Honours Computer Architecture 2019 Prac 2 – solutions

You may confer on issues you do not understand but each answer must be your individual work. Note that you need to write enough so I can assess how well you understood – it you just write out an answer with no detail of how you got there, I can't assess understanding.

understanding of memory system and pipeline concepts	15
clear and accurate calculation methods – show how you got there	15
correct answers	15
readable, well presented, minimal typos	5
Total	50

Several hand-ins did not include a name or student number. Since I processed them directly off saving attachments from emails I don't think I had any wrong.

Also, please try to integrate everything into one document and create a PDF if you can because this is much less likely to lose formatting when opened on a different computer.

Part 1: Memory

In each of the following take care to quantify your answer in the terms requested. Assume the following information as a baseline for each part of this question:

- 2GHz clock
- 2 instructions per clock in the absence of stalls
- L1 cache access takes 2 cycles but is fully pipelined
- L2 cache access takes 10 cycles but results in a stall
- DRAM access takes 50 cycles

Instruction mix: 20% branches, 20% loads, 10% stores, remainder ALU
State any assumptions needed to answer (e.g. filling in missing detail).

1. What is the time per instruction in ns in the absence of any stalls?

2GHz clock: time per tick $1/2 \times 10^{9}$s = 0.5ns; if there are 2 instructions per clock average without stalls = 0.25ns per instruction.

2. If 1% of instructions result in a miss to L2 and 10% of L2 references result in a miss to DRAM, what is the average number of clock cycles per instruction (CPI) in the absence of any stalls not related to memory? *Hint: think through whether these are global or local miss rates.*

*Biggest issue: **confusion over L1 accesses taking 2 cycles but fully pipelined**. The way this works: in the 5-stage pipeline model, 1 cycle is allowed for instruction fetch and memory access (time for data transfer) stages. To implement a faster clock speed without having to make cache faster, you could split the fetch and memory stages into two stages so progress through a pipeline would look like this:*

```
IF1 IF2 ID EX MEM1 MEM2 WB
```

*Possibly other stages could be split too, to make it easier to fit the logic critical path into a clock cycle – using a very deep pipeline is referred to as **superpipelining** – Intel does this in many designs.*

Once you have this straight and with 2 instructions per clock, you are on average completing 2 instructions per clock so IPC = 2 and CPI = 0.5.

2 instructions per clock: CPI = 0.5.

10% of L2 references misses as a global rate: 1% (0.01 as a fraction) of all miss from L1, so this is 1%x10% = 0.1% (or 0.001 as a fraction) global miss rate from L2.

As a rough approximation, work on CPI = 0.5 and do not attempt to work out effect of stalls on 2nd instruction that could be simultaneous with the missing instruction. Assume that a miss from L2 also incurs the hit cost to L2 since the tags need to be updated.

So overall CPI based on execution time formula (3.1, p 44): 0.5 + 0.01x10+0.001x50 = 0.65.

Variation: I assume here that an L1 miss does not incur an extra L1 lookup latency (hidden by pipeline) but OK if you add 2 cycles for each L1 miss.

3. With the given DRAM parameters, for a cache block size of 32 bytes, how wide must the bus be to fit miss to a DRAM into 50 cycles?

Amazing how hard some of you made this… I got some answers like 6 bits wide. Think through whether an answer makes sense.

50 cycles is the time for one DRAM access so for a miss to be limited to this time, the bus must be wide enough to do this in one access, so 32B or 256b.

4. A design is proposed where a large hardware write buffer eliminates stalls for write misses. This design results in a smaller L2 cache, causing misses to DRAM to increase to 11% of L2 accesses. Calculate CPI in this new scenario and contrast with your answer to (2).

We need to make an assumption on write misses. The miss rates do not break down misses into instruction fetch misses, data read misses or data write misses. The given instruction mix is 10% stores but without any knowledge of what fraction of instruction fetches miss, we have to estimate how this breaks down. In general, data references are more likely to miss than instruction references because code tends to have tighter locality (a loop re-executes a small section of code but can reach a lot of data e.g. by following pointers or array referencing). It could be an under-estimate if 10% of misses are data writes (though I note on the other hand, a write is very likely to follow a read at the same location so this is at best a guess), so let us used this number for a ballpark estimate.

Also since no detail is supplied, assume that write miss latency can be hidden at all levels – but we can apply the miss fraction result at L1 and when we count misses at L2, we don't count the saved write misses.

Assumption: the write buffer hides the latency of 10% of all misses. We therefore reduce the effective number of misses from L1 to 0.9%, or 0.09 as a fraction. L2 misses increase as a fraction of this to 0.09x0.1 = 0.009. So the revised CPI calculation is

0.5 + 0.009x10+0.0009x50 = 0.635.

The difference is not very big – the speedup is 0.65÷0.635 = 1.02. In a real system, we may expect to see a bigger gain if we focus on saving in L1, since any stalls for a miss may slow the pipeline for other instructions but since we are not modelling the pipeline this effect is not modelled.

