Quiz #2

<table>
<thead>
<tr>
<th>Name</th>
<th>Solutions</th>
<th>Athena login name</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recitation section</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>□ WF 10, 34-302 (Francis)</td>
<td>□ WF 2, 34-302 (Robert)</td>
<td>□ WF 12, 35-310 (Kendall)</td>
<td></td>
</tr>
<tr>
<td>□ WF 11, 34-302 (Francis)</td>
<td>□ WF 3, 34-302 (Robert)</td>
<td>□ WF 1, 35-310 (Kendall)</td>
<td></td>
</tr>
<tr>
<td>□ WF 12, 34-302 (Grace)</td>
<td>□ WF 10, 35-310 (Sara)</td>
<td>□ opt-out</td>
<td></td>
</tr>
<tr>
<td>□ WF 1, 34-302 (Grace)</td>
<td>□ WF 11, 35-310 (Sara)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please enter your name, Athena login name, and recitation section above. Enter your answers in the spaces provided below. Show your work for potential partial credit. You can use the extra white space and the backs of the pages for scratch work.
Problem 1. Sequential Circuits in Minispec (27 points)

Thanks to your mastery of 6.004, you have landed an internship at Cyberdyne Systems. Cyberdyne is revolutionizing artificial intelligence by building hardware to process binarized neural networks: networks where inputs and outputs are single-bit values.

Your task is to design a binarized perceptron, the circuit at the heart of this computation. This circuit implements the following function:

\[
p(\vec{a}, \vec{w}) = \begin{cases} 
+1 & \text{if } \vec{a} \cdot \vec{w} = \sum_{i=0}^{n-1} a_i w_i \geq 0 \\
-1 & \text{otherwise}
\end{cases}
\]

where \( \vec{a} \) and \( \vec{w} \) are \( n \)-element vectors where each element can be \(-1\) or \(+1\). In other words:

\[
\vec{a} = (a_0, a_1, ..., a_{n-1}) \text{ where } a_i \in \{-1, +1\}
\]

\[
\vec{w} = (w_0, w_1, ..., w_{n-1}) \text{ where } w_i \in \{-1, +1\}
\]

The \( a_i \) and \( w_i \) inputs, as well as the output, can only be \(-1\) or \(+1\), so we encode each value with a single bit. Specifically, we use binary 0 to encode value \(-1\) and binary 1 to encode \(+1\).

We can implement this circuit by (1) performing the individual one-bit multiplications \( a_i w_i \), (2) summing them up, and (3) checking whether the sum is 0 or higher. We will implement combinational and sequential versions of these circuits, and analyze their tradeoffs.

(A) (2 points) Fill in the truth table for multiplication of \( a_i w_i \) below. Note that a binary 0 encodes value \(-1\), so this is not quite like standard binary multiplication.

<table>
<thead>
<tr>
<th>( a_i )</th>
<th>( w_i )</th>
<th>( a_i w_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (-1)</td>
<td>0 (-1)</td>
<td>1 (+1)</td>
</tr>
<tr>
<td>0 (-1)</td>
<td>1 (+1)</td>
<td>0 (-1)</td>
</tr>
<tr>
<td>1 (+1)</td>
<td>0 (-1)</td>
<td>0 (-1)</td>
</tr>
<tr>
<td>1 (+1)</td>
<td>1 (+1)</td>
<td>1 (+1)</td>
</tr>
</tbody>
</table>

(B) (2 points) Implement the parametric function \( \text{binMul} \) in Minispec. \( \text{binMul} \) takes \( n \)-bit inputs \( a \) and \( w \) that encode \( \vec{a} \) and \( \vec{w} \), and returns an \( n \)-bit output that encodes their individual multiplications (essentially implementing \( n \) parallel copies of the truth table above).

\[
\text{function Bit#}(n) \text{ binMul#}(\text{Integer } n)(\text{Bit#}(n) a, \text{Bit#}(n) w);
\]

\[
\text{return } a \land \neg w; \quad // \text{or } \neg(a \land w)
\]

endfunction
Summing up the individual $a_i w_i$ values and checking that the total is non-negative is equivalent to counting the number of +1s and checking whether the total is greater than or equal to $n/2$.

