**Question #1** (4 points)

Let’s consider the Branch Prediction Techniques.

You are requested to:

1. Discuss why branch prediction techniques are used and how they can be classified;
2. Discuss the general characteristics of the Branch History Table;
3. Draw and explain the basic implementation of a Branch History Table;
4. Discuss possible Branch History Table improvements and alternatives.

**Write your answer here.**

**Question #2** (4 points)

Let's consider a MIPS architecture using a *Branch History Table* (BHT) composed of *16 1-bit entries*. Let's assume that this architecture executes the following code, which counts the number of values equal to zero and one contained in vector *vec* and then writes the result in the variables *res0* and *res1*. 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: | . byte | 0, 2, 2, 1, 0, 0, 1 | # input vector |
| res0: | .space | 1 | # number of values equal to zero |
| res1: | .space | 1 | # number of values equal to one |
| … |  |  |  |
| 0x0000 |  | daddui r1, r0, 1 | # initialize the value used as a comparator |
| 0x0004 |  | daddui r2, r0, 0 | # initialize the pointer |
| 0x0008 |  | daddui r3, r0, 0 | # initialize the counter of elements equal to zero |
| 0x000c |  | daddui r4, r0, 0 | # initialize the counter of elements equal to one |
| 0x0010 |  | daddui r5, r0, 6 | # initialize the value containing the size of the vector |
| 0x0014 | cyc: | beq r2, r5, term | # condition for exiting the cycle |
| 0x0018 |  | lb r6, vec(r2) | # load an element from vec |
| 0x001c |  | beq r6, r0, eq0 | # check if the loaded value is equal to zero |
| 0x0020 |  | beq r6, r1, eq1 | # check if the loaded value is equal to one |
| 0x0024 |  | j nxt | # values different from zero or one are skipped |
| 0x0028 | eq0: | daddui r3, r3, 1 | # increment the counter of elements equal to zero |
| 0x002c |  | j nxt | # go to the next cycle |
| 0x0030 | eq1: | daddui r4, r4, 1 | # increment the counter of elements equal to one |
| 0x0034 | nxt: | daddui r2, r2, 1 | # increment the pointer |
| 0x0038 |  | j cyc | # next cycle |
| 0x003c | term: | sb r3, res0(r2) | # store the result |
| 0x0040 |  | sb r4, res1(r2) | # store the result |
| 0x0044 | 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, 1 |  |  |  |  |  |  |  |  |  |
| 0x0004 |  | daddui r2, r0, 0 |  |  |  |  |  |  |  |  |  |
| 0x0008 |  | daddui r3, r0, 0 |  |  |  |  |  |  |  |  |  |
| 0x000c |  | daddui r4, r0, 0 |  |  |  |  |  |  |  |  |  |
| 0x0010 |  | daddui r5, r0, 6 |  |  |  |  |  |  |  |  |  |
| 0x0014 | cyc: | beq r2, r5, term | 5 | not | not | not | not | not | not | Not | Not |
| 0x0018 |  | lb r6, vec(r2) |  |  |  |  |  |  |  |  |  |
| 0x001c |  | beq r6, r0, eq0 | 7 | not | taken | taken | not | not | not | Not | not |
| 0x0020 |  | beq r6, r1, eq1 | 8 |  |  | not | not | not | not | not | taken |
| 0x0024 |  | j nxt |  |  |  |  |  |  |  |  |  |
| 0x0028 | eq0: | daddui r3, r3, 1 |  |  |  |  |  |  |  |  |  |
| 0x002c |  | j nxt |  |  |  |  |  |  |  |  |  |
| 0x0030 | eq1: | daddui r4, r4, 1 |  |  |  |  |  |  |  |  |  |
| 0x0034 | nxt: | daddui r2, r2, 1 |  |  |  |  |  |  |  |  |  |
| 0x0038 |  | j cyc |  |  |  |  |  |  |  |  |  |
| 0x003c | term: | sb r3, res0(r2) |  |  |  |  |  |  |  |  |  |
| 0x0040 |  | sb r4, res1(r2) |  |  |  |  |  |  |  |  |  |
| 0x0044 | 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, 1 |  |  |  |  |  |  |  |  |  |
| 0x0004 |  | daddui r2, r0, 0 |  |  |  |  |  |  |  |  |  |
| 0x0008 |  | daddui r3, r0, 0 |  |  |  |  |  |  |  |  |  |
| 0x000c |  | daddui r4, r0, 0 |  |  |  |  |  |  |  |  |  |
| 0x0010 |  | daddui r5, r0, 6 |  |  |  |  |  |  |  |  |  |
| 0x0014 | cyc: | beq r2, r5, term | 5 | Not | Not | Not | not | not | not | not | taken |
| 0x0018 |  | lb r6, vec(r2) |  |  |  |  |  |  |  |  |  |
| 0x001c |  | beq r6, r0, eq0 | 7 | Not | taken | taken | taken | taken | not |  |  |
| 0x0020 |  | beq r6, r1, eq1 | 8 |  |  |  |  | taken | taken |  |  |
| 0x0024 |  | j nxt |  |  |  |  |  |  |  |  |  |
| 0x0028 | eq0: | daddui r3, r3, 1 |  |  |  |  |  |  |  |  |  |
| 0x002c |  | j nxt |  |  |  |  |  |  |  |  |  |
| 0x0030 | eq1: | daddui r4, r4, 1 |  |  |  |  |  |  |  |  |  |
| 0x0034 | nxt: | daddui r2, r2, 1 |  |  |  |  |  |  |  |  |  |
| 0x0038 |  | j cyc |  |  |  |  |  |  |  |  |  |
| 0x003c | term: | sb r3, res0(r2) |  |  |  |  |  |  |  |  |  |
| 0x0040 |  | sb r4, res1(r2) |  |  |  |  |  |  |  |  |  |
| 0x0044 | halt |  |  |  |  |  |  |  |  |  |  |

