Problem M4.1: Fully-Bypassed Simple 5-Stage Pipeline

We have reproduced the fully bypassed 5-stage MIPS processor pipeline from Lecture 7 in Figure M4.1-A. In this problem, we ask you to write equations to generate correct bypass and stall signals. Feel free to use any symbol introduced in the lecture.

Problem M4.1.A — Stall

Do we still need to stall this pipeline? If so, explain why. (1) Write down the correct equation for the stall condition and (2) give an example instruction sequence which causes a stall.

Problem M4.1.B — Bypass Signal

In Lecture 7, we gave you an example of bypass signal (ASrc) from EX stage to ID stage. In the fully bypassed pipeline, however, the mux control signals become more complex, because we have more inputs to the muxes in the ID stage.

Write down the bypass condition for each bypass path in Mux 1. Please indicate the priority of the signals; that is, if all bypass conditions are met, indicate which signals have the highest and the lowest priorities.

Bypass $_{\text{EX->ID}}$ ASrc = $(rs_D{=}ws_E)$.we-bypass$_E$.re1$_D$ (given in Lecture 7)

Bypass $_{\text{MEM->ID}}$ =

Bypass $_{\text{WB->ID}}$ =

Priority:

Problem M4.1.C — Partial Bypassing

While bypassing gives us a performance benefit, it may introduce extra logic in critical paths and may force us to lower the clock frequency. Suppose we can afford to have only one bypass in the datapath. How would you justify your choice? Argue in favor of one bypass path over another.

