CS342 Midterm Review

Jay Noppone P

Dedicated to Chapter 1 and 3 Midterm

1 Introduction

The following pages are for use of reviewing for Prof. Fazli's CS342 Midterm Exam for semester Summer 2024. Motivations include wanting to practice my LaTeX skills, and also entertaining myself while also re-consuming the class material, which will firmly ingrain the content into my memory.
This review will be structured as a small rundown on Chapter 1, then Chapter 3, and then practice problems different than those given in the homework. Note that some abbreviations mentioned comes from the slides, and this brief review will require basic reading from said slides. If there's any inaccurate information, bad typos, or anything of the sort, let me know.

2 Chapter 1

2.1 CPU Time

CPU Time can be represented in 3 significant ways:

$$\text{CPU Time} = \text{Clock Cycles} \times \text{Clock Cycle Time}$$
$$\text{CPU Time} = \frac{\text{Clock Cycles}}{\text{Clock Rate}}$$
$$\text{CPU Time} = \frac{\text{IC} \times \text{CPI}}{\text{Clock Rate}}$$

Notice that knowing these three formulas also tells you that **Clock Cycles** can also be represented as **IC** $\times$ **CPI**, which is useful since questions may come up involving calculating total Clock Cycles given CPI value(s) and IC value(s). Above formulas will help with the next section.

2.2 Average CPI

Calculating the average CPI of an implementation/program often follows:

$$\text{Average CPI} = \frac{\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i}{\text{Total Instruction Count}}$$

where n represents the total instruction classes within a program. Obviously, the formula looks different than the slides, but the one above is just more simple for understanding for calculating the average CPI when prompted to do so.

2.3 Speedup

Although this is introduced right in the beginning, it makes sense to talk about this in the end since this requires the CPU/Execution time already calculated, which should be already done at this point. Calculating speedup is very simple:

$$\text{Speedup} = \frac{\text{CPU Time}_B}{\text{CPU Time}_A}$$

Based on Homework 1 given, when given a problem about a new CPU time calculated, the new CPU time would best be CPU Time_A instead of B.

2.4 Amdahl's Law

Key components with problems relating to Amdahl's Law includes the percentage of the program that is affected by a given improvement, and the factor of improvement. (Example: "30% of the program is improved by 10 times", 30% = affected, 10 = factor of improvement)
In order to calculate the new percentage after improvement, say we use the example mentioned above. Having 30% of the program improved by 10 times would cause for the computation: $\frac{0.30}{10} = 0.03$. This results in a new total percentage: $0.03 + 0.70 = 0.73$, where the 0.70 is the unaffected factor of the program. Finally, to calculate the overall speedup, we use the below equation:

$$\text{Overall Speedup} = \frac{1}{\text{unaffected} + \text{improved affected}}$$

In our example, this would be $\frac{1}{0.73} = 1.37$, meaning with this improvement, the implementation is 1.37x faster.

2.5 SPEC Benchmark and Performance

Most likely, there will only be two types of problems that are unique to this topic when given a benchmark table. In order to calculate the SPECratio of a program, use the below small formula:

$$\text{SPECratio} = \frac{\text{Reference Time}}{\text{CPU/Execution Time}}$$

Other than the above, the second problem type would be asking for a geometric mean (or "the SPEC benchmark") of the entire SPEC table given. All we would have to do is find with the formula:

$$\text{Geometric Mean} = \sqrt[n]{\Pi_{i=1}^{n} \text{SPEC Ratio}_i}$$

3 Chapter 3

One might say I would be wasting my time writing all of the subsections for unoptimized multiplication and division, but I honestly don't know how Fazli's exams look, so better to be overprepared right?

