1. What is the range of addresses in KB for conditional branches in MIPS (KB = 1024 Bytes)?
    1. Addresses between 0 and 64K-1
    2. Addresses between 0 and 256K-1
    3. Addresses up to about 32K before the branch to about 32K after
    4. Addresses up to about 128K before the branch to about 128K after

Solution:

As shown on page 119 (Section 2.10) of your textbook, branch instructions in MIPS get 16 bits for the address they are trying to branch to. This means we can only branch to addresses up to 2^{16} away from the current PC (branches are PC-relative). However, we are able to branch forward or backwards, so we need 1 bit to represent the sign. This reduces the range to 2^{15}. But, we also need to consider that branches must be word-aligned, hence we need to multiply by 4. Thus the formula is:

$$New\ Address = CurrPC + BranchImmediate * 4$$

All of this means that 4 is the correct answer, because 2^{15} = 32K * 4 = 128K.

2. What is the range of addresses for jump and JAL (jump and link) in MIPS (K = 1024)?
    1. Addresses between 0 and 64M – 1
    2. Addresses between 0 and 256M – 1
    3. Addresses up to about 32M before the branch to about 32M after
    4. Addresses up to about 128M before the branch to about 128M after
    5. Anywhere within a block of 64M addresses where the PC supplies the upper 6 bits
    6. Anywhere within a block of 256M addresses where the PC supplies the upper 4 bits

Solution:

As is also shown on page 119 (Section 2.10) of your textbook, jump instructions in MIPS have 26 bits for the address they are jumping to. Note that jumps jump to an address, and are not PC-relative – this means that they can jump anywhere within the addressable block (this rules out 3 and 4). We know that the lowest 2 bits will be 0 to make the address word aligned, which means the PC can only supply 4 bits – this rules out 5. Also, the PC supplying the 4 most significant bits means that the addresses could be in a range that does not start at 0 – this rules out 1 and 2. That leaves 6 as the correct answer.

Thinking about the problem from the opposite direction: given that jump instructions in MIPS have 26 bits for the address they are jumping to, this means we can jump to addresses up to 2^{26} (64 MiB). However, the 2 least significant bits of the address are also 0, which means the address can be up to 256M addresses, instead of 64M.

3. Consider a program running on your computer. You apply a technique that accelerates non-memory instructions in your program by 4x. This speeds up the entire program execution by 3x. Now would it be better for performance to (i) eliminate half of the memory instructions in your program or (ii) improve the technique such that it accelerates non-memory instructions by 6x?

Solution:

First we need to find the fraction, *f*, of instructions that are non-memory instructions, since these are the instructions our optimization helps with. To find *f*, we can use Amdahl's Law, since we know that only the portion of the program that is non-memory instructions is sped up by the optimization:

$$\frac{1}{1-f+\frac{f}{4}} = 3$$

Solving for f, we get f = 8/9. This means that 8/9ths of the instructions are non-memory instructions.

Now, we can analyze whether (i) or (ii) is better. To do so, we can again use Amdahl's Law, but now we adjust the terms according to what (i) or (ii) changes.

(i) Eliminating half of the memory instructions is like making half of the memory instructions (which make up 1/9*1/2 = 1/18th of the instructions in the program) infinitely fast. Thus:

$$\frac{1}{\left(\frac{1-f_{nonmem}}{2}\right)+\frac{\left(\frac{1-f_{nonmem}}{2}\right)}{\infty}+\frac{(f_{nonmem})}{4}}$$

$$\frac{1}{\left(\frac{1}{18}\right)+\frac{\left(\frac{1}{18}\right)}{\infty}+\frac{\left(\frac{8}{9}\right)}{4}} = 3.6X$$

(ii) Here we simply change the speedup for the fraction of non-memory instructions to 6 instead of 4:

$$\frac{1}{\left(\frac{1}{9}\right)+\frac{\left(\frac{8}{9}\right)}{6}} = 3.86X$$

Thus (ii) is a better choice.

4. Consider a processor architecture with an average CPI of 1.8. You propose a change to the architecture that is able to eliminate half of all memory read instructions in the program but incurs a single-cycle

overhead on all memory write instructions. Given the typical instruction breakdowns that you expect in your programs (table below), is this change a worthy trade-off for performance?

