These are practice exam questions for the material covered since the mid-term exam. Please note that the final exam is cumulative. See the practice mid-term exam questions for material from the first half of the course.

1. When performing signed division, the sign of the remainder should be the same as the sign of the dividend. Why?

(Note: It took me some time to come up with this formal proof. I won’t expect you to be able to do so during an exam).

By the definition of division, when dividing the dividend D by the divisor V results in a quotient Q and the remainder R, the following holds:

\[ D = (Q \times V) + R \]  

(equation 1)

Furthermore, an axiom of division (as practiced by humans and computers) is that it produces a quotient Q such that the sign of \((Q \times V)\) is the same as the sign of D. Another axiom says that when both D and V are non-negative, the following inequality holds:

\[ D \geq Q \times V \]

In order to satisfy equation 1, above, in this case, R must be non-negative.

Now suppose the D and/or V are negative. The algorithm for signed division described in the book simply performs division using the absolute values of D and V and adjusts the sign of the resulting quotient and remainder. We show that the remainder should have the same sign as D by considering each case separately.

- **case 1:** \(D < 0\) and \(V > 0\) (so the quotient \(Q\) will be negative). In this case, \(|D| = -D\) and we perform the division using \(|D|\) which gives us:

\[ |D| = |Q| \times V + R \]

and the inequality

\[ |D| \geq |Q| \times V \]

Since \(|D| = -D\), and \(|Q| = -Q\), from the second equation we get

\[-D \geq (-Q \times V)\]

and thus

\[ D \leq (Q \times V) \]

Therefore, in order for \(D = Q \times V + R\) to be satisfied, \(R \leq 0\).

- **case 2:** \(D < 0\) and \(V < 0\) (so \(Q\) will be positive). Since \(|D| = -D\) and \(|V| = -V\), performing the division using their absolute values gives us

\[ |D| = Q \times |V| + R \]

and
Thus, 
\[-D \geq Q \times -V\]
and thus 
\[D \leq Q \times V\]
Therefore, in order for \(D = Q \times V + R\) to be satisfied, \(R \leq 0\).

- case 3: \(D > 0\) and \(V < 0\) (so \(Q\) is negative). Performing the division gives us
  \[D = |Q| \times |V| + R\]
and
\[D \geq |Q| \times |V|\]
Thus,
\[D \geq -Q \times -V\]
and thus,
\[D \geq Q \times V\]
Therefore, in order for \(D = Q \times V + R\) to be satisfied, \(R \geq 0\).

In each case, \(R\) has the same sign as \(D\).

2. Suppose there was a large floating point representation which used 16 bits for the exponent. If the exponent was represented using a bias, what would you expect the bias to be?

Just as the exponent bias on the IEEE floating point number with 8 exponent bits is \(2^8/2 - 1 = 127\), you would expect the exponent bias on a number with 16 exponent bits to be \(2^{16}/2 - 1 = 32767\). This divides the range of possible exponent values almost equally between negative and positive exponents. The “- 1” in \(2^{16}/2 - 1\) arises from the fact that you need to reserve one exponent value, namely 0x0000, to be used in the special value representing the number 0.

3. The following questions refer to figure 5.19 in Patterson & Hennessy.
   a. Why is the value of the PC register sent to an adder?

   To increment the value of the PC by 4 to point to the next instruction.

   b. With regard to the mux appearing to the left of the register file, under what circumstances is its control line ("RegDst") set to 0?

   When the second register operand in an instruction is the target register for a register write. This occurs when an LW instruction is executed.

   c. Following the sign extend of bits 0-15 of an instruction, why is a "shift left 2" performed?

   Because the target address of a branch instruction is given as the offset in words, not bytes, from the value of PC+4. In order to obtain a byte address, the word offset is multiplied by 4 (i.e. shifted left by 2) before being added to the result of PC+4.

   d. Why are separate instruction and data memories required?
Because in the single-cycle implementation shown in figure 5-19, only a single access to memory is possible. Thus, if the memories were not separated, an instruction could be fetched or data could be read/written, but not both. The processor would not be able to execute a complete instruction in a single cycle in that case.

e. What is the purpose of the and-gate connected to the "zero" output line of the ALU?