3.1 Multiplication

3.1.1 Unoptimized Multiplication

Unoptimized multiplication is practically the same as traditional on-paper long multiplication. We have three registers for the multiplier, multiplicand, and the product. The product is preset to bits of 0s, and continuously computes its sum with the multiplicand based on the current LSB of the multiplier. If the multiplier flags as 1, proceed the summation; otherwise, do not compute the sum. Note that at each step, no matter if summation occurs or not, the multiplier register shifts right, and multiplicand shifts left. Typically, the number of steps is equal to the length of the operands' digit count. Below is the problem given in the second homework.

Multiplier	Multiplicand	Product
0111	00001101	00000000
0011	00011010	00001101
0001	00110100	00100111
0000	01101000	[**01011011**]

In the above, multiplier shifts right, multiplicand shifts left, and product is computed the same way as: product += multiplicand ONLY IF the LSB of the multiplier is a 1.

3.1.2 Optimized Multiplication

Optimized multiplication is more compressed, but is definitely more efficient and can be seen more simple when fully understood. With this, we have registers for the multiplicand, addition (initialized as nothing at start), and a product register. The basis of the operation is checking the flag in the LSB of the product, and only applying summation with the left half of the product + the multiplicand when this flag is a 1 (otherwise, only apply the left half of the product). The summation will be an extra bit long because after performing the addition, we overwrite the left half of the product with the new summation, then shifting the entire register to the right.

Multiplicand	Addition	Product
1101	null	00000111
1101	0000+1101=01101	01101 0111→0110 1011
1101	0110+1101=10011	10011 1011→1001 1101
1101	1001+1101=10110	10110 1101→1011 0110
1101	1011+0000=01011	01011 0110→[**0101 1011**]

Note that the → above represents the shifting to the right after the summation has been computed.

3.2 Division

3.2.1 Unoptimized Division

Unoptimized division is straight forward at this point. Simply have a quotient register set to bits of 0s, a divisor register with a bit extension of 0s to its right, and a remainder register with a bit extension of 0s to its left. To compute division the "naive" way, continuously try to subtract remainder - divisor; if successful, set the remainder register to the difference, set the quotient's LSB to 1, shift the quotient left, and shift the divisor right. Otherwise, do not change the remainder register, but set the quotient's LSB to 0, shif the quotient left, then shift the divisor right. Note for unoptimized division, an 8-bit problem will often lead to 8+1 steps, but for optimized division, it would lead to 8 steps.

Quotient	Divisor	Remainder
null	0010 0000	0000 0111
0000	0001 0000	0000 0111
000**0**	0000 1000	0000 0111
00**00**	0000 0100	0000 0111
0**001**	0000 0010	(0000 0111)-(0000 0100)→(0000 0011)
[**0011**]	0000 0001	(0000 0011)-(0000 0010)→(0000 [**0001**])

After the process above, the quotient is 0011, with the remainder 0001 To avoid confusion during computation, simply start by initializing your first row. Then, try to subtract remainder - divisor. Then after that, you update the remainder, and then shift the divisor.

3.2.2 Optimized Division

First thing you should absolutely think of is Lab 2 in CS343, where we have a HI-LO representation of the final quotient, which is simply the (Remainder)(Quotient) structure. Technically in optimized division, there's really just the divisor and the remainder register (initialized with the dividend). The dividend is bit extended to the left, and we start processing by immediately shifting the remainder register to the left, attempting to subtract first half of reemainder - divisor; if the difference is successful or greater than 0, overwrite the current first half with the compute difference, and mark the LSB of the remainder with a 1. Otherwise, mark the LSB with a 0, and do not overwrite.

Divisor	After-Shifting-Left	Remainder Register
0010	null	0000 0111
0010	0000 111?	0000-0010$\not>$0: 0000 111**0**
0010	0001 110?	0001-0010$\not>$0: 0001 110**0**
0010	0011 100?	0011-0010>0: 0001 100**1**
0010	0011 001?	0011-0010>0: **0001 0011**

Same as the unoptimized division, we reach a quotient of 0011...remainder of 0001

3.3 IEEE Floating Point

Three important concepts to implement here are: precision of converting a decimal to a binary number, normalization, and the format of an IEEE Floating-Point number.

