Problems from Past Exams

1 (15%). Remember that typically, we have only room for 16-bit immediate values in the instructions (see the format of typical instructions given below).

Describe at least 2 different solutions (not just slight variations of each other) for storing a 32-bit immediate value in a register using instruction formats shown above.

**Key:**

a). One easy solution will be to have two different instructions: Load Immediate Upper or Load Immediate Lower, to indicate if you are loading into the upper half or lower half.

In this case, the Rs src is ignored.

b). We can also load always into lower half (and the upper half will not be effected); then shift the value to the left by 16 positions, and then load the remaining 16 bits in the lower half.

Again, Rs src is ignored.

c). A third way (used in PowerPC), is to add 16 bit immediate value to either the lower half or upper half of a register. In this case, we add the 16 bits to the either the upper or lower half of Rs rc (other bits are un-effected). So, we can first zero Rs rc, add 16 bit immediate to the upper half, and then add 16 bit to the lower half of Rs rc.

d). We can also add 16 bits always to the lower half of Rs rc. First zero Rs rc, add 16-bit immediate to the lower half; shift left Rs rc by 16 bit positions, and then add 16 bit immediate to the lower half.

e). We can use two words for the instruction, the second 32-bits will contain the immediate value. This is what one of you mentioned -- I gave full credit, but this does not conform to the question, since my question mentions using only 32-bit instructions

2 (25%). Consider a machine which showed the following benchmark results

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Cycles to execute</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>30%</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>25%</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>20%</td>
</tr>
<tr>
<td>D</td>
<td>4</td>
<td>15%</td>
</tr>
<tr>
<td>E</td>
<td>5</td>
<td>10%</td>
</tr>
</tbody>
</table>

What is the CPI for machine based on this data? Assuming a 125Mhz clock, what is the MIPS rating?
The compiler group decided to improve the performance. The new optimization resulted in the following.

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>Percentage of instructions as compared to unoptimized programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>95%</td>
</tr>
<tr>
<td>B</td>
<td>85%</td>
</tr>
<tr>
<td>C</td>
<td>80%</td>
</tr>
<tr>
<td>D</td>
<td>90%</td>
</tr>
<tr>
<td>E</td>
<td>75%</td>
</tr>
</tbody>
</table>

That is, if the unoptimized program executed 200 type B instructions, the optimized program will need only 170 type B instructions.

What is the new CPI after the compiler optimization? What is the MIPS rating?

**Key:** For the first part, the CPI is given by

\[
1 \times 0.3 + 2 \times 0.25 + 3 \times 0.2 + 4 \times 0.15 + 5 \times 0.1 = 2.5
\]

That is, on the average an instruction takes 2.5 cycles. Since the clock is operated at 125 Mhz (125 Million cycles per second), the MIPS rating is given by that is 50 MIPS.

For the second part, we can construct the actual instruction frequencies. We can find the CPI rating as follows.

\[
1 \times 0.3 \times 0.95 + 2 \times 0.25 \times 0.85 + 3 \times 0.2 \times 0.8 + 4 \times 0.15 \times 0.9 + 5 \times 0.1 \times 0.75
\]

\[= 2.105\]

In other words, after compiler optimization, the benchmark has a CPI of 2.105 cycles per instruction. Using the same clock of 125 Mhz, the MIPS rating is or 59.38 MIPS.

3. (15%) We have studied the Amdahl’s law, which states that the speed-up that can be achieved using an enhancement is limited by the fraction \(f\) of the time this enhancement can be applied. This can also be described in terms of the *increased capacity* resulting from the enhancement. In other words, since the enhancement decreases the execution time, we can do more work in the same amount of time. Write an expression that shows the increased capacity when a fraction \(f\) is speeded up by a factor \(s\). What is the upper bound on the increased capacity?

**Key.** Remember that we use Amdahl’s law to indicate speed up which is given by

\[
\text{Speed-Up} = 1/[(1-f) + f/s] = s/[s - (s-1)*f]
\]

In this problem we are rephrasing the equation. Let us look at the amount of time reduction using the enhancement. Time reduction is given by (original time – new time)
Time saved = \(1 - [(1-f) + f/s)] = f(s-1)/s.\)

To complete one unit of work in the enhanced mode we need \([(1-f) + f^*s]. So how much extra work can we do in \(f(s-1)/s\) time units?

Enhanced capacity = \[f(s-1)/s]/[(1-f)+f^*s] = [(s-1)*f]/[s-f(s-1)]\]

The maximum enhanced capacity happens when \(f= 1\) and is given by \(s-1\).

