CPSC 330 Midterm Review

Hardware and Performance

The following equations are given:

1. $CPU\ Time = \frac{Clock\ Cycles}{Clock\ Rate}$
2. $CPU\ Time = Instructions \times \frac{Cycles}{Instruction} \times \frac{Seconds}{Cycle}$
3. $CPU\ Time = Clock\ Cycles \times Cycle\ Time$
4. $CPI = \frac{Clock\ Cycles}{Instructions}$
5. $CPU\ Time = Instruction\ Count \times CPI \times Cycle\ Time$
6. $CPU\ Time = Instruction\ Count \times \frac{CPI}{Clock\ Rate}$
7. $(Clock\ Cycles)_{Total} = \sum_{i=1}^{n} (CPI_i \times Count_i)$
8. $MIPS = \frac{Instruction\ Count}{CPU\ Time \times 10^6}$
9. $Power = Capacitive\ Load \times (Voltage)^2 \times Frequency$

Question 1: How can we improve performance?

Solution: There are several factors that can help to improve performance of a system. Consider the following equation:

$$\frac{seconds}{program} = \frac{cycles}{program} \times \frac{seconds}{cycle} = \frac{cycles/program}{clock\ rate}$$

From this equation, we can tell that there are a few ways to increase performance. For example, we can

1. We can **decrease** the number of cycles in a program. If we look at the right-most side of the equation, we can see that decreasing the numerator will decrease the $\frac{seconds}{program}$.
2. By a similar argument, we can say that **increasing** the clock rate will also decrease the $\frac{seconds}{program}$.
3. Increasing the cycles is the same as **decreasing** the clock cycle time.

Question 2: Suppose we have two implementations of the same ISA on machines A and B. Machine A has a clock rate of 4GHz and a CPI of 2.0, and machine B has a clock rate of 2GHz and a CPI of 1.2. If we run the same program on both machines, which machine is faster, and by how much?

Solution: Recall the equations we have above. In our case, we will be looking at equation [5]. Since the program is the same on both machines, the number of instructions will cancel out. We want to find the ratio of the speeds, and then convert them into percentages.

$$\begin{aligned}
\frac{CPU\ Time_A}{CPU\ Time_B} &= \frac{Instruction\ Count \times \frac{CPI_A}{Clock\ Rate_A}}{Instruction\ Count \times \frac{CPI_B}{Clock\ Rate_B}} \\
&= \frac{\frac{CPI_A}{Clock\ Rate_A}}{\frac{CPI_B}{Clock\ Rate_B}} \\
&= \frac{CPI_A \times Clock\ Rate_B}{Clock\ Rate_A \times CPI_B} \\
&= \frac{2.0 \times 2}{1.2 \times 4} \\
&= 83.3\%
\end{aligned}$$

So, Machine A is 83.3% faster than machine B.

Question 3: Two computers, C_1 and C_2 have the following metrics when running the same program:

	Computer C_1	Computer C_2
# Instructions	10 billion	8 billion
Clock Rate	4GHz	4GHz
CPI	1.0	1.1

Which computer is faster, and which has higher MIPS?

Solution: Let's address the first part of the question: *Which computer is faster?*. To do this, we can use equation [6]. We will have the following:

$$\begin{aligned} CPU\ Time_1 &= 10e9 \times \frac{1.0}{4e9} \\ &= \frac{10}{4} \\ &= 2.5 \end{aligned} \qquad \begin{aligned} CPU\ Time_2 &= 8e9 \times \frac{1.1}{4e9} \\ &= \frac{8.8}{4} \\ &= 2.2 \end{aligned}$$

So we can see that C_2 is faster than C_2. Now lets answer the second part of the conversation. To find the MIPS, we need to use [8]:

$$\begin{aligned} MIPS_1 &= \frac{10e9}{4e9} \\ &= 2.5 \end{aligned} \qquad \begin{aligned} MIPS_2 &= \frac{8e9}{4e9} \\ &= 2 \end{aligned}$$

And we are done.

Question 4: A compiler designer is trying to decide between two code sequences for a particular machine. Based on the hardware implementation, there are three different classes of instructions: Class A, Class B, and Class C, and they require one, two, and three cycles, respectively.