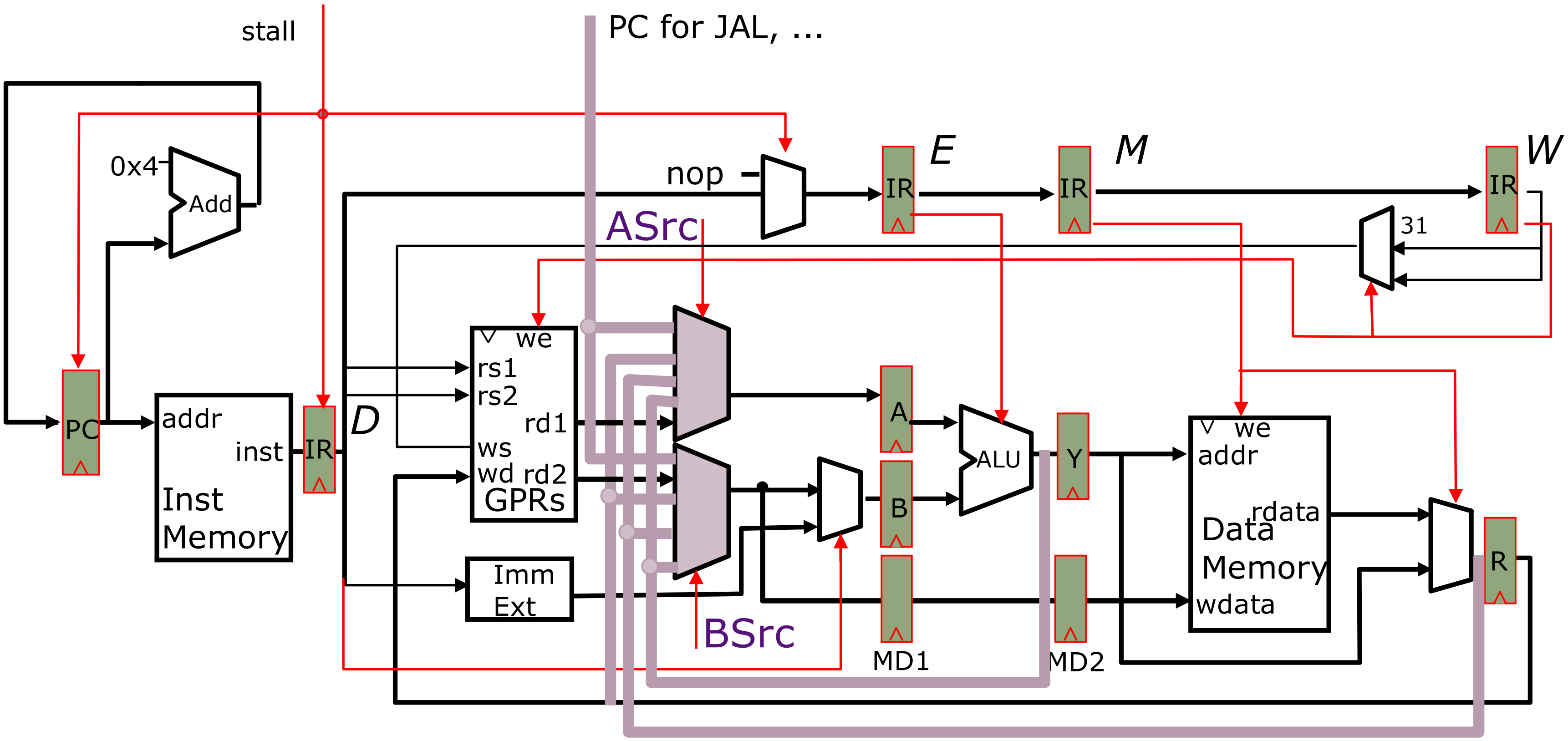


Figure M4.1-A: Fully-Bypassed MIPS Pipeline

Problem M4.2: Basic Pipelining

Unlike the Harvard-style (separate instruction and data memories) architectures, machines using the Princeton-style have a shared instruction and data memory. In order to reduce the memory cost, Ben Bitdiddle has proposed the following two-stage Princeton-style MIPS pipeline to replace a single-cycle Harvard-style pipeline from our lectures.

Every instruction takes exactly two cycles to execute (i.e., instruction fetch and execute) and there is no overlap between two sequential instructions; that is, fetching an instruction occurs in the cycle following the previous instruction's execution (no pipelining).

Assume that the new pipeline does not contain a branch delay slot. Also, don't worry about self-modifying code for now.

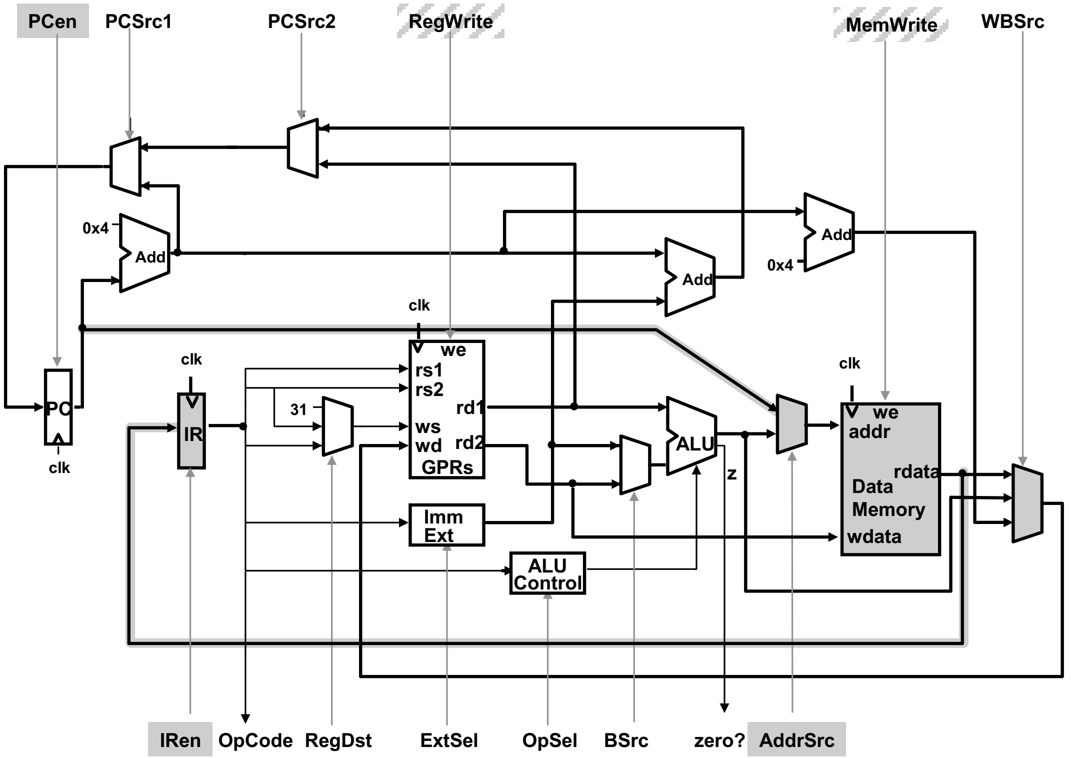


Figure M4.2-A: Two-stage pipeline, Princeton-style

Problem M4.2.A **Mux Control Signals (1)**

Please complete the following control signals. You are allowed to use any internal signals (e.g., OpCode, PC, IR, zero?, rd1, data, etc.) but not other control signals (ExtSel, IRSrc, PCSrc, etc.).