It is used to set the appropriate value on the control line for the multiplexor that selects between PC+4 and the target address of a BEQ instruction. If the current instruction is a BEQ instruction, then the “branch” control line will be set to true. If the zero output of the ALU is also true, indicating that the two register operands are equal, then the and-gate will send 1 to the mux, selecting the target address of the branch instruction as the value to be written to the PC.

f. If, in a given cycle, the value of the "read register 1" line of the register file is the same as the value of the "write register" line, what is to prevent the value on the "write data" line from being propagated immediately to the "read data 1" output line?

The registers are constructed from flip-flops, thus their output only changes upon the falling edge of the clock. That is, a value be written to a register can only change upon the next falling edge, and thus cannot affect a value currently being read.

4. Figure 5.29 on page 372 of the text shows the simple, single-cycle datapath extended to implement a jump instruction. In order to accomplish this, another output line, the "jump" line, is added to the Control component. The opcode of a jump instruction is 2. Demonstrate how the Control should be extended by adding a new column to figure C.4 on page C-7 of the text, and by building a new Control out of gates, by extending figure C.5 on page C-8.

Since the opcode of the jump instruction is \( 2 = 000010 \), only the op1 input to the control is set to 1. When a jump instruction is being executed, the following control output lines in figure 5.29 are important:

- **jump**: must be set to 1 to select the target address for the next instruction
- **MemWrite**: must be set to 0 to ensure that memory isn’t modified
- **RegWrite**: must be set to 0 to ensure that a register isn’t modified

We don’t care about the status of the rest of the control lines, since the results computed by the ALU, etc. are not used. Thus, the column to be added to figure C.4 would look like:

<table>
<thead>
<tr>
<th>inputs</th>
<th>op5</th>
<th>op4</th>
<th>op3</th>
<th>op2</th>
<th>op1</th>
<th>op0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>outputs</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0</td>
</tr>
</tbody>
</table>
MemRead  x  
MemWrite  0  
Branch    x  
ALUOp1    x  
ALUOp0    x  
jump      1  (oops, new row needed as well)

The control logic in figure C.5 would be extended as follows:

• a new AND gate (with bubbles) would be added to the row of AND gates, which computes

\[ \overline{op5} \cdot \overline{op4} \cdot \overline{op3} \cdot \overline{op2} \cdot \overline{op1} \cdot \overline{op0} \]

• a new “jump” output line would be connected to the output of the new AND gate, above.

Notice that since the RegWrite and MemWrite lines are not connected to the new AND gate, and none of the other AND gates will produce an output of 1 when the opcode is 000010, the RegWrite and MemWrite lines will be set to 0 when a jump instruction is executed. This is the desired behavior.

5. What are the characteristics of the MIPS Instruction Set Architecture (ISA) that facilitate pipelined execution?

• Uniform instruction size, to facilitate a simple instruction fetch stage
• Register operands appear in that same fields of the instruction, over a wide range of instructions, facilitating a simple operand-fetch mechanism which can occur in the same stage as instruction decoding.
• Only simple operations, such as register-to-register ALU operations and simple load/store operations are supported, facilitating a simple execution stage.

6. The five stages of the simple MIPS pipeline we covered in class are: instruction fetch, instruction decode and register read, execute or calculate address, memory access, and register write. Describe the purpose of each of these stages.

• Instruction Fetch: Send the value of the PC register as an address to instruction memory and read the value of the instruction stored at that address.
• Instruction Decode and Register Read: Set the values of the control lines, based on the opcode and func fields of the instruction. Also, set the read select lines of the register file according to the operand fields of the instruction, and read the corresponding data on the output lines of the register file.
• Execute or calculate address: In the case of R-Type instructions, send the output of the register file to the ALU in order to calculate the desired arithmetic result. In the case of BEQ, send the target offset to the adder to calculate the new PC value of the next instruction. In the case of memory instructions (lw/sw), send the address field to the ALU to be added to the value of the base register being output from the register file.
• Memory Access: In the case of memory instructions (lw/sw), take the address computed by the ALU in the previous stage and use it to either read data from memory (in the case of an LW) or store the value of a register into memory.

