CompArch: exam

Renaud Pacalet

10 February 2022

You can use any document but communicating devices are strictly forbidden. Please number the different pages of your paper and indicate on each page your first and last names. You can write your answers in French or in English, as you wish. Precede your answers with the question's number. If some information or hypotheses are missing to answer a question, add them. If you consider a question as absurd and thus decide to not answer, explain why. If you do not have time to answer a question but know how to, briefly explain your ideas. Note: copying verbatim the slides of the lectures or any other provided material is not considered as a valid answer. Advice: quickly go through the document and answer the easy parts first.

1 Branch prediction (6 points)

Definitions and notations

- **Miss-Prediction per executed Branch Instruction (MPBI)**: the number of times a given branch instruction has been wrongly predicted divided by the total number of times this same branch instruction has been executed. The lower the MPBI, the better the prediction.
- M_∞: for a given branch instruction, the limit of the MPBI when the number of times the branch instruction is executed tends to infinity, if this limit exists. Undefined if it does not exist.
- **Branch outcome**: the actual decision (**not the prediction**) for a given branch instruction; Taken or Not taken, denoted `T` and `N`, respectively.
- **Periodic infinite sequence of outcomes**: a sequence of outcomes that starts with a finite sequence, the stem, which can be empty, and continues with a finite cycle that repeats infinitely. We represent these sequences as `STEM(CYCLE)*` where `STEM` is the shortest possible stem and `CYCLE` is the shortest possible cycle. Example: `TNTN NNNTTTT NNNTTTT NNNTTTT...` is a periodic infinite sequence of outcomes, its shortest possible stem is `TNTN`, its shortest possible cycle is `NNNTTTT` and we represent it as `TNTN(NNNTTTT)*`.

Problem

Consider the MIPS32 assembly code of Listing 1.

```
1 .data
2 flk_data:         .word   2
3
```

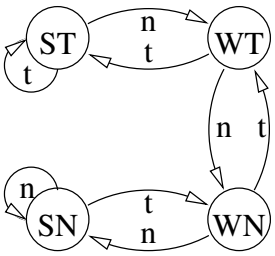

Figure 1: The 2-bits saturating counter

```
4  .text
5  flk:
6          addiu   $sp,    $sp,    -8      # $sp <- $sp-8
7          sw      $ra,    0($sp)          # mem[$sp] <- $ra
8          sw      $s0,    4($sp)          # mem[$sp+4] <- $s0
9          lw      $s0,    flk_data        # $s0 <- mem[flk_data]
10         xori    $s0,    $s0,    1       # $s0 <- $s0 xor 1
11         sw      $s0,    flk_data        # mem[flk_data] <- $s0
12 flk_loop:
13         jal     flk_sub                 # call flk_sub
14         addiu   $s0,    $s0,    -1      # $s0 <- $s0-1
15 BRANCH: bnez    $s0,    flk_loop        # if $s0 != 0 branch to flk_loop
16         lw      $s0,    4($sp)          # $s0 <- mem[$sp+4]
17         lw      $ra,    0($sp)          # $ra <- mem[$sp]
18         addiu   $sp,    $sp,    8       # $sp <- $sp+8
19         jr      $ra                     # return
```

Listing 1: The MIPS32 assembly code of the `flk` function

The shown `flk` function uses a 32-bits data word located in memory at address represented by label `flk_data` (declared line 2). This data word is initialized with value 2 and we assume that it is modified only by line 11, nowhere else in the entire program. We do not know what the `flk_sub` function (called line 13) does but we know that it respects the conventions seen during the lectures; in particular, calling `flk_sub` does not modify registers `$sp`, `$ra`, and `$s0`.

In the following we denote `BRANCH` the branch instruction at line 15 and labelled `BRANCH`. Try to understand the behavior of the `flk` function and answer the following questions:

Q1.1 Do you think the `flk` function respects the procedure call conventions we studied during the lectures, and used during the labs? Explain why.

Q1.2 The first time the `flk` function is executed, how many times is the `flk_sub` function called? And the second time? And the third time?

Q1.3 Assuming the `flk` function is called an infinite number of times, what is the sequence of outcomes of the `BRANCH` instruction? Is it periodic? If yes represent it using the `STEM(CYCLE)`* notation.

We consider the 4-states branch predictor studied during the lecture on pipelines, named the 2-bits saturating counter, and which state diagram is represented on Figure 1.

Q1.4 How many bits are needed inside the Branch Prediction Unit (BPU) to store the current state of one such predictor?