The first code sequence has 5 instructions: 2 of A, 1 of B, and 2 of C

The second sequence has 6 instructions: 4 of A, 1 of B, and 1 of C

Which sequence will be faster, and by how much? What is the CPI for each sequence?

Solution: Recall equation [4]. It tells us that the CPI will be equal to the clock cycles divided by the number of instructions. To solve this problem, we need to realize that the total number of cycles will be the sum of each cycle. In other words, we have

$$\begin{aligned} CPI_1 &= \frac{1+1+2+3+3}{5} \\ &= \frac{10}{5} \\ &= 2 \end{aligned} \qquad \begin{aligned} CPI_2 &= \frac{1+1+1+1+2+3}{6} \\ &= \frac{9}{6} \\ &= 1.5 \end{aligned}$$

So we have the CPI for each sequence. Since we have the same machine, the ratios for both [6] and [5] will have elements that cancel out. Let's use [5] in this case, as it will be simpler.

$$\begin{aligned} \frac{CPU\ Time_1}{CPU\ Time_2} &= \frac{5 \times 2 \times Cycle\ Time}{6 \times 1.5 \times Cycle\ Time} \\ &= \frac{10}{9} \\ &= 1.11 \end{aligned}$$

This tells us that sequence 1 is faster than sequence 2, and in fact it is 11% faster.

At this point, we will introduce a table that shows us what factors affect the CPU Time. Recall from [5] that The number of instructions, the CPI, and the cycle time all influence the CPU Time. An "X" implies that the CPU time factor is directly affected, and a $\sim$ implies that it indirectly affects a factor of the CPU Time.

	# Instructions	CPI	Clock Cycles
Algorithms	X	$\sim$	
Language	X	$\sim$	
Compiler	X	X	
ISA	X	X	X
Technology			X

Question 5: Consider the following table which describes the attributes of a certain machine:

Op	Frequency	CPI	Frequency $\times$ CPI
ALU	50%	1	0.5
Load	20%	5	1.0
Store	10%	3	0.3
Branch	20%	2	0.4
			$\sum = 2.2$

a. How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?

b. How does this compare with using branch prediction to shave a cycle off the branch time?

c. What if two ALU instructions could be executed at once?

Solution: Let's start with part *a*. To solve this, we will have to recalculate the total sum of Frequency $\times$ CPI. Luckily, all the values will stay the same except of the CPI for the load, which is being changed to 2. The new value for Frequency $\times$ CPI is now 0.2×2, or 0.4. This makes the total sum 1.6, and the ratio of the old sum to the new sum is $2.2 : 1.6$, or 37.5%.

We will do the same thing for part *b*. In this case we will have one less cycle for the branch time, bringing the Frequency $\times$ CPI for the Branch operation to 0.2. Our new sum is 2.0, which means that we have a ratio of $2.2 : 2.0$, or 10%

Part *c* is a little different, but not too far off from *a* and *b*. In this case, it's not the CPI that's changing, but rather the frequency. We can process 2 ALU instructions at once, so the frequency will be half of what it currently is. In other words, we will have a frequency of 25%, instead of 50%. This makes the Frequency $\times$ CPI for the ALU 0.25, and the total sum is now 1.95. Now we have a ratio of $2.2 : 1.95$, or 12.8%.

MIPS Assembly

Question 1: Translate the following piece of pseudocode into MIPS instructions, with $t0 and $t1 as temporary registers.

```
f = (g + h) - (i + j);
```

Solution: This is simple. Just remember that R-Type MIPS instructions (e.g. add, sub, ...) take three arguments: the register to store the value into, the left-hand operand, and the right-hand operand. We have as follows:

```
add $t0, g, h
add $t1, i, j
sub f, $t0, $t1
```

Question 2: Assume A is an array of 100 words. Variables g and h are associated with registers $s1 and $s2 and the starting (base) address of the array is in $s3. Use $t0 as a temporary register. Translate the following pseudocode into MIPS assembly.

```
g = h + A[8];
```

Solution: The trick here is to remember that in a 32-bit system, each address in memory is separated by 4 bytes. In other words, if you want to access the 9th index of A, you need to move four spaces in memory, rather than just one. We also need to remember the syntax for an offset. We have two ways of doing this: we can use the add instruction on the address, or we can specify an offset. Since the base address of the array is stored in $s3, we need to offset by 8×4, or 32. We will have the following:

```
lw $t0, 32($s3)
add $s1, $s2, $t0
```