Part 2: Pipeline timing

For this question, use the following code in the machine code notation used in the course. The instruction set is described in the notes (RISC-V with simplified register naming), as is the 5-stage pipeline to be used in this question.

```
# registers:
# R1: base address of a; a copy we can change
# R2: base address of b; a copy we can change
# R3: N; constant in this code
# R4: i; loop counter
# R5: holds value of a[i]
# R6: holds value of b[i]
# R7: temp

         li R4, 0            # for (int i = 0; i < N; i++) {
         j fortest           # test loop at end
forbody: lw R6, 0(R2)        #   if (b[i] % 2 != 0)
         ori R7, R6, 1       #   // will be 1 if b[1] is odd
         bzero R7, else
         li R5, 2            # a[i] = b[i] * 2
         mult R5, R6, R5
         j endif             # else
else:    move R5, R6         #   a[i] = b[i]
endif:   sw R5, 0(R1)
         addi R4, R4, 1      # increment loop counter
         addi R1, R1, 4      # increment base addresses by 4:
         addi R2, R2, 4      #   (word size in bytes)
fortest: blt R4, R3, forbody # }
```

1. Draw a timing diagram (in the style of Figure 4.4, p 62 of the notes) illustrating the time it takes to do two (2) iterations of the loop under the following assumptions:

   a. No stalls.

      This one should not have been too hard except that the **if** branch outcome is not as clear as intended (see more detail in part b). You can here magically assume that the branch outcome does not waste any time since I say no stalls. This is not totally unrealistic as a branch target buffer containing the successor instruction could load that without having to go back to memory.

      This is easiest to answer in either a table or a spreadsheet – for ease of marking, my preference is that you include it in your final document saved as PDF – not terribly hard in LaTeX for example and not that bad in MS Word.

To see what's going on I list all the code with labels so it is possible to see which code is skipped when a branch is taken (or not). If you don't do this, take care to omit the correct instructions based on the branch outcome.

	instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
forbody:	`lw R6, 0(R2)`	F	D	X	M	W																			
	`ori R7, R6, 1`		F	D	X	M	W																		
	`bzero R7, else`			F	D	X	M	W																	
	`li R5, 2`				F	D	X	M	W																
	`mult R5, R6, R5`					F	D	X	M	W															
	`j endif`						F	D	X	M	W														
else:	`move R5, R6`							F																	
endif:	`sw R5, 0(R1)`							F	D	X	M	W													
	`addi R4, R4, 1`								F	D	X	M	W												
	`addi R1, R1, 4`									F	D	X	M	W											
	`addi R2, R2, 4`										F	D	X	M	W										
fortest	`blt R4, R3, forbody`											F	D	X	M	W									
	`loop successor`												F												
forbody:	`lw R6, 0(R2)`												F	D	X	M	W								
	`ori R7, R6, 1`													F	D	X	M	W							
	`bzero R7, else`														F	D	X	M	W						
	`li R5, 2`															F									
	`mult R5, R6, R5`																								
	`j endif`																								
else:	`move R5, R6`															F	D	X	M	W					
endif:	`sw R5, 0(R1)`																F	D	X	M	W				
	`addi R4, R4, 1`																	F	D	X	M	W			
	`addi R1, R1, 4`																		F	D	X	M	W		
	`addi R2, R2, 4`																			F	D	X	M	W	
fortest	`blt R4, R3, forbody`																				F	D	X	M	W

Timing depends on whether the branch is taken or not – as shown, it is not taken so we abandon the fetch of the instruction after `j endif` as the jump skips the next instruction. An iteration of the loop would do the opposite if `b[i]` is even. Not enough information is supplied to know if this is the case. So do next iteration with the **if** doing the opposite to illustrate both cases. Since we eliminate all stalls, we assume some magic whereby `j` and branch instructions can happen without delay – if you assumed that was impossible and got 27 clocks, fine.

b. Redraw the timing diagram for maximum possible forwarding and explain how stalls can be eliminated, given these design parameters:

- registers update in the first half of a cycle and can be read in the second half
- backward branches predicted as taken; forward branches predicted as not taken.

Biggest issue here: including all the right instructions. You need to skip any not included based on the **if** branch outcome. I meant this to alternate taken-not taken but actually as written it doesn't do that as it is testing contents of the array, which is not known. In a situation like like, state your assumption on what the branch outcome would be.

To do this properly you need to included the loop-end branch.

		clock number																												
	instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
forbody:	lw R6, 0(R2)	F	D	X	M	W																									
	ori R7, R6, 1		F	D	–	X	M	W																							
	bzero R7, else			F	–	D	X	M	W																						
	predict not taken																														
	li R5, 2					F	D	X	M	W																					
	mult R5, R6, R5						F	D	X	M	W																				
	j endif							F	D	X	M	W																			
else:	move R5, R6								F																						
endif:	sw R5, 0(R1)									F	D	X	M	W																	
	addi R4, R4, 1										F	D	X	M	W																
	addi R1, R1, 4											F	D	X	M	W															
	addi R2, R2, 4												F	D	X	M	W														
fortest	blt R4, R3, forbody													F	D	X	M	W													
	predict taken														F																
forbody:	lw R6, 0(R2)															F	D	X	M	W											
	ori R7, R6, 1																F	–	D	X	M	W									
	bzero R7, else																		F	D	X	M	W								
	predict not taken																														
	li R5, 2																			F	D										
	mult R5, R6, R5																				F										
	mispredict																														
	j endif																														
else:	move R5, R6																					F	D	X	M	W					
endif:	sw R5, 0(R1)																						F	D	X	M	W				
	addi R4, R4, 1																							F	D	X	M	W			
	addi R1, R1, 4																								F	D	X	M	W		
	addi R2, R2, 4																									F	D	X	M	W	
fortest	blt R4, R3, forbody																										F	D	X	M	W