This makes sense since when \(f=1\), speed up is \(s\). So, we can do \(s-1\) additional units of work in the same time.

The lower bound on enhanced work capacity happens when \(f=0\); as you can from my equation, when \(f=0\), there is no enhanced capacity as expected.

Let us see what are the implications of phrasing Amdahl’s law in this manner. Consider the example from page 30 of our textbook, where we are given \(f= 0.4\) and \(s=10\). The resulting speed up is 1.66.

But in terms of increased capacity we get 0.5625. That means we can do 56% more work in the same time. This phrasing makes us feel a bit more comfortable than saying that you can get only 1.66 times speed up with 10 times enhancement.

A variation of this increased capacity law was originally derived by John Gustavson of Ames Research Labs at Iowa State University and is the foundation of “Scalability” studies of parallel processing algorithms.

I am surprised that most of you failed to see what I am asking in this problem. Most of you were trying work with the speed-up equation.

4. (25%) Consider the following possible enhancements for floating point applications
   Make multiplication 10 times faster
   Make Load/Store 4 times faster
   Make add 2 times faster

a). Using the averages for SPEC 2000 benchmarks shown on page 139 of the test book, indicate the performance improvements (speed-up) that is possible with each of the enhancements.

b). If you can choose any two enhancements, together, which two will produce highest improvements?

Key. From page 139, we have 8% FP multiply and 7% FP Add. For Load and Store, we include all Load and Stores (both FP and integer), we have a total of 15+2+15+7 = 39%. But we only use FP Load and Store we have 22%.
a). Making FP Multiply 10 times faster gives us
\[ \frac{1}{0.92 + \frac{0.08}{10}} = \frac{1}{0.928} = 1.0776 \text{ or } 7.76\% \text{ speedup} \]

Making Load/Stores 4 times faster
- Only FP load stores
  \[ \frac{1}{0.78 + \frac{0.22}{4}} = \frac{1}{0.835} = 1.198 \text{ or } 19.8\% \text{ speedup} \]
- All Load/Stores
  \[ \frac{1}{0.61 + \frac{0.39}{4}} = \frac{1}{0.7075} = 1.413 \text{ or } 41.3\% \]

Making FP adds 2 times faster
\[ \frac{1}{0.93 + \frac{0.07}{2}} = \frac{1}{0.965} = 1.036 \text{ or } 3.6\% \]

Thus Load and Store improvement achieves highest speed-ups

b).

- Multiply + Load/Store (only FP load/store)
  \[ \frac{1}{0.7 + \frac{0.08}{10} + \frac{0.22}{4}} = \frac{1}{0.763} = 1.311 \text{ (or } 31.1\%) \]
- Multiply + all Load/store
  \[ \frac{1}{0.53 + 0.008 + 0.0975} = \frac{1}{0.6355} = 1.574 \text{ (or } 57.4\%) \]
- Multiply + Add
  \[ \frac{1}{0.85 + 0.008 + 0.035} = \frac{1}{0.893} = 1.12 \text{ (or } 12\%) \]
- Add and FP Load/Store
  \[ \frac{1}{0.71 + 0.035 + 0.055} = \frac{1}{0.8} = 1.25 \text{ (or } 25\%) \]
- Add and all Load/Store
  \[ \frac{1}{0.54 + 0.035 + 0.0975} = \frac{1}{0.6725} = 1.487 \text{ (48.7\%) } \]

5. (20%) Assume you have a 32-bit address. Show how the 32-bits are used if your cache is 32-Kbytes for each of the following case. Also show to which cache set the (hex) address ABCD8F8 falls to in each of these cases.

a). 32-byte cache lines and direct mapped
b). 64-byte cache lines and direct mapped
c). 32-byte cache lines and 4-way set associative.

Key.

Since we have a 32Kbyte cache, we need 15 bits to address each byte of this cache.

a). Since we are using 32-byte cache lines, we will have 1024 cache lines. We need the 5 least significant bits to find a bit within a cache line, and we will use the next 10 significant bits to locate a cache set (here each set has one element since we are looking at direct mapped cache.)
The address ABCED8F8 translates to 1010 1011 1100 1110 1 10100 111 1 1000
Thus the set number is 101 1000 111 (711) and the byte is 20.

b). With 64-byte lines we have 512 cache lines; we need least 6 bits to find a byte (byte offset) and we need next 9 bits to index a set (or a line since each set has only one line in direct mapped)

The address now maps as follows 1010 1011 1100 1110 1 101 1000 11 1 1000
Thus the set number is 101 1000 11 (355) and the byte is 56.

c). Note that like in (a) we still have only 1024 lines of 32-bytes each. However since he have a 4-way set associative cache, we only have 1024/4 = 256 sets (each set containing 4 lines).

