Computer Architectures 02LSEOV	Delivery date: October 30th 2024, 11.59 PM
Laboratory 4 REVALOR RICCARDO s339423	Expected delivery of lab_04.zip must include: - This file, filled with information and possibly compiled in a pdf format.

This lab will focus on putting into practice some of the concepts seen during the lectures. In particular, the proposed exercises will focus on the functioning of caches, superscalar processors, and branch prediction techniques.
These exercises are similar to those that can be found during the exam. In this regard, it should be noted that, during the exam itself, the use of the calculator is forbidden.

Question #1

Let's consider a MIPS architecture using a *Branch History Table* (BHT) composed of ***8*** *1-bit entries*. It executes the instructions reported in the table below, which also indicates the hex address of the corresponding memory cell. Assuming that before executing the code fragment the BHT is full of null values (corresponding to the prediction Not Taken, NT), you are asked to compute the number of mispredicted branches during the execution of the code.To calculate the BHT entry corresponding to each branch instruction, remember to exclude the last two bits from the instruction address as they are always 0. Write in the highlighted cells whether the prediction (PRED) of the current branch and the outcome (RES) of the software is *Taken* (T) or *NotTaken* (NT).

If I make a jump og just 4 bytes (ex from 0x004c to 0x0050), the branch was NOT taken.
If I make a jump > 4 bytes or jump backwards the branch was TAKEN.

Address	Code	Address in Binary	BHT Entry #	PRED	RES
0x0048	dmul r4, r5, r6				
0x004c	bne r4, r1, lab1	... *0000 0000 010**01**100*	*3*	*NT*	*NT*
0x0050	daddui r7, r7, 1				
0x0054	j chk (*)				
0x0068	daddi r5, r5, -1				
0x006c	bnez r4, term	... *0000 0000 011**01**100*	*3*	*NT*	*T*
0x0048	dmul r4, r5, r6				
0x004c	bne r4, r1, lab1	... *0000 0000 010**01**100*	3	*T*	*T*
0x0058	bne r4, r2, lab2	*...0000000001011**000***	6	*NT*	*NT*
0x005c	daddui r8, r7, 1				
0x0060	j chk (*)				
0x0068	daddi r5, r5, -1				
0x006c	bnez r4, term	... *0000 0000 010**01**100*	3	*T*	*T*
0x0048	dmul r4, r5, r6				
0x004c	bne r4, r1, lab1	... *0000000001**01**100*	3	*T*	*T*
0x0058	bne r4, r2, lab2	... *0000000001011**000***	6	*NT*	*T*
0x0064	daddui r9, r8, 1				
0x0068	daddi r5, r5, -1				
0x006c	bnez r4, term	... *0000 0000 010**01**100*	3	*T*	*NT*

0x0070	halt				

Note: the instructions marked with (*) are different: they have different addresses.

a) Based on the results obtained, fill in the BHT content at the end of the run and indicate the number of mispredicted branches.

The number of mispredicted branches during the execution of the code is: 3

BHT - Final content

Entry 0	NT	Entry 2	NT	Entry 4	NT	Entry 6	**T**
Entry 1	NT	Entry 3	**NT**	Entry 5	NT	Entry 7	NT

b) Next, reconstruct the program structure by examining the sequence of instructions.

You have to look at the adx because they say the sequence of instructions, with adx in order.

Code:

```
term:
0x0048      dmul r4, r5, r6
0x004c      bne r4, r1, lab1
0x0050      daddui r7, r7, 1
0x0054      jchk
lab1:
0x0058      bne r4, r2, lab2
0x005c      daddui r8, r7, 1
0x0060      jchk
lab2:
0x0064      daddui r9, r8, 1
chk:
0x0068      daddi r5, r5, -1
0x006c      bnez r4, term
0x0070      halt
```

Question #2

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 is initially empty (i.e., full of 0s), report the content of the BTB after the execution of each instruction:

