1 Introduction

This programming assignment will require you to design a system with two ALUs that can perform arithmetic operations on 32-bit operands in a cascaded design. Operands either come from external 32-bit inputs or feedback from one of two registers. This assignment has the following functionality:

- The following ALU operations:
  - 32-bit addition
  - 32-bit multiplication
  - 32-bit shift-logical-right
  - 32-bit compare-if-equal
- Logic to control arithmetic operations, based on a select input.
- Logic to control ALU and 64-bit register inputs using multiplexers.
- Verification of the design using a test bench.
2 Description

The figure below shows the architecture of the system:

Overall Design of PA1

The ALU operation of the inputs and values stored in the assigned register bank are performed continuously. X[31:0] and Y[31:0] serve as external input signals to the system, and are 32-bits wide. The arithmetic logical units ALU1 and ALU2 have operations controlled by the alu1_sel and alu2_sel signals, which are described in table 1 below:

<table>
<thead>
<tr>
<th>alu1_sel / alu2_sel</th>
<th>OPERATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>addition</td>
</tr>
<tr>
<td>01</td>
<td>multiplication</td>
</tr>
<tr>
<td>10</td>
<td>right-logical shift</td>
</tr>
<tr>
<td>11</td>
<td>compare-if-equal</td>
</tr>
</tbody>
</table>
The \texttt{r1\_en} and \texttt{r2\_en} signals control the 64-bit registers R1 and R2. When ‘1,’ the register should propagate the value from D → Q on the \textbf{rising edge} of \texttt{clk}. Otherwise, the register should instead hold the old value of Q.

The register reset signal \texttt{reset} \textit{synchronously} clears the values of R1 and R2. When brought to a ‘1,’ the values held by R1 and R2 are set to 0 on the rising edge of \texttt{clk}. When the signal is ‘0,’ the registers operate as normally. The reset signal should remain ‘0,’ except when you need to clear the outputs of the registers. Initially, the register output values should be 0 by asserting the reset signal at the beginning of simulation.

There are also 3 MUX control signals: \texttt{m1\_sel}, \texttt{m2\_sel}, and \texttt{m3\_sel}. When these select signals are ‘0,’ the MUX should select the input A to propagate to the output. Otherwise, B should be propagated. M1 chooses between the input \texttt{Y} and the output \texttt{Z1} as the operand A of ALU2, while M2 and M3 select a feedback signal \texttt{Z1} or \texttt{Z2} as the operand B to ALU2 and ALU1, respectively.

Each ALU is expected to perform an unsigned integer add, multiply, logical right shift, and compare-if-equal operation. We do not require structural VHDL for these operations. You should use behavioral operators and the appropriate IEEE libraries for your design. One design option is to use constrained unsigned to represent the input operand (i.e. \texttt{X: in unsigned (31 downto 0); for the X operand}) and use \texttt{IEEE.numeric_bit.all} library to perform the add and multiply operations. The comparison operation can be performed via library functions, or simply the \texttt{=} operator. \textbf{Note that for the right shift operation, input B is the value to be shifted right by a single bit, and input A is unused.} All non-shift operations use both operands A and B.

The register output values, \texttt{Z1} and \texttt{Z2}, are 64-bit buses. As mentioned above, the values of the multiplexer select signals determine which outputs to use for the ALU1 and ALU2 feedback inputs. The lower 32-bit output of each register bank is connected to the inputs of the multiplexers (i.e. any ALU input signals coming from register outputs need to be truncated appropriately to match the 32-bit widths of each ALU input). Keep in mind we do not consider overflow in the design of our digital system, so these conditions may be ignored.

You will use separate VHDL modules for i) the register bank (\texttt{dff.vhd}) ii) the ALU (\texttt{alu.vhd}) and iii) the overall PA1 module (\texttt{pa1.vhd}). You are given top-level modules with entity instantiations, and you are expected to write the architecture for each module. In the \texttt{pa1} module specifically (\texttt{pa1.vhd}), you are given the architecture declaration, signal definitions, and entity instantiations, and you are required to fill in the logic for each signal and the port maps for each of these entity instantiations. Note, entity instantiation is an alternative way to instantiate modules. You can read more about entity instantiation and how it works compared to component instantiation \url{[here]}. It is recommended that you use explicit port connection definitions in your port map definitions, which was used exclusively in the examples from the hyperlink provided above. You will be graded on the design of these modules and their functionality.
3 Test Bench

A good design develops a test bench. Complete the following simulations using test benches `testbench1.vhd` and `testbench2.vhd`. The first test bench is provided and described in section 3.1. The second test bench in section 3.2 gives you a high level function that you need to implement. You should use `testbench1.vhd` as a reference to help you when writing your own code in `testbench2.vhd`.

3.1 Calculating 3D Distance with a Test Bench

In the provided example test bench (`testbench1.vhd`) we use the digital design to find a 3D point’s (squared) distance from origin. Given a point’s coordinate, \((x, y, z)\), and the origin coordinate, \((0, 0, 0)\), we calculate the squared distance between them using the following equation:

\[ r^2 = x^2 + y^2 + z^2 \quad (1) \]

When you are finished writing each module (`dff.vhd`, `alu.vhd`, and `pa1.vhd`), simulate the design using `testbench1.vhd`. To select the current simulation, right click “testbench1” and select “set as top” (it should be bold). In the test bench, the coordinate \((x, y, z) = (5, 6, 7)\) is used to test the functionality by default. After testing and confirming the result, feel free to change the values of the defined constants (lines 30, 31, and 32 in `testbench1.vhd`) to different \((x, y, z) = (a, b, c)\) to test your code with alternate coordinates. The test bench data flow for generic coordinates \((x, y, z) = (a, b, c)\) is as follows:

- Reset the system so R1 and R2 are zero. Set the inputs \(X \leftarrow a, Y \leftarrow b\). Set up \(R1 \leftarrow X + R1\) and \(R2 \leftarrow Y + R1\). Wait one clock cycle \((R1 = a\) and \(R2 = b\)).
- Set up \(R1 \leftarrow X \cdot R1\) and \(R2 \leftarrow Y \cdot R2\). Wait one clock cycle \((R1 = a^2\) and \(R2 = b^2\)).
- Set input \(X \leftarrow 0\). Set up \(R1 \leftarrow X \cdot R1\) and \(R2 \leftarrow R1 + R2\). Wait one clock cycle \((R1 = 0\