Chapter 2

1. Equation 4, which relates the performance of an ideal pipeline to pipeline depth, looks very similar to Amdahl's law. Describe the relationship between the terms in these two equations, and develop an intuitive explanation for why the two equations are so similar.

   The nonpipelined combinational delay T is comparable to the serial execution time under Amdahl's law. Pipelining effectively parallelizes this delay over k pipe stages. However, just as there is a serial portion (1-p) under Amdahl's law that cannot be parallelized, the latch overhead S cannot be parallelized. Hence the speedup equations look very similar.

2. Using Equation 7, the cost/performance optimal pipeline depth k_{opt} can be computed using parameters G, T, L, and S. Compute k_{opt} for the pipelined floating-point multiplier example in Section 2.1 by using the chip count as the cost terms (G = 175 chips and L = 82/2 = 41 chips per interstage latch) and the delays shown for T and S (T = 400ns, S = 22ns). How different is k_{opt} from the proposed pipelined design?

   $$k_{opt} = \sqrt{\frac{GT}{LS}} = \sqrt{\frac{175x400}{41x22}} = \sqrt{77.6} = 8.8$$

   This differs substantially from the 3 stage pipeline.

3. Identify and discuss two reasons why Equation 4 is only useful for naive approximations of potential speedup from pipelining.

   Some possible reasons are: the metrics for cost (G and L) are not very useful in modern technology, since wire delay is more important than area; combinational logic cannot always be pipelined uniformly over k stages; the latch overhead varies depending on where the latches are placed due to fan-in and fan-out variation in the combination logic; etc.

4. Consider that you would like to add a *load-immediate* instruction to the TYP instruction set and pipeline. This instruction extracts a 16-bit immediate value from the instruction word, sign-extends the immediate value to 32 bits, and stores the result in the destination register specified in the instruction word. Since the extraction and sign-extension can be accomplished without the ALU, your colleague suggests that such instructions be able to write their results into the register in the decode (ID) stage. Using the hazard detection algorithm described in Figure 2-15, identify what additional hazards such a change might introduce.

   Since there are now 2 stages that write the register file (ID and WB), WAW hazards may also occur in addition to RAW hazards. WAW hazards exist with respect to instructions that are ahead of the load immediate in the pipeline. WAR hazards do exist since the ID register write stage is earlier than the RD register read stage (assuming a write-before-read register file). If the register file is read-before-write, the ID write occurs after the RD read, and therefore WAR hazards do not exist.

5. Ignoring pipeline interlock hardware (discussed in Problem 6), what additional pipeline resources does the change outline in Problem 4 require? Discuss these resources and their cost.

Since there are 2 stages that write the register file (ID and WB), the register file must have two write ports. Additional write ports are expensive, since they require the RF array and bitcells to be redesigned to support multiple writes per cycle. Alternatively, a banked RF design with bank conflict resolution logic could be added. However, this would require additional control logic to stall the pipeline on bank conflicts.

6. Considering the change outlined in Problem 4, redraw the pipeline interlock hardware shown in Figure 2-18 to correctly handle the load-immediate instructions.

   The modified figure should show the ID stage destination latch connected to a second write port register identifier input. Further, comparators that check the ID stage destination latch against the destination latches of instructions further in the pipeline should drive a stall signal to handle WAW hazards.

7. Consider that you would like to add byte-wide ALU instructions to the TYP instruction set and pipeline. These instructions have semantics that are otherwise identical to the existing word-width ALU instructions, except that the source operands are only one byte wide and the destination operand is only one byte wide. The byte-wide operands are stored in the same registers as the word-wide instructions, in the low-order byte, and the register writes must only affect the low-order byte (i.e., the high-order bytes must remain unchanged). Redraw the RAW pipeline interlock detection hardware shown in Figure 2-18 to correctly handle these additional ALU instructions.