Address	Code		
	.data		
	vec:	.byte 15, 12, 9, 8, 6, 4	; input vector
	mul2:	.space 1	; no. of values equal to or multiples of 2
	mul3:	.space 1	; no. 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, 6	; 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:	lbu r7, vec(r4)	; load an element from *vec*
0x001c		ddiv r8, r7, r1	; Barrett reduction (mod calculation), $n = 2$
0x0020		dmulu r8, r8, r1	; Barrett reduction (mod calculation), $n = 2$
0x0024		dsubu r8, r7, r8	; Barrett reduction (mod calculation), $n = 2$
0x0028		bnez r8, m3	; jump to *m3* if the mod is not equal to zero
0x002c		daddui r5, r5, 1	; increment the counter of values equal to or multiples of 2
0x0030	m3:	ddiv r8, r7, r2	; Barrett reduction (mod calculation), $n = 3$
0x0034		dmulu r8, r8, r2	; Barrett reduction (mod calculation), $n = 3$
0x0038		dsubu r8, r7, r8	; Barrett reduction (mod calculation), $n = 3$
0x003c		bnez r8, nxt	; jump to *nxt* if the mod is not equal to zero

```
0x0040          daddui r6, r6, 1        ; increment the counter of values equal to
                                          or multiples of 3
0x0044   nxt:   daddui r4, r4, 1        ; increment the pointer
0x0048          bne r3, r4, cyc         ; condition for exiting the cycle
0x004c   term:  sb r5, mul2(r0)         ; store the number of values equal to or
                                          multiples of 2
0x0050          sb r6, mul3(r0)         ; store the number of values equal to or
                                          multiples of 3
0x0054          halt                    ; termination of the program
```

a) First, try to understand what the code does using the code itself and the comments provided.

Explanation:
The program contains a cycle (cyc) that has 6 iterations. In every one of them one of the elements of the vector vec stored in memory is selected and it's checked wether it's a multiple of 2 or 3. In both cases a counter (related to the values that are multiple to either 2 or 3) is then updated. At the end of the iterations, both counters are stored in memory.

b) Then, fill in the contents of this table to understand which entry of the BTB each instruction will refer to. To calculate the BTB entry corresponding to each branch instruction, remember to exclude the last two bits from the instruction address, as they always equal 0.

Instruction	Address (Hex)	Address (Binary)	Entry #
bnez r8, m3	*0x0028*	0000 0000 0000 0000 0000 0000 0010 1000	*2*
bnez r8, nxt	0x003c	00000000000000000000000000111100	7
bne r3, r4, cyc	0x0048	00000000000000000000000001001000	2

c) Subsequently, run the program and describe the change of BTB during its execution. At the end, report the total number of correct and incorrect predictions.

0. BTB initial content

Entry #	Address	Target
0	*0x00000000*	*0x00000000*
1	*0x00000000*	*0x00000000*
2	*0x00000000*	*0x00000000*
3	*0x00000000*	*0x00000000*

Entry #	Address	Target
4	*0x00000000*	*0x00000000*
5	*0x00000000*	*0x00000000*
6	*0x00000000*	*0x00000000*
7	*0x00000000*	*0x00000000*

1. ***Cycle #1***: BTB content after ***bnez r8, m3*** execution
   NOT FOUND IN BTB, BUT TAKEN →**PUT ENTRY IN BTB**

Entry #	Address	Target
0	*0x00000000*	*0x00000000*
1	*0x00000000*	*0x00000000*
2	***0x00000028***	***0x00000030***
3	*0x00000000*	*0x00000000*

Entry #	Address	Target
4	*0x00000000*	*0x00000000*
5	*0x00000000*	*0x00000000*
6	*0x00000000*	*0x00000000*
7	*0x00000000*	*0x00000000*

0. ***Cycle #1***: BTB content after ***bnez r8, nxt*** execution
PREDICTED NOT TAKEN, ACTUALLY NOT TAKEN →**NORMAL EXECUTION**

Entry #	Address	Target
0	*0x00000000*	*0x00000000*
1	*0x00000000*	*0x00000000*
2	***0x00000028***	***0x00000030***
3	*0x00000000*	*0x00000000*

Entry #	Address	Target
4	*0x00000000*	*0x00000000*
5	*0x00000000*	*0x00000000*
6	*0x00000000*	*0x00000000*
7	***0x00000000***	***0x00000000***