We use one such predictor to predict the outcomes of the `BRANCH` instruction. We measure the prediction efficiency with the MPBI. We assume that the predictor is initialized in the Strong Taken (ST) state and that it is not polluted by other branch instructions elsewhere in the program.

Q1.5 What is the MPBI for the `BRANCH` instruction after the first execution of `flk`?

Q1.6 For the `BRANCH` instruction, does M_∞ exist? If yes what is its value?

We want to improve the prediction efficiency with a two-levels local prediction. We thus equip our BPU with Branch History Shift Registers (BHSR) and Pattern History Tables (PHT).

Q1.7 What is the minimum lenght L of the BHSR needed to obtain $M_\infty = 0$ for the `BRANCH` instruction, assuming again that one entry of the BPU is entirely dedicated to this branch instruction and is not polluted by other branch instructions?

We decide to implement our BPU with $2^{10} = 1024$ entries and a direct-mapped architecture. Each entry of our new BPU contains a tag (to distinguish different branch instructions with same index), a target address (the destination of the branch if it is taken), one single valid bit for the whole entry, the BHSR, and the current states of the saturating counter predictors.

Q1.8 How many bits of storage are now needed inside the BPU to store the 1024 entries?

2 Representation of numbers (2 points)

Definitions and notations

There are several ways to represent signed integers using bits. In computer systems, the two most frequently encountered are **sign and magnitude** and **two's complement**. In the following we denote $x_{n-1}x_{n-2}\dots x_1x_0$ the n-bits representation of integer X. In both representations x_0 is the **Least Significant Bit** (LSB). In **sign and magnitude** x_{n-1} is the **sign** bit. In **two's complement** x_{n-1} is the **Most Significant Bit** (MSB).

Problem

Consider the 3 integer numbers -1024, 519 and -653.

Q2.1 We want to represent these 3 integers in two's complement on the **same** number of bits n. What is the minimum possible value of n (only one answer expected)?

Q2.2 Represent these 3 integers in n-bits two's complement, where n is your **unique** answer to the preceding question.

Let $a_{p-1}a_{p-2}\dots a_1a_0$ be the p-bits two's complement representation of integer A.

Q2.3 What is the minimum number of bits q needed to represent any possible value of A in sign and magnitude (only one answer expected)?

Q2.4 Explain how to obtain the q-bits *sign and magnitude* representation of A, where q is your **unique** answer to the preceding question.

3 Caches (6 points)

Definitions and notations

- The **breakdown** of a data structure is the partitioning of the data structure in individual fields and, for each field, its bit-width and a description of its role.
- In a set-associative cache the lines in a set are numbered starting from 0.
- The various cache operations on a line in a set are:
  - **Filled**, when it was not valid yet and receives a block.
  - **Updated**, when the block it caches is modified by the CPU.
  - **Read**, when the block it caches is read by the CPU.
  - **Replaced (or evicted)**, when the previously cached block is replaced by another one.
  - **Invalidated**, when, for any reason, it is decided that the block it caches shall not be cached anymore. When a line is invalidated it becomes invalid and ready to be filled again.
- **Extra cache data**: control and management information stored in the cache (tags, flags, replacement policy information...)
- **Net cache data**: copies of memory data stored in the cache, that is, everything except *extra cache data*.

Cache architecture

We consider a tiny computer system where addresses are byte addresses, with a 16-bits CPU (data words are 16-bits) and 24-bits addresses (total address space: 16 MBytes). We consider a 3-ways set-associative, write back, write allocate, data cache with 8 data words per line. There is no cache coherence, it would be useless in this system. The total net cache data capacity is 12kB (12×1024 bytes).

Q3.1 What is the breakdown of a 24-bits address? Explain your reasoning.

Q3.2 What is the breakdown of a cache line? Explain your reasoning.

Q3.3 Under what condition a line must be replaced (evicted) in a set?

The replacement policy is Least Recently Used (LRU). We recall that the replacement policy is used to select which line of a set is replaced when needed. With the LRU policy it is always the line that has been accessed (that is, filled, updated, read or replaced) less recently than the other lines in the set.

Q3.4 We denote x the number of bits per set that are used to implement the LRU policy; what is the minimum value of x? Explain your reasoning.

Q3.5 Propose a way to implement the LRU policy (you can use more bits than your previous answer for the minimum value of x). Explain the algorithm you use to decide which line to replace, and how your LRU management data evolve on each operation on a set (line fill, update, read, replace or invalidate). Do not forget to consider situations where a set is not full (all or some lines are not valid).