   Each register specified in the pipeline latches (destination and two sources) should now include a type identifier to differentiate byte and word references. The logic should be augmented to generate a stall signal whenever a source matches an earlier destination, but the read cannot be satisfied by the inflight write since the operand types do not match (i.e. a byte write cannot bypass to a word read). An alternative solution is to add a third register file read port, along with a bypass network and hazard detection logic for this third read port. The third read port will be used by byte-wide instructions to read the register word so the new low-order byte can be merged into it and the entire word written back (or bypassed to dependent instructions).

8. Consider adding a store instruction with indexed addressing mode to the TYP pipeline. This store differs from the existing store with *register+immediate* addressing mode by computing its effective address as the sum of two source registers, that is, stx r3,r4,r5 performs r3←MEM[r4+r5]. Describe the additional pipeline resources needed to support such an instruction in the TYP pipeline. Discuss the advantages and disadvantages of such an instruction.

   This instruction must read three operands from the register file. Hence a third read port must be added, along with hazard detection logic and bypass networks for this third operand. Alternatively, an underpipelined implementation that stalls the pipeline for this instruction can be used. This implementation will never outperform an “add, st” pair where a separate add performs the indexed addressing mode computation.

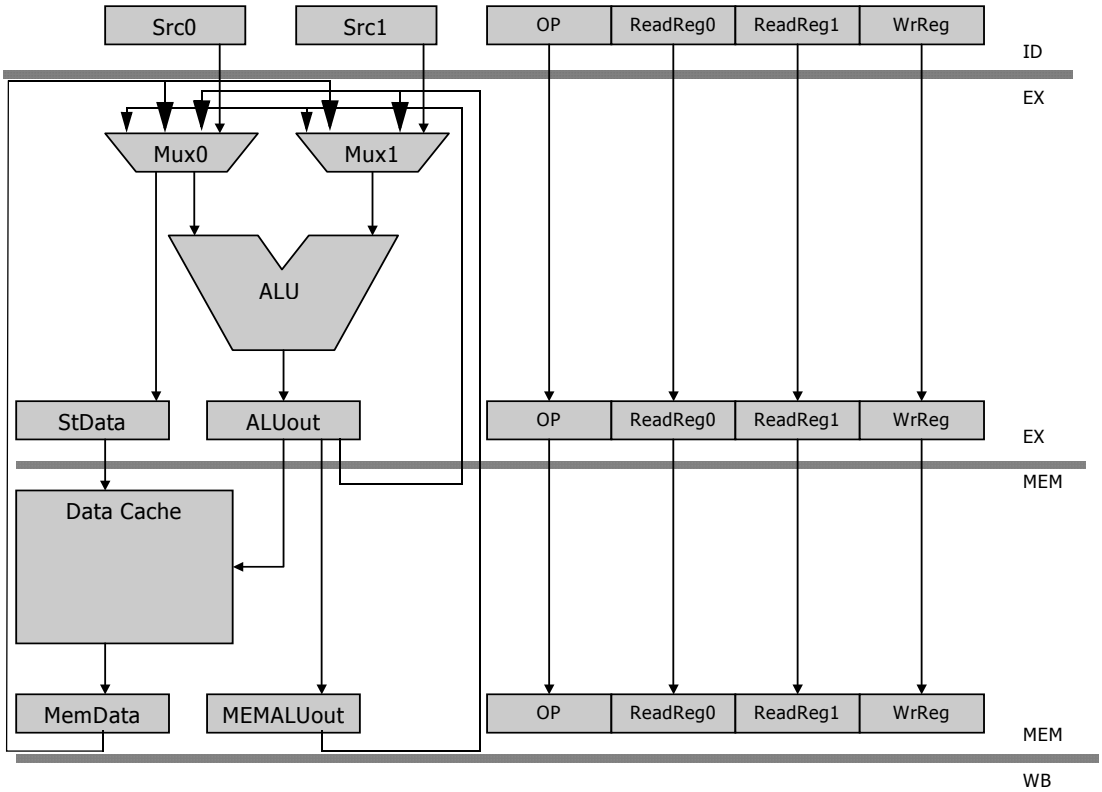
9. Consider adding a *load-update* instruction with *register+immediate and postupdate* addressing mode. In this addressing mode, the effective address for the load is computed as register+immediate, and the resulting address is written back into the base register. That is, lwu r3,8(r4) performs r3←MEM[r4+8]; r4←r4+8. Describe the additional pipeline resources needed to support such an instruction in the TYP pipeline.