A bit trickier this time; I showed the instructions that are skipped. If your answer is much different, check where you differ from me. We can't eliminate the stall after the load entirely as we have to wait for the memory operation to complete before the next instruction can start its execute.

2. Would a 2-level branch predictor make a difference in this case, versus the simple branch predictor of part (1)(b)? Explain.

   The common problem here was answering as if I asked about a ***2-bit*** predictor, not a ***2-level*** predictor. A 2-bit predictor (notes p 65) is stymied by a regular repeating pattern whereas a 2-level predictor (notes p 75 – this section needs some edits for clarity) is designed for this exact situation.

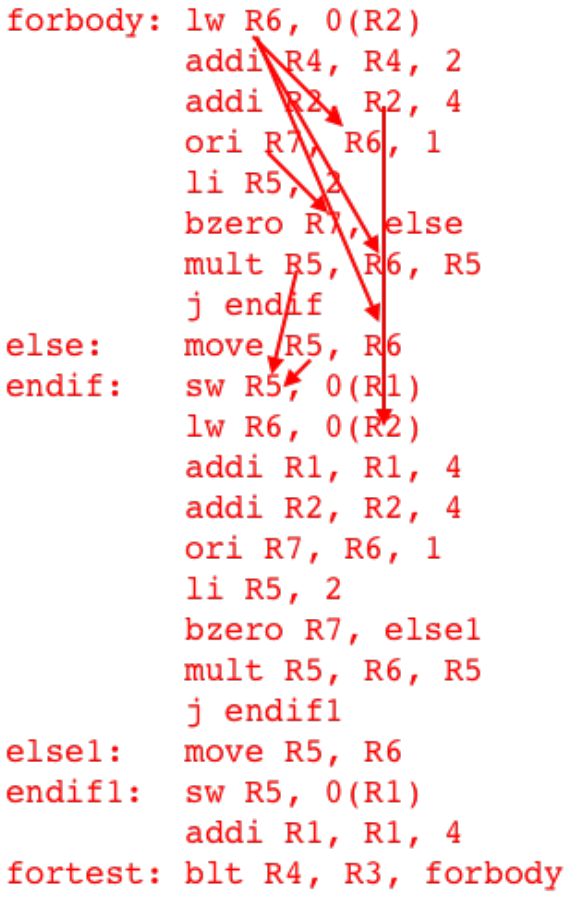
   It depends whether the values of b[i] follow a predictable pattern as they determine the if statement blanch outcome.

3. We now explore the benefit of loop unrolling. Assume we always do an even number of iterations.

   a. Unroll the loop once (write out two iterations of the loop body) and only as necessary rename registers and reorder code to minimize dependences. Mark name dependences using the arrow notation used in the course.

      To do this properly you should rename the repeated labels so they can be told apart. It is ***not necessary*** to do aggressive reordering or renaming as the only dependences that cause stalls in this case follow the `lw` instruction and only one stall needs to be removed in each case.

This question does NOT require a timing diagram. I only show the arrows in the first half since dependences very far apart have no effect.



```
forbody: lw R6, 0(R2)
         addi R4, R4, 2
         addi R2, R2, 4
         ori R7, R6, 1
         li R5, 2
         bzero R7, else
         mult R5, R6, R5
         j endif
else:    move R5, R6
endif:   sw R5, 0(R1)
         lw R6, 0(R2)
         addi R1, R1, 4
         addi R2, R2, 4
         ori R7, R6, 1
         li R5, 2
         bzero R7, else1
         mult R5, R6, R5
         j endif1
else1:   move R5, R6
endif1:  sw R5, 0(R1)
         addi R1, R1, 4
fortest: blt R4, R3, forbody
```

   b. How many stalls are removed in the unrolled version of the loop compared with that of part (1)(b) – aggressive forwarding, no rewriting?

We can lose all the stalls from the load instructions as 2 other instructions can go between the load and the or immediate that uses its result; in the second half. Only one would do as the result of the load is available in the MEM stage.

c. How would Tomasulo's algorithm help in generalising the example?

The biggest issue here is not relating the answer to the example.

Its register renaming would not be much help here though it has the effect of automatically unrolling the loop and can reorder code so it could result in some small gains particularly the stalls for the load. Unrolling the loop generalises more easily in hardware as you don't need to know how many unrolls will work (with a **for** loop, you can work out in advance how many iterations it does in some cases so you can e.g. split the loop into going once then unrolling into 2 instances if the number of iterations is an odd number).