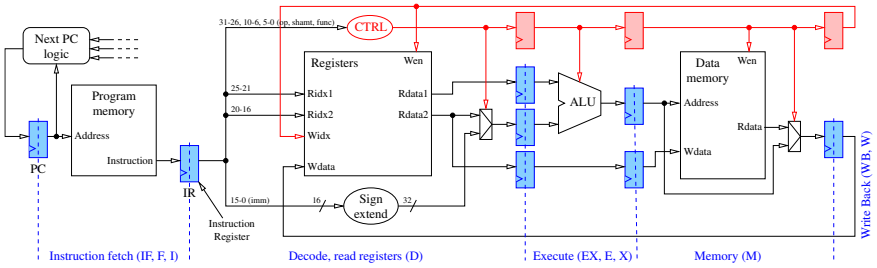

Figure 2: The 5-stages pipelined MIPS microprocessor

Q3.6 The LRU policy is implemented using the minimum number of bits x. What is the breakdown of a cache set? Explain your reasoning.

Q3.7 What is the total size (in bits) of the cache? Explain your reasoning.

4 Pipe-line hazards (6 points)

4.1 Pipeline data hazards

In this problem we use an Instruction Set Architecture (ISA) that is almost the same as the MIPS32 ISA, except that there are no delay slots (in the real MIPS32 ISA there is one delay slot after all control flow instructions, that is, branch, jump, jump-and-link...). Consider the MIPS assembly code of Listing 2. The code uses only true MIPS instructions, no pseudo-instructions.

```
1  .data 0x1001fff8
2  flk_data: .word 2
3
4  .text
5  flk:
6          addiu   $sp,    $sp,    -8          # $sp <- $sp-8
7          sw      $ra,    0($sp)              # mem[$sp] <- $ra
8          sw      $s0,    4($sp)              # mem[$sp+4] <- $s0
9          lui     $t0,    0x1001              # $t0 <- 0x10010000
10         ori     $t0,    $t0,    0xfff8      # $t0 <- 0x1001fff8 (flk_data)
11         lw      $s0,    0($t0)              # $s0 <- mem[$t0+0]
12         xori    $s0,    $s0,    1           # $s0 <- $s0 xor 1
13         sw      $s0,    0($t0)              # mem[$t0+0] <- $s0
14 flk_loop:
15         jal     flk_sub                     # call flk_sub
16         addiu   $s0,    $s0,    -1          # $s0 <- $s0-1
17         bne     $s0,    $zero,  flk_loop        # if $s0 != 0 branch to flk_loop
18         lw      $s0,    4($sp)              # $s0 <- mem[$sp+4]
19         lw      $ra,    0($sp)              # $ra <- mem[$sp]
20         addiu   $sp,    $sp,    8           # $sp <- $sp+8
21         jr      $ra                         # return
```

Listing 2: The MIPS assembly code of the `flk` function

Assume we want to run this code on the 5-stages pipelined MIPS microprocessor we studied during the lectures and which is represented on Figure 2.

Assume also that there are enough hardware resources to completely avoid structural hazards (two different instructions in two different pipeline stages never compete to access the same resource). However, the implementation is a very naive one where data and control hazards have not been considered at all yet: nothing has been done

to handle them. As a consequence, running the code will not produce the expected results: the hardware is bogus.

There is a data hazard between the two first instructions (lines 6 and 7). There is another data hazard between instructions at lines 11 and 12. There is a control hazard due to the instruction at line 17. For each of these 3 hazards:

Q4.1 Explain why there is a hazard.

Q4.2 Explain what is the expected behavior, what (bogus) behavior is observed instead if we execute the code as it is on the hardware as it is, and why it is different from the expected behavior.

Q4.3 Explain what pure software solution(s) could be used to solve or mitigate the hazard and obtain the expected behavior without modifying the hardware implementation or the ISA. What would be the performance impact?

Q4.4 Explain what pure hardware solution(s) could be used to solve or mitigate the hazard and obtain the expected behavior by modifying only the hardware implementation but not the ISA and not the software. What would be the performance impact?

Q4.5 Are they other solutions you can think of and that could be used to solve or mitigate the hazard?

There are other pipeline hazards in this code.

Q4.6 List two other data hazards, for each identify (by their line number) the two involved instructions and the involved register. Find at least one other control hazard and explain why it is a control hazard.