(C) (3 points) Complete the implementation of the parametric `countOnes` function, which takes an $n$-bit input and returns the count of ones in that input (e.g., `countOnes#(8)(8'b01010001) = 3'd3`). You can use any Minispec operators. **For each function, make sure the return value matches the expected bit width.**

```verbatim
// Base case
function Bit#(1) countOnes#(1)(Bit#(1) x);
    return _____________x__________________ ;
endfunction

// General case, assume n is a power of 2
function Bit#(log2(n)+1) countOnes#(Integer n)(Bit#(n) x);
    Bit#(log2(n)) lowerOnes = countOnes#(n/2)(x[n/2-1:0]);
    Bit#(log2(n)) upperOnes = countOnes#(n/2)(x[n-1:n/2]);
    return ________{1'b0, lowerOnes} + {1'b0, upperOnes}_______ ;
endfunction
```

(D) (2 points) Complete the implementation of the parametric perceptron function below, which completes your combinational design.

```verbatim
function Bit#(1) perceptron#(Integer n)(Bit#(n) a, Bit#(n) w);
    Bit#(n) x = binMul(a, w);
    Bit#(log2(n)+1) ones = countOnes(x);
    return ___ ones >= n/2 ? 1 : 0_____ ;
    // or ones[log2(n)] | ones[log2(n)-1]
endfunction
```
(E) (7 points) Complete the multi-cycle sequential perceptron below, implemented as a Minispec module. This sequential circuit receives a single input \((a_i, w_i)\) each cycle, and keeps a running count of the number of \(a_iw_i\) products that have been 1 so far. The cycle after \(n\) inputs have been received, the module should produce a single-bit output with the result.

Inputs are fed through the \(\text{in}\) input, and the \(\text{getResult}\) method produces the result of the computation. A new input may not be given every cycle, so \(\text{in}\) is a Maybe type. Similarly, \(\text{getResult}\) returns a Maybe type, which should be valid only when \(n\) inputs have been received. If a new input arrives after having completed a computation, the module should begin a new computation, and \(\text{getResult}\) should return Invalid until a new result is computed.

```plaintext
typedef struct {Bit#(1) ai, Bit#(1) wi} PerceptronArgs;

module MultiCyclePerceptron#(Integer n);
    Reg#(Bit#(log2(n)+1)) i(0);
    Reg#(Bit#(log2(n)+1)) ones(0);

    input Maybe#(PerceptronArgs) in default = Invalid;

    rule tick;
        if (isValid(in)) begin
            let args = fromMaybe(?, in);
            Bit#(1) x = binMul#(1)(args.ai, args.wi);
            if (i < n) begin
                i <= ________ i + 1__________________________ ;
                ones <= ______ones + zeroExtend(x)___________ ;
            end
            else
                i <= _______ 1_______________________________ ;
                ones <= ______ zeroExtend(x)__________________ ;
        end
    endrule

    method Maybe#(Bit#(1)) getResult;
        return ____ (i == n)? Valid(ones >= n/2? 1 : 0) : Invalid _____ ;
    endmethod
endmodule
```
(F) (5 points) Analyze the critical-path delay, throughput, and area of your combinational and multi-cycle sequential implementations. Complete the table below showing the asymptotic behavior of each metric as a function of $n$, using order-of notation. To do this, the following table shows how delay and area scale with $n$ for different types of Minispec operators. Show your work for partial credit.

**Hint:** You may find the following relationship useful in your analysis:
$$\sum_{i=0}^{k} \frac{1}{2^k} = 1 + \frac{1}{2} + \frac{1}{4} + \cdots < 2.$$ Therefore, $n + \frac{n}{2} + \frac{n}{4} + \cdots < 2n$.

<table>
<thead>
<tr>
<th>Operator type</th>
<th>Critical-path delay ($t_{PD}$)</th>
<th>Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitwise logical (e.g., and, or, xor, negation)</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Arithmetic (e.g., addition, comparisons, shifts)</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Combinational perceptron</th>
<th>Critical-path delay ($t_{PD}$)</th>
<th>Throughput</th>
<th>Area</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\Theta(\frac{n}{n})$</td>
<td>$\Theta(\frac{1}{n})$</td>
<td>$\Theta(n \log n)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Multi-cycle perceptron</th>
<th>Critical-path delay ($t_{PD}$)</th>
<th>Throughput</th>
<th>Area</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\Theta(\frac{n}{n^2})$</td>
<td>$\Theta(\frac{1}{n^2})$</td>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>

**Combinational perceptron:**

delay = binMul delay + countOnes delay + comparison with n/2 delay
binMul - $\Theta(1)$
countOnes – addition is $\Theta(n)$ but each recursive call creates an adder that is half as long so delay is: $n + \frac{n}{2} + \frac{n}{4} + \cdots < 2n$
comparison - $\Theta(n)$

So critical path delay is $\Theta(n)$.
This means that the throughput is $\Theta(1/n)$.

area = binMul area + countOnes area + comparison with n/2 area
binMul - $\Theta(n)$
countOnes – even though each recursive call creates an adder that is half as large, it creates two such adders (one for lower and another for upper) so the total area is $n \log n$.
comparison - $\Theta(n)$

So total area dominated by $n \log n$ term - $\Theta(n \log n)$

**Multi-Cycle perceptron:**

cycle delay = 1 binMul delay + 1 countOnes delay
binMul - $\Theta(1)$
countOnes – addition is $\Theta(n)$

so cycle delay is $\Theta(n)$
A new output is produced every n cycles, so total latency is $\Theta(n^2)$, which means that the throughput is $\Theta(1/n^2)$.