% of Program	Instruction Type
10%	Addition/subtraction
5%	Multiplication
40%	Memory read
10%	Memory write
15%	Logic
20%	Branch

Solution:

After removing half of memory read instructions, the % of memory writes = 0.1 / (1 – 0.4/2) = 0.125.

(Another way to think about this: if we had 100 instructions originally, now we have 80 instructions after removing the 20 memory reads. 10 memory writes / 80 total instructions = 0.125)

Now we can apply the speedup equation, with our new values for the number of accesses, to see when this tradeoff is worthwhile:

$$Speedup = \frac{execution\ time_{old}}{execution\ time_{new}}$$

$$Speedup = \frac{instructions_{old} * CPI_{old} * f}{instructions_{new} * CPI_{new} * f}$$

$$Speedup = \frac{\#\ instructions_{old} * CPI_{old}}{\left(\left(1 - \frac{0.4}{2}\right) * \#instructions_{old}\right) * \left(CPI_{old} + 1 * \left(\frac{0.1}{1 - \frac{0.4}{2}}\right)\right)}$$

$$Speedup = \left(\frac{1}{1 - \frac{0.4}{2}}\right) * \left(\frac{1.8}{1.8 + \frac{0.1}{1 - \frac{0.4}{2}}}\right) = 1.17X$$

Thus it would be worthwhile for a 1.17X speedup (i.e., the change helps).

Note: when we apply the speedup equation here, we are using the Iron Law of Performance to compare the relative speedup of the two variants, where the frequency remains constant and thus cancels. The trick though is the dynamic instruction count and CPI are different for the new version (since we remove 20% of reads and the writes take an extra cycle), so we need to determine how to express this information. However, we can express the new version's execution time as a ratio of the old version's execution time – all of the remaining instructions take at least as long as they did before (the left part of the denominator), and the additional cycle for the memory writes must be added to this (the right part of the denominator).

5. Rewrite the following MIPS assembly code such that it produces the same output (*$s2*) but executes less instructions:

```
addi $t3, $s3, 4
add $t3, $s5, $t3
lbu $t2, 0($t3)
add $s2, $s4, $t2
```

Solution:

```
add $t1, $s5, $s3
lbu $t2, 4($t1)
add $s1, $s4, $t2
```

6. Consider the following MIPS assembly code:

```
Loop: sll $t1, $s3, 2
      add $t1, $t1, $s6
      lw $t0, 0($t1)
      bne $t0, $s5, Exit
      addi $s3, $s3, 1
      j Loop
Exit:
```

(a) Write down the machine code of these assembly instructions:
2points each for correct answer, award partial credit 0.5 points for last fill in the blank if the exit label address is incorrect, but rest is correct.

Answer from what we've learned thus far:

Assembly Program Instruction	Machine code (in hexadecimal)
`add $t1, $t1, $s6`	0x01364820
`addi $s3, $s3, 1`	0x22730001
`lw $t0, 0($t1)`	0x8D280000
`bne $t0, $s5, Exit`	0x15150003

Answer once we learn about branch delay slots:

Assembly Program Instruction	Machine code (in hexadecimal)
`add $t1, $t1, $s6`	0x01364820
`addi $s3, $s3, 1`	0x22730001
`lw $t0, 0($t1)`	0x8D280000
`bne $t0, $s5, Exit`	0x15150002

The trick here for determining the address of Exit is that it comes 3 instructions after the branch and that branch addresses are PC relative (i.e., they get added to the current PC). As we discussed above in problem 1A, we know that branch addresses will be multiplied by 4 to make them word aligned. Thus, it would seem that the address for Exit should be 3. However, MIPS uses something called a branch delay slot, which we will discuss later in the semester. For the purpose of this

problem, all you need to know about the branch delay slot is that it reduces the number of instructions away Exit is from the branch by one. Thus the address for the bne is 2.

7. The above assembly code is compiled from the following C statement:

```
while (A[i] == k) i++;
```

Which registers contain:

- Integer variable i? 1 point
- Integer variable k? 1 point
- Base address of integer array A (&A[0])? 2 points

Solution:

[$s3] = i
[$s5] = k
[$s6] = &A[0]