Now the address map looks like 1010 1011 1100 1110 1 10 1100 0111 1 1000.
The set number is 1100 0111 (199) and byte is 20.

6. (30%) Consider the following problem for computing miss penalties and CPI due to cache misses. It takes It takes 4 cycles to send an address and initiate a DRAM memory access. It takes 4 cycles to read (or write) a 4-byte word from memory (using non-interleaved memory). It takes 1 cycle to send (or receive) a 4-byte word.

a). If your cache uses 32-byte cache lines, how long will it take to exchange data between the main memory (DRAM) and the cache (i.e., miss penalty).

b). 30% of all instructions access data (load and store instructions). We will assume split data caches (separate Instruction and Data cache) and that the instruction cache miss rate is 5% while the data cache miss rate is 10%. If the CPI on a cache hit is 1 cycle, what is the CPI with cache misses?

c). Let us assume that we are using write-back caches and that 50% of the cache lines are dirty. What is the CPI under this assumption?

d). If we are using 4-way interleaved memory, what is the miss penalty. Note that all 4 memory modules can read their respective words in parallel but need to send one word at a time.

Key.

a). Here we have a latency of 4 cycles. For each word (4-bytes), we need 4 cycles to access the data and 1 cycle to transmit the data. Since we have 32-byte cache lines, we need to access and transmit 8 words.

Miss Penalty = 4+ 8*(4+1) = 44 cycles
b). Since we have split caches, let me compute the CPI contribution due to instructions and data separately.

Instruction CPI = 1 + 5%*44 = 1 + 2.2 = 3.2 cycles
Data CPI = 30%[1 + 10%*44] = 30%*5.4 = 1.62 cycles

Total = 4.82 cycles

c). If we have write back cache, only the Data CPI will change; on a miss we have 1.5 memory accesses since 50% we need to write a dirty line back to memory.

Instruction CPI = 1 + 5%*44 = 1 + 2.2 = 3.2 cycles
Data CPI = 30%[1 + 10%*1.5*44] = 2.28 cycles
Total = 5.48 cycles

d). If we use a 4-way interleaved, we only 4 cycle latency to initiate access to all 4 modules, the four modules can access a 4-byte word in 4 cycles in parallel; then each module will need 1 cycle to send its word in series. Since we need 8 words, we need 2 accesses to the 4-way interleaved memory.

Miss Penalty = 4 + 2*(4+ 4*1) = 20 cycles.

7. (15%) Consider a processor with a 16 Kbyte 4-way set associative cache, with 8-byte lines. Show how you will translate 32-bit address to find the data in cache. Show the cache set to which the following (hex) address falls: ABCED8F8. How many bits are needed for tags in this cache?

**Key:**
A 16K Byte cache with 8-byte line will contain 2048 lines and with 4-way set associativity, we will have 512 sets (each set with 4 lines). Thus we need 9-bits to locate a set, and 3-bits to locate a byte within a line (since each line has 8bytes). So we will divide a 32 bit address as follows.

<table>
<thead>
<tr>
<th>20-bit Tag</th>
<th>9-bit set index</th>
<th>3-bit</th>
</tr>
</thead>
</table>

Address ABCED8F8 = 1010 1011 1100 1110 1101 1000 1111 1000
Yields 1 000 1111 1 = 287 as the set number (and byte zero in the line)
This leave the tag as ABCED = 1010 1011 1100 1110 1101

There are a total of 2048 lines and each line will have 20 bit tags. So the total number of bits for tags = 20*2048 = 40,960 bits

8. (25%) The instruction set of your architecture has 32-bit addresses, with each addressable item being a byte. You elect to design a direct mapped cache with a line size of 32 bytes. Assume that there are 32-sets in the cache.

a). Show how the 32-bit physical address is treated in performing a cache reference
b). Consider the following sequence of addresses (all are hex numbers). Compute the miss rate. Assume that none of the above addresses are initially in the cache. Show what will be the tags in the cache set(s) that contain these references, after each reference.

1) B01AA050; 2) B01AA073; 3) B2FE3057; 4) B01AA07E;
5) B01AA050; 6) B2FE3057; 7) B01AA073

c). Consider now designing a direct mapped cache with a victim cache with 2 entries (a fully associative cache). What is the miss rate?