$$ \text{area} = 1 \text{ binMul area} + 1 \text{ countOnes area} + \text{ comparison with } n/2 \text{ area} = \Theta(n) + \Theta(n) + \Theta(n) = \Theta(n) $$

(G) (2 points) In previous exercises, you probably used a comparator to check whether the count of ones is n/2 or larger. However, this adds $\Theta(n)$ delay. **Assume that n is a power of 2.** To improve performance, we want to perform this comparison with n/2 in $\Theta(1)$ delay. Complete the function below, which should return 1 if the count of ones is greater than or equal to n/2. Its delay should be $\Theta(1)$.

```
function Bit#(1) fastComparator#(Integer n)(Bit#(log2(n)+1) ones);

    return _____ ones[log2(n)] | ones[log2(n)-1]_________ ;

endfunction
```

If ones $\geq$ n/2, then either MSB of count OR bit just before MSB must be a 1. Log2(n) gives you MSB of ones.
(H) (4 points) You are tasked with building Skynet, Cyberdyne’s upcoming neural accelerator. Skynet must achieve high throughput on very large neural networks (with $n$ reaching millions of bits). Thus, you set out to build a pipelined perceptron that achieves $\Theta(1)$ throughput.

To ease your task, you are given a parametric PipelinedCountOnes module that implements a pipeline for the countOnes operation with $\Theta(1)$ throughput (i.e., constant critical-path delay regardless of $n$). In addition, you may assume that you have a working $\Theta(1)$ version of the fastComparator function. Use this module and this function to build your pipelined perceptron module.

Your pipeline should implement valid bits (similar to e.g. the sorting pipeline from lab 5). Follow the usual convention that pipelines must have a register at the output. You don’t know the implementation details of PipelinedCountOnes, other than it uses valid bits and that it has a register at its output.

```verilog
module PipelinedCountOnes#(Integer n);
    input Maybe#(Bit#(n)) in default = Invalid;
    method Maybe#(Bit#(log2(n)+1)) getResult;
    ...
endmodule

typedef struct {Bit#(n) a, Bit#(n) w} PerceptronArgs#(Integer n);

module PipelinedPerceptron#(Integer n);
    PipelinedCountOnes#(n) countPipe;
    Reg#(Maybe#(Bit#(1))) outputReg(Invalid);
    input Maybe#(PerceptronArgs#(n)) in default = Invalid;
    rule tick;
        if (isValid(in)) begin
            let args = fromMaybe(?, in);
            Bit#(n) x = binMul#(n)(args.a, args.w);
            countPipe.in = ______ Valid(x)______________________ ;
        end else begin
            countPipe.in = ______ Invalid_______________________ ;
        end
        if (isValid(countPipe.getResult)) begin
            Bit#(log2(n)+1) ones = fromMaybe(?, countPipe.getResult);
            outputReg <= ______ Valid(fastComparator#(n)(ones))_______ ;
        end else begin
            outputReg <= ______ Invalid___________________________ ;
        end
    endrule
    method Maybe#(Bit#(1)) getResult = outputReg ;
endmodule
```
Problem 2. Arithmetic Pipelines (15 points)

You are an intern at Zoo Company, maker of the FROG circuit, depicted on the right. The circuit has two inputs, F and R, and two outputs, O and G. You are told that the circuit functions, but its throughput is too low. You decide to take a look and try to pipeline the circuit.

For each of the questions below, please create a valid K-stage pipeline of the given circuit. Each component in the circuit is annotated with its propagation delay in nanoseconds. Show your pipelining contours and place large black circles (●) on the signal arrows to indicate the placement of pipeline registers. Give the latency and throughput of each design, assuming ideal registers (t_{PD}=0, t_{SETUP}=0). Note that an invalid pipeline will receive 0 points.

(A) (4 points) Show the maximum-throughput 2-stage pipeline using a minimal number of registers. What is the latency and throughput of the resulting circuit? Pay close attention to the direction of each arrow.

Latency (ns): 22

Throughput (ns⁻¹): 1/11
(B) (4 points) Show the **maximum-throughput 3-stage pipeline** using a minimal number of registers. What is the latency and throughput of the resulting circuit?

Latency (ns): **30**

Throughput (ns⁻¹): **1/10**
(C) (4 points) Show the maximum-throughput pipeline using a minimal number of registers. What is the latency and throughput of the resulting circuit? For full credit your pipelined circuit should use the smallest number of pipeline stages required to achieve maximum throughput.

