 0x00000000 | 0x00000000 |
| 2 | *0x00003728* | *0x0000372c* |  | 6 | 0x00003634 | 0x00003644 |
| 3 | 0x0000367c | 0x0000370c |  | 7 | 0x00000000 | 0x00000000 |

|  |  |
| --- | --- |
| Total number of correct predictions: \_\_2\_\_ | Total number of incorrect predictions: \_6\_\_\_ |

**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

SHR BX,1

ADD BX, BX

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 exploreMaze subroutine in ARM assembly language to find the exit of a maze.

The maze is saved as a NUM\_ROW \* NUM\_COL matrix of bytes. The character ‘X’ indicates a wall, and the space indicates a passage. It is guaranteed that all the borders of the maze are walls.

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

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| X | X | X | X | X | X | X | X | X |
| X |  |  |  | X |  |  |  | X |
| X |  | X | X | X | X | X |  | X |
| X |  | X |  |  |  |  |  | X |
| X |  |  |  | X |  |  |  | X |
| X | X | X | X | X |  | X |  | X |
| X |  |  |  |  |  | X |  | X |
| X | X | X | X | X | X | X | X | X |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 9 | A | B | C | D | E | F | 10 | 11 |
| 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 1A |
| 1B | 1C | 1D | 1E | 1F | 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 | 28 | 29 | 2A | 2B | 2C |
| 2D | 2E | 2F | 30 | 31 | 32 | 33 | 34 | 35 |
| 36 | 37 | 38 | 39 | 3A | 3B | 3C | 3D | 3E |
| 3F | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |

For the sake of clarity, the hexadecimal numeration of the cells according to the row-major order is reported on the right.

Assuming that the cell in the top left corner has coordinates (0, 0), the initial cell in the maze is at (1, 1). In the example, the initial cell is numbered 0xA and highlighted in green.

The exit of the maze has coordinates (NUM\_ROW – 2, NUM\_COL – 2). In the example, it is numbered 3D and highlighted in red.

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

* number of rows
* number of columns
* address of the *maze* matrix
* address of an empty array of bytes, which will contain the path from the initial position to the final position.

The subroutine implements the following algorithm to move through the maze:

1. the current cell in the maze matrix is marked as ‘V’ in order to signal that it has been visited
2. if the cell at the right (same row, next column) is not a wall and has not been already visited:
   1. save the index (i.e., the hexadecimal number) of the current cell in the *path* array
   2. current cell = cell at the right
3. else if the cell at the left (same row, previous column) is not a wall and has not been visited:
   1. save the index of the current cell in the *path* array
   2. current cell = cell at the LEFT
4. else if the cell at the top (previous row, same column) is not a wall and has not been visited:
   1. save the index of the current cell in the *path* array
   2. current cell = cell at the TOP
5. else if the cell at the bottom (next row, same column) is not a wall and has not been visited:
   1. save the index of the current cell in the *path* array
   2. current cell = cell at the BOTTOM
6. else (i.e., no movements allowed):
   1. if the *path* array contains one or more elements:
      * current cell = last element of the *path* array
      * remove the last element of the *path* array (e.g., you can overwrite it with 0)
   2. else:
      * return 0
7. if current cell = final position
   1. return 1
8. else:
   1. go to step 1.

In the considered example, the exploreMaze subroutine will return 1 (meaning that a path to the end of the maze exists) and write in the *path* array the following values: A, 13, 1C, 25, 26, 27, 1E, 1F, 20, 21, 22, 2B, 34, 3D. If the exit can not be reached from the initial cell, the subroutine returns 0 and the array will be empty.

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 a path.

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. Furthermore, an unitialized array of chars with NUM\_ROWS \* NUM\_COLUMNS is defined.

The value 'X' is assigned to all the cells in the borders. The cells (1, 1) and (NUM\_ROWS -2, NUM\_COLUMNS – 2) are initialized with the value ' '. For each remaining cell, a pseudo-random value in the range [0, 72] is generated: if the pseudo-random value is lower than 40, the cell is initialized with ' ', otherwise with 'X'. The sequence of pseudo-random numbers is obtained by means of the following linear congruential generator: value\_new = (value\_old \* 11 + 6) % 73, 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 Key1, 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 200, then the sequence of pseudo-random numbers is
   * value1 = (200 \* 11 + 6) % 73 = 16
   * value2 = (16 \* 11 + 6) % 73 = 36
   * value3 = (36 \* 11 + 6) % 73 = 37
   * value4 = (37 \* 11 + 6) % 73 = 48
   * …

With that sequence of values, the first three cells will be initialize with ' ', and the last one with 'X'.

1. Finally, call the exploreMaze subroutine.

Steps 2 and 3 have to be implemented inside the handler of button Key1.