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

**BHT - Final content**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Entry 0 | 0 |  | Entry 4 | 0 |  | Entry 8 | 1 |  | Entry 12 | 0 |
| Entry 1 | 0 |  | Entry 5 | 1 |  | Entry 9 | 0 |  | Entry 13 | 0 |
| Entry 2 | 0 |  | Entry 6 | 0 |  | Entry 10 | 0 |  | Entry 14 | 0 |
| Entry 3 | 0 |  | Entry 7 | 0 |  | Entry 11 | 0 |  | Entry 15 | 0 |

**Question 3** (6 points)

Two sparse matrices 200\*200 each one storing 4 binary values on 7 bits different from zero, are represented by two arrays of 4 24-bits-entries each,

EL1 DB 4\*3 DUP(?)

EL2 DB 4\*3 DUP(?)

where each entry stores

Index i (on 8 bits), index j (on 8 bits), value different from zero (on 8 bits, but smaller than 128)

For example

130, 189, 7 → refers to cell (130,189) storing the value 7

It is requested to write an 8086 program which computes the addition of the two sparse matrices and represents the result in the same form/representation in the array

ADDITION DB 4\*2\*3 DUP(?)

Please observe/comply with the following:

* Sparse matrices are matrices where all elements are zero, except a few cells which are different from zero; sparse matrices are represented (as above) by storing the indexes of the cells different from zero together with the corresponding value.
* Sparse matrices are NEVER represented as full matrices.
* Solving the problem by passing through a full matrix is not permitted, as all operations should be made on the sparse matrices representations EL1, EL2 and ADDITION
* Having the values of EL1 and EL2 smaller than 128 guarantees that an addition will not generate an overflow

Example:

|  |  |
| --- | --- |
| EL1 | EL2 |
| 101, 97, 6 | 33, 85, 10 |
| 33, 85, 2 | 61, 13, 99 |
| 68, 86, 9 | 77, 77, 77 |
| 15, 9, 2 | 68, 86, 100 |

Leads to

|  |
| --- |
| ADDITION |
| 101, 97, 6 |
| 33, 85, 12 |
| 68, 86, 109 |
| 15, 9, 2 |
| 61, 13, 99 |
| 7, 77, 77 |

Observe that there are two elements in EL1 and EL2 which have the same i and j, and therefore do add up; on the other hand, these present only in EL1 or EL2, are simply copied to ADDITION.

