|  |  |
| --- | --- |
| **Computer Architectures**  **02LSEOV** | Delivery date:  October 30th 2024, 11.59 PM |
| **Laboratory**  **4** | Expected delivery of lab\_03.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).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Address** | **Code** | **Address in Binary** | **BHT Entry #** | **PRED** | **RES** |
| 0x0048 | dmul r4, r5, r6 |  |  |  |  |
| 0x004c | bne r4, r1, lab1 | *… 0000 0000 010****011****00* | *3* | *NT* | *NT* |
| 0x0050 | daddui r7, r7, 1 |  |  |  |  |
| 0x0054 | j chk (\*) |  |  |  |  |
| 0x0068 | daddi r5, r5, -1 |  |  |  |  |
| 0x006c | bnez r4, term | *… 0000 0000 011****011****00* | *3* | *NT* | *T* |
| 0x0048 | dmul r4, r5, r6 |  |  |  |  |
| 0x004c | bne r4, r1, lab1 | … 0000 0000 010**011**00 | 3 | T | T |
| 0x0058 | bne r4, r2, lab2 | … 00000000010**110**00 | 6 | NT | T |
| 0x005c | daddui r8, r7, 1 |  |  |  |  |
| 0x0060 | j chk (\*) |  |  |  |  |
| 0x0068 | daddi r5, r5, -1 |  |  |  |  |
| 0x006c | bnez r4, term | … 0000 0000 010**011**00 | 3 | T | T |
| 0x0048 | dmul r4, r5, r6 |  |  |  |  |
| 0x004c | bne r4, r1, lab1 | … 00000000011**011**00 | 3 | T | T |
| 0x0058 | bne r4, r2, lab2 | … 00000000010**110**00 | 6 | T | NT |
| 0x0064 | daddui r9, r8, 1 |  |  |  |  |
| 0x0068 | daddi r5, r5, -1 |  |  |  |  |
| 0x006c | bnez r4, term | … 0000 0000 010**011**00 | 3 | T | NT |
| 0x0070 | halt |  |  |  |  |

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

1. 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: \_\_\_\_

**BHT - Final content**

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

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

Code:

**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 |

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

Explanation:

1. 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 | 0000 0000 0000 0000 0000 0000 0011 1100 | 7 |
| bne r3, r4, cyc | 0x0048 | 0000 0000 0000 0000 0000 0000 0100 1000 | 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.

1. BTB initial content

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Entry #** | **Address** | **Target** |  | **Entry #** | **Address** | **Target** |
| 0 | *0x00000000* | *0x00000000* |  | 4 | *0x00000000* | *0x00000000* |
| 1 | *0x00000000* | *0x00000000* |  | 5 | *0x00000000* | *0x00000000* |
| 2 | *0x00000000* | *0x00000000* |  | 6 | *0x00000000* | *0x00000000* |
| 3 | *0x00000000* | *0x00000000* |  | 7 | *0x00000000* | *0x00000000* |

1. ***Cycle #1***: BTB content after ***bnez r8, m3*** execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Entry #** | **Address** | **Target** |  | **Entry #** | **Address** | **Target** |
| 0 | *0x00000000* | *0x00000000* |  | 4 | *0x00000000* | *0x00000000* |
| 1 | *0x00000000* | *0x00000000* |  | 5 | *0x00000000* | *0x00000000* |
| 2 | ***0x00000028*** | ***0x00000030*** |  | 6 | *0x00000000* | *0x00000000* |
| 3 | *0x00000000* | *0x00000000* |  | 7 | *0x00000000* | *0x00000000* |

1. ***Cycle #1***: BTB content after ***bnez r8, nxt*** execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Entry #** | **Address** | **Target** |  | **Entry #** | **Address** | **Target** |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. ***Cycle #1***: BTB content after ***bne r3, r4, cyc*** execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Entry #** | **Address** | **Target** |  | **Entry #** | **Address** | **Target** |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. ***Cycle #2***: BTB content after ***bnez r8, m3*** execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Entry #** | **Address** | **Target** |  | **Entry #** | **Address** | **Target** |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. ***Cycle #2***: BTB content after ***bnez r8, nxt*** execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Entry #** | **Address** | **Target** |  | **Entry #** | **Address** | **Target** |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. ***Cycle #2***: BTB content after ***bne r3, r4, cyc*** execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Entry #** | **Address** | **Target** |  | **Entry #** | **Address** | **Target** |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. ***Cycle #3***: BTB content after ***bnez r8, m3*** execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Entry #** | **Address** | **Target** |  | **Entry #** | **Address** | **Target** |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. ***Cycle #3***: BTB content after ***bnez r8, nxt***execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Entry #** | **Address** | **Target** |  | **Entry #** | **Address** | **Target** |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. ***Cycle #3***: BTB content after ***bne r3, r4, cyc***execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Entry #** | **Address** | **Target** |  | **Entry #** | **Address** | **Target** |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. ***Cycle #4***: BTB content after ***bnez r8, m3*** execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Entry #** | **Address** | **Target** |  | **Entry #** | **Address** | **Target** |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. ***Cycle #4***: BTB content after ***bnez r8, nxt***execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Entry #** | **Address** | **Target** |  | **Entry #** | **Address** | **Target** |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. ***Cycle #4***: BTB content after ***bne r3, r4, cyc***execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Entry #** | **Address** | **Target** |  | **Entry #** | **Address** | **Target** |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. ***Cycle #5***: BTB content after ***bnez r8, m3*** execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Entry #** | **Address** | **Target** |  | **Entry #** | **Address** | **Target** |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. ***Cycle #5***: BTB content after ***bnez r8, nxt***execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Entry #** | **Address** | **Target** |  | **Entry #** | **Address** | **Target** |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. ***Cycle #5***: BTB content after ***bne r3, r4, cyc***execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Entry #** | **Address** | **Target** |  | **Entry #** | **Address** | **Target** |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. ***Cycle #6***: BTB content after ***bnez r8, m3*** execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Entry #** | **Address** | **Target** |  | **Entry #** | **Address** | **Target** |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. ***Cycle #6***: BTB content after ***bnez r8, nxt***execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Entry #** | **Address** | **Target** |  | **Entry #** | **Address** | **Target** |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