Example syntax: PCEn = (OpCode == ALUOp) or ((ALU.zero?) and (not (PC == 17)))

You may also use the variable S which indicates the pipeline's operation phase at a given time.

```
S := I-Fetch | Execute  (toggles every cycle)
```

PCEn =

IREn =

AddrSrc = Case ______________

_____________ => PC

_____________ => ALU

Problem M4.2.B — Modified pipeline

After having implemented his proposed architecture, Ben has observed that a lot of datapath is not in use because only one phase (either I-Fetch or Execute) is active at any given time. So he has decided to fetch the next instruction during the Execute phase of the previous instruction.

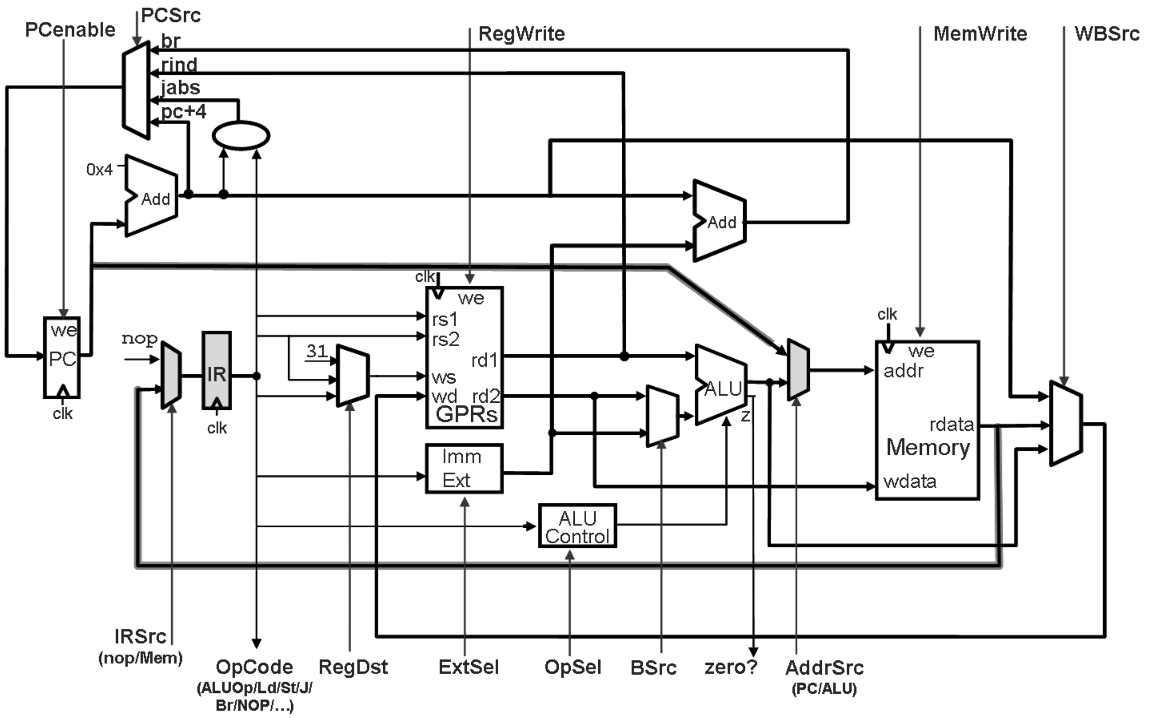


Figure M4.2-B: Modified Two-stage Princeton-style MIPS Pipeline

Do we need to stall this pipeline? If so, for each cause (1) write down the cause in one sentence and (2) give an example instruction sequence. If not, explain why. (Remember there is **no** delay slot.)

Problem M4.2.C — Mux Control Signals (2)

Please complete the following control signals in the modified pipeline. As before, you are allowed to use any internal signals (e.g., OpCode, PC, IR, zero?, rd1, data, etc.) but not other control signals (ExtSel, IRSrc, PCSrc, etc.)