Latency (ns): 28

Throughput (ns⁻¹): 1/7
(D) (3 points) The Zoo Company has now tasked you with adding a new module, the PANDA, to their mammal circuit. The mammal circuit currently contains multiple pipelined modules: ZEBRA, SLOTH, and GIRAFFE. The maximum throughput and number of registers used for each module, when used on its own, is listed below. Assume that the output of ZEBRA is an input to SLOTH and that the output of SLOTH is an input to GIRAFFE. You now want to take the output of GIRAFFE and feed it into a new module, the PANDA. You are choosing between two different pipelined versions of the PANDA, which differ in maximum throughput and number of registers used. The Zoo Company wants to maximize throughput of the mammal circuit while minimizing area (i.e. number of registers). Which new module is more appropriate to add in order to meet these design goals? Explain your answer.

Modules currently in the mammal circuit:

<table>
<thead>
<tr>
<th>Module</th>
<th>Throughput (ns⁻¹)</th>
<th>Number of Registers Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZEBRA</td>
<td>1/4</td>
<td>11</td>
</tr>
<tr>
<td>SLOTH</td>
<td>1/9</td>
<td>4</td>
</tr>
<tr>
<td>GIRAFFE</td>
<td>1/5</td>
<td>8</td>
</tr>
</tbody>
</table>

Choices for the PANDA module:

<table>
<thead>
<tr>
<th>Version</th>
<th>Throughput (ns⁻¹)</th>
<th>Number of Registers Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red PANDA</td>
<td>1/8</td>
<td>7</td>
</tr>
<tr>
<td>Giant PANDA</td>
<td>1/6</td>
<td>10</td>
</tr>
</tbody>
</table>

Version (circle one): Red PANDA

Explanation:

The throughput of the original mammal circuit is 1/9 ns⁻¹, constrained by the SLOTH module, meaning the fastest clock period we can run this circuit at is \( t_{CLK} = 9 \) ns. Both options for the PANDA module have throughput greater than 1/9 ns⁻¹, meaning they are both able to run at \( t_{CLK} = 9 \) ns. Since the PANDA module is not the limiting factor in the mammal circuit’s throughput, the resulting throughput will be the same regardless of which version we choose. Thus, we want to choose the version with the smaller area, which is the Red PANDA, since it uses a smaller number of registers (and registers take up space).
Problem 3. RISC-V Processor (17 points)

Mel Rose is writing a RISC-V assembly program to estimate whether the world has more wheels or more doors. In order to calculate the estimated number of wheels and doors in the world, this program includes counters that increment when some condition is met. To optimize his program, Mel wants to add the following conditional increment instruction to the RISC-V ISA:

\[ \text{cinc } rd, rs1, rs2 \]

The behavior of the cinc instruction is as follows.

\[
\text{if } (\text{reg}[rs1] == \text{reg}[rs2]) \quad \text{reg}[rd] \leftarrow \text{reg}[rd] + 1
\]

(A) (3 points) Mel hopes to be able to simply add new signals as inputs to the selection multiplexers in the existing RISC-V processor. Can the cinc instruction be added with the existing processor components as described in lecture and shown below? If not, describe all the limitations that prevent it.

Can cinc be implemented with existing components? **YES NO**

Explanation of all limitations if it can’t be implemented:

1. Reading from 3 registers while register file has 2 read ports
2. ALU can’t be used for comparison and increment at the same time.
3. According to the diagram, the ALU A input comes from rs1, but this instruction requires for rs1 and rd as first source operand.
Along with the rest of the world, Mel has moved on from the whole wheels versus doors debate and is now writing a different RISC-V program. He’s finding that he often needs to modify an individual bit in a given register, while keeping every other bit in the register the same as it was before. He would like to simplify his program by implementing the two following instructions in the RISC-V ISA.

Set bit: `setb rd, imm`
\[
\text{reg}[rd] \leftarrow \text{reg}[rd] \mid (1 \ll imm)
\]
- `setb x5, 9` would set the 9th bit in `x5` to be 1, while keeping every other bit in `x5` the same as it was originally.

Clear bit: `clrb rd, imm`
\[
\text{reg}[rd] \leftarrow \text{reg}[rd] \& \sim(1 \ll imm)
\]
- `clrb x5, 6` would set the 6th bit in `x5` to 0, while keeping every other bit in `x5` the same as it was originally. *Note: The ~ (bitwise NOT) operation can be carried out by XORing with 0xFFFFFFFF.*

To assist him in performing these operations, Mel has added a shifter to the processor circuit as shown in the diagram. It takes in a single input (in this case `imm`) and outputs 1 left-shifted by that input value.
The encoding of \texttt{setb} is as follows.

\begin{tabular}{|c|c|c|c|c|c|}
\hline
\hline
000000 & 0000 & imm[4:0] & 000 & rd & 0101011 \\
\hline
\end{tabular}

The encoding of \texttt{clrb} is as follows:

\begin{tabular}{|c|c|c|c|c|c|}
\hline
\hline
000000 & 0000 & imm[4:0] & 001 & rd & 0101011 \\
\hline
\end{tabular}

Mel plans to support \texttt{setb} and \texttt{clrb} by only making changes to the decoder, leaving the rest of the hardware as it appears in the diagram above. Due to company-imposed area constraints, Mel cannot add any more logic gates to the processor circuit.