This instruction performs two register writes. It can either be underpipelined, by forcing the second write to stall the pipeline, or, to maintain a fully pipelined implementation, a second write port must be added to the register file. In addition, the hazard detection and bypass network must be augmented to handle this special case of a second register write.

10. Given the change outlined in Problem 9, redraw the pipeline interlock hardware shown in Figure 2-20 to correctly handle the load-update instruction.

    The figure should be modified to include the changes described above.

11. Bypass network design: given the following ID, EX, MEM, and WB pipeline configuration, draw all necessary Mux0 and Mux1 bypass paths to resolve RAW data hazards. Assume that load instructions are always separated by at least one independent instruction (possibly a NOP) from any instruction that reads the loaded register (hence you never stall due to a RAW hazard).



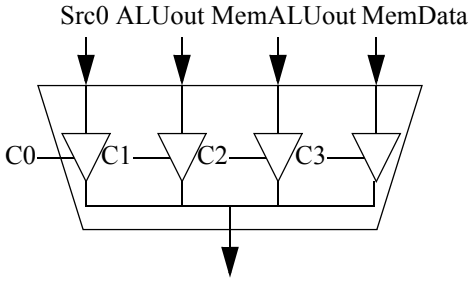
12. Given the forwarding paths in Problem 11, draw a detailed design for Mux0 and Mux1 that clearly identifies which bypass paths are selected under which control conditions. Identify each input to each mux by the name of the pipeline latch that it is bypassing from. Specify precisely the Boolean equations that are used to control Mux0 and Mux1. Possible inputs to the Boolean equations are:

    ❑ID.OP, EX.OP, MEM.OP = {‘load’, ‘store’, ‘alu’, ‘other’}

    ❑ID.ReadReg0, ID.ReadReg1 = [0..31,32] where 32 means a register is not read by this instruction

- EX.ReadReg0, etc. as in ID stage
- MEM.ReadReg0, etc. as in ID stage
- ID.WriteReg, EX.WriteReg, MEM.WriteReg = [0..31,33] where 33 means a register is not written by this instruction

Draw Mux0 and Mux1 with labeled inputs; you do not need to show the controls using gates. Simply write out the control equations using symbolic OP comparisons, etc. (e.g. Ctrl1 = (ID.op == 'load') & (ID.WriteReg==MEM.ReadReg0)).



The two muxes are identical, except that the left input is connected to Src1 instead of Src2. Similarly, the control equations for the right mux are the same as below except they refer to ReadReg1 instead of ReadReg0.

C0 = ~C1 & ~C2 & ~C3

C1 = (EX.WrReg == ID.ReadReg0) & (EX.op == 'alu')

C2 = (Mem.WrReg == ID.ReadReg0) & (Mem.op == 'alu') & ~C1

C3 = (Mem.WrReg == ID.ReadReg0) & (Mem.op == 'load') & ~C1

13. Given the IBM experience outlined in Section 2.2.4.3, compute the CPI impact of the addition of a level-zero data cache that is able to supply the data operand in a single cycle, but only 75% of the time. The level-zero and level-one caches are accessed in parallel, so that when the level-zero cache misses, the level-one cache returns the result in the next cycle, resulting in one load-delay slot. Assume uniform distribution of level-zero hits across load delay slots that can and cannot be filled. Show your work.

    The CPI effect of unfilled load delay slots is 0.0625. Since the L0 cache can satisfy 75% of these loads, 75% of this CPI term disappears. Hence, the CPI is reduced by .75 x .0625 = .047, resulting in a load CPI adder of .25x 0.0625 = 0.016

14. Given the assumptions of Problem 11, compute the CPI impact if the level-one cache is accessed sequentially, only after the level-zero cache misses, resulting in two load-delay slots instead of one. Show your work.