PCEnable =

AddrSrc = Case ______________

_____________ => PC

_____________ => ALU

IRSrc = Case ______________

_____________ => nop

_____________ => Mem

Problem M4.2.D

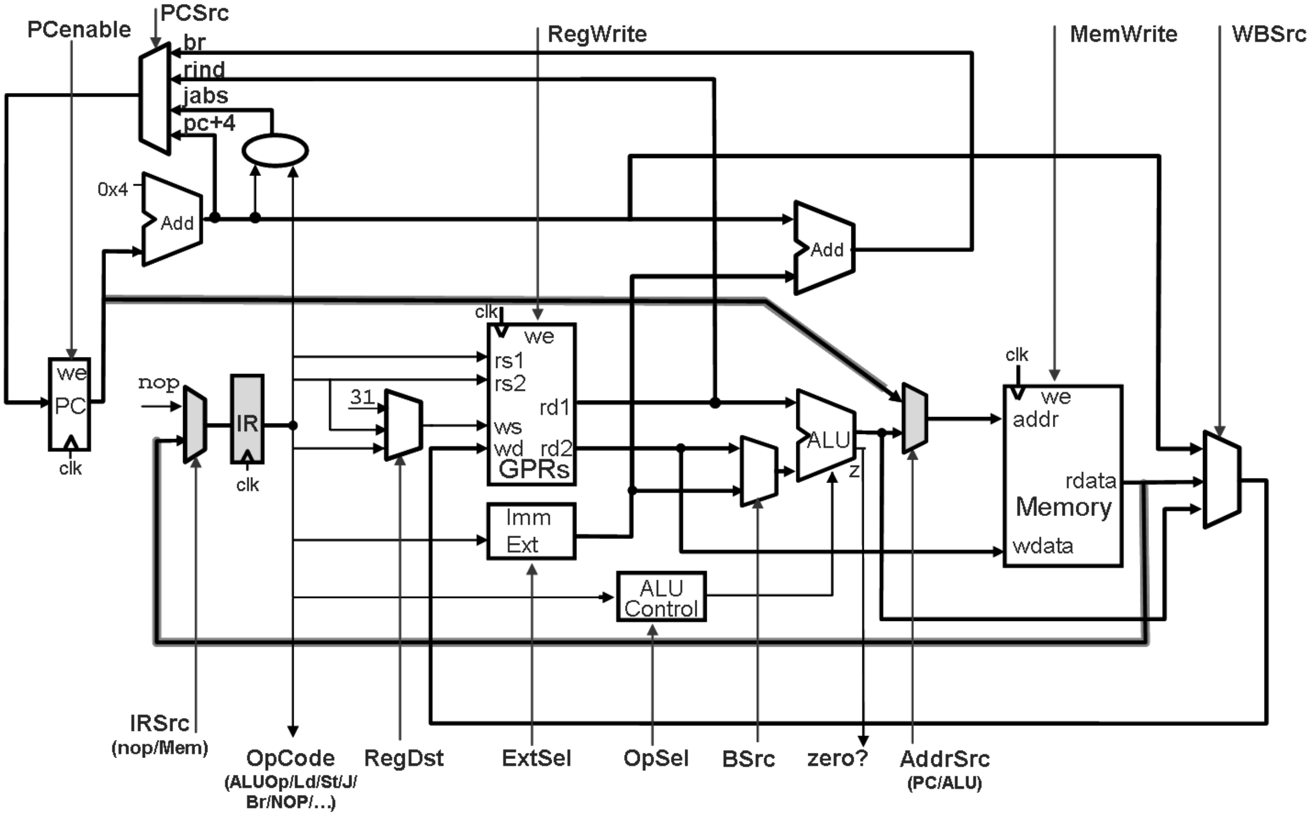
Now we are ready to put Ben's machine to the test. We would like to see a cycle-by-cycle animation of Ben's two-stage pipelined, Princeton-style MIPS machine when executing the instruction sequence below. In the following table, each row represents a snapshot of some control signals and the content of some special registers for a particular cycle. Ben has already finished the first two rows. Complete the remaining entries in the table. Use * for "don't care".