(B) (10 points) Given that \texttt{setb} can be implemented using the existing hardware in our processor circuit (including the new shifter), please indicate the necessary values of the following control signals output by the decoder.

\textbf{MWR: Read, Write}

\textbf{AluFunc: Add, Sub, And, Or, Xor, Slt, Sltu, Sll, Srl, Sra}

\begin{align*}
\text{BSEL: } & \quad 2 \\
\text{PCSEL: } & \quad 0 \\
\text{WERFSEL: } & \quad 1 \\
\text{WDSEL: } & \quad 1 \\
\text{MWR: } & \quad \text{Read} \\
\text{AluFunc: } & \quad \text{Or}
\end{align*}

Using the binary encoding of \texttt{setb}, determine the necessary register and immediate values that must be outputs from the decoder to support the \texttt{setb instruction} (N/A if output is not needed). Write your answer in terms of the bits in \texttt{inst}, the 32-bit binary encoding of the instruction.

\begin{align*}
\text{rd: } & \quad \text{inst}[11:7] \\
\text{rs1: } & \quad \text{inst}[11:7] \\
\text{rs2: } & \quad \text{N/A} \\
\text{imm: } & \quad \text{inst}[19:15]
\end{align*}
(C) (4 points) clrb cannot be implemented using the existing hardware. Please describe all the hardware changes required to support this instruction.

**What are all the changes required to support clrb?**

1. **New ALU operation that can perform & and ~ operations.**
2. **New decode logic for clrb that matches decode logic required for setb.**
Problem 4. Caches (16 points)

Buoyed by your success at the Zoo Company, you have been brought on board the LEAP-FROG project, the latest and greatest in cache design. The LEAP-FROG cache features an industry leading 3-way set-associative cache with 4 cache sets and a block size of 8 words. Data words and addresses are 32 bits.

(A) (2 points) Which address bits should the LEAP-FROG cache use for its tag, index, and block offset?

Address bits for tag: \text{addr}[\underline{31} : \underline{7}]

Address bits for index: \text{addr}[\underline{6} : \underline{5}]

Address bits for block offset: \text{addr}[\underline{4} : \underline{2}]

(B) (2 points) If the LEAP-FROG cache were instead 6-way set-associative and had a block size of 4 words, how would each of the following change? The total number of words in the cache remains unchanged.

Number of tag bits (select one of the choices below):

\begin{itemize}
  \item UNCHANGED
  \item $+1$
  \item $-1$
  \item $2x$
  \item $0.5x$
  \item CAN'T TELL
\end{itemize}

Number of cache sets (select one of the choices below):

\begin{itemize}
  \item UNCHANGED
  \item $+1$
  \item $-1$
  \item $2x$
  \item $0.5x$
  \item CAN'T TELL
\end{itemize}

(C) (6 points) The current contents of the LEAP-FROG cache are shown below. All values are in hex. Assume that any hex digit not shown is zero. Would the following memory accesses result in a hit or a miss? If it is a hit, give the value returned. If it is a miss, explain in a few words. Assume that index 0 corresponds to the first row of each way.

32-bit address: \text{0x72C8}

\begin{itemize}
  \item Index: \underline{2}
  \item Tag: \text{0x\underline{E5}}
  \item Block offset: \underline{2}
\end{itemize}

Return value if hit (0x)/ Explanation if miss: \underline{\text{Miss, tag matches but line is invalid}}

32-bit address: \text{0x3AE4}

\begin{itemize}
  \item Index: \underline{3}
  \item Tag: \text{0x\underline{75}}
  \item Block offset: \underline{1}
\end{itemize}

Return value if hit (0x)/ Explanation if miss: \underline{0x36}

6.004 Spring 2022 - 16 of 26 - Updated Quiz #2
You have decided to benchmark the performance of the LEAP-FROG cache with a short assembly sequence that copies an array A at address 0x1000 to an array B at address 0x2000.

```
.l = 0
    lui x1, 0x1  // address of Array A
    lui x2, 0x2  // address of Array B
    addi x3, x0, 0    // index
    addi x4, x0, 1000 // words to copy
loop:
    slli x5, x3, 2
    add x6, x5, x1
    add x7, x5, x2
    lw x8, 0(x6)
    sw x8, 0(x7)
    addi x3, x3, 1
    blt x3, x4, loop
end:
    unimp
```

(D) (1 points) How many instruction fetches and data accesses occur per iteration of the loop?

Number of instruction fetches: 7
Number of data accesses: 2
(E) (3 points) What is the steady-state hit ratio for the loop after many iterations?