For the 75% of loads that hit the L0, there is no impact, and CPI is reduced by 0.047 as in Problem 13. For the 25% that miss the L0, there are now two load delay slots. 65% of the time, both delay slots can be covered, leading to no CPI adders. 10% of the time, only one slot can be covered, while 25% of the time, neither can be covered. Hence, the CPI adder for loads is now 0.25 x (0.25 x (0.10 x 1 + 0.25 x 2)) = 0.0375.

15. The IBM study of pipelined processor performance assumed an instruction mix based on popular C programs in use in the 1980s. Since then, object-oriented languages like C++ and Java have become much more common. One of the effects of these languages is that object inheritance and polymorphism can be used to replace conditional branches with virtual function calls. Given the IBM instruction mix and CPI shown in the following table, perform the following transformations to reflect the use of C++/Java, and recompute the overall CPI and speedup or slowdown due to this change:
    - Replace 50% of taken conditional branches with a load instruction followed by a jump register instruction (the load and jump register implement a virtual function call).
    - Replace 25% of not-taken branches with a load instruction followed by a jump register instruction.

Instruction type	Old Mix %	Latency	Old CPI	Cycles	New Mix %	Instructions	Cycles	New CPI
Load	25.0%	2	0.50	500	30.5%	305	610	0.58
Store	15.0%	1	0.15	150	15.0%	150	150	0.14
Arithmetic	30.0%	1	0.30	300	30.0%	300	300	0.28
Logical	10.0%	1	0.10	100	10.0%	100	100	0.09
Branch - T	8.0%	3	0.24	240	4.0%	40	120	0.11
Branch - NT	6.0%	2	0.12	120	4.5%	45	90	0.09
Jump	5.0%	2	0.10	100	5.0%	50	100	0.09
Jump register	1.0%	3	0.03	30	6.5%	65	195	0.18
Total	100.0%		1.54	1540	105.5%	1055	1665	1.58
Increase in CPI: 1.58/1.54 =			1.024805					
Increase in pathlength:			1.0550					
Total slowdown (product of the two)			1.081169	or	8%	slowdown		

16. In a TYP-based pipeline design with a data cache, load instructions check the tag array for a cache hit in parallel with accessing the data array to read the corresponding memory location. Pipelining stores to such a cache is more difficult, since the processor must check the tag first, before it overwrites the data array. Otherwise, in the case of a cache miss, the wrong memory location may be overwritten by the store. Design a solution to this problem that does not require sending the store down the pipe twice, or stalling the pipe for every store instruction. Referring to Figure 2-15, are there any new RAW, WAR, and/or WAW memory hazards?

    One possible design is to do the tag check for the store in the EX stage, but buffer the store data. On a store miss, the pipeline will stall until the line is filled. On a hit (or once the miss is resolved), the store data stays in the buffer until the next store comes down the pipe. In parallel with checking the tag for the next store, the data for the previous store is written into the data array. This is a fully pipelined solution.