Label	Address	Instruction
I_1	100	ADD
I_2	104	LW
I_3	108	J I_7
I_4	112	LW
I_5	116	ADD
I_6	120	SUB
I_7	312	ADD
I_8	316	ADD

Time	PC	"IR"	PCenable	PCSrc1	AddrSrc	IRSrc
t_0	I_1:100	-	1	pc+4	PC	Mem
t_1	I_2:104	I_1	1	Pc+4	PC	Mem
t_2						
t_3						
t_4						
t_5						
t_6						

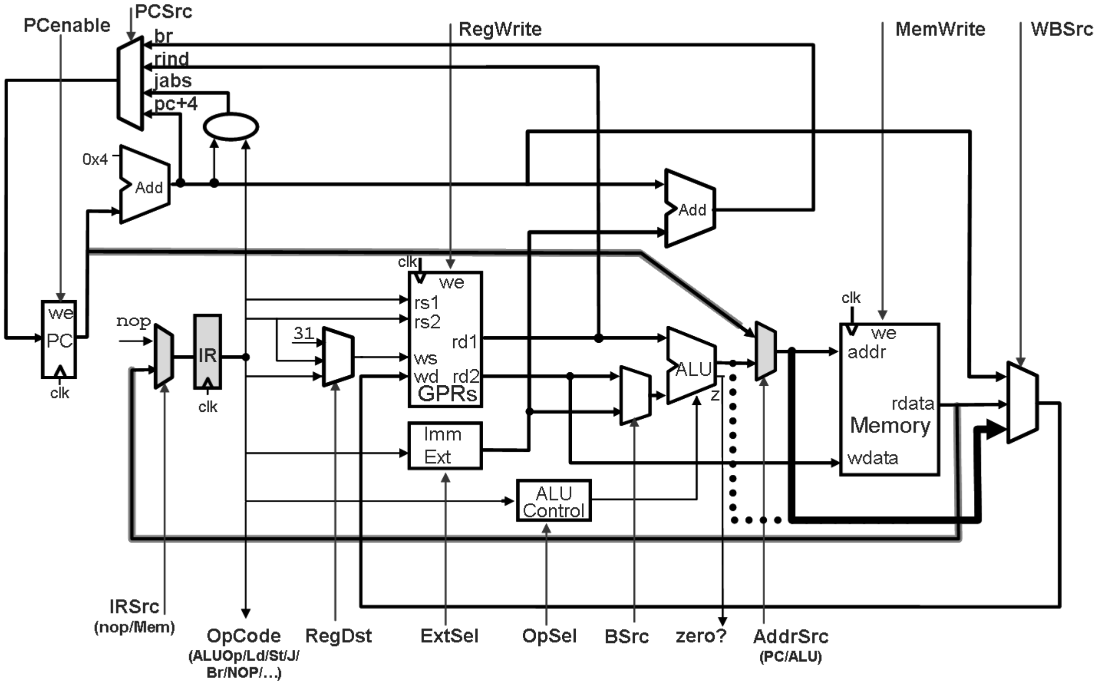
Problem M4.2.E Self-Modifying Code

Suppose we allow self-modifying code to execute, i.e., store instructions can write to the portion of memory that contains executable code. Does the two-stage Princeton pipeline need to be modified to support such self-modifying code? If so, please indicate how. You may use the diagram below to draw modifications to the datapath. If you think no modifications are required, explain why.



Problem M4.2.F

To solve a chip layout problem Ben decides to reroute the input of the WB mux to come from after the AddrSrc MUX rather than ahead of the AddrSrc MUX. (The new path is shown with a bold line, the old in a dotted line.) The rest of the design is unaltered.



How does this break the design? Provide a code sequence to illustrate the problem and explain in one sentence what goes wrong.

Problem M4.2.G Architecture Comparison

Give one advantage of the Princeton architecture over the Harvard architecture.

Give one advantage of the Harvard architecture over the Princeton architecture.

Problem M4.3: Processor Design (Short Yes/No Questions)

The following statements describe two variants of a processor which are otherwise identical. In each case, circle "**Yes**" if the variants might generate different results from the same compiled program, circle "**No**" otherwise. You must also briefly explain your reasoning. Ignore differences in the time that each machine takes to execute the program.

Problem M4.3.A Interlock vs. Bypassing

Pipelined processor A uses interlocks to resolve data hazards, while pipelined processor B has full bypassing.