**Key.** You need to play a closer attention to tags. Simply because two references fall to the same set, it may not mean a hit or a miss, unless you compare the tags. If two references fall to the same set and if their tags match, then it is a hit. Otherwise it is a miss.

We are looking at a 32 bit virtual address. Since we have a line size of 32 bytes, the least significant 5 bits are used as an address of the byte in a line. Since we have 32 sets in the cache, the next 5 bits will be used as an index into the cache (to find one of the 32 sets). That leaves 22 bits for cache Tag.

b). Let us first find the tags and sets for the 7 references.

<table>
<thead>
<tr>
<th>Ref#</th>
<th>Tag</th>
<th>Set #</th>
<th>Byte Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>1011 0000 0001 1010 1010 00</td>
<td>00010 (2)</td>
<td>10000</td>
</tr>
<tr>
<td>2.</td>
<td>1011 0000 0001 1010 1010 00</td>
<td>00011 (3)</td>
<td>10011</td>
</tr>
<tr>
<td>3.</td>
<td>1011 0010 1111 1110 0011 00</td>
<td>00010 (2)</td>
<td>10111</td>
</tr>
<tr>
<td>4.</td>
<td>1011 0000 0001 1010 1010 00</td>
<td>00011 (3)</td>
<td>11110</td>
</tr>
<tr>
<td>5.</td>
<td>1011 0000 0001 1010 1010 00</td>
<td>00010 (2)</td>
<td>10000</td>
</tr>
<tr>
<td>6.</td>
<td>1011 0010 1111 1110 0011 00</td>
<td>00010 (2)</td>
<td>10111</td>
</tr>
<tr>
<td>7.</td>
<td>1011 0000 0001 1010 1010 00</td>
<td>00011 (3)</td>
<td>10011</td>
</tr>
</tbody>
</table>

Note that references 1, 3, 5 and 6 all fall to the same set. Comparing the tags, you will find that the tags are different (from 1 to 3, from 3 to 5 and from 5 to 6). However, even though references 2, 4 and 7 fall to the same set, their tags match. So, there are actually referencing different bytes of the same line. Let us consider which references cause misses.

<table>
<thead>
<tr>
<th>Ref#</th>
<th>Miss or Hit</th>
<th>Any Victim?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Miss</td>
<td>none (cold start)</td>
</tr>
</tbody>
</table>
Of the 7 references, we have 5 misses. The miss rate = 5/7.

c). If we use a victim cache, the victims shown earlier can be stored in the cache. This eliminates the misses caused by references 5 and 6 -- these references can be satisfied by the victim cache. So, the new miss rate is 3/7.

9. (30%). Consider a system with L-1 and L-2 caches, main memory and disk (or secondary memory) with the following parameters. It takes 10 ns to access L-1 cache, 100 ns to access L-2 Cache, 500 ns to access RAM and 10 ms to access disk. A program makes 250,000 memory references of which 215,000 are hits in L-1 cache, 235,000 are hit in L-2 cache, 245,000 are hit in RAM.

a). What are the L-1, L2 cache and RAM miss ratios?

b). Estimate the average access time assuming all memory accesses are reads.

c). If we are using write-back policy for L-1, 75% of all memory accesses are reads and 50% of all cache lines are dirty, what is the effective memory access time?

Key:

a). L-1 Miss rate = 35000/250000 = 0.14 (or 14%)
   local L-2 miss rate = 15000/250000 = 0.06 (6%)
   global L-2 miss rate = 15000/35000 = 0.4286 (42.86%)
   global RAM miss rate = 5000/250000 = 0.02 (2%)
   local RAM miss rate = 5000/15000 = 0.33 (33%)

b). Access time = (215000/250000)*10 + (20000/250000)*100 + (10000/250000)*500
   + (5000/250000)*10,000,000
   = 0.86*10 + 0.08*100 + 0.04*500 + 0.02*10,000,000 = 200,036.6 ns

c). We have a write back L-1, on a miss in L-1 we may have to write the dirty line back to memory 50% of the time. The information that 75% of all accesses are reads is not relevant, since either on a read or a write, a miss will cause a dirty line to be written back. So, on a miss in L-1, we have to allow for 1.5 accesses to L2.

Access time = (215000/250000)*10 + 1.5*(20000/250000)*100 + (10000/250000)*500
   + (5000/250000)*10,000,000
   = 0.86*10 + 1.5*0.08*100 + 0.04*500 + 0.02*10,000,000 = 200,040.6 ns

10. (15%) Suppose we have the following parameters on a two level cache system (L1 is
closer to CPU).
L1 Cache:
  Size = 4Kbytes
  Hit rate: 1 cycle
  Miss rate: 0.10 misses per memory reference
  Miss penalty: 3 cycles if found in L2 cache
L2 Cache:
  Size = 64Kbytes
  Miss rate: 0.02 misses per memory reference
  Miss penalty: 10 cycles

Assume that 20% of all instructions reference memory (i.e., load and store).