**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 instruction set of ARM v7-M contains an instruction for the signed division with 32-bit operands: SDIV Rd, Rn, Rm where Rd is the quotient, Rn is the dividend, and Rm is the divisor. You have to write the SDIV64 subroutine in ARM assembly language, which computes the signed division between a 64-bit dividend and a 32-bit divisor, and returns a 32-bit quotient. The subroutine receives in input:

* U: the upper half of the dividend
* L: the lower half of the dividend
* D: the divisor

The following algorithm is used to compute the quotient Q:

1. if the dividend and the divisor have the same sign:
2. flag = true
3. else
4. flag = false
5. if the dividend is negative:
6. dividend = opposite(dividend)
7. if the divisor is negative:
8. divisor = opposite(divisor)
9. for i = 0 to 31:
10. double the dividend: U = 2U, L = 2L
11. if U >= D:
12. U = U – D
13. set i-th bit of Q to 1
14. else:
15. set i-th bit of Q to 0
16. if flag == false
17. Q = opposite(Q)
18. return Q

You can not use multiplication. In particular:

* At points 6, 8, and 17, you have to compute the two's complement (instead of multiplying by -1) to get the opposite of the number. Two's complement is achieved by inverting all bits and adding 1 to the entire inverted number.
* At point 10, you have to use the left shift (instead of multiplication) to double the dividend. The most significant bit of L becomes the least significant bit of U.

At points 13 and 15, the bits are counted from left to right. E.g., bit 0 is the most significant one.

An example is given here considering 4-bit registers (the dividend is expressed with 8 bits). You can easily extend it to the 32-bit case.

Example: U = 1101, L = 1010, D = 1011

dividend and the divisor have the same sign => flag = true

dividend is negative => U = 0010, L = 0110

divisor is negative => D = 0101

Iteration i = 0

U = 0100, L = 1100

U < D -> bit 0 of Q is 0

Iteration i = 1

U = 1001, L = 1000

U >= D -> U = 1001 – 0101 = 0100 bit 1 of Q is 1

Iteration i = 2

U = 1001, L = 0000

U >= D -> U = 1001 – 0101 = 0100 bit 2 of Q is 1

Iteration i = 3

U = 1000, L = 0000

U >= D -> U = 1000 – 0101 = 0011 bit 3 of Q is 1

The quotient is Q = 0111

flag = true => Q = 0111

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)

Write the SDIV64S subroutine, which extends the subroutine of the previous exercise by setting the flags in the Program Status Register (PSR). The bits in PSR are:

* Bit 31 (most significant bit): negative flag
* Bit 30: zero flag
* Bit 29: carry flag
* Bit 28: overflow flag
* Bit 27: sticky saturation flag
* Bit 26, 25, 15-10: interrupt-continuable instruction (ICI) bits, IF-THEN instruction status bit
* Bit 24: Thumb state
* Bit 8-0: exception number.

At the end of the procedure, the flags have to be:

* negative flag = first bit of the quotient
* zero flag = 1 if the quotient is zero; zero flag = 0 if the quotient is different from 0
* carry flag = 0
* overflow flag = 1 if the quotient can not be stored in 32 bits; overflow flag = 0 otherwise.

The procedure does not change the other bits in PSR.

The value of the overflow flag can be determined at the beginning of the subroutine by comparing:

* |U|: the absolute value of the upper half of the dividend U (i.e., U if U is positive, -U if U is negative)
* |D|/2: half of the absolute value of the divisor D.

If |D|/2 > |U|, then the overflow flag is 0, otherwise is 1.

Note: you do not have to copy duplicated code from the previous exercise. Write new instructions; you can write comments to indicate duplicated section of code.