2. ***Cycle #1***: BTB content after ***bne r3, r4, cyc*** execution
ENTRY NOT FOUND IN BTB, ACTUALLY TAKEN →**PUT ENTRY IN BTB**

Entry #	Address	Target
0	*0x00000000*	*0x00000000*
1	*0x00000000*	*0x00000000*
2	***0x00000048***	***0x00000018***
3	*0x00000000*	*0x00000000*

Entry #	Address	Target
4	*0x00000000*	*0x00000000*
5	*0x00000000*	*0x00000000*
6	*0x00000000*	*0x00000000*
7	***0x00000000***	***0x00000000***

1. ***Cycle #2***: BTB content after ***bnez r8, m3*** execution
ENTRY NOT FOUND IN BTB, ACTUALLY NOT TAKEN →**NORMAL EXECUTION**

Entry #	Address	Target
0	*0x00000000*	*0x00000000*
1	*0x00000000*	*0x00000000*
2	***0x00000048***	***0x00000018***
3	*0x00000000*	*0x00000000*

Entry #	Address	Target
4	*0x00000000*	*0x00000000*
5	*0x00000000*	*0x00000000*
6	*0x00000000*	*0x00000000*
7	***0x00000000***	***0x00000000***

2. ***Cycle #2***: BTB content after ***bnez r8, nxt*** execution
PREDICT NOT TAKEN, ACTUALLY NOT TAKEN → **NORMAL EXECUTION**

Entry #	Address	Target
0	*0x00000000*	*0x00000000*
1	*0x00000000*	*0x00000000*
2	***0x00000048***	***0x00000018***
3	*0x00000000*	*0x00000000*

Entry #	Address	Target
4	*0x00000000*	*0x00000000*
5	*0x00000000*	*0x00000000*
6	*0x00000000*	*0x00000000*
7	***0x00000000***	***0x00000000***

3. ***Cycle #2***: BTB content after ***bne r3, r4, cyc*** execution
PREDICT TAKEN, ACTUALLY TAKEN →**CORRECT**

Entry #	Address	Target
0	*0x00000000*	*0x00000000*
1	*0x00000000*	*0x00000000*
2	***0x00000048***	***0x00000018***
3	*0x00000000*	*0x00000000*

Entry #	Address	Target
4	*0x00000000*	*0x00000000*
5	*0x00000000*	*0x00000000*
6	*0x00000000*	*0x00000000*
7	*0x00000000*	*0x00000000*

4. ***Cycle #3***: BTB content after ***bnez r8, m3*** execution
NOT FOUND IN BTB, ACTUALLY TAKEN →**PUT ENTRY IN BTB**

Entry #	Address	Target
0	*0x00000000*	*0x00000000*
1	*0x00000000*	*0x00000000*
2	***0x00000028***	***0x00000030***
3	*0x00000000*	*0x00000000*

Entry #	Address	Target
4	*0x00000000*	*0x00000000*
5	*0x00000000*	*0x00000000*
6	*0x00000000*	*0x00000000*
7	*0x00000000*	*0x00000000*

5. ***Cycle #3***: BTD content after ***bnez r8, nxt*** execution
   NOT FOUND IN BTB, ACTUALLY NOT TAKEN →**NORMAL EXECUTION**

Entry #	Address	Target
0	*0x00000000*	*0x00000000*
1	*0x00000000*	*0x00000000*
2	***0x00000028***	***0x00000030***
3	*0x00000000*	*0x00000000*

Entry #	Address	Target
4	*0x00000000*	*0x00000000*
5	*0x00000000*	*0x00000000*
6	*0x00000000*	*0x00000000*
7	***0x00000000***	***0x00000000***

6. ***Cycle #3***: BTD content after ***bne r3, r4, cyc***execution
   NOT FOUND IN BTB, ACTUALLY TAKEN →**PUT ENTRY IN BTB**

Entry #	Address	Target
0	*0x00000000*	*0x00000000*
1	*0x00000000*	*0x00000000*
2	***0x00000048***	***0x00000018***
3	*0x00000000*	*0x00000000*