• Register write: In the case of R-Type and LW instructions, the value computed by the ALU or fetched from memory is written back to the register file.

7. Suppose a new, more complicated, instruction is desired for this simple pipelined MIPS processor. Suppose, also, it could be implemented by either
   (a) adding new logic to the execute stage of the pipeline, or
   (b) adding a new stage (i.e., a 6th stage) altogether.

   Assume that choice (a) would slow the execute stage down by 20%. Since choice (b) increases the number of stages by 20%, is the overall effect on execution the same? If not, describe how each choice affects execution time and indicate which choice is better.

   The effect of the two choices on overall performance is extremely different. On a pipelined machine that can start a new instruction every cycle, the time taken by a program that executes \( I \) instructions is:

   \[ (I + (L-1)) \times C \]

   where \( L \) is the length (i.e., number of stages) of the pipeline and \( C \) is the cycle time.

   Generally, \( I \) will be much, much larger than \( L \) and will thus determine the overall execution time. In our simple MIPS processor, \( L \) is 5 so the execution time would be:

   \[ (I + 4) \times C \]

   If choice (a), above, is taken, then since in our simple pipelined MIPS processor, all stages of a pipeline take one cycle, the cycle time of the processor would have to be increased by 20%. Thus, the new time to execute the program would be

   \[ (I + 4) \times 1.2C = 1.2((I + 4) \times C) \]

   Thus, assuming the number of cycles required to execute a program doesn’t change, the program will run 20% slower. If choice (b) is taken, each instruction takes 6 cycles to complete instead of 5. However, a new instruction can still start executing every cycle and the cycle time hasn’t increased. Thus, the total number of cycles for the same program would be:

   \[ (I + 5) \times C = ((I + 4) \times C) + C \]

   an increase of only 1 cycle over the entire execution! (NOTE: An unrealistic aspect of this question is that it is generally very difficult to add a new complicated instruction by simply adding a stage in the pipeline without affecting the cycle time. That’s the motivation behind the RISC philosophy of keeping instructions simple).

8. Define the terms "structural hazard", "control hazard", and "data hazard" in the context of pipelines. Which of these hazards is addressed by a hardware branch predictor (which guesses whether a branch will be taken or not)? For one of the other hazards, suggest a way, either in software or hardware, the effect of that hazard could be reduced.
A structural hazard refers to the situation in which two different instructions currently occupying two different stages in a pipeline require the same resource (e.g. bus, memory, ALU). Thus, both stages cannot execute simultaneously and a bubble is introduced in the pipeline. A control hazard refers to the situation in which an instruction currently in the pipeline is a conditional branch instruction and thus it cannot be immediately determined which is the next instruction to feed into the pipeline. A data hazard refers to the situation in which the value of an operand register required by an instruction is still being computed by a previous instruction, and thus the pipeline must stall until the result is available in the register.

A branch predictor ameliorates the effect of a control hazard, by guessing which instruction will follow the branch instruction and feeding it into the pipeline. If the guesses are correct a high percentage of the time, there will be few stalls of the pipeline. A hardware solution to the structural hazard problem is to replicate resources that might be required in multiple stages. A hardware solution to the data hazard problem lies in “forwarding”, where a result computed in one stage of a pipeline can be passed directly to another stage of the pipeline without having to wait for a register to be written. A software solution to the data hazard problem is to have the compiler generate code in which instructions that depend on each other are not adjacent in the instruction stream.

9. Define the terms "spatial locality" and "temporal locality", and explain how caches are used to exploit them for a performance benefit. Be specific in the different ways that caches exploit these two phenomena.