Yes / No

Problem M4.3.B Delay Slot

Pipelined processor A uses branch delay slots to resolve control hazards, while pipelined processor B kills instructions following a taken branch.

Yes / No

Problem M4.3.C Structural Hazard

Pipelined processor A has a single memory port used to fetch instructions and data, while pipelined processor B has no structural hazards.

Yes / No

Problem M5.1: Pipelined Cache Access

This problem requires the knowledge of Lecture 3. Please, review it before answering the following questions. You may also want to take a look at pipeline lectures if you do not feel comfortable with the topic.

Problem M5.1.A

Ben Bitdiddle is designing a five-stage pipelined MIPS processor with separate 32 KB direct-mapped primary instruction and data caches. He runs simulations on his preliminary design, and he discovers that a cache access is on the critical path in his machine. After remembering that pipelining his processor helped to improve the machine's performance, he decides to try applying the same idea to caches. Ben breaks each cache access into three stages in order to reduce his cycle time. In the first stage the address is decoded. In the second stage the tag and data memory arrays are accessed; for cache reads, the data is available by the end of this stage. However, the tag still has to be checked—this is done in the third stage.

After pipelining the instruction and data caches, Ben's datapath design looks as follows:

I-Cache Address Decode	I-Cache Array Access	I-Cache Tag Check	Instruction Decode & Register Fetch	Execute	D-Cache Address Decode	D-Cache Array Access	D-Cache Tag Check	Write-back

Alyssa P. Hacker examines Ben's design and points out that the third and fourth stages can be combined, so that the instruction cache tag check occurs in parallel with instruction decoding and register file read access. If Ben implements her suggestion, what must the processor do in the event of an instruction cache tag mismatch? Can Ben do the same thing with load instructions by combining the data cache tag check stage with the write-back stage? Why or why not?

Problem M5.1.B

Alyssa also notes that Ben's current design is flawed, as using three stages for a data cache access won't allow writes to memory to be handled correctly. She argues that Ben either needs to add a fourth stage or figure out another way to handle writes. What problem would be encountered on a data write? What can Ben do to keep a three-stage pipeline for the data cache?

Problem M5.1.C

With help from Alyssa, Ben streamlines his design to consist of eight stages (the handling of data writes is not shown):

I-Cache Address Decode	I-Cache Array Access	I-Cache Tag Check, Instruction Decode & Register Fetch	Execute	D-Cache Address Decode	D-Cache Array Access	D-Cache Tag Check	Write-Back

Both the instruction and data caches are still direct-mapped. Would this scheme still work with a set-associative instruction cache? Why or why not? Would it work with a set-associative data cache? Why or why not?

Problem M5.1.D

After running additional simulations, Ben realizes that pipelining the caches was not entirely beneficial, as now the cache access latency has increased. If conditional branch instructions resolve in the Execute stage, how many cycles is the processor's branch delay?

Problem M5.1.E

Assume that Ben's datapath is fully-bypassed. When a load is executed, the data becomes available at the end of the D-cache Array Access stage. However, the tag has not yet been checked, so it is unknown whether the data is correct. If the load data is bypassed immediately, before the tag check occurs, then the instruction that depends on the load may execute with incorrect data. How can an interlock in the Instruction Decode stage solve this problem? How many cycles is the load delay using this scheme (assuming a cache hit)?

Problem M5.1.F

Alyssa proposes an alternative to using an interlock. She tells Ben to allow the load data to be bypassed from the end of the D-Cache Array Access stage, so that the dependent instruction can execute while the tag check is being performed. If there is a tag mismatch, the processor will wait for the correct data to be brought into the cache; then it will re-execute the load and all of the instructions behind it in the pipeline before continuing with the rest of the program. What processor state needs to be saved in order to implement this scheme? What additional steps need to be taken in the pipeline? Assume that a **DataReady** signal is asserted when the load data is available in the cache, and is set to 0 when the processor restarts its execution (you don't have to worry about the control logic details of this signal). How many cycles is the load delay using this scheme (assuming a cache hit)?