Entry #	Address	Target
4	*0x00000000*	*0x00000000*
5	*0x00000000*	*0x00000000*
6	*0x00000000*	*0x00000000*
7	***0x00000000***	***0x00000000***

7. ***Cycle #4***: BTD content after ***bnez r8, m3*** execution
   NOT FOUND IN BTB, ACTUALLY NOT TAKEN →**NORMAL EXECUTION**

Entry #	Address	Target
0	*0x00000000*	*0x00000000*
1	*0x00000000*	*0x00000000*
2	***0x00000048***	***0x00000018***
3	*0x00000000*	*0x00000000*

Entry #	Address	Target
4	*0x00000000*	*0x00000000*
5	*0x00000000*	*0x00000000*
6	*0x00000000*	*0x00000000*
7	***0x00000000***	***0x00000000***

8. ***Cycle #4***: BTD content after ***bnez r8, nxt***execution
   NOT FOUND IN BTB, ACTUALLY TAKEN →**PUT ENTRY IN BTB**

Entry #	Address	Target
0	*0x00000000*	*0x00000000*
1	*0x00000000*	*0x00000000*
2	***0x00000048***	***0x00000018***
3	*0x00000000*	*0x00000000*

Entry #	Address	Target
4	*0x00000000*	*0x00000000*
5	*0x00000000*	*0x00000000*
6	*0x00000000*	*0x00000000*
7	***0x0000003c***	***0x00000044***

9. ***Cycle #4***: BTD content after ***bne r3, r4, cyc***execution
   PREDICT TAKEN, ACTUALLY TAKEN →**CORRECT**

Entry #	Address	Target
0	*0x00000000*	*0x00000000*
1	*0x00000000*	*0x00000000*
2	***0x00000048***	***0x00000018***
3	*0x00000000*	*0x00000000*

Entry #	Address	Target
4	*0x00000000*	*0x00000000*
5	*0x00000000*	*0x00000000*
6	*0x00000000*	*0x00000000*
7	***0x0000003c***	***0x00000044***

10. ***Cycle #5***: BTD content after ***bnez r8, m3*** execution
    NOT FOUND IN BTB, ACTUALLY NOT TAKEN →**NORMAL EXECUTION**

Entry #	Address	Target
0	*0x00000000*	*0x00000000*
1	*0x00000000*	*0x00000000*

Entry #	Address	Target
4	*0x00000000*	*0x00000000*
5	*0x00000000*	*0x00000000*

2	***0x00000048***	***0x00000018***
3	*0x00000000*	*0x00000000*

6	*0x00000000*	*0x00000000*
7	***0x0000003c***	***0x00000044***

11. ***Cycle #5***: BTB content after ***bnez r8, nxt***execution
PREDICT TAKEN, ACTUALLY NOT TAKEN →**INCORRECT**

Entry #	Address	Target
0	*0x00000000*	*0x00000000*
1	*0x00000000*	*0x00000000*
2	***0x00000048***	***0x00000018***
3	*0x00000000*	*0x00000000*

Entry #	Address	Target
4	*0x00000000*	*0x00000000*
5	*0x00000000*	*0x00000000*
6	***0x00000000***	***0x00000000***
7	***0x00000000***	***0x00000000***

12. ***Cycle #5***: BTB content after ***bne r3, r4, cyc***execution
PREDICT TAKEN, ACTUALLY TAKEN →**CORRECT**

Entry #	Address	Target
0	*0x00000000*	*0x00000000*
1	*0x00000000*	*0x00000000*
2	***0x00000048***	***0x00000018***
3	*0x00000000*	*0x00000000*

Entry #	Address	Target
4	*0x00000000*	*0x00000000*
5	*0x00000000*	*0x00000000*
6	*0x00000000*	*0x00000000*
7	***0x00000000***	***0x00000000***

13. ***Cycle #6***: BTB content after ***bnez r8, m3*** execution
ENTRY NOT PRESENT IN BTB, NOT TAKEN →**NORMAL EXECUTION**

