Question 1 {10 pts}

Let's say that you are an engineer at LEG microsystems and you currently have a pipelined processor design with a cycle time of 10 ns. You know that your favorite benchmark program exhibits an average CPI of 1.6 on this processor. You also know that 10% of the instructions in this workload are branches and that your branch prediction scheme is 90% accurate. Every branch misprediction costs you 2 cycles of delay on this processor. You are considering a new processor design where you decrease the cycle time to 9 ns by increasing the depth of the pipeline. In this new design the cost of a misprediction will increase to 7 clock cycles but everything else will stay the same. Compute the average CPI on the new processor for the benchmark. Will your benchmark program run faster or slower on this new processor? Show your work.

Answer:

In the new processor the misprediction penalty increases by \((7 - 2) = 5\) cycles for 10% of the branches. Hence new CPI = \(1.6 + 0.1 \times 0.1 \times 5 = 1.65\)

Time taken on workload = Number of Instructions \(N\) * CPI * cycle time

Time taken on old machine = \(N \times 1.6 \times 10 = 16N\)
Time taken on new machine = \(N \times 1.65 \times 9 = (1.65 \times 9)n\)

Note \(16 - (1.65 \times 9) = 1.6 - 0.45 = 1.15\) so the old machine will take longer than the new machine – the new machine is faster.

Question 2 {10 pts}

In class we discussed the design of a Carry Save Adder which is sometimes used in the context of fast multiplier designs. The basic CSA block takes as input 3 n-bit inputs and produces as a result 2 n-bit outputs. Your job is to complete the following Verilog module which implements the desired functionality.

module CSA (A, B, C, S, CO);
parameter n=8;
input [n-1:0] A, B, C;
output [n-1:0] S, CO;

assign CO = (A&B) | (B&C) | (A&C);       // Carry Out outputs
assign S = (A ^ B) ^ C;                  // Sum outputs

endmodule
Consider the standard 5 stage pipeline discussed in class and in the text: Fetch, Decode, eXecute, Memory and Write back.

Now imagine that we disable forwarding from the MEM/WB register to the Execute stage and we change the stall logic to ensure that all instructions get executed correctly. Note we retain the assumption that the register file is written during the first phase of the clock and read on the second phase so reading and writing a register in a single clock cycle is O.K.

Fill in the pipeline table below, which shows the progress of the following instructions through the processor over the clock cycles. You must use *s to denote cycles where an instruction is stalled in a pipeline phase.

Remember in MIPS the left most register in the instruction is the one that is being written if any.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $3,$2,$1</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub $4,$3,$2</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add $5,$3,$4</td>
<td>F</td>
<td>D</td>
<td>*</td>
<td>*</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $6,8(5)$</td>
<td>F</td>
<td>*</td>
<td>*</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw $7,3(6)$</td>
<td>F</td>
<td>D</td>
<td>*</td>
<td>*</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or $5,$6,$3</td>
<td>F</td>
<td>*</td>
<td>*</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Question 4 {10 pts}

The original version of the MIPS ISA sported a feature known as a delayed branch which meant that the instruction immediately after a conditional branch instruction like beq was always executed regardless of whether the branch was taken or not.

1) Explain why the designers of the MIPS ISA thought that this was a good idea.
2) Explain why this delayed branch feature isn’t used much in more modern processor designs and ISAs.

Answer:
The branch delay slot was a good idea in the context of a 5 stage pipeline. The designers noticed that simple branches like beq could be resolved in the decode phase of the pipeline and doing this reduced the penalty for branch misprediction to a single cycle. Meaning that you would only have to flush the instruction in the fetch stage. The designers then went one step further and said that instead of flushing that next instruction they would just go ahead and execute it to avoid the stall penalty entirely. The hope was that smart compilers would be able to schedule a useful instruction in that delay slot that did not depend upon the branch outcome so you could still do useful work.

As pipelines got longer to further increase clock speeds it became impractical to resolve branches by the 2nd stage of the pipeline. This meant that to carry the idea forward you would need to introduce even more delay slots into the ISA. This was very unattractive since it was hard enough for compilers to find useful work to fill a single delay slot – filling 4 or 5 slots was considered a problem. Attention turned to other methods of reducing control hazards like more accurate branch prediction and speculative execution. These solutions required more sophisticated hardware but they made life easier on the programmer and they were ultimately more compatible with deeper pipelines.
Question 5 {10 pts}

The following table shows the state of a Correlated Predictor associated with a 3 bit Branch History Register, BHR. We show the state of the predictor during the first branch that the processor encounters and show the sequence of branch outcomes that the processor encounters. Your job is to fill in all the missing entries in this table. You should assume that new bits are shifted into the right most bit of the BHR.

<table>
<thead>
<tr>
<th>BHR</th>
<th>NNN</th>
<th>NNT</th>
<th>NTN</th>
<th>NTT</th>
<th>TNN</th>
<th>TNT</th>
<th>TTN</th>
<th>TTT</th>
<th>Prediction</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>NNN</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>T</td>
</tr>
<tr>
<td>NNT</td>
<td>T</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>NTT</td>
<td>T</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>N</td>
<td>T</td>
</tr>
<tr>
<td>TTT</td>
<td>T</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>N</td>
<td>T</td>
<td>T</td>
<td>N</td>
<td>T</td>
</tr>
<tr>
<td>TTN</td>
<td>T</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>NNT</td>
<td>T</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>T</td>
<td>N</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
</tr>
<tr>
<td>NTT</td>
<td>T</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>T</td>
</tr>
<tr>
<td>TNN</td>
<td>T</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

Question 6 {10 pts}

In this question we want you to chart how the state of a saturating 2 bit counter will change as it encounters a series of branch outcomes by filling in the table below. All of the branch outcomes come from the same branch. Use N to denote the hard no state, n to denote soft no, t to denote yes and T to denote hard yes.