Problem M5.1.G

Ben is worried about the increased latency of the caches, particularly the data cache, so Alyssa suggests that he add a small, unpipelined cache in parallel with the D-cache. This "fast-path" cache can be considered as another level in the memory hierarchy, with the exception that it will be accessed simultaneously with the "slow-path" three-stage pipelined cache. Thus, the slow-path cache will contain a superset of the data found in the fast-path cache. A read hit in the fast-path cache will result in the requested data being available after one cycle. In this situation, the simultaneous read request to the slow-path cache will be ignored. A write hit in the fast-path cache will result in the data being written in one cycle. The simultaneous write to the slow-path cache will proceed as normal, so that the data will be written to both caches. If a read miss occurs in the fast-path cache, then the simultaneous read request to the slow-path cache will continue to be processed—if a read miss occurs in the slow-path cache, then the next level of the memory hierarchy will be accessed. The requested data will be placed in both the fast-path and slow-path caches. If a write miss occurs in the fast-path cache, then the simultaneous write to the slow-path cache will continue to be processed as normal. The fast-path cache uses a no-write allocate policy, meaning that on a write miss, the cache will remain unchanged—only the slow-path cache will be modified.

Ben's new pipeline design looks as follows after implementing Alyssa's suggestion:

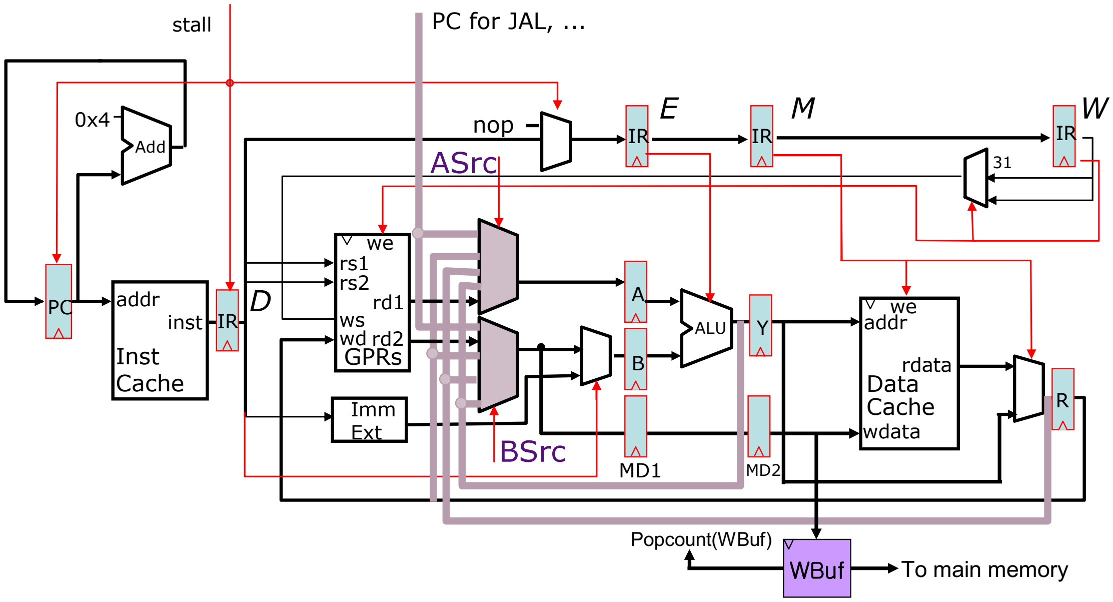
I-Cache Address Decode	I-Cache Array Access	I-Cache Tag Check, Instruction Decode & Register Fetch	Execute	Fast-Path D-Cache Access and Tag Check & Slow Path D-Cache Address Decode	Slow-Path D-Cache Array Access	Slow-Path D-Cache Tag Check	Write-Back

The number of processor pipeline stages is still eight, even with the addition of the fast-path cache. Since the processor pipeline is still eight stages, what is the benefit of using a fast-path cache? Give an example of an instruction sequence and state how many cycles are saved if the fast-path cache always hits.

Problem M5.2: Write Buffer for Data Cache (2005 Fall Part C)

In order to boost the performance of memory writes, Ben Bitdiddle has proposed to add a write buffer to our 5-stage fully-bypassed MIPS pipeline as shown below. Assuming a write-through/write no-allocate cache, every memory write request will be queued in the write buffer in the MEM stage, and the pipeline will continue execution without waiting for writes to be completed. A queued entry in the write buffer gets cleared only after the write operation completes, so the maximum number of outstanding memory writes is limited by the size of the write buffer.

Please answer the following questions.