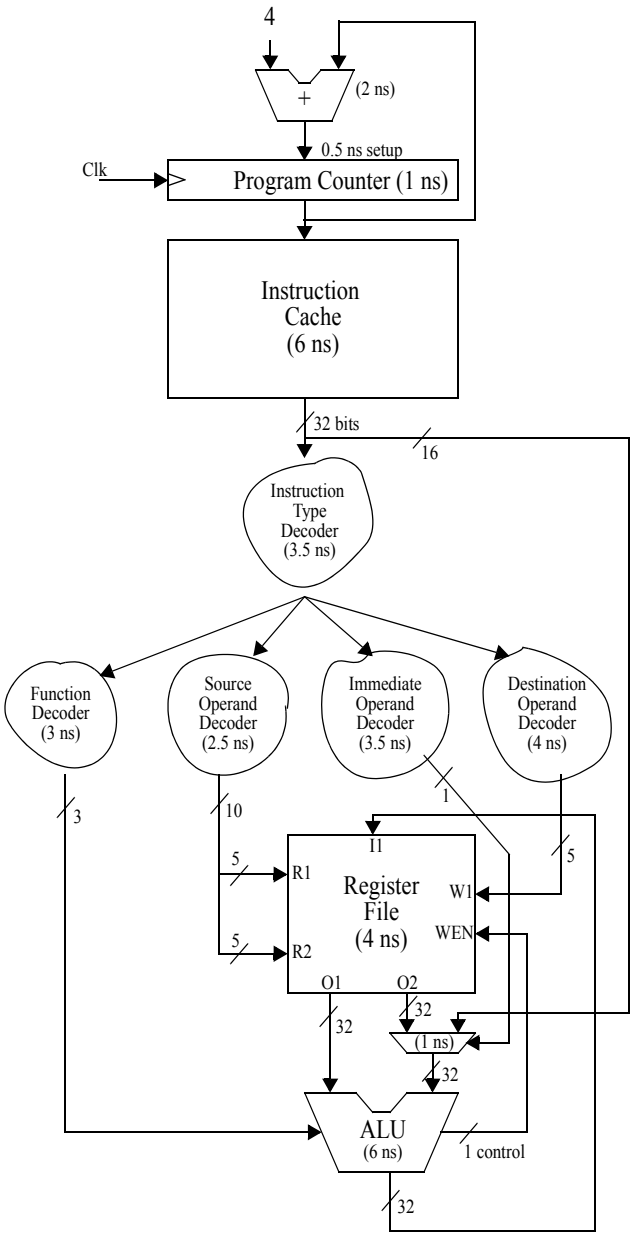
However, there is now a later memory write stage (since the write doesn't occur until the next store reaches the MEM stage, it appears as if the store operations are in an arbitrarily deep pipeline with the write stage at the bottom of that pipeline). Hence, RAW memory hazards are now possible, since a subsequent load from the same address could enter the MEM stage before the store data has been written from the buffer to the cache. Hence, hazard detection logic must compare the address of the data in the store buffer with subsequent loads, and must either stall the load to allow the store to write into the cache or bypass the data to the load directly from the buffer.

17. The MIPS pipeline shown in Table 2-7 employs a two-phase clocking scheme that makes efficient use of a shared TLB, since instruction fetch accesses the TLB in phase one and data fetch accesses in phase two. However, when resolving a conditional branch, both the branch target address and the branch fall-through address need to be translated during phase one--in parallel with the branch condition check in phase one of the ALU stage--to enable instruction fetch from either the target or the fall-through during phase two. This seems to imply a dual-ported TLB. Suggest an architected solution to this problem that avoids dual-porting the TLB.

    Two solutions are possible. In either case, the instruction translation for the branch instruction is reused for the subsequent fetch. The first solution requires that all branch targets lie within the same physical page as the branch instruction. Hence, the physical page number of the branch instruction can be reused with the branch target. The compiler and programmer must take special care to ensure that this is the case. Alternatively, the branch fall-through path can be restricted to be on the same page. In this scenario, the pipeline reuses the physical page number of the branch when fetching the fall-through path, and uses the TLB to translate the target address. This restriction is simpler, since it only forbids the compiler or programmer from placing a branch instruction at the end of a physical page. Whenever a branch does fall into such a location, the compiler can pad it with NOPs to place it on the next page.

