Problem 1

a) The total propagation delay will be \((40 + 10 + 10) = 60\) pS. The throughput is \(\frac{1}{60}\).

b) In the case that there is an input register before the first pipeline stage:

The unit should be divided into two pipeline stages- one for the multipliers and one for both adders. The multiplication stage takes 5 pS for the input register propagation delay, 40 pS for the multiply, and 5 pS for the register setup, resulting in a total delay of 50 pS. The adder stage takes 5 pS for the register propagation delay, 20 pS for the add, and 5 pS for the output register setup time, for a total of 30 pS.

The maximum throughput is then \(\frac{1}{50}\) with a minimum latency of 100 pS over 2 clock cycles.

If there is no register before the first pipeline stage, then the multiply section is 5 pS faster from skipping the register propagation delay. This results in a throughput of \(\frac{1}{45}\) and a latency of 90 pS.

c) Keep in mind that this is a pipelining of the entire LEE circuit, not just the single multiply unit. The entire LEE circuit has 4 multipliers in parallel, followed sequentially by a small adder and 3 large adders.

The multipliers are still the slowest part of the circuit. When pipeline registers are added, the delay through the multiply stage is 35 pS. The adders can be grouped into two more stages: the small adder and a large adder, two large adders.

With the described three stage circuit, the throughput is \(\frac{1}{35}\) with a latency of 115 pS over 3 clock cycles.

If one ignores the initial input register, a throughput of \(\frac{1}{30}\) and latency 120 pS can be achieved with a 4 stage pipeline.

d) Each average triangle takes 7 evaluations \(\times 30\) pixels \(\times \) about 35 pS per evaluation = 7350 pS.

Since there are \(10^{12}\) pS in a second, there can be \(\frac{10^{12}}{7350}\) or about 136 million triangles per second.
Problem 2

a) The output results are:

\[
\begin{align*}
O_0 &= C \\
O_1 &= A + C \\
O_2 &= 2A + C \\
O_3 &= 3A + C \\
O_4 &= 4A + C \\
O_5 &= 5A + C \\
O_6 &= 6A + C \\
O_7 &= 7A + C
\end{align*}
\]

b) A ‘divide by 2’ circuit can be created by making a circuit that shifts values to the right by one. Recall that shift left and right is identical to multiplying and dividing by powers of two. To shift right, we construct a circuit that discards the right-most bit, shifts all other bits right by one, and introduces a new 0 as the value for the left-most bit.

The original LEE was \(Ax + By + C\). Let us first introduce \(C' = By + C\). Since \(B\) and \(C\) are constant for any evaluation, we can change \(y\) by simply changing \(C'\). We then use this \(C'\) instead of \(C\) in the newly designed LEE circuit.

c) Since each output in computed using a chain of 6 adders, the propagation delay is 60 pS. Since there are 64 evaluations output, the throughput is then \(64/60\) LEEs per pS or about \(1.07 \times 10^{12}\) LEEs per second. Since there are 7 adders in each of the 9 component blocks, there is a total of 63 adders in the design.

d) To achieve maximum throughput, each ‘row’ of adders is put into a pipeline stage. This results in 3 pipeline stages for a single block, while the entire parallel design has a 6 stage deep pipeline.

Each stage requires 5 pS for the register propagation, 10 pS for the add, and another 5 pS for the register setup. Since 64 evaluations are output per cycle, there is a throughput of \(64/20\) LEEs per pS or about \(3.2 \times 10^{12}\) LEEs per second. The minimum latency is 120 pS for all six pipeline stages.

e) From the last problem, there were \(3.2 \times 10^{12}\) LEEs per second. Each triangle uses 30 pixels, and each pixel needs 7 evaluations. This leads to a total of \(\frac{1}{30} \times \frac{1}{7} \times 3.2 \times 10^{12}\) or about \(1.52 \times 10^{10}\) triangles per second. However, only 40% of those evaluations are valid, so the final rate is about \(6.1 \times 10^{9}\) triangles per second.