Entry #	Address	Target
0	*0x00000000*	*0x00000000*
1	*0x00000000*	*0x00000000*
2	***0x00000048***	***0x00000018***
3	*0x00000000*	*0x00000000*

Entry #	Address	Target
4	*0x00000000*	*0x00000000*
5	*0x00000000*	*0x00000000*
6	*0x00000000*	*0x00000000*
7	***0x00000000***	***0x00000000***

14. ***Cycle #6***: BTB content after ***bnez r8, nxt***execution
NOT FOUND IN BTB, ACTUALLY TAKEN →**PUT ENTRY IN BTB**

Entry #	Address	Target
0	*0x00000000*	*0x00000000*
1	*0x00000000*	*0x00000000*
2	***0x00000048***	***0x00000018***
3	*0x00000000*	*0x00000000*

Entry #	Address	Target
4	*0x00000000*	*0x00000000*
5	*0x00000000*	*0x00000000*
6	*0x00000000*	*0x00000000*
7	***0x0000003c***	***0x00000044***

15. ***Cycle #6***: BTB content after ***bne r3, r4, cyc***execution
PREDICT TAKEN, ACTUALLY NOT TAKEN →**INCORRECT**

Entry #	Address	Target
0	*0x00000000*	*0x00000000*
1	*0x00000000*	*0x00000000*
2	***0x00000000***	***0x00000000***
3	*0x00000000*	*0x00000000*

Entry #	Address	Target
4	*0x00000000*	*0x00000000*
5	*0x00000000*	*0x00000000*
6	*0x00000000*	*0x00000000*
7	***0x0000003c***	***0x00000044***

Number of correct predictions: __3__ Number of incorrect predictions: __2__

Question #3

Consider a processor connected to a memory divided into 64 blocks and equipped with a direct mapped cache of 8 lines.

Supposing that the cache is initially empty and that the processor performs the sequence of memory accesses shown in the table below, determine for each of them the corresponding cache line accessed and whether a hit (H) or a miss (M) occurred.

Hint: In a direct mapped cache, the line i in which a block k is stored is given by the following formula:

$$i = k \bmod N$$

where N is the number of cache lines.

Block	Accessed Line	H/M
21	5	M
28	4	M
30	6	M
10	2	M
7	7	M
60	4	M
30	6	H
31	7	M
32	0	M
33	1	M
60	4	H
21	5	H
25	1	M
28	4	M
27	3	M
14	6	M

a) Based on the results obtained, fill in the cache content at the end of the run andindicate the total number of hits and misses.

Number of hits: __3__ Number of misses: __13__

Cache - Final content

Line 0	32	Line2	10	Line 4	28	Line6	14
Line 1	25	Line3	27	Line 5	21	Line7	31

Question #4

Consider a processor connected to 64 KB of memory and equipped with a set-associative cache consisting of 4 sets of 4 lines each, for a total of 16 different lines, each of 32 bytes. Assuming that:

- The cache is initially empty.
- Within each set, the lines are filled in the order given by their index.
- The adopted replacing algorithm is the Least Recently Used (LRU).

Given the sequence of memory accesses shown in the table below, determine the corresponding set and line being accessed. Use the cache schema provided below to assist in calculating the associated line. This way, you will get the final cache status. Finish by writing and the total number of hits and misses that occurred.

Hint: In a set associative mapped cache consisting of s sets, the set k in which a block i is stored is given by the following formula:

$$k = i \bmod s$$

The blocki can be put into any of the W lines of the set k.

In case of full sets, I use **LRU** to determine which slot to erase.

