Problem 1: Memory Modeling

1.1 Briefly explain what is meant by the term Big Endian in the context of this course. Answer: If you don’t think of the term “memory address” when you see the terms Big-Endian and Little-Endian, then you don’t know your definition.

Typically, we refer to memory as byte addressable. This means that aligned accesses occur at byte boundaries, and that byte, half-word, word, and double-word loads and stores must occur at predefined locations, as shown in the table below. Note that all data access instructions are considered to be integer instructions because they require the computation of the effective address using the INT ALU unit.

<table>
<thead>
<tr>
<th>Dest/Src Register</th>
<th>Memory Access</th>
<th>MIPS64 OP Code</th>
<th>Effective Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPR (R0-R31)</td>
<td>load byte</td>
<td>LB</td>
<td>0 mod 1</td>
</tr>
<tr>
<td></td>
<td>store byte</td>
<td>SB</td>
<td>0 mod 1</td>
</tr>
<tr>
<td></td>
<td>load half word</td>
<td>LH</td>
<td>0 mod 2</td>
</tr>
<tr>
<td></td>
<td>store half word</td>
<td>SH</td>
<td>0 mod 2</td>
</tr>
<tr>
<td></td>
<td>load word</td>
<td>LW</td>
<td>0 mod 4</td>
</tr>
<tr>
<td></td>
<td>store word</td>
<td>SW</td>
<td>0 mod 4</td>
</tr>
<tr>
<td></td>
<td>load double word</td>
<td>LD</td>
<td>0 mod 8</td>
</tr>
<tr>
<td></td>
<td>store double word</td>
<td>SD</td>
<td>0 mod 8</td>
</tr>
<tr>
<td>FPR (F0-F31)</td>
<td>load sp word</td>
<td>L.S</td>
<td>0 mod 4</td>
</tr>
<tr>
<td></td>
<td>store sp word</td>
<td>S.S</td>
<td>0 mod 4</td>
</tr>
<tr>
<td></td>
<td>load dp word</td>
<td>L.D</td>
<td>0 mod 8</td>
</tr>
<tr>
<td></td>
<td>store dp word</td>
<td>S.D</td>
<td>0 mod 8</td>
</tr>
</tbody>
</table>

Just like books can be placed on a shelf on their sides, with their spines readable right side up or upside down (alternately left to right or right left), or even their spines in, multi-byte words can be stored in memory with two specific byte orderings, as indicated by the location of the MSB, the most significant bit (often, the sign bit), and the location of the least significant bit (the LSB).

While the table indicates how big each word or double word is, and the address at which any such entity resides, it doesn’t tell you the byte address of the MSB (most significant bit) or the LSB (least significant bit). Typically, the LSB is the bit which is the coefficient of \(2^0\), and the MSB is the bit which is the coefficient of \(2^{31}\) (or \(2^{63}\)) when the word (or double word) is interpreted as an unsigned binary number. That is the focus of the Little-Endian or Big-Endian organization—how individual bytes of a multi-byte word are stored in memory. The MSB is the big end, and the LSB is the little end. The way to understand this is to remember the source of the name.

*Little Endian means, literally, little end, or LSB, in first.*

*Big Endian means, literally, big end, or MSB, in first.*

So, in Little-Endian organization, the LSB is the byte at address 0 mod 4 of a word, and at address 0 mod 8 of a double word. The MSB is at address 3 mod 4 for a word, and at address 0 mod 7 of a double word.

In Big-Endian organization, the MSB is the byte at address 0 mod 4 of a word, and at 0 mod 8 of a double word. For a word, the LSB is at an address which is 3 mod 4, word, and at address 7 mod 8 of a double word.
Note that a big-endian byte-level organization says nothing about the bit-level organization. Some machines are byte-level big-endian, and bit-level big-endian. Others are byte-level big-endian, and bit-level little-endian. Both bit-flavors of little-endian machines exist as well, but, this is the last you will hear of bit-level ordering.

For purposes of this course, we will always write the MSB on the left and the LSB on the right. Since MIPS assumes a Big Endian organization, the book will label the MSB as bit 0, and the LSB as bit 31 in a word, and is bit 63 in a double word.

1.2 A 256K cache is implemented with a block-size of 4K bytes. Diagram the memory address, assuming that the organization of the cache is direct mapped. Be sure to label all parts of the 48-bit wide address correctly.

**Answer:** This one is the straightforward one. That is, it works just like the formula in the book.

Block offset: 12 bits because \(4K = 2^2 \times 2^{10} = 2^{12}\)

Index: 6 bits because \(2^6 = 256K/(4K \times 1) = 2^{18-12}\)

Tag: 30 bits because \(30 = 48 - 18\)

Also, order matters: “Tag Index Offset”, with MSB on the far left.

1.3 Suppose that the 256K cache described above is implemented using a slightly different organization strategy. How is the cache organized if the index is 5-bits long? Explain your answer for full credit.

**Hint:** Diagram the memory address and analyze how you would identify a given block in the cache.

**Answer:** First, we know the organization strategy is different. Since the number of index bits is non-zero, we know it can’t be fully associative. And, since the problem states that the organization strategy has changed, it can’t be direct mapped.

Note: if it were direct mapped, we’d end up with 8K size blocks, because we’d have 32 sets instead of 64. However, no where in the problem is there any hint of a block size change. That is, a block size change is not considered to be an organizational change.