What is the CPI of this processor (In a perfect cache system, each instruction takes 1 cycle)?

**Key:** The number of memory references per instruction = 1.2 (one for instruction and 0.2 for operand references).

So, let us compute the average memory access time (or the number of memory stalls in an imperfect world).

Since 90% of the references are satisfied by L1 cache, we have 1.2*0.9 as the stalls when a memory references is a L1 hit.

On a L1 miss, if we have a L2 hit, the stalls needed is: 1.2*0.1 *0.98* 3. If we have a L2 miss, then the stalls needed is: 1.2*0.1*0.02 *10. So, the average number of stalls is given by

\[
1.2*0.9 + 1.2*0.1 *0.98* 3 + 1.2*0.1*0.02 *10 = 1.4568 \text{ Cycles}
\]

Now, since the CPI in the perfect world is 1, the CPI including the memory accesses is given by 1 +1.4568 = 2.4568.

I gave full credit if you computed the memory stalls correctly.

As an aside, if we did not have an L2 cache, then all cache misses will have a penalty of 10 cycle. So, the stalls would have been:

\[
1.2*0.9 + 1.2*0.1*10 = 2.28 \text{ cycles}
\]

and the CPI would have been 3.28 cycles.

11. (30%) Consider the following program segment in C.

```c
for (k= 0; k < 50; k++)
{
    for (i =0; i < 100; i ++)
```
Let us only worry about memory accesses generated by the arrays a, b, x, new_x (ignoring accesses for scalar variables like i, j, k, sum) and each element is one word (32 bits).

a). What is the expected cache miss rate if each cache block is 16 bytes (and we will assume the cache is large enough so that any data brought into cache is not displaced). Initially the data is not in cache (cold start)

b). What is the expected cache miss if the cache block size is increased to 32-bytes each?

Key:

a). Each line of the cache is 16 byte and will hold 4 array elements. So each time a line is brought into cache, 4 consecutive array elements are brought into cache. Matrices are stored row-major and thus the 4 elements belong to the same row.

Consider the loop

```c
    for (j =0; j < 100; j ++)
        if ( i !=j) sum = sum + a[i][j]*x[j];
    new_x[i] = (b[i] - sum)/a[i][ i];
```  

Here we are access the elements of the ith row. So we will have 100/4 = 25 misses for accessing a[i][*].

So for the outer i-loop which is repeated 100 times, we will have 25 misses for each a[i][*].

Total a misses = 100*25 = 2500

We will have to bring x, new_x, and b arrays only once, and we will have 100/4 = 25 misses each for b, new_x and x arrays.

```c
    for (i =0; i < 100; i++)
        x[i] = new_x[i];
```

will not generate any additional misses since new_x is already in cache.

So, the total number of misses = 2500+25+25+25 = 2575 misses.

Number of accesses:

Note that in

```c
    for (j =0; j < 100; j ++)
        if ( i !=j) sum = sum + a[i][j]*x[j];
```
we have 100 accesses a and 100 access to x.
Then if we consider the outer loop
for (i =0; i < 100; i ++)
{
    sum = 0;
    for (j =0; j < 100; j ++)
        if ( i !=j) sum = sum + a[i][j]*x[j];
    new_x[i] = (b[i] - sum)/a[i][ i];
}

For each i iteration we have 100+1 (for a[i][i]) accesses to a; 100 access to x, 1 access to
new_x and one access to b. The total number of access are
100*(101) access to a;
100*100 access to x
100 accesses to new_x
100 access to b
The total number of access = 20,300

The
for (i =0; i < 100; i ++)
    x[i] = new_x[i];
}

will add 100 accesses each to x and new_x

The accesses for each k loop add to 20,500
Since we have 50 values for k, the total number of access = 50*20,500 =102,500

So miss rate = 2575/102500 =0.25%

b). If the line size is doubled to 32 bytes, each line will have 8 elements. The total
number of misses will be halved for accessing each array
We will have 100/8 = 13 misses (note we will have 4 wasted elements in the last access)
for accessing each row of a, and 13 misses accessing each of the b, new_x and x arrays.

Total misses = 100*13 + 13 +13+13 = 1339 misses

The miss rate = 1339/102500 =0.13%