a [Pipelined Performance] (1pt) Three interns, α, β, and θ are in charge of performance characterization of the MN-4363 processor, which represents a basic MIPS implementation. First, all tried to estimate the average CPI (cycles per instruction). To this end, for a benchmarking program, all relied on the ratio of two measurements: $\frac{\texttt{the total number of cycles it took to execute the program}}{\texttt{the number of instructions}} = \frac{\texttt{C}}{\texttt{I}}$.

- α counted the number of instructions *fetched* to estimate `I`.
- β counted the number of instructions *issued* to estimate `I`.
- θ counted the number of instructions *retired* to estimate `I`.

Note: I(nstruction)F(etch)-I(nstruction)D(ecode)-EX(ecute)-MEM(ory)-W(rite)B(ack) constitute the 5 pipeline stages. An instruction is "issued" as it moves from ID to EX; and "retired" as it leaves WB. Recall that branches can earliest be resolved at ID.

a-1 Whose CPI estimation you think is the most accurate? Consider two cases: (i) if branch prediction is implemented; (ii) if branch prediction is not implemented.

The safest approach would be θ's, i.e. counting the number of instructions retired over the entire execution time (in cycles) of the workload mix. The ratio of the cycle count and the instruction count would render the average CPI. The cycle count would already include all hazard-induced stall cycles, while the retired instruction count would exclude potential noise due to nops or instructions on the mispredicted path of branches.

(i) Since there is no guarantee for each fetched and issued instruction to contribute to actual computation as long as the instruction follows a to-be-resolved branch in program order, α and β's approaches may lead to under-estimates for the average CPI. The difference in `I` estimates would decrease as the accuracy of branch prediction increases.

(ii) If the processor excludes branch prediction, all three approaches in counting the total number of instructions would be equivalent (excluding other potential sources of speculative execution). [0.25pts]

a-2 How would the 3 estimates compare? How would each capture the impact of control flow hazards (assuming that branch prediction is implemented)? Consider different pipeline stages a branch can get resolved at.

Fetched vs. issued vs. retired count under MIPS: Even if we rely on branch prediction, branches can earliest be resolved at decode. Thus, fetched count may exceed retired count. If branches are resolved at decode, issued count becomes equivalent to retired count, since in this case, only one instruction following the branch can be fetched at most before the branch outcome gets resolved. On the other hand, if branches are resolved at EX or later stages, issued count may exceed retired count (in this case, instructions on the mispredicted path would be able to get issued before the branch gets resolved); but would not necessarily be equivalent to the fetched count (since fetched instructions on the mispredicted path may get flushed upon the resolution of the branch before being issued). [0.25pts]

a-3 Explain the implications of the three approaches under forwarding.

Impact of forwarding: Forwarding reduces the number of stall cycles due to data hazards. The cycle count captures this effect. Hence, under forwarding (for all instructions on the correct path), the instruction count fetched, issued, and retired would all be the same. If, as in classic MIPS, EX is where a branch can latest be resolved, and EX represents the main destination for forwarding, forwarding wouldn't help instructions on the mispredicted path – which all would be residing in predecessor stages of EX as the branch is getting resolved in EX. [0.25pts]

b [Pipelined Execution] (1pt) α, the infamous architect of the MN-4363 processor – a pipelined implementation of the basic MIPS ISA, wants to implement forwarding from EX(ecute) and MEM(ory) stages to EX. To this end he connected

- The inputs of EXMEM register to the inputs of the IDEX register.
- The inputs of MEMWB register to the inputs of the IDEX register.

He added muxes at the inputs of the IDEX register to control where the inputs to the EX stage should be coming from – i.e. I(nstruction)D(ecode) stage, forward from (the outputs of) EX, or forward from MEM. He claims that he will be able to cover any dependency by inclusion of these two forwarding paths.

Would this implementation work? Explain your answer considering clocking of pipeline registers.

Note: I(nstruction)F(etch)-I(nstruction)D(ecode)-EX(ecute)-MEM(ory)-W(rite)B(ack) constitute the 5 pipeline stages. The pipeline register between two stages is named by concatenating the acronyms for the preceding and succeeding pipeline stages. I.e. IDEX is the register between ID and EX stages; EXMEM, between EX and MEM; MEMWB, between MEM and WB. As opposed to α's suggestion, in a classic MIPS implementation, connecting the outputs of EXMEM to the output of IDEX (over muxes) takes care of forwarding from EX to EX. In a classic MIPS implementation, connecting the outputs of MEMWB to the output of IDEX (over muxes) takes care of forwarding from MEM to EX. The control inputs of the muxes are set to trigger forwarding as necessary.

