CMSC 411 Computer Architecture

Sample Final Exam I
Summer, 1997

Problem 1: Truth or Dare

The dreaded true-false section has magically appeared. Two points each, and be a tad careful, as some are easy, and some are tricky. Circle the number of each **true** statement.

1. The instruction count per program and the total execution time of the program are independent.
2. Amdahl's law demonstrates that overall speedup is limited primarily by the fraction of the original process that is enhanceable.
3. Commercial benchmarks provide excellent estimates of system performance and can be used to compare general purpose systems.
4. The performance of a computer architecture can usually be improved by increasing the clock rate.
5. Adding a new type of ALU instruction to a computer's Instruction Set Architecture has no effect upon the total CPI.
6. Making the common case fast is a good rule of thumb for designing computer architectures.
7. All instructions in a General Purpose Register machine of type (0,3) must have 3 operands.
8. Interleaved memory decreases the delays associated with contention for a single memory during the access of logically sequential addresses.
9. Increasing the depth of a pipeline reduces the potential for WAW hazards.
10. MFLOPS is a reasonable measure of performance for a machine dedicated to numerically intensive calculations, such as data encryption and decryption.
11. An enhancement to machine A is desirable when the ratio of the average execution time on A' (the enhanced machine) to the average execution time on A exceeds one (1).
12. The width of a clock cycle in a pipelined architecture must be as least the length of the slowest stage.
13. The instruction set architecture of a machine uniquely determines the hardware on which the code must be run.
14. Compiler optimization is insufficient to remove control hazards.
15. The 2-1 cache law predicts the reduction of cache misses when a single level cache is replaced by a 2-level cache.
16. Pipelined architectures were originally developed independently from cache technology.
17. The 90-10 law states that 90% of the exceptions that must be handled during program execution arise in 10% of the code.
18. Cache block size is a factor in computing both the hit rate and the miss penalty.
19. Tomasulo's algorithm solves the problems associated with RAR and RAW data hazards.
20. Quantitative design methods advocate the use of numerical estimates of key system parameters in choosing features to be implemented in a system under development.

Problem 2: An Overview for the Short Sighted

The following section requires short answers. Complete sentences are not necessary. However, make *sure* you address the question I'm asking. Facts are nice, but your analysis of them is what I"m going for here. Counter-examples are acceptable as well. Oh, and each one is worth (4) points.

2.1 Explain why the statement "Make the common case fast." is often called a corollary of Amdahl's Law.

2.2 Can there ever be a pipeline hazard involving register R0? Explain your answer for full credit.

2.3 Explain an advantage and a disadvantage of the "offset-based" addressing used in DLX.

2.4 Does the instruction set architecture uniquely determine the hardware organization that supports it?

2.5 How does an accumulator based architecture differ from a GPR (general purpose register) architecture? (Cite and explain 2 differences for full credit).

2.6 What effect would you expect the addition of forwarding and bypassing hardware to a pipelined architecture to have upon the average program CPI? Again, explain your answer.

2.7 If Amdahl's law gives a speedup of 2.5 for program A, and 3.14159 for program B, what can you conclude about the execution times of the two programs after the enhancement.

Problem 3: Enhancing Performance I: Excess Speedup

Hopefully, you'll be able to handle these performance enhancing techniques without too much trouble. Don't worry about doing the arithmetic. I'm extremely fond of fractions, since they are exact, while many decimals aren't.

Suppose that you install the HyperOptomIZing Compiler, abbreviated below by HO, which increases the speed of all floating point operations by a factor of **K**. No other operations are affected by this upgrade. You compile a program using both the old compiler and the new compiler, and compare the execution times. Let T denote the original execution time, and T_{HO} denote the enhanced execution time.

3.1 If half of the original execution time is spent on floating point instructions, what must K be to achieve an effective (or overall) speedup of 4.

3.2 Write an expression for the new execution time, T_{HO}, in terms of K, x, and y, where

x is the amount of time spent performing floating point operations before HO is used.

y is the amount of time spent performing non-floating point operations before HO is used.

3.3 Express Amdahl's law in terms of the variables x, y, K, T, and T_{HO}. You may not need all of these variables. However, they should be sufficient to answer this question.

3.4 If the new floating point code runs 10 times faster than the old, the original code ran for 50 seconds, and the effective (overall) speedup is 5, how much time does the new code spend executing floating point instructions?

Problem 4: A Little DLX Code

This problem requires you to write and debug some DLX code. Don't forget that without comments I have no idea of your intentions. I will withhold points for otherwise correct work in the absence of comments.

(8) 4.1 Consider the trinomial

$$y = x^2 + Bx + C$$

The x-intercepts of the graph of this equation, if any, can be derived by evaluating the following.

$$X_1 = (-B + sqrt(B^2 - 4 * C))/2$$

$$X_2 = (-B - sqrt(B^2 - 4 * C))/2$$

Unfortunately, the only computer available to solve this problem is a *sequential* machine with faulty division circuits. The sole square root function assumes that its argument is non-negative. And, the exponent subroutine was time-stamped proprietary code that expired last week.