Block	Block (Binary)	Accessed Set	Accessed Line	H/M
352	*0001 0110 0000*	*0*	*0*	*M*
590	*0010 0100 1110*	*2*	*8*	*M*
618	*0010 0110 1010*	*2*	*9*	*M*
1679	*0110 1000 1111*	3	12	*M*
2001	*0111 1101 0001*	1	4	*M*
590	*0010 0100 1110*	2	8	*H*
685	*0010 1010 1101*	1	5	*M*
1318	*0101 0010 0110*	2	10	*M*
1987	*0111 1100 0011*	3	13	*M*
125	*0000 0111 1101*	1	6	*M*
1993	*0111 1100 1001*	1	7	*M*
766	*0010 1111 1110*	2	11	*M*
921	*0011 1001 1001*	1	4	*M*
900	*0011 1000 0100*	0	2	*M*
819	*0011 0011 0011*	3	14	*M*
1098	*0100 0100 1010*	2	8	*M*
1465	*0101 1011 1001*	1	5	*M*

Cache

	Set 0		Set 1		Set 2		Set 3
Line 0	*352*	Line 4	921	Line 8	*1098*	Line 12	1679
Line 1	900	Line 5	1465	Line 9	*618*	Line 13	1987
Line 2		Line 6	125	Line 10	1318	Line 14	819
Line 3		Line 7	1993	Line 11	766	Line 15	

Number of hits: __1__ Number of misses: __16__

Question #5

Let's consider a superscalar MIPS64 architecture implementing dynamic scheduling, speculation, and multiple issues and composed of the following units:

- An issue unit able to process 1 instruction per clock period; in the case of a branch instruction, only one instruction is issued per clock period
- A commit unit able to process 1 instruction per clock period
- The following functional units (for each unit, the number of clock periods to complete one instruction is reported):
  - 1 unit for memory access: 1 clock period
  - 1 unit for integer arithmetic instructions: 1 clock period
  - 1 unit for branch instructions: 1 clock period
  - 1 unit for FP multiplication (pipelined): 3 clock periods
  - 1 unit for FP division (unpipelined): 10 clock periods
  - 1 unit for other FP instructions (pipelined): 2 clock periods
  - 1 Common Data Bus (CDB).
- Let's also assume that:
  - Branch predictions are always correct
  - All memory accesses never trigger a cache miss.

Use the following table to describe the behavior of the processor during the execution of the first two iterations of the loop, computing the total number of required clock cycles. For the EXE stage, it is recommended to add a lowercase letter to help distinguish between the different functional units available.

Loop #	Instruction	ISSUE	EXE	MEM	CDB	COMMIT	Notes
1	l.d f1, v1(r1)	*1*	*2m*	*3*	*4*	*5*	
1	l.d f2, v2(r1)	*2*	*3m*	*4*	*5*	*6*	
1	add.d f5, f1, f2	*3*	*6a*		*8*	*9*	*Wait for f2(5)*
1	l.d f3, v3(r1)	4	5m	6	7	10	
1	l.d f4, v4(r1)	5	6m	7	9	11	CDB already used at 8
1	sub.d f6, f3, f4	6	10a		12	13	Wait for f3(7), f4(9)
1	mul.d f7, f5, f6	7	13x		16	17	Wait for f5(9), f6(12)
1	div.d f7, f7, f8	8	17d		27	28	Wait for f7(16)
1	s.d f7, v5(r1)	9	10m			29	Wait for f7(27) (at COMMIT)
1	daddui r1, r1, 8	10	11i		13	30	CDB already used at 12
1	daddi r2, r2, -1	11	12i		14	31	CDB already used at 13
1	bnez r2, loop	12	15b			32	Wait for r2(14)
2	l.d f1, v1(r1)	13	14m	15	17	33	CDB already used at 16
2	l.d f2, v2(r1)	14	15m	16	18	34	CDB already used at 17
2	add.d f5, f1, f2	15	19a		21	35	Wait for f1(17),f2(18)
2	l.d f3, v3(r1)	16	17m	18	19	36	

2	l.d f4, v4(r1)	17	18m	19	20	37	
2	sub.d f6, f3, f4	18	21a		23	38	Wait for f3(19), f4(20)
2	mul.d f7, f5, f6	19	24x		28	39	Wait for f5(21), f6(23), CDB already used at 27
2	div.d f7, f7, f8	20	29d		39	40	Wait for f7(28)
2	s.d f7, v5(r1)	21	22m			41	Wait for f7(39) (at COMMIT)
2	daddui r1, r1, 8	22	23i		24	42	
2	daddi r2, r2, -1	23	24i		25	43	
2	bnez r2, loop	24	26b			44	Wait for r2(25)

.