EX to EX: α is forwarding from the output of the combinational EX logic to the inputs of the pipeline register feeding the EX logic. Assume that two dependent instructions are issued back to back (a (P)roducer-(C)onsumer pair), and at clock cycle t, P is in EX, and C is in ID. The data to be forwarded from P to C will be ready at the end of cycle t, or at the beginning of cycle $t+1$. To enforce correct operation, as C enters EX at the beginning of cycle $t+1$, we should have the forwarded value ready at the inputs of EX logic. At the beginning of cycle $t+1$, IDEX may capture the value to be forwarded indeed. However the forwarded value is the output of combinational EX logic which is sensitive to changes at the input, i.e. IDEX. Eventually, IDEX may get the wrong value. In the classic implementation, since a register output is forwarded (as opposed to the output of a combinational circuit), the forwarded value remains stable.

A similar explanation holds for forwarding from *MEM to EX*. [1pt]

c [Memory] (1pt) Your labmate, α, is in charge of the memory hierarchy design of the MN-4363 processor.

c-1 α claims that a write-through cache can be made to perform very similarly to a write-back cache if enhanced with a write buffer. If the write buffer is large enough, no stalls would be observed as a write-through cache processes a write hit.

- Explain how a write buffer can help.
- Explain whether α's observation is correct.

A write buffer can help in hiding the performance overhead induced by the updates of the copies residing in lower levels, upon a write hit in a write-through cache. The idea is queuing the lines to be updated in a write buffer (of n entries) and resuming execution (following the write – store in program order) as the entries of the write buffer are processed one by one. In this manner, the updates to the lower levels proceed simultaneously with execution of instructions following the store in program order (as long as these instructions do not depend on a store in the write buffer).

A restriction to α's observation is if the number of write requests generated by the processor per second exceeds the number of updates that can be serviced by the lower levels per second significantly. If the gap between processor speed in generating writes and the update speed is huge, increasing the write buffer size wouldn't prevent stalls – even a large buffer will saturate after one point. [0.5pts]

c-2 According to α, a write-through policy should be preferred for faster operation, since a write-through cache can handle the write and the tag check in parallel (which is not the case for a write-back cache).

- Is α correct that a write-back cache cannot handle the actual write and the tag check in parallel? Explain.
- Is α's insight about write-through caches correct? Explain.

α is correct: In a write-back cache, if the tag comparison and the actual write are conducted in parallel, and the corresponding line is dirty, we may loose the only existing valid (up-to-date) copy of this line. In a write-through cache, such an overwrite would be safe, since last time the write-through cache was updated, the memory already got the most up-to-date copy.

While write-through caches are simpler to implement to accommodate the parallel tag check, the update overhead may wipe out the corresponding performance benefits. [0.5pts]

d [Quantitative Analysis] (1pt) Your class-mate θ's study on usage of high-level language constructs suggests that procedure calls are one of the most expensive operations. He came up with an ISA which reduces the loads and stores normally associated with procedure calls and returns. The first thing he did was run some experiments with and without this optimization. His experiments deployed the same state-of-the-art optimizing compiler that will be used with either version of the computer. These experiments revealed the following information: (i) The clock cycle time of the optimized version is 5% lower than the unoptimized version. (ii) Thirty percent of the instructions in the unoptimized version are loads or stores. (iii) The optimized version executes two-thirds as many loads and stores as the unoptimized version. For all other instructions the dynamic execution counts are unchanged. (iv) Every instruction (including load and store) in the unoptimized version takes one clock cycle. (v) Due to the optimization, the procedure call and return instructions take one extra cycle in the optimized version, and these instructions account for 5% of total instruction count in the optimized version.

Is θ's optimization working? Justify your answer.

Execution times (E) to be compared: $I(nstruction)\ C(ount) \times CPI \times clk\ (period)$.

$E_{unoptimized} = IC_{unoptimized} \times CPI_{unoptimized} \times clk_{unoptimized} = IC_{unoptimized} \times 1 \times clk_{unoptimized}$.

$E_{optimized} = IC_{optimized} \times CPI_{optimized} \times clk_{optimized}$. 30% of instructions are load/store, and the optimized version executes 2/3 of them; hence, the optimized version executes $30\% \times 1/3 = 10\%$ less instructions. $E_{optimized} = (0.9 \times IC_{unoptimized}) \times (0.95 \times 1 + 0.05 \times 2) \times (0.95 \times clk_{unoptimized})$.

$\implies E_{optimized} \approx 0.89775 \times E_{unoptimized}$. The optimization is working.

0.1pts for comparing execution times. 0.3pts for each correct component of calculation: IC, CPI and clk.