Spatial locality refers to the property of programs that, once a datum or instruction is in memory is accessed, it is very likely that a neighboring datum or instruction will be accessed soon. Temporal locality refers to the property of programs that, once a datum or instruction in memory is accessed, it will likely be accessed again soon.

Caches exploit spatial locality by storing blocks (called “cache blocks” or “cache lines”) of instructions or data that are contiguous in memory. Thus, once a block has been brought into the cache due to a cache miss on a particular address, it is likely that, upon an access to a neighboring address, the desired data will be in the cache. Caches exploit temporal locality by storing recently accessed data or instructions. Thus, when the same data or instruction is desired again shortly thereafter, it is found in the cache.

10. Suppose physical addresses are 32 bits wide. Suppose there is a cache containing 256K words of data (not including tag bits), and each cache block contains 4 words. For each of the following cache configurations,
   a. direct mapped
   b. 2-way set associative
   c. 4-way set associative
   d. fully associative

specify how the 32-bit address would be partitioned. For example, for a direct mapped cache, you would need to specify which bits are used to select the cache entry and which bits are used to compare against the tag stored in the cache entry.
Since there are 4 bytes/word, the lowest two bits are used to select the byte offset. Since there are 4 words per cache block, the next two lowest bits are used to for the word offset in the the block. The partition of the rest of the 32-bit address then depends on the associativity of the cache, as follows:

- **direct mapped:** Since there are 4 words per block and 256K words in the cache, there are \( \frac{256K}{4} = 64K \) blocks in the cache. Thus, \( \log(64K) = 16 \) bits of the address are needed to specify the cache block. The remaining bits are used for the tag. The partition would be:
  
  bits 0-1: byte offset  
  2-3: word offset  
  4-19: cache block index  
  20-31: tag

- **2-way set associative:** Since there two blocks per set, and 64K blocks in the cache (see above), there must be 32K sets. Thus, the number of bits required to specify a set is \( \log(32K) = 15 \). The remaining bits are used for the tag, as follows:
  
  bits 0-1: byte offset  
  2-3: word offset  
  4-18: set index  
  19-31: tag

- **4-way set associative:** Since there four blocks per set, and 64K blocks in the cache (see above), there must be 16K sets. Thus, the number of bits required to specify a set is \( \log(32K) = 14 \). The remaining bits are used for the tag, as follows:
  
  bits 0-1: byte offset  
  2-3: word offset  
  4-17: set index  
  18-31: tag

- **fully associative:** Since a block can be placed anywhere in the cache, all the remaining bits (other than the word and byte offsets) must be used as the tag. Hence,
  
  bits 0-1: byte offset  
  2-3: word offset  
  4-31: set index
11. Draw the implementation of the 2-way set associative version of the above cache, at the level of detail shown in figure 7.19 on page 574 of the text. Be sure to include the logic for selecting the desired word of the cache block.
12. Cache misses can be characterized as one of the following: compulsory misses, capacity misses, and conflict misses. Describe how each of these kinds of misses can be addressed in the hardware.

**Compulsory misses**, i.e. cache misses that occur the first time a datum or instruction is accessed, can be often reduced by increasing the size of the cache block. Since increasing the size of the cache block causes more neighboring locations to be cached when a cache miss occurs, it increases the likelihood that a datum that has not yet been accessed will be in the cache later on when it is accessed.

**Capacity misses**, i.e. cache misses that occur because the cache isn’t large enough to hold all the blocks that a program is currently accessing, can be reduced by increasing the size of the cache.

**Conflict misses**, i.e. cache misses that occur because two or more blocks currently in use map to the same location in the cache, can be reduced by increasing the associativity of the cache. The greater the associativity, the larger the set size. In this way, two blocks that map to the same set can be accommodated within the set.

13. Suppose you own a computer that exhibits the following properties on the programs that you run:
   - the pipeline can accept a new instruction every cycle
   - the cache can provide data every cycle (i.e. no penalty for cache hits)
   - the instruction cache miss rate is 2.5%
   - the data cache miss rate is 3.5%
   - 30% of instructions are memory instructions
   - the cache miss penalty is 80 cycles.