Thus, we can conclude that it is set associative. The question is, how many way set associative is it? Note that the 5 bits in the index mean that there are 32 sets. We have to determine the number of slots per set.

Now, by inspection, perhaps, you can tell that there are 2 slots per set.

However, we can also use the standard formula, assuming that it’s \(x\)-way set associative, solving

\[
2^5 = 256K/(4K \times x)
\]

for \(x\), yields \(x = 2\).
1.4 Given a memory address, explain how to determine whether or not you have a read hit. Be specific regarding your use of the fields in the address and data areas of the memory.

**Answer:** Here are the basics. I’ve left out details regarding actual access of tags—such as if they are checked in parallel or sequentially.

1. Check the index bits to determine which set to examine.
2. Check the cache tag field for a match to the tag bits.
3. If a tag match is found, and the entry is valid, then you have a read hit.

1.5 A draft version of an architecture textbook claims that good memory organization can prevent at least 99% of all cache misses from occurring. Do you believe this statement, or do you think that there might be a typographical error on this page? Make sure you justify your answer clearly using the appropriate memory-related terminology.

**Answer:** The most basic answer is that no one can ever totally eliminate compulsory misses. So, unless this is an embedded system with an extremely restricted set of programs that are started and never re-started, it’s not remotely possible to get the advertised improvements.

Sorry, Genghis! It’s off to the garage for you.
Problem 2: Primary Processes

2.1 Explain how a (1,3) GPR instruction set architecture (ISA) differs from a (0,3) GPR ISA. 

Answer: The ALU instructions of a (0,3) machine can never have a memory reference as an operand. These are typically called register-register machines.

The ALU instructions of a (1,3) machine are allowed to have up to one (1) memory reference. They are typically called memory-register machines.

So, the instructions (DADD R1, R2, R3) and (DADDI R1, R1, #10) are valid for both ISA’s.

The instructions (DADD R1, (R2), R3) and (DADDI R1, (R2), #34) are only valid for the (1,3) ISA.

2.2 Suppose your memory is byte addressable and Little Endian, with four (4) bytes per word and eight (8) bytes per double word. Write an expression for Z, the address from which I would load the most significant byte (MSB) of a double word.

That is, choose A and B such that any byte loaded from an address, Z, where

\[ Z \equiv A \mod B \]

will be properly aligned, and will contain the most significant bit (msb) of the associated double word.

Answer: This answer is embedded in the answer to problem 1.1. Because memory is little endian, the least significant byte (LSB), which happens to contain the least significant bit (lsb), of a word must be stored at an address Y, where

\[ Y \equiv 0 \mod 4 \]

That means that the MSB (the most significant byte, which contains the msb) must be stored at address W, where

\[ W \equiv 3 \mod 4 \]

Similarly, for a double word of 8 bytes, the LSB is at address

\[ U \equiv 0 \mod 8 \]

That means that the MSB (the most significant byte, which contains the msb) must be stored at address W, where

\[ Z \equiv 7 \mod 8 \]
2.3 Write an expression for memory access time, given that the miss rate is 10% and the miss penalty is 500 times the hit time (HT).

**Answer:** Since the actual hit time was not included in the problem, you get to have the blessed variable HT in your answer. Let AT be the memory access time. Then, we have:

\[ AT = HT + 0.10 \times 500HT \]

While it doesn’t make a difference in the answer to this problem if the hit time, HT, is in terms of number of clock cycles or time, it DOES matter if that number, 500, only applies if the HT is in terms of number of clock cycles, and not, say some portion of seconds.

2.4 Suppose that we consider a slightly different memory model. The miss penalty is still 500 times the hit time. However, separate memories are used for instructions and data. Furthermore,

* The miss rate for **instructions** is 7%,
* The miss rate for **data** is 13%.
* Loads transfer data from memory to registers, and make up 30% of the the instructions.
* Stores transfer data from registers to memory, and comprise 15% of the instructions.

Write an expression for the CPU execution time when the cache is not perfect.

**Answer:** This was the hard nasty question, because it made you think. First, you are supposed to know that CPU = IC \* CPI \* CC is the equation for CPU execution time.

Second, you are supposed to recognize that the memory related stuff has NO impact on the number of instructions (IC) and the clock cycle time (CC). So, the problem boils down to one that asks:

What is the CPI, assuming that the cache is NOT perfect?

From the wording of the problem, we have to distinguish between instruction memory access and data memory access. However, the read miss penalty is identical to the write miss penalty. So, we don’t have to separate memory reads from writes.

One way to do this is to write:

\[ CPI = CPI_{\text{perfect}} + CPI_{\text{ins misses}} + CPI_{\text{data misses}} \]

\[ CPI_{\text{ins misses}} = \text{ins miss rate} \times \text{ins miss penalty} \]

\[ CPI_{\text{data misses}} = \%\text{data access} \times \text{ins miss rate} \times \text{data miss penalty} \]

Note: if the miss penalty differed for reads and writes, we’d have to break down the right hand side further into (data miss rate * read miss penalty * percentage of loads) added to (data miss rate * write miss penalty * percentage of stores)

But, we don’t have that. So, we get the following pieces to work with.

\[ CPI_{\text{ins misses}} = 0.07 \times 500HT \]

\[ CPI_{\text{data misses}} = 0.45 \times 0.137 \times 500HT \]

Now, notice that for this problem, the hit rate (HT) is assumed to be in terms of number of clock cycles.