Before we continue, lets talk about MIPS instructions in more detail. MIPS is built for a 32-bit system, and as a result, MIPS instructions are all 32 bits. We have three types of instructions, R-Type, I-Type, and J-Type. R-Type instructions deal with arithmetic operations, such as add, sub, mult, slt, and the like. I-Type instructions deal with memory, and include instructions like lw, sw, la, and others. I-Type also deals with instructions that load values immediate J-Type instructions deal with branching and jumps, which includes instructions such as j, beq, and bne. All MIPS instructions have the same number of bits, which is best illustrated by the R-Type format:

op - 6 bits	rs - 5 bits	rt - 5 bits	rd - 5 bits	shamt - 5 bits	funct - 6 bits

Note that the instruction type changes the meaning of the bits. For example, consider an I-Type instruction as follows:

```
lw $t0, 400($t1)
```

The opcode for this instruction is $(35)_{10}$ (you would have to look this up in a table). The values for rs and rt are 9_{10} and 8_{10}, respectively. However, we won't use the last 16 bits the same way we would use it for an R-Type instruction. For an I-Type instruction, the last 16 bits are used for the offset. In this case, they 32 bit representation of this instruction would look like:

$(35)_{10}$ - 6 bits	$(9)_{10}$ - 5 bits	$(8)_{10}$ - 5 bits	$(400)_{10}$ - 16 bits

Now consider the R-Type add instruction:

```
add $t0, $s2, $t0
```

Which would result in the following bits:

$(0)_{10}$ - 6 bits	$(18)_{10}$ - 5 bits	$(8)_{10}$ - 5 bits	$(8)_{10}$ - 5 bits	$(0)_{10}$ - 5 bits	$(32)_{10}$ - 6 bits

As you can see, an R-Type instruction uses each section, while the I-Type does not. Let's look at a J-Type instruction, which will only have two sections

```
j branchname
```

This instruction results in the following table:

$(2)_{10}$ - 6 bits	$(24)_{10}$ - 26 bits

It's worth mentioning that there are also FR-Type and FI-Type instructions. The difference between FR/FI-Type and R/I-Type is that FR/FI-Type deal with floating point numbers. The instruction format is the same, but rather than rs, rt, rd, and shamt, we have fmt, ft, fs, and fd.

Question 3: Given the following MIPS code, create a trace table:

```
0x400:a addi    $t0, $zero, 3
0x404:b add     $v0, $t0, $t0
0x408:c addi    $t0, $t0, -1
0x40c:d bne     $t0, $zero, b
0x410:e j       e
```

Solution: For each instruction, we will analyze the values of the program counter and the $t0 and $v0 registers. Let's take it instruction by instruction:

a) `0x400:a addi $t0, $zero, 3`

In this case, there is nothing in $v0, so we just forget about it for now. The program counter is the element that holds the current address in memory, so in this step it contains 0x400. Finally, we are adding $3 + 0$ and storing the value in $t0, which makes $t0 = 3.

b) `0x404:b add $v0, $t0, $t0`

Now, we are adding $t0 to itself and storing the value in $v0. We know from *a)* that $t0 contains 3, so $v0 = 6. The program counter has increased by 4.

c) `0x408:c addi $t0, $t0, -1`

We are essentially decrementing $t0 here, so $t0 will contain 2. $v0 is unchanged, and the program counter is incremented by 4.

d) `0x40c:d bne $t0, $zero, b`

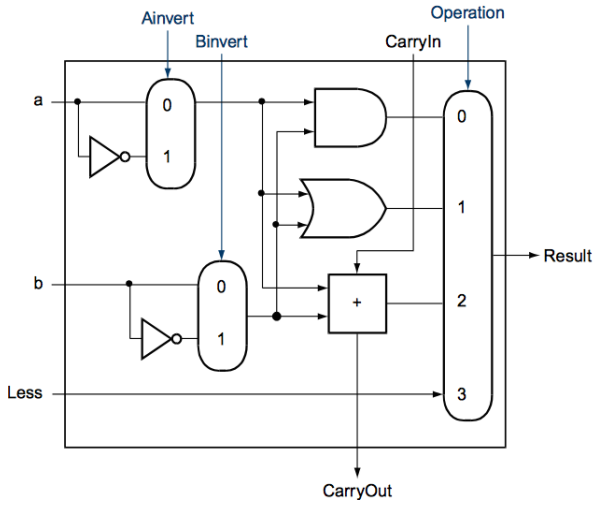
Here is where things get interesting. Recall that a ben instruction says that if (in this case) $t0 is not equal to $zero, then goto b. Since $t0 is not 0, we jump back up to b. The program counter in this step is incremented by 4, and $v0 and $t0 do not change.