Suppose, also, that you have decided to purchase a new computer. For the budget allocated, you can either
   - purchase a machine with a processor and cache that is twice as fast as your current one (memory speed is the same as the old machine, though), or
   - purchase a machine with a processor and cache that is the same speed as your old machine but in which the cache is twice as large.

Assume, for the purposes of this problem, that the cache miss rate for the programs you run will drop by 40% with this larger cache (although this is generally not true in the real world). Which computer are you best off purchasing? Explain in detail, showing the relative performance of each choice.

When running a program that executes \( I \) instructions, the total number of cycles executed by a processor will be:
\[
I + (I \times \text{instruction-miss-rate} \times \text{miss-penalty}) + \\
(I \times \text{fraction-of-memory-instructions} \times \text{data-miss-rate} \times \text{miss-penalty})
\]

(note: this ignores the small constant \(L\), the length of the pipeline, discussed in question 7).

Thus, the number of cycles required to execute the program on the old machine is:

\[
I + (I \times 0.025 \times 80) + (I \times 0.3 \times 0.035 \times 80) = I + 2I + .84I = 3.84I
\]

If you choose to buy the machine with the processor that is twice as fast, the miss penalty doubles to 160 cycles (because memory hasn’t gotten faster). Thus, the number of cycles required to execute the program is:

\[
I + (I \times 0.025 \times 160) + (I \times 0.3 \times 0.035 \times 160) = I + 4I + 1.68I = 6.68I
\]

However, since the new processor is twice as fast, in the time that the new machine takes to execute 5.68I cycles, the old machine would only have executed \(6.68I/2 = 3.34I\) cycles. Thus, the speedup of the machine with the fast processor over the old machine is \(3.84I/3.34I = 1.15\), only a 15% increase in performance.

If you choose to buy the new machine with the bigger cache, the miss penalty is the same as the old machine, but the miss rates fall by 40%. Thus, the instruction miss rate will be \(2.5\% \times 60\% = 1.5\%\) and the data miss rate will be \(3.5\% \times 60\% = 2.1\%\). The number of cycles required to execute the program will be

\[
I + (I \times 0.015 \times 80) + (I \times 0.3 \times 0.021 \times 80) = I + 1.2I + .50I = 2.70I
\]

This gives a speedup over the old machine of \(3.84I/2.70I = 1.42\), a 42% increase in performance. In addition, this new machine outperforms the new machine with the faster processor by a factor of \(3.34I/2.70I = 1.24\). In other words, since the new machine with a bigger cache outperforms the new machine with the fast processor by 24%, that’s the one you should buy.

14. Describe the number of bits required in each entry of a TLB that has the following characteristics:

- Virtual addresses are 32 bits wide
- Physical addresses are 31 bits wide
- The page size is 2K bytes
- The TLB contains 16 entries of the page table
- The TLB is direct-mapped

Each entry of the TLB contains the following items:

- a valid bit
- the physical page number of the desired virtual page
- a tag used to see if the desired entry of the page table is stored in the TLB

(note: we’ll ignore the “dirty” bit, discussed in the text but not in class). Since physical
addresses are 31 bits and each page is 2K = 2^{11} bytes, there are 2^{31}/2^{11} = 2^{20} physical pages. Thus, since 20 bits are required to distinguish among 2^{20} physical pages, the size of the physical page number in the page table (and TLB) is 20 bits.

In order to determine how many bits are required for the tag, we first need to consider how the 32-bit virtual address is partitioned. Since the virtual address space is 2^{32} bits and the page size is 2^{11} bits, the number of virtual pages is 2^{32}/2^{11} = 2^{21}. Thus, 21 bits are required to represent a virtual page number. Since the TLB contains 16 entries, log(16) = 4 bits of the virtual page number are used to select the entry in the TLB. The rest of the bits of the virtual page number, namely 21 - 4 = 17 bits, are used as a tag. Therefore, the total number of bits in a TLB entry is: 1 + 17 + 20 = 38 bits.