1. ***Cycle #6***: BTB content after ***bne r3, r4, cyc***execution

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Entry #** | **Address** | **Target** |  | **Entry #** | **Address** | **Target** |
| 0 |  |  |  | 4 |  |  |
| 1 |  |  |  | 5 |  |  |
| 2 |  |  |  | 6 |  |  |
| 3 |  |  |  | 7 |  |  |

Number of correct predictions: \_\_\_\_ Number of incorrect predictions: \_\_\_\_

**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 mod N*

*where N is the number of cache lines.*

|  |  |  |
| --- | --- | --- |
| **Block** | **Accessed Line** | **H/M** |
| *21* | *5* | *M* |
| *28* |  |  |
| *30* | *6* | *M* |
| *10* |  |  |
| *7* |  |  |
| *60* |  |  |
| *30* | *6* | *H* |
| *31* |  |  |
| *32* |  |  |
| *33* |  |  |
| *60* |  |  |
| *21* |  |  |
| *25* |  |  |
| *28* |  |  |
| *27* |  |  |
| *14* |  |  |

1. 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: \_\_\_\_ Number of misses: \_\_\_\_

**Cache - Final content**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Line 0 |  |  | Line2 |  |  | Line 4 |  |  | Line6 |  |  |
| Line 1 |  |  | Line3 |  |  | Line 5 |  |  | Line7 |  |  |

**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 mod s*

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 |  |  |  |  |
| 2001 |  |  |  |  |
| 590 |  |  |  |  |
| 685 |  |  |  |  |
| 1318 |  |  |  |  |
| 1987 |  |  |  |  |
| 125 |  |  |  |  |
| 1993 |  |  |  |  |
| 766 |  |  |  |  |
| 921 |  |  |  |  |
| 900 |  |  |  |  |
| 819 |  |  |  |  |
| 1098 |  |  |  |  |
| 1465 |  |  |  |  |

**Cache**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Set 0 |  |  | Set 1 |  |  | Set 2 |  |  | Set 3 |  |
| Line 0 | *352* |  | Line 4 |  |  | Line 8 | *590* |  | Line 12 |  |  |
| Line 1 |  |  | Line 5 |  |  | Line 9 | *618* |  | Line 13 |  |  |
| Line 2 |  |  | Line 6 |  |  | Line 10 |  |  | Line 14 |  |  |
| Line 3 |  |  | Line 7 |  |  | Line 11 |  |  | Line 15 |  |  |

Number of hits: \_\_\_\_ Number of misses: \_\_\_\_

**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) |  |  |  |  |  |  |
| 1 | l.d f4, v4(r1) |  |  |  |  |  |  |
| 1 | sub.d f6, f3, f4 |  |  |  |  |  |  |
| 1 | mul.d f7, f5, f6 |  |  |  |  |  |  |
| 1 | div.d f7, f7, f8 |  |  |  |  |  |  |
| 1 | s.d f7, v5(r1) |  |  |  |  |  |  |
| 1 | daddui r1, r1, 8 |  |  |  |  |  |  |
| 1 | daddi r2, r2, -1 |  |  |  |  |  |  |
| 1 | bnez r2, loop |  |  |  |  |  |  |
| 2 | l.d f1, v1(r1) |  |  |  |  |  |  |
| 2 | l.d f2, v2(r1) |  |  |  |  |  |  |
| 2 | add.d f5, f1, f2 |  |  |  |  |  |  |
| 2 | l.d f3, v3(r1) |  |  |  |  |  |  |
| 2 | l.d f4, v4(r1) |  |  |  |  |  |  |
| 2 | sub.d f6, f3, f4 |  |  |  |  |  |  |
| 2 | mul.d f7, f5, f6 |  |  |  |  |  |  |
| 2 | div.d f7, f7, f8 |  |  |  |  |  |  |
| 2 | s.d f7, v5(r1) |  |  |  |  |  |  |
| 2 | daddui r1, r1, 8 |  |  |  |  |  |  |
| 2 | daddi r2, r2, -1 |  |  |  |  |  |  |
| 2 | bnez r2, loop |  |  |  |  |  |  |