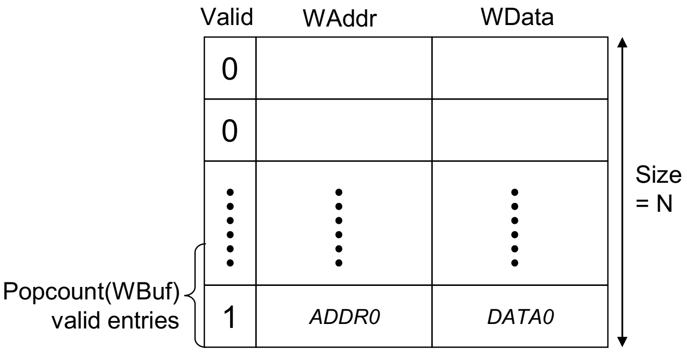
Problem M5.2.A

Ben wants to determine the size of the write buffer, so he runs benchmark X to get the observation below. What will be the average number of writes in flight (=the number of valid entries in the write buffer on average)?

1) The CPI of the benchmark is 2.
2) On average, one of every 20 instructions is a memory write.
3) Memory has a latency of 100 cycles, and is fully pipelined.

Problem M5.2.B

Based on the experiment in the previous question, Ben has added the write buffer with N entries to the pipeline. (Do not use your answer in M5.2A to replace N.) Now he wants to design a stall logic to prevent a write buffer overflow. The structure of the write buffer is shown in the figure below. `Popcount(WBuf)` gives the number of valid entries in the write buffer at any given moment.

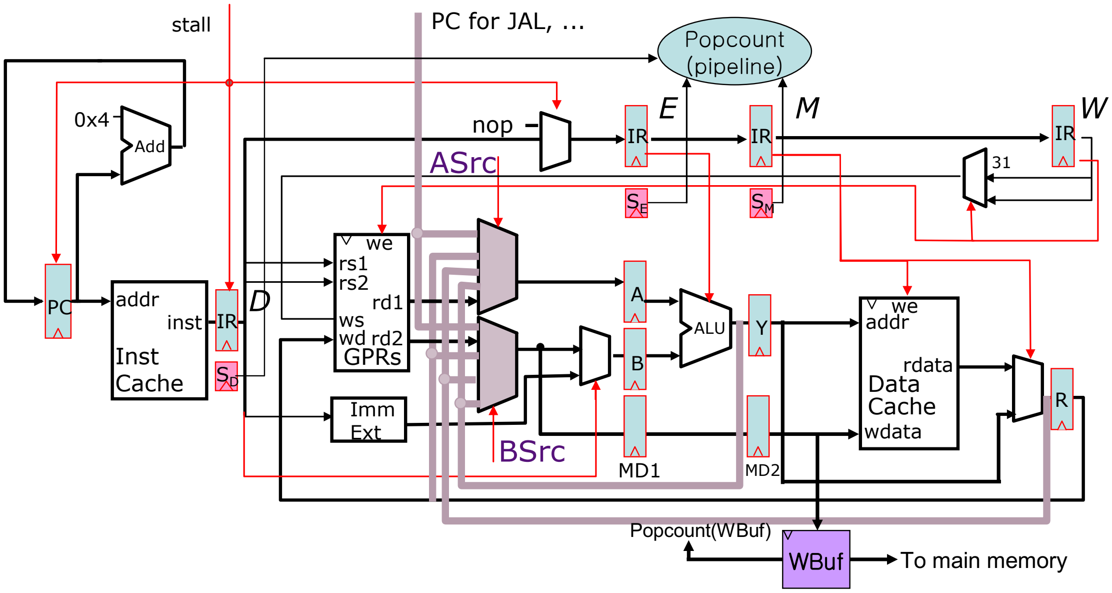


Please write down the stall condition to prevent write buffer overflows. You should derive the condition without assuming any modification of the given pipeline. You can use Boolean and arithmetic operations in your stall condition.

Stall =

Problem M5.2.C

In order to optimize the stall logic, Ben has decided to add a predecode bit to detect store instructions in the instruction cache (I-Cache). That is, now every entry in the I-Cache has a store bit associated with it, and it propagates through the pipeline with an S_{stage} bit added to each pipeline register (except the one between MEM and WB stages) as shown below. `Popcount(Pipeline)` gives the number of store instructions that are in flight (= number of S_{stage} bits set to 1).



How will this optimization change the stall condition, if at all?

Stall =