Steady-state hit ratio: ___ 35/36 ___

Because the cache is 3-way set-associative there are no conflict misses. The only misses are from compulsory misses for loads and stores, which occur 1/8 of the time because the block size is 8. Each iteration has 9 memory accesses, so across 8 iterations there are 72 accesses of which 2 are misses. So our hit rate is 70/72 = 35/36.

(F) (2 points) Are there any conflict misses for the loop in steady state? (these are misses due to insufficient associativity in the cache) If yes explain why they occur, if no explain how the design of the LEAP-FROG cache avoids conflict misses.

Because the cache is 3-way set-associative there are no conflict misses. Instruction accesses, array A, and array B can all go in different ways of the cache.
Problem 5. Pipelined Processors (18 points)

Consider the following loop, which runs on a standard 5-stage RISC-V processor with full bypassing. Assume that branches are always predicted not taken and that branch decisions are made in the EXE stage. Assume that the loop repeats many times and its currently in the middle of its execution.

In case you need them, extra pipeline diagrams are available at the end of the quiz (page 18).

```
loop:
    slli x5, x3, 2
    add x6, x5, x1
    add x7, x5, x2
    lw x8, 0(x6)
    sw x8, 0(x7)
    addi x3, x3, 1
    blt x3, x4, loop
    ori x5, x2, 2
    and x7, x5, x7
end:
```

(A) (5 points) Fill in the pipeline diagram for cycles 200-207 assuming that at cycle 200 the `slli` instruction is fetched. Draw arrows indicating each use of bypassing. You can ignore any cells shaded in gray.

<table>
<thead>
<tr>
<th></th>
<th>200</th>
<th>201</th>
<th>202</th>
<th>203</th>
<th>204</th>
<th>205</th>
<th>206</th>
<th>207</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>slli</td>
<td>add</td>
<td>add</td>
<td>lw</td>
<td>sw</td>
<td>addi</td>
<td>addi</td>
<td>addi</td>
</tr>
<tr>
<td>DEC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>add</td>
<td>add</td>
<td>sw</td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NOP</td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

During cycles 200-207, how many cycles are wasted due to stalls? How many are wasted due to annulments?

Number of cycles wasted due to stalls: _____2_______

Number of cycles wasted due to annulments: _____0_______
loop:
  slli x5, x3, 2
  add x6, x5, x1
  add x7, x5, x2
  lw x8, 0(x6)
  sw x8, 0(x7)
  addi x3, x3, 1
  blt x3, x4, loop
  ori x5, x2, 2
  and x7, x5, x7
end:

(B) (5 points) Fill in the pipeline diagram for cycles 300-307 assuming that at cycle 301 the `addi` instruction enters the decode stage. **Draw arrows indicating each use of bypassing.** You can ignore any cells shaded in gray. The code is repeated above for your convenience.

<table>
<thead>
<tr>
<th></th>
<th>300</th>
<th>301</th>
<th>302</th>
<th>303</th>
<th>304</th>
<th>305</th>
<th>306</th>
<th>307</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>addi</td>
<td></td>
<td>blt</td>
<td>ori</td>
<td>and</td>
<td>slli</td>
<td>add</td>
<td>add</td>
</tr>
<tr>
<td>DEC</td>
<td></td>
<td>addi</td>
<td></td>
<td>blt</td>
<td>ori</td>
<td>NOP</td>
<td>slli</td>
<td>add</td>
</tr>
<tr>
<td>EXE</td>
<td></td>
<td></td>
<td></td>
<td>addi</td>
<td>blt</td>
<td>NOP</td>
<td>NOP</td>
<td>slli</td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addi</td>
<td>blt</td>
<td>NOP</td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NOP</td>
</tr>
</tbody>
</table>

During cycles 300-307, how many cycles are wasted due to stalls? How many are wasted due to annulments?

\[
\begin{align*}
\text{Number of cycles wasted due to stalls: } & \quad 0 \\
\text{Number of cycles wasted due to annulments: } & \quad 2
\end{align*}
\]

(C) (2 points) Suppose that for cost reasons, you need to give up one of your bypass paths. If your overall goal is to minimize the steady state number of cycles per iteration of the loop above, which bypass path should you give up and why.

`WB \rightarrow DEC` is only used once whereas the other two bypass paths are used twice. So removing `WB \rightarrow DEC` will only add one stall cycle whereas removing either of the other bypass paths would add 2 stall cycles.
(D) (6 points) Consider executing the following code snippet on the following 3 broken 5-stage pipelined RISC-V processors:

**P1:** Register values read from EXE to DEC bypass are read as 0, registers read from register file or other bypass paths are read correctly.

**P2:** Register values read from MEM to DEC bypass are read as 0, registers read from register file or other bypass paths are read correctly.