Now we can continue to fill out the table. We know that it will continue to branch until $t0 holds the value of $zero. The table is as follows:

Trace Table											
Step	1	2	3	4	5	6	7	8	9	10	11
pc	0x400	0x404	0x408	0x40c	0x404	0x408	0x40c	0x404	0x408	0x40c	0x410
$t0	3	3	2	2	2	1	1	1	0	0	0
$v0	-	6	6	6	4	4	4	2	2	2	2

The ALU in Hardware

Now we will talk about the ALU in terms of hardware. For the next few problems, we will consider the following diagram, which is one ALU unit. In a 32-bit system, there would be (you guessed it) 32 of these units.



We actually can observe quite a lot about the ALU pictured above. For example, lets imagine that we have $a = 1$, and $A\ invert = 1$. Then we know that the value from the MUX that handles the a input will output 0, since the select line choses the inverted a. The same can be said for the MUX that handles the b signal. There is also another MUX that handles the resulting output of the ALU.

You can imagine that in a 32-bit system, this arrangement would not be very efficient. We won't go into too much detail, but the strategy to implement the ALU in a 32-bit (or even 64-bit) is **not** to align each of the ALU's in series. Rather, the hardware uses a *carry lookahead* to help make the system more efficient Let's talk about how different instructions are implemented in the ALU.

Question 1: Describe how the following instruction could be implemented in the ALU:

```
slt $t0, $t1, $t2
```

Solution: Recall what the *slt* instruction does. If \$t1 < \$t2, then \$t0 receives the value of 1; otherwise, \$t0 gets 0. So the question we are really asking is "How can the ALU determine if a value is less than another value?" We can tell when two values are equal by subtracting them and seeing if the value is zero, so that takes care of the case where the values are equal. Now we need a way to determine if \$t1 > \$t2. To do this, we can subtract \$t2 from \$t1, and if the value is greater than 0, then we know that \$t1 > \$t2. If neither of these are true, then we set all the bits in \$t0 to 0. The hardware is implemented using adders. For the hardware to take the 2's compliment, it must invert the and then add 1 to the result. For example, we can try the following:

a) Let $a = 0011110$ and $b = 0101110$. Determine the value of f given the following:

```
slt f, a, b
```

Remember that in order to subtract binary numbers, we need to find the 2's compliment of the second number and add it into the first. Alternatively, we can convert the numbers into base 10 and then back into base 2. We will do it the second way here. Changing both into decimal give us:

$$(0011110)_2 = (0 + 0 + 2^4 + 2^3 + 2^2 + 2 + 0)_{10} = (30)_{10}$$

$$(0101110)_2 = (0 + 2^5 + 0 + 2^3 + 2^2 + 2 + 0)_{10} = (46)_{10}$$

It's very easy to see here that $a < b$, so we know that $f = 1$. Now let's try it with the 2's compliment, since the hardware cannot change binary into decimal. We only need to change a into the 2's compliment.

$$a: \ 0011110 \rightarrow 1100001$$

$$\begin{array}{rr} & 1100001 \\ + & 1 \\ \hline & 1100010 \end{array}$$

Now we add b to the 2's compliment of a:

$$\begin{array}{rr} & 1100010 \\ + & 0101110 \\ \hline & 10010000 \end{array}$$

Note that we have a signed int here, so we can just ignore the signed bit. This gives us what we expected: that is $a < b$, or that $b - a > 0$. Thus, f will be set to 00000001.

Question 2: Describe how the following instruction could be implemented in the ALU:

```
beq $t0, $t1, label
```

Solution: This is a very simple problem. We already looked at the *slt* instruction, and we stated that we can tell if $\$t0 == \$t1$ if their difference is zero, and the program counter will change to the address of *label*.