Problem 3

a) The \texttt{abnz} instruction will use the immediate format (since the address is supplied in immediate form).
b) Control should be set as follows:

<table>
<thead>
<tr>
<th>Control</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCSEL</td>
<td>asdf</td>
</tr>
<tr>
<td>WASEL</td>
<td>1</td>
</tr>
<tr>
<td>SEXT</td>
<td>1</td>
</tr>
<tr>
<td>BSEL</td>
<td>0</td>
</tr>
<tr>
<td>WDSEL</td>
<td>1</td>
</tr>
<tr>
<td>ALUFN:sub</td>
<td>1</td>
</tr>
<tr>
<td>ALUFN:bool</td>
<td>xx</td>
</tr>
<tr>
<td>ALUFN:shift</td>
<td>0</td>
</tr>
<tr>
<td>ALUFN:math</td>
<td>1</td>
</tr>
<tr>
<td>WR</td>
<td>0</td>
</tr>
<tr>
<td>WERF</td>
<td>1</td>
</tr>
<tr>
<td>ASEL</td>
<td>0</td>
</tr>
</tbody>
</table>

c) The datapath only supports subtracting $rt$ from $rs$. Either the instruction would behave differently than sub, or the datapath would need to be modified.

d) To find the percent difference, divide the difference by the original amount.

$$\frac{(3N + 4) - (2N + 5)}{(3N + 4)} \geq 0.25$$

When $N \geq 8$, sum2 is faster than sum1.

e) The branch decision for abnz is not known until the ALU stage.

This would require 2 delay slots after the branch instruction.

In order to predict the branch decision early, an adder and comparator would be needed in the Register-Fetch stage. Clearly, this is much more complex and slower than the simple comparator logic needed for the other branch instructions.

f) Standard:

```
addi $t0, $t0, 0    #sum stored in $t0
    addi $t1, $t1, 0  #i = 0
    addi $t2, $0, N   #N stored in $t2
    slt $t2, $t1, $t2
    beq $t2, $0, end
loop: sll $t1, $t1, 2
    lw $t3, x($t1)
    add $t0, $t0, $t3 #sum = sum + x[i]
    addi $t1, $t1, 1  #i++
    addi $t2, $0, N   #N stored in $t2
    slt $t2, $t1, $t2
    bne $t2, $0, loop
end:
```
ABNZ code:

```
addi $t0, $t0, 0  #sum stored in $t0
addi $t1, $t1, 0  #i = 0
addi $t2, $0, N
beq $t2, $t1, end
loop: subi $t2, $0, N   # -N stored in $t2
    sll $t1, $t1, 2
    lw $t3, x($t1)
    add $t0, $t0, $t3  #sum = sum + x[i]
    addi $t1, $t1, 1   # i++
    abnz $t1, $t2, loop
end:
```

Note that the problem states that the sum must be computed in the same order as the C code (no counting backwards).

Problem 4

a) In the case of 1000 adds, on the original pipeline, the first add would take 5 cycles to complete. Since the other 999 adds will be pipelined, one will finish on each following cycle for 999 cycles. The total on the original pipeline is thus 1004 cycles for 1000 adds.

On the modified pipeline, the first add will take only 4 cycles to complete. Each sequential cycles completes another add. Thus the total is 1003 cycles for 1000 adds.

b) One instruction will complete per clock cycle on the original pipeline.

c) Bud’s pipeline is ‘faster’ by allowing instructions to skip the MEM stage. If a 5 stage instruction such as lw or sw uses the MEM stage, it will use the WB stage on the next cycle. Thus, if a 4 stage instruction follows a 5 stage instruction, the 4 stage instruction will have to stall until the WB stage is free.

```
<table>
<thead>
<tr>
<th>Stage</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>lw</td>
<td>add</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RF</td>
<td>lw</td>
<td>add</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU</td>
<td>lw</td>
<td>add</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td>lw</td>
<td>stall</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td>lw</td>
<td>add</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

d) Bud’s pipeline seems much faster when executing a single instruction. However, in most cases, long sequences of instructions are going to be executed. As part b) showed, long sequences execute at nearly the same speed on the both pipelines. Part c) showed that when the MEM stage is used, Bud’s pipeline runs at the same speed as the original design.

Bud’s design does not increase performance by any noticeable amount.
Problem 5

a) To emulate \texttt{lw} $t0, 0xoffset(t1)$:

\begin{verbatim}
addi $as, $t1, 0xoffset
lw $t0, ($as)
\end{verbatim}

To emulate \texttt{sw} $t0, 0xoffset(t1)$:

\begin{verbatim}
addi $as, $t1, 0xoffset
sw $t0, ($as)
\end{verbatim}

b) A bypass path is needed from WDSEL to the ASEL and BSEL muxes. This is needed when the value from memory is used in the next instruction. Consider the following sequence of instructions:

\begin{verbatim}
lw $t0, ($t1)
addi $t1, $t0, $t0
\end{verbatim}

c) No interlocks are required, since the MEM stage completes in time for its value to be used in the next cycle.