**P3:** Register values read from WB to DEC bypass are read as 0, registers read from register file or other bypass paths are read correctly.

```
addi x1, x0, 1
addi x2, x0, 2
addi x3, x0, 3
add  x4, x3, x2
add  x5, x3, x2
add  x6, x4, x2
```

What value when end up in x6 after running this code snippet on each of the 3 broken processors? The following pipeline diagram is provided for your convenience, but you are not required to fill it out.

<table>
<thead>
<tr>
<th></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>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>addi</td>
<td>addi</td>
<td>addi</td>
<td>add</td>
<td>add</td>
<td>add</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td></td>
<td>addi</td>
<td>addi</td>
<td>addi</td>
<td>add</td>
<td>(x3,x2)</td>
<td>add</td>
<td>(x3,x2)</td>
<td>add</td>
<td>(x4,x2)</td>
</tr>
<tr>
<td>EXE</td>
<td></td>
<td></td>
<td>addi</td>
<td>addi</td>
<td>addi</td>
<td>x3</td>
<td>add</td>
<td>x4</td>
<td>add</td>
<td>x5</td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td>addi</td>
<td>addi</td>
<td>x2</td>
<td>add</td>
<td>x3</td>
<td>add</td>
<td>x4</td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td>addi</td>
<td>addi</td>
<td>x1</td>
<td>add</td>
<td>x2</td>
<td>add</td>
<td>x3</td>
</tr>
<tr>
<td>Next cycle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXE comp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P1: x4=0+2=2</td>
<td>P1: x5=3+2=5</td>
<td>P1: x6=2+2=4</td>
<td>P2: x4=3+0=3</td>
<td>P2: x5=0+2=2</td>
</tr>
</tbody>
</table>

x6 after running code on P1: _____4_____

x6 after running code on P2: _____2_____

x6 after running code on P3: _____7_____
Problem 6. Pipelined Processor Performance (19 points)

Alice and her teammates are designing their own RISC-V processor with integer multiplication support. It supports a `mul` instruction that multiplies two source registers and produces a 32-bit output.

Some of the following questions use pipeline diagrams. In case you need them, extra pipeline diagrams are available at the end of the quiz (pages 19-21).

They use the following program to test their processor. Assume the branch will always be taken. All instructions after the `ret` are NOP.

```
LOOP:    lw x14, 4(x12)
        mul x10, x14, x10
        andi x12, x14, 255
        bnez x12, LOOP
EXIT:    sw x12, 0(x11)
        ret
```

At first, Alice tries to build a single-cycle processor, as shown below.

(A) (2 points) Assume the processor has ideal registers ($t_{\text{setup}} = t_{\text{hold}} = t_{\text{pd}} = t_{\text{cd}} = 0\text{ns}$) and all memory operations can be done in a single cycle (i.e., combinational reads). Use the following table to derive the minimal clock period, the average number of cycles per instruction, and the overall performance of the processor measured as the average number of instructions per nanosecond when running the test code above.

<table>
<thead>
<tr>
<th>Logic Block</th>
<th>Propagation Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>3\text{ns}</td>
</tr>
<tr>
<td>DEC</td>
<td>2\text{ns}</td>
</tr>
<tr>
<td>EXE</td>
<td>10\text{ns}</td>
</tr>
<tr>
<td>MEM</td>
<td>3\text{ns}</td>
</tr>
<tr>
<td>WB</td>
<td>2\text{ns}</td>
</tr>
</tbody>
</table>

Clock period (ns): ____20____

Average cycles per instruction: ____1____

Average number of instructions per nanosecond: ____1/20____
Bob recalls that pipelining could be used to improve the clock frequency. He modifies Alice’s design as shown in the following figure. Assume full bypassing is used and branch decision is made in the EXE stage. The IF stage speculatively fetches the instruction at PC+4 and annuls incorrectly fetched instructions after a branch misprediction.

Bob uses the same test program to test the performance his processor. The test benchmark is repeated below for your convenience.

```plaintext
LOOP:   lw  x14, 4(x12)
        mul x10, x14, x10
        andi x12, x14, 255
        bnez x12, LOOP
EXIT:   sw  x12, 0(x11)
        ret
```

(B) (3 points) Calculate the clock period, average cycles per instruction, and average number of instructions per nanosecond when executing the test code. Assume that bypassing is not in the critical path. You may use the pipeline diagram below, but we will only grade your final answers.

<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>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>lw</td>
<td>mul</td>
<td>andi</td>
<td>andi</td>
<td>andi</td>
<td>bnez</td>
<td>sw</td>
<td>ret</td>
<td>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>lw</td>
<td>mul</td>
<td>mul</td>
<td>mul</td>
<td>andi</td>
<td>bnez</td>
<td>sw</td>
<td></td>
<td></td>
<td>NOP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>mul</td>
<td>andi</td>
<td>bnez</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>mul</td>
<td>andi</td>
<td>bnez</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Clock period (ns): _____10_____

Average cycles per instruction: _____2_____  

Average number of instructions per nanosecond: _____1/20_____  

8 cycles per loop iteration (4 instructions), so average cycles per instruction = 2.

