Pipelining HW

Q. Can a MIPS SW instruction executing in a simple 5-stage pipelined implementation have a data dependency hazard of any type resulting in a nop bubble? If so, show an example; if not, prove it cannot happen. You may assume any data forwarding paths that could prevent a bubble have been implemented.

Q. Consider a dynamic instruction execution (an execution trace, in other words) that consists of repeats of code in this pattern:

```
label_k:
LW $1, offset_i( $2 )
BReq $1, $ZER0, label_i
```

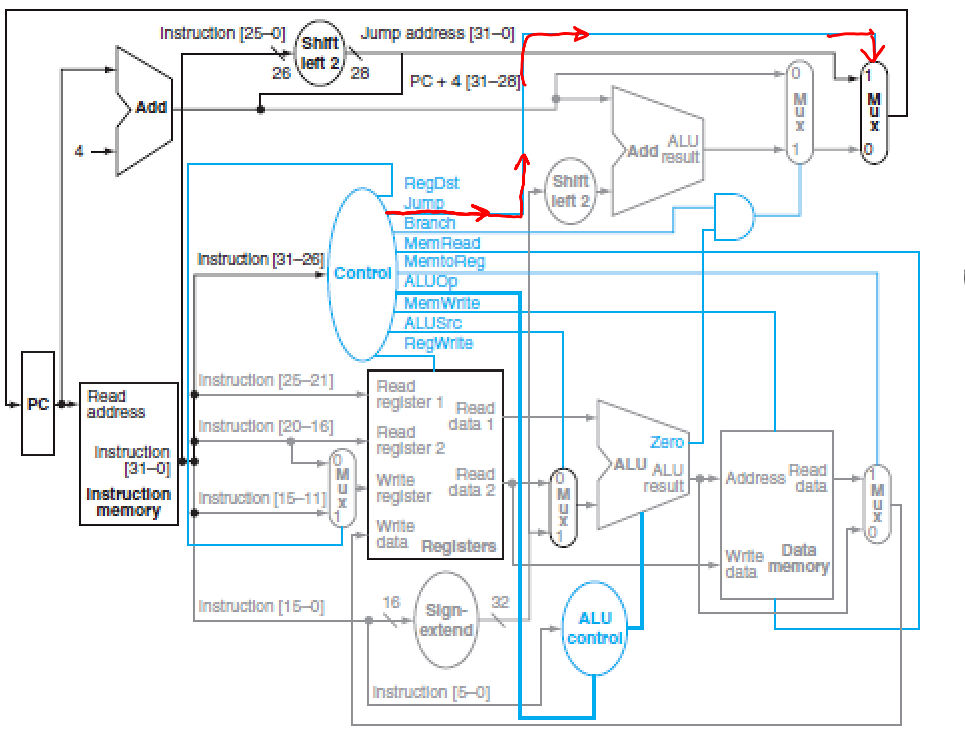
Suppose this runs on M1, a 5-stage pipelined MIPS machine with full data forwarding and stalls and nullification as needed for data and control hazards. Ignoring pipeline fill and drain overhead, what is the average branch CPI if every branch is taken? What is the average LW CPI? What is the overall average CPI?

Q. Compare the performance of M1 with M2, a simple 5-stage pipelined MIPS with no forwarding or stalls or nulls: all hazards are handled by the compiler inserting NOPS as needed. Also compare to M3, a single-cycle MIPS implementation with a 800ps clock cycle. The pipelined machines have a 200ps clock cycle.

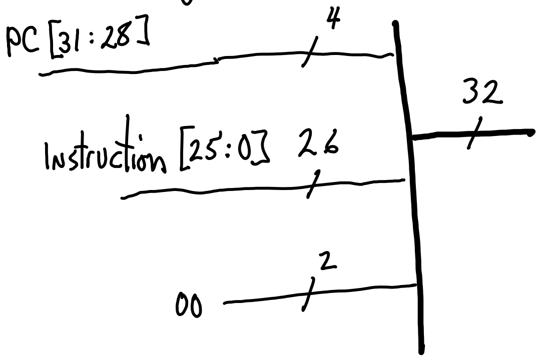
Q. The previous question did not consider hierarchical refinement. That is, we might consider implementing parallelism in various ways in the functional units by, for instance, interleaving and pipelining. As a limiting case, using Amdahl's law and without any restrictions on the possible improvements except minimum flip-flop timing considerations, derive an absolute limit on speedup.

Q. The single-cycle MIPS implementation presented in the P&H textbook has a micro-controller ROM presented in Figure 4.22. Only one state is needed for execution of each instruction: There is one ROM word for each opcode and the instruction's opcode bits are fed directly to the ROM's address input. Figure 4.24 (below) shows the single-cycle MIPS, including elements that implement the J, or jump, instruction. Add a row to the ROM to implement control signals for J-format instructions.

NB--Having 6-bit opcodes implies a 64-word ROM. We are only implementing five of those for the opcodes 000000 (R-format), 100011 (LW), 101011 (SW), 000100 (BR), and 000010 (J). We ignore the other ROM words. Figure 4.22 has been rearranged (below) to make the ROM look like a typical memory layout.