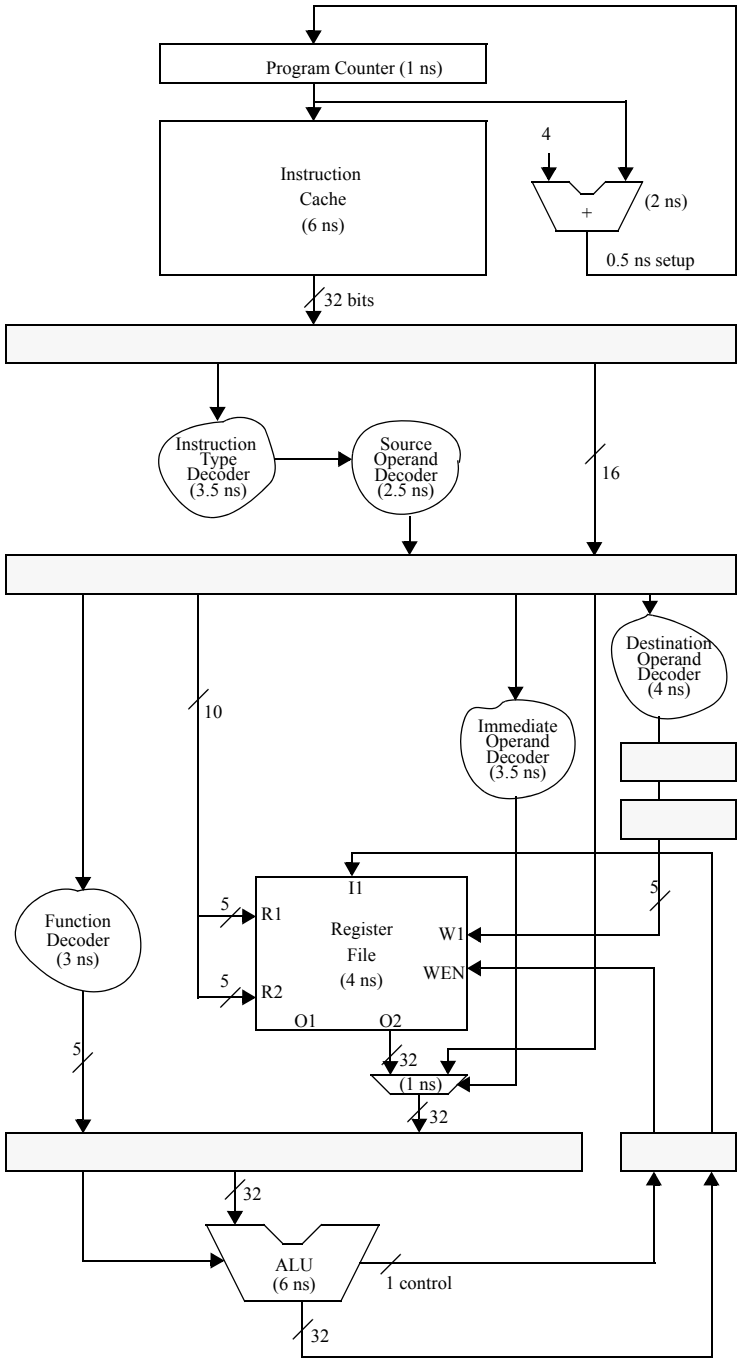
Problem 18 through Problem 24: Instruction Pipeline Design

This problem explores pipeline design. As discussed during lecture, pipelining involves balancing the pipe stages. Good pipeline implementations minimize both internal and external fragmentation to create simple balanced designs. Below is a non-pipelined implementation of a simple microprocessor that executes only ALU instructions, with no data hazards:



18. Pipeline implementation: Generate a pipelined implementation of the simple processor outlined above that minimizes internal fragmentation. Each sub-block in the diagram above is a primitive unit that cannot be further partitioned into smaller ones. The original functionality must be maintained in the pipelined implementation. Show the diagram of your pipelined implementation. Pipeline registers have the following timing requirements:

- 0.5 ns Setup time
- 1 ns Delay time (from clock to output)



19. Compute the latencies (in ns) of the instruction cycle of the non-pipelined and the pipelined implementations.

    Non-pipelined: PC delay(1ns) + ICache(6ns) + Itype Decode(3.5ns) + Src. Deccode(2.5ns) + Reg. Read(4ns) + Mux(1ns) + ALU(6ns) + Reg. Write(4ns) = 28 ns.

    or

    Non-pipelined: Add(2ns) + PC setup(0.5) + PC delay(1ns) + ICache(6ns) + Itype Decode(3.5ns) + Src. Deccode(2.5ns) + Reg. Read(4ns) + Mux(1ns) + ALU(6ns) + Reg. Write(4ns) = 30.5 ns.