Your job is to write a code fragment that will correctly compute the values of the x-intercepts without crashing (because broken or non-existent commands were used). You may assume that the subroutine *sqrt*() exists (meaning don't write it because I WILL NOT GRADE IT), and returns its value to location 0, for simplicity. Make sure that you include any additional assumptions that you make in writing the code, such as where the program constants are stored!!!

(6) 4.2 Write the code fragments needed to convert a 64 bit word stored beginning at location 64 in Big Endian mode to one stored in the same location, but in Little Endian mode.

(8) 4.3 Now it is your turn to evaluate my code. The fragment below is intended to run on a sequential machine that supports DLX. Unfortunately, I have forgotten the DLX syntax. For each line of code, record the line number and "correct" if there are no errors. If errors are present, either fix them or tell me what's wrong.

Line	Operator	Operands	Comments (?)
(1)	LF	R5, 10(R0)	; Load register R5
(2)	ADDD	R4, R0, R0	; Initialize R4
(3)	ADD	R3,100(R4), R5	
(4)	ADDI	R0, R5, R5	;Increase R5
(5)	SW	104(R4+R3), R5	; Save register R5
(6)	SUB	R1, R5, R5	; Clear register R1
(7)	SD	103(R4), F5	
(8)	ADDI	R5, R5, #-42,708	; Reload register R5
(9)	SW	R5, 108(R4)	; Save final R5 value

Problem 5: Hazardous Waste

This problem deals with the following code fragment, to be executed on the basic 5-stage DLX pipeline, with a single memory, no forwarding and no bypassing, unless otherwise stated. For your inconvenience, no comments are included. Happy pipelining!

Line	Operator	Operands
(1)	LW	R5, 20(R0)
(2)	ADD	R4, R0, R0
(3)	SW	100(R4), R5
(4)	ADD	R5, R5, R5
(5)	SW	104(R4), R5
(6)	ADD	R5, R5, R5
(7)	SW	106(R4), R5
(8)	ADD	R5, R5, R5
(9)	SW	108(R4), R5

(5) 5.1 Evaluate the hazard potential of the above code. For each hazard class present (control, structural, or data), identify **one** occurrence. Give the type of hazard, the line(s) involved in the hazard, and suggest a fix. Even if every line contains a hazard, I only want to hear about at most 3: 1 control, 1 structural, and 1 data—assuming such beasts exist in the code.

(10) 5.2 Compute the CPI for the code fragment above, assuming that it is executed using the **basic 5-stage DLX pipeline, with NO forwarding, NO bypassing, and NO split-memory**.

(5) 5.3 Suppose now that the same code is executed on a DLX pipeline with a split memory, full forwarding, and full bypassing. Identify any data hazards that will persist in the new configuration. That is, are there any data hazards present in the code that forwarding, bypassing, and separate memories cannot fix? If so, identify the line number(s) and operands involved, and classify the data hazards by type.

Problem 6: Benched Metrics and Performance

This is the portion of our show where we dredge out the dreaded CPU performance equations. In this case, we are interested in comparing metrics or *measures of performance* other than speedup. You'll need the formulas for that, for MIPS, and for MFLOPS. So either find them, or derive them before proceeding. (This is probably the easiest problem here. So, don't let the length scare you.)

(5) 6.1 Express execution time, as defined by the CPU performance equation, in terms of the following metrics. That is, if T represents the execution time of a program, write "$T =$" some expression that uses each measurement below and any other variables that you need. Make sure you define other variables that you use.

Show your work for full credit.

6.1.a MFLOPS (1×10^6 floating-point operations per second)

6.1.b MIPS (1×10^6 instructions per second)

(9) 6.2 Suppose we assume that a program is compiled using both the original compiler, O, and the new HO compiler. Assume that the machine clock rate is 100Mhz. Furthermore, the table below gives the instruction counts (in millions) for the three classes of instructions listed.

Code Source	# ALU	# Load/Store	# Branches
Compiler O	5	1	1
Compiler HO	10	1	1

6.2.a Compute the execution time of each version using the CPU execution time equation.

6.2.b Compute the MIPS rating of each version.

6.2.c Compute the MFLOPS rating of each version.

For grading purposes, please organize your results in a table.

(3) 6.3 Regardless of your results from part 4.2, suppose that the execution time of the version produced by compiler O is less than than produced by compiler HO. What can you conclude about the effectiveness of the new compiler?

(3) 6.4 This time, suppose that the execution time for the HO code is less than that for the O code. Suppose that the MIPS rating of the code produced by compiler O is higher than that of HO, and the MFLOPS ratings are reversed. Speculate why this occurs. What can you conclude about the effectiveness of MIPS and MFLOPS as predictors for this program?