4 instr/80 ns = 1/20 instr/ns.
Carol thinks the EXE stage may be the reason for the low performance improvement, so she splits the EXE stage into 3 pipeline stages. The bypass and branch decision logic of the previous EXE stage is now moved to EX3 stage.

The test benchmark is repeated below for your convenience.

```
LOOP:
  lw    x14, 4(x12)
  mul   x10, x14, x10
  andi  x12, x14, 255
  bnez  x12, LOOP
EXIT:
  sw    x12, 0(x11)
  ret
```

(C) (6 points) Fill in the following pipeline diagram. Use arrows to denote all uses of bypass paths. Then, calculate the clock period, average cycles per instruction, and average number of instructions per nanosecond for the test benchmark running on this processor.

<table>
<thead>
<tr>
<th>Logic Block</th>
<th>Propagation Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX1</td>
<td>4ns</td>
</tr>
<tr>
<td>EX2</td>
<td>3ns</td>
</tr>
<tr>
<td>EX3</td>
<td>3ns</td>
</tr>
</tbody>
</table>

```
Cycle 1 2 3 4 5 6 7 8 9 10 11 12
IF lw mul andi andi andi andi andi bnez sw sw sw sw ret
DEC lw mul mul mul mul mul andi bnez bnez bnez bnez sw
EX1 lw NOP NOP NOP NOP mul andi NOP NOP bnez
EX2 lw NOP NOP NOP NOP mu1 andi NOP NOP NOP
EX3 lw NOP NOP NOP NOP mul andi NOP
MEM lw NOP NOP NOP NOP mu1 andi
WB lw NOP NOP NOP NOP NOP mul
```

Clock period (ns): _____ 4 _____

Average cycles per instruction: _____ 3.5 _____

Average number of instructions per nanosecond: _____ 1/14 _____

Branch decision is made when bnez gets to EX3 in cycle 14, so lw is fetched again in cycle 15.
14 cycles per loop iteration (4 instructions), so average cycles per instruction = 14/4 = 3.5
4 instr/56 ns = 1/14 instr/ns.
Dave finds that the critical path of the EXE stage is actually the multiplier used for the mul instruction. He splits the multiplier from the EXE stage and makes them run in parallel so that non-mul instructions only need 1 cycle to complete. The branch decision logic is still in EX stage.

The MUX before the MEM stage uses the following policy: it will select the valid instruction from EX and MUL3, and if both instructions from EX and MUL3 are valid, it will select the one from MUL3 and stall EX and all earlier pipeline stages.

The test benchmark is repeated below for your convenience.

LOOP:  
```assembly
lw x14, 4(x12)
mul x10, x14, x10
andi x12, x14, 255
bnez x12, LOOP
```

EXIT:  
```assembly
sw x12, 0(x11)
ret
```

The timing parameters for Dave’s processor are shown below.

<table>
<thead>
<tr>
<th>Logic Block</th>
<th>Propagation Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX</td>
<td>4ns</td>
</tr>
<tr>
<td>MUL1</td>
<td>4ns</td>
</tr>
<tr>
<td>MUL2</td>
<td>3ns</td>
</tr>
<tr>
<td>MUL3</td>
<td>3ns</td>
</tr>
</tbody>
</table>

(D) (6 points) Fill in the following pipeline diagram. **Use arrows to denote all uses of bypass paths.** Then, calculate the clock period, average cycles per instruction, and average number of instructions per nanosecond for the test benchmark running on this processor. Fill in these values on the next page.
Clock period (ns): _____4_____

Average cycles per instruction: _____2_____

Average number of instructions per nanosecond: _____1/8_____ 

8 cycles per loop iteration (4 instructions), so average cycles per instruction = 8/4 = 2
4 instr/32 ns = 1/8 instr/ns.

Also accepted 9 cycles per loop iteration if you thought lw could not be fetched until bnez leaves EX stage.
In this case CPI is 2.25 and average number of instructions per ns is 1/9.

Eve carefully examines the WB stage in the pipeline diagram of Dave’s design and is concerned that his design may produce incorrect results in some rare cases.

(E) (2 points) Write a sequence of instructions that will break Dave’s design. Briefly explain what goes wrong.

mul x10, x11, x12
and x10, x1, x2

<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>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>mul</td>
<td>and</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>mul</td>
<td>and</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL1/EX</td>
<td>mul</td>
<td>and</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL2</td>
<td>mul</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL3</td>
<td>mul</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>and</td>
<td>mul</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td>and</td>
<td>mul</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The result of the mul will overwrite the result of the and.

END OF QUIZ 2!