CSE 351 Winter 2011 Assignment 4 Answer Key

The answers below are samples that got full credit. Some shorter answers also got full marks, but I thought these explained their reasoning concisely and well.

1. **Basic, Sequential Datapath Operation**

Consider the sequential Y86 implementation shown in Figure 4-22 (page 376) of the text (and elaborated upon in Figure 4-23, page 378). We're going to develop a similar datapath by starting with support for only a single instruction, and then introducing instructions and the components they require one by one. Each component of the datapath may take a non-zero amount of time to do its work; this time is called the "latency" of the component. Suppose that the major components have the following latencies:

Instruction memory	400 picoseconds (ps) (400 * 10^{-12} seconds)
PC increment	150 ps
Register file	250 ps
ALU	220 ps
Condition codes/signals	100 ps
Data memory	500 ps

Assume that other components of the processor have negligible latency. For instance, it takes no time to choose the next PC based on the instruction and/or jump target. Also remember that the hardware can perform independent operations in parallel, i.e., at once.

a. Let's start by designing a super-lightweight Y86 processor, capable of executing only the JMP instruction (unconditional jump). Of the major components above, which ones do we need? How many instructions per second can this processor execute?

b. Now suppose we want also to execute NOP instructions. How long would each instruction take to execute? How many instructions per second can this processor execute?

c. Suppose we now add support for ADDL. What components do we need? How many instructions per second can this processor execute?

d. Finally, suppose we now add support for POPL. What components do we need? How many instructions per second can this processor execute?

(a) To implement only a jump instruction, we would only need instruction memory. The PC does not need to be incremented because we never go to the next instruction, only to the instruction specified by the jump instruction. This processor could execute 2500 million instructions per second.

(b) Adding NOP instruction capability means that in addition to the instruction memory, we need a PC increment component to increment the PC sequentially. The PC incrementer can run in parallel to the instruction fetch, so the rate is unchanged at 2.5 GIPS.
[I graded parts c and d based on your answer here, so if you added 150 ps the error doesn't cascade.]

(c) The ADDL instruction takes two arguments, the registers that are to be added (therefore we need the register file – 250ps). We also need the ALU in order to be able to add the register values together (220ps). Since we're not worried about conditional stuff at this point, and we don't need to access memory, no condition codes/data memory are needed (if you really wanted to include writing the condition codes as a result of ADDL, then you could still do that, but it wouldn't affect the latency: you can write the CC registers in parallel at the same time as you write-back to the registers). At the end, we need to write the added value back into the registers, so we need an additional access to the register file. Each of these steps must take place in sequence – the ALU needs the register values, and the write-back needs the added value. Therefore the total latency of the processor is 400 + 250 + 220 + 250 = 1120ps, for a speed of 8.9e8 instructions per second.
[Your answer here should be 720 ps greater than your part b answer. So if you added 150 ps for the PC incrementer in part b, and 100 ps for the Condition codes here, and left out the 250 ps write-back, you could get 1120 ps but it wouldn't be right.]

(d) The data memory component in addition to all the other components must now be included as the return address must be read from memory. The time to execute the instruction is now 1620 ps with the memory access so about 617 million instructions can be executed per second.

2. Pipelined Implementation

Now consider a pipelined implementation. (Figure 4-52, page 419 of the text, may be useful.) Each of the main components in the pipelined implementation have latencies the same as in question 1. Writing the pipeline registers requires 20 ps.

a. What is the maximum number of instructions per second that the pipelined implementation can execute?

b. Pipelining is a form of parallel execution, and so its performance is affected by dependences among the instructions available for execution at any moment. Give an example of an instruction sequence for which the pipeline implementation **cannot** achieve its maximum theoretical instruction execution rate (even if you let someone very clever fill in the details that are missing from our coarse description of the pipeline).

(a) Theoretically the pipelined implementation can execute as fast as the slowest stage in the processor plus the latency to write the pipeline registers. 500ps + 20 ps = 520 ps, so the pipelined implementation can execute 1.9*10^9 instructions per second.