NB- the jump address is formed this way:



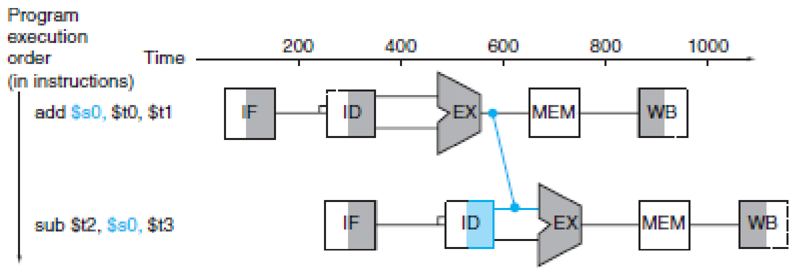
OPCODE (address)

ROM content

OPCODE (address)	Branch	MemW	MemR	RegW	Mem2reg	ALUsrc	RegDst	Jump
000000	0	0	0	1	0	0	1	
100011	0	0	1	1	1	1	0	
101011	0	1	0	0	X	1	X	
000100	1	0	0	0	X	0	X	
000010								

* X means "don't care"

Q. Figure 4.29 (below) in the P&H text shows pipeline graphical notation for execution of an ADD instruction. Gray areas represent active hardware elements in each clock cycle. The register file is shown twice, once in ID and once in WB. Since writes into the register file occur on the falling clock edge, the register file is gray for the first half of the WB cycle as data is sampled. An instruction reads the register file during the second half of its ID cycle, so the right half of the register file is gray in ID. The register file hardware is shown twice for each instruction's execution. (Memory is not gray because ADD does not access memory.) By this convention, if the register file's registers were positive-edge triggered, the register file would be gray for the entire ID cycle as the read would initiate at the positive clock edge that moved the ADD instruction into the ID stage and would not complete until next positive clock edge clocked the data read into the EX stage. How would you handle write-back if the register file were positive-edge triggered?



An execution trace contains the following mix of MIPS instructions: 45% 3-register operate, 25% loads, 10% stores, 20% branches (1/2 are taken). We are considering our options for pipelining the 1-cycle MIPS implementation. We are considering a 5-stage implementation. The clock cycle time is 800 ps for the single-cycle implementation and 200 ps for the pipelined version.

Q. Let's analyze an optimistic limiting case. Let's assume all operate instructions have forwarding to avoid data dependency stalls. Further, let's assume all loads and stores cause no bubbles because the compiler always moves independent instructions into any delay slots. Further, assume all branching avoids data dependencies in a similar manner. With branch hazard detection, the only bubbles will come from taken branches. What is the average CPI in this case (ignore pipe fill and drain overhead)? What is the speed-up of the pipelined version versus the single-cycle version?

Q. A pessimistic assumption is just the opposite: all possible delays are incurred. Find the CPI and speed-up in this case.

Q. Going back to the optimistic case, let's suppose we spend more to make the pipeline deeper: We split each stage into k sub-stages. The pipeline depth is then 5k at an added expense of O(5k) dollars. What is the limiting speed-up in this case?

Q. Considering the costs, which alternative seems the most practical? Explain.

Suppose functional-unit delays in 5-stage pipelined implementation are (200, 100, 250, 200, 100) in ps (IMEM, DECODE, EXECUTE, DMEM, WRITE-BACK). We are considering two possible improvements: (1) add look-ahead-carry circuitry which typically speeds up an ALU by 25% or (2) replace the register file with a new one containing double-word registers and double the datapath to DMEM. So, data fetches are not slowed, and data fetching only needs half the number of LW executions (program data access pattern permiting).

Q. Assuming both improvements cost about the same, which is most cost-effective? Is doing both a possibility worth considering? Assume the typical job mix is 50% ALU operations, 35% load-store, and 15% branches. Analyze using Amdahl's Law.

Q. Now redo the analysis comparing execution times directly. That is, account for CR speedup and code size reduction.

Q. In a 5-stage pipelined MIPS, an exception can be caused by an instruction in any functional unit except WB. In fact, four instructions could cause exceptions simultaneously. On top of that, a hardware interrupt might also occur in the same cycle. The cause register can record all the causes. If the EPC is to hold the address of only one instruction, which instruction's address is the most logical? Does it matter to the functionality of the operating system?

Q. Suppose an instruction stream looks like this (first executed is at top):

LW
ADD
LW
SUB
JR
ADD

Suppose the 5-stage MIPS processor is executing in user mode and the JR (jump via register) instruction's argument register contains 80008000. Which instructions will continue execution and which will be flushed from the pipeline? In addition, suppose the SUB instruction causes an overflow exception. The cause register acts like a stack and can push another cause. What happens to the OS routine that started in response to the first exception? What address could be put in the EPC to handle this. Assuming we did as you suggest, does this processor have precise exceptions?

Q. Suppose we add 1-bit branch prediction to our 5-stage MIPS, but we put the branch instruction's full 32-bit address into our branch prediction table instead of just the low address bits. Does this solve the problem of mis-prediction on entering a loop? How?

Q. Suppose we know that $2 always has the value 400 before entering the loop. Show how a compiler could use unrolling and register renaming to take advantage of a VLIW machine which has two parallel integer units.

```
Loop:
   SUB $4, $3, $1
   ADD $4, $6, $4
   SW $4, base( $2 )
   SUB $2, $2, 4
   BRneq $2, $ZERO, Loop
```