<table>
<thead>
<tr>
<th>Outcome</th>
<th>T</th>
<th>T</th>
<th>T</th>
<th>N</th>
<th>N</th>
<th>T</th>
<th>N</th>
<th>N</th>
<th>T</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>N</td>
<td>n</td>
<td>t</td>
<td>T</td>
<td>t</td>
<td>n</td>
<td>t</td>
<td>n</td>
<td>N</td>
<td>n</td>
</tr>
</tbody>
</table>

Note that the state refers to the state of the 2bC before it sees the outcome.

Question 7 {10 pts}

Your cousin, Crazy Eddie, read on the Internet that floating point operations were fraught with problems. He was especially shocked to hear that on a computer \(((1+x) - 1)\) might not be equal to \(x\) if \(x\) is a really small value. Now he is fundamentally distrustful of all floating point math and he worries about what would happen if he multiplied two small numbers. Is Crazy Eddie right to be worried about multiplication as well? Could he get into a situation where \((x*y)/x\) is very different from \(y\)? Explain your answer briefly.

Answer:
Crazy Eddie is in general correct to be concerned about numerical precision but the situation for multiplication is not quite as bad as the situation for addition. In addition bad things often happen when two incommensurate quantities are added because of the nature of floating point addition and the fact that one of the numbers will get shifted with respect to the other. In multiplication we are multiplying two numbers \((a \times 2^m) \times (b \times 2^n)\) where \(a\) and \(b\) are normalized to values between 1 and 2, ignoring sign which is handled trivially. The result of the multiplication is \((a \times b) \times (2^{(m+n)})\) so we only need to multiply the significands and add the exponents. Since the two mantissas \(a\) and \(b\) are normalized this multiplication is usually pretty accurate so in most cases \((x \times y) / x\) is very close to \(y\) even if the numbers have very different magnitudes. In some cases you may end up with a situation where the value of \((m+n)\) overflows the values that can be stored in the exponent field. In this case you would have a problem but this is much less likely in practice.

**Question 8** {10 pts}

Explain briefly what the RAS (Return Address Stack) is used for in the context of branch prediction. Similarly explain the role of the Branch Target Buffer (BTB).

The RAS or return address stack is used to keep track of return addresses that are pushed on the stack during subroutine calls. This cache provides a way for the branch prediction unit to correctly deduce the next instruction when the processor returns from a subroutine. Typically these addresses are not predictable since a subroutine could be called from multiple locations in the code but this mechanism provides a mechanism to help the processor correctly guess where the return will go.

The branch target buffer is used for more traditional PC relative branching instructions. It is the place where the processor stores the destination associated with the branches it encounters in the program. This cache stores a mapping between the PCs of branch instructions and the program location that they branch to. This information helps the processor to fetch the next predicted instruction.

**Question 9** {15 pts}

Your cousin, Crazy Eddie, has read a lot of C code and noticed that many experienced programmers are fond of ternary assignment statements like \(A = B ? C : D;\). In order to better support these and other constructs Eddie proposes adding a new instruction to the MIPS ISA with the following semantics.

\[
\text{MOVC} \$1, \$2, \$3 : \text{if } \$3 \neq 0, \$1 = \$2 \text{ otherwise } \$1 \text{ is unchanged.}
\]

Eddie makes the following arguments, 1) This instruction will allow us to code ternary operations in fewer instructions 2) This instruction is actually pretty easy to add to our standard 5 stage pipeline (see figure from Question 3) 3) This instruction
improves life from the perspective of control hazards. Do you agree with Eddie’s arguments? give your reasons for each case. Make sure your answer addresses how you would need to change the datapath and control of the processor and what impacts this would have on data and control hazards.

This is a very useful instruction for implementing ternary instructions. Consider the statement

A = B ? C : D; where A,B,C and D are all integer values for simplicity.

Let us imagine that they are assigned to registers as follows $1 = A, $2 = B, $3 = C, $4 = D.

Without the MOVC instruction we need to implement this using some form of branching. Logically

A = D;
If (B != 0) { A = C; }

We can implement this as follows – taking advantage of the delayed slot semantics

    beq $0, $2, tag
    addi $1, $4, 0 ; A = D – this will always happen since its in the delay slot
    addi $1, $3, 0 ; A = C – this will get skipped if B == 0
Tag:  rest of the program

With the new instruction we could code up the ternary assignment as follows

addi $1, $3, 0 ; $1 = $3, A = C
movc $1, $4, $2 ; $1 = $4 if (B == 0)

Two instructions is pretty compact since we need at least three for the earlier solution and one of those is a branch which we want to avoid if possible.

Implementing this instruction in the 5 stage pipeline is not that hard (MOV $1, $2, $3). Basically what we need to do is check the value of the determining register, in this case $3, in the execute stage. If the value is non-zero the instruction proceeds through just like an arithmetic instruction with the destination register set ($1) and the value field set to ($2). If it is zero then the instruction is flushed by zeroing the control signals associated with this instruction for subsequent pipeline stages and basically inserting a nop in the pipe.

We would need all of the usual forwarding between the Memory and Writeback stages and the execute stage and we would also need to forward the results from
this instruction back to subsequent instructions if the system decides that the
destination register is in fact going to be written.

The instruction does have a positive impact on control hazards in the following
sense. Without it we would have to code this instruction up using beq or some other
conditional branch. With every conditional branch there is the possibility of
misprediction which would introduce stalls and flushes into the pipeline. This
instruction is deterministic in the sense that we do not need to insert any stalls into
the pipeline regardless of whether the move was made or not.