3.3.1 Precision and Normalization

In precision where we convert a decimal number to a binary number, we simply multiply the decimal number by 2, and noting down the MSB of the given decimal number as one of the digits of the binary number conversion. Example:
0.593 x 2 = **1**.186
0.186 x 2 = **0**.372
0.372 x 2 = **0**.744
0.744 x 2 = **1**.488
0.488 x 2 = **0**.976
Above, 0.593 is precisely converted to 0.10010. We would want to normalize this for the purpose of finally converting into IEEE Floating Point format. Normalization is the same logic as writing scientific notation, let's write 0.10010 as, instead, 1.001×2^{-1}. Note that no matter what, the MSB of the decimal multiplying by 2 would have to be 0, none of the other digits will be affected.

3.3.2 Conversion

IEEE Floating Point format is written as a three components: [S][Exponent][Fraction], with a formula given $x = (-1)^S \times (1 + Fraction) \times 2^{Exponent-Bias}$ where x simply represents the original number in the IEEE format.
An approach to when given a decimal number, for example, -0.75_{10}, would be to first identify whether this is a negative or positive. If negative, set S = 1 to allow the $(-1)^1$ to compute as -1. Otherwise, apply the double negative. Then, convert the decimal number into binary through precision. After calculating, $0.75_{10} = 0.11_2$. Then, normalize: 1.1×2^{-1}. We now treat the exponential value -1 to be equal to Exponent-Bias. Say, hypothetically, the bias was 16, then we would have Exponent-16=-1, which implies that Exponent=15. Finally, we need the value Fraction, which we can extract from the 1.1: 1.1 = (1 + 0.1), thus

Fraction has the value of 0.1. This gives us x being:

$$x = (-1)^1 \times (1 + 0.01) \times 2^{15-16}$$
$$S = 1, \quad \text{Fraction} = 0.01, \quad \text{Exponent} = 15$$

Result: 1 01000 11110 00000

4 Practice Problems

4.0.1 Problem 1

Consider the below table:

Processor	Clock Rate	CPI Class A	CPI Class B
P1	1.5GHz	2	3
P2	1.75GHz	1	2
P3	4GHz	1	3

Say that the total instruction count throughout is 10^2, distributed into 40% class A and 60% class B.

1. Calculate the total clock cycles on each processor
2. Calculate the average CPI of each processor
3. Calculate the CPU time of each processor
4. Consider a situation where class A was improved by 40 times, what would be the overall speedup throughout all implementations?

4.0.2 Problem 2

Consider the SPEC benchmark table below:

Program	Instruction Count x 10^9	Exec Time	Ref Time	SPECratio
perl	159	120	1400	?
bzip2	2389	604.76	9,650	15.96
gcc	1,050	469.47	8,050	17.15
golang	1,658	511.57	10,490	20.51
minecraft	336	360	9,120	?

1. If the clock rate were to be 4.5GHz, what would the CPI of **perl** and **minecraft** be?
2. Calculate the SPECratio of **perl** and **minecraft**.
3. Calculate the geometric mean of the above SPEC benchmark.

4.0.3 Problem 3

1. Compute $1001_2 \times 0101_2$ with unoptimized multiplication
2. Compute $1001_2 \times 0101_2$ with optimized multiplication

4.0.4 Problem 4

1. Compute $1100_2 \times 0100_2$ with unoptimized division
2. Compute $1100_2 \times 0100_2$ with optimized division

For the following problems, assume the same system as demonstrated in Homework 2 where a "half type" system has only 16 bits with the mantissa 10 bits long, and the exponent 5 bits long.

4.0.5 Problem 5

1. Compute the floating point for 3.625_{10}
2. Compute the floating point for -5.8125_{10}

4.0.6 Problem 6

1. Compute the decimal values for 0110 0111 0001 0000
2. Compute the decimal values for 1100 1001 1000 0000