    Pipelined: 5 stages * 7.5ns cycle time = 37.5ns

20. Compute the machine cycle times (in ns) of the non-pipelined and the pipelined implementations.

    Non-pipelined: machine cycle = instruction cycle = 28ns.

    Pipelined cycle time: 7.5ns

21. Compute the (potential) speedup of the pipelined implementation in Problem 18 over the original non-pipelined implementation.

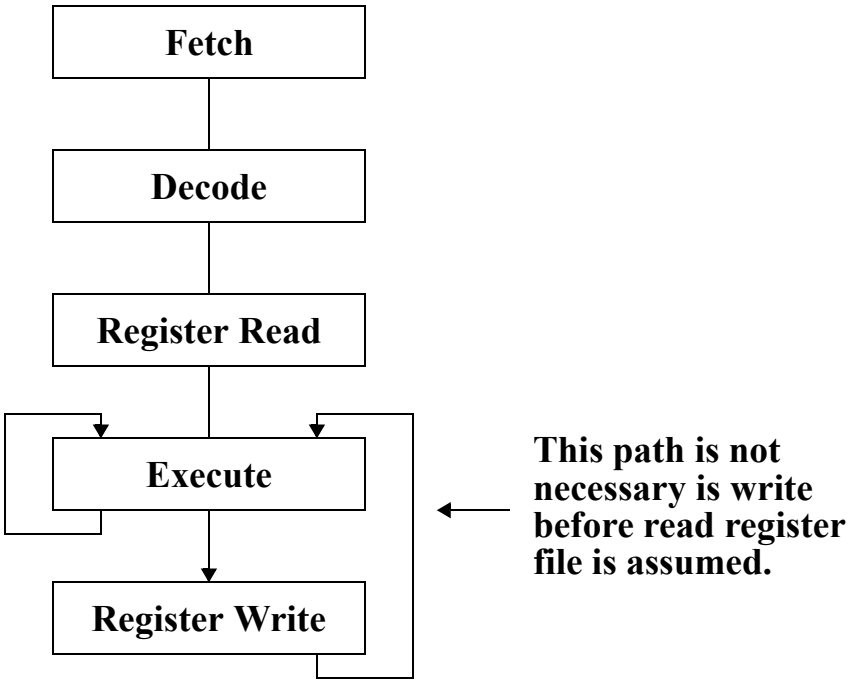
    Potential speedup is not 5x, because of additional overhead from the pipeline registers. The non-pipelined solution finishes an instruction once every 28ns. The pipelined solution finishes an instruction once every 7.5ns. The speedup is

    28/7.5 = 3.73 speedup

22. What microarchitectural techniques could be used to further reduce the machine cycle time of pipelined designs? Explain how the machine cycle time is reduced.

    An analysis of the frequency bottlenecks of the pipelined solution is expected. Any talk about predecode bits in the instruction cache to reduce the decode time, faster instruction cache implementation, faster ALU implementation, or superpipelining.

23. Draw a simplified diagram of the pipeline stages in Problem 18**;** you should include all the necessary data forwarding paths. This diagram should be similar to Figure 2-16.



24. Discuss the impact of the data forwarding paths from Problem 23 on the pipeline implementation in Problem 18. How will the timing be affected? Will the pipeline remain balanced once these forwarding paths are added? What changes to the original pipeline organization of Problem 18 might be needed?

    Expect generic discussion of adding muxes and comparators to catch the forwarding case. Also had to mention the muxes are in the execute stage, and the overall timing increase of the execute stage. The timing increase in the execute stage may or may not increase the timing of the pipeline implemented in Part A.