(b)
```
irmovl  $8, %ecx
irmovl  $64, %ebx
mrmovl  0(%ebx), %ebx
addl    %ecx, %ebx
```
There is a RAW dependency with %ebx that can't be fixed with renaming or forwarding. A bubble will occur since a value is being loading into %ebx from memory, which occurs two stages after the register, and we'll need to wait for it to load before the adding op.
[Anything involving a memory read (mrmovl, popl) followed by a use of the value just fetched (e.g., in an addl or another mrmovl) is a good example.]

3. Compilers and Pipelining

Although the pipelined implementation gets the right results no matter what instruction sequence it is asked to execute, some instruction sequences execute more quickly than others. This can motivate the compiler to optimize the instruction sequence it emits by ordering instructions for efficient pipelined processing. Imagine that the compiler does this using a two-step process. First, it emits "the natural" instruction sequence for the code it has compiled, then as a "second-pass" goes over that instruction sequence and re-orders instructions for efficient pipelined execution.

a. How can the compiler determine which re-orderings are correct, i.e., get the same results as the original sequence? Briefly describe the criterion that ensures correctness.

b. Give a correct, more efficient re-ordering for this instruction sequence:

```
addl    r2, r3
subl    r3, r4
mrmovl  -16(%ebp), r2
mrmovl  8(r2), r3
addl    r3, r4
```

(a) The criterion to ensure correctness is that program execution gives the same results as if executed sequentially as written. To determine this, the compiler must *[perform a dependence analysis and]* make sure that no RAW is disturbed (these are true dependencies), that any series of WAWs ends with the same final W (to maintain the same final state after execution), that all WAW do not disturb the expected values (by writing a value using that value before a previous instruction finishing writing that value), and that all WARs are maintained (can't write something that first needs to finish being read (would disturb the read)).
[Even false dependencies must be respected here, because the compiler doesn't do register renaming.]

(b) As it is now, this code uses forwarding between the addl and subl commands (a RAW dependence), but uses a bubble between the two mrmovl commands and between the mrmovl and the addl – two bubbles representing two "useless" cycles. We can reduce this by reordering the instructions to create one fewer bubble.

```
addl    r2, r3
mrmovl  -16(%ebp), r2    (now no bubble after this – useful subl instruction instead)
subl    r3, r4
mrmovl  8(r2), r3        (bubble after this instruction is still necessary)
addl    r3, r4
```

4. Register Renaming

The compiler has to encode instructions according to the rules of the ISA. The x86 ISA, which dates back to a couple of years after the Magna Carta, was designed when hardware was expensive to build, and so has only 8 registers. The compiler therefore has to generate code that specifies only 8 registers.

Hardware is cheaper to build now, and implementations can easily afford building many more than 8 registers. Register renaming is a way to use additional registers, even though the compiler has generated code that names only 8 of them. For instance, the hardware might have 32 physical registers. When it fetches an instruction, it is free to change the register names used in the instruction (which are necessarily limited to the range 0 to 7) so that the instruction actually uses registers chosen by the hardware (which can be in the range 0 to 31).

Register renaming allows the hardware to remove false (WAW and WAR) dependences, which in turn allows it to reorder instructions in ways that otherwise wouldn't have been correct. In the following, assume that the hardware fetches instructions in order, possibly renames registers and re-orders the instructions, and then inserts them into the pipeline.

a. Apply register renaming and instruction reordering to the code sequence from question 3b so that it executes perfectly efficiently in a pipelined implementation. Assume that when these instructions start execution the data they want to work on is in the registers they name. (For example, the current value of r2 is what that code sequence expects.) Further, assume that none of the "hidden registers" (8-31) have anything useful in them, but **all** of the ISA registers (0-7) do.
   Note: *For reasons that may be clearer after answering the next question, the results of the code sequence **do not** need to end up in the registers named by the original code sequence (i.e., they do not need to end up in registers 2, 3, and 4). They can end up anywhere you like.*

b. Register renaming requires that the hardware be prepared to re-write both the source and the destination register numbers in instructions it fetches, as well as to figure out a beneficial re-ordering of instructions. In this question, we're concerned only with re-writing source registers, i.e., we're ignoring how the hardware decides whether/how to re-write the destination register number and how to re-order instructions, we'll just assume that it can (and does) do it.
   Assuming that the hardware fetches instructions "in order" (i.e., in the order dictated by the simple, sequential execution model of the ISA), does re-naming and re-ordering, and then inserts the re-named/re-ordered instruction stream into the pipeline, describe a procedure and data structure the hardware can maintain to determine how to re-write **source** register names. You should assume that the unspecified process of choosing destination register names knows about and updates the data structure you describe. You should also assume that pipeline executing the instructions with renaming is able to resolve any hazards, exactly as the simple five-stage pipeline (without renaming) in earlier questions did.
   Note: *I think it will be easier to figure this out yourself than to try to get Google to answer it for you, because the online descriptions consider more complicated situations than the one in this problem and address also renaming of destination registers. Because of that, if you want to try to answer this problem by "researching" known techniques, I'm all for it -- you'll learn a ton. (To repeat, though, I think that's more work than just thinking about it and answering it, and of course my opinion is that the thinking part of that is worth doing.) Search keywords it might take you a while to find: Tomasulo's Algorithm.*

(a) The following re-ordering of instructions will execute at maximum efficiency on a pipelined processor (forwarding would still need to be used to handle some dependencies, but stalling would not be necessary with this ordering so the efficiency would not be affected). False dependencies have been eliminated by re-naming registers, but the order of instructions that have true dependences have not changed relative to each other (the subl still comes after the first addl, the second addl still comes after the second mrmovl which comes after the first mrmovl), so the output is not affected.

```
mrmovl  -16(%ebp), r8
addl    r2, r3
mrmovl  8(r8), r9
subl    r3, r4
addl    r9, r4
```

Both memory reads have an independent instruction after them so the dependent instructions can get their results through forwarding without a hole (it is "filled" by the independent instruction).
[Although most people used two registers, one will do.]

(b) A simple solution is to have a short array of size 8, and each index of the array corresponds to each of the source registers, and when it rewrites a new register, we would place the number or value of the new register for register renaming into the value under the index of the original source register. That way, every time in the future when we encounter the old source register, we would be able to reference and access the new 'renamed' register that the old register was pointing to within the array that we created.
[You needed a data structure that maps one register name to another. The main idea here is that for each ISA register, r0–r7, you need to remember what physical register the value named by it in the original code is actually in. Because you're going through the instructions sequentially, the "current value" for r0–r7 is exactly one place—you don't need to keep any kind of history around. So you just need a simple table (array) that you consult whenever an instruction needs to read a register/value.]

5. **Bonus Question** [Optional]
   In question 3 the compiler did instruction re-ordering. In question 4, the hardware did both register renaming and instruction re-ordering. If register re-naming isn't implemented in the hardware, is there any point to implementing re-ordering there? (That is, can everything that the hardware achieve by re-ordering be done by the compiler instead, when there is no register re-naming?)

Yes the hardware should implement instruction reordering because the compiler cannot know at runtime the exact sequence of instructions that will be executed because of jumps and user input etc. The hardware does know about the sequence being executed at runtime because it's the one doing the executing and therefore can do some optimization that the compiler couldn't.
[The main reason is branches -- the hardware can see past branch execution (with prediction & pre-fetching), but the compiler can, at best, only guess (less accurately than the hw). Further, the compiler may not have the branch target available when it's compiling. And, finally, the branch target may be the target of many different branches, so that optimizing for it statically leads to conflicting results (whereas at runtime the branch target is being reached from some specific branch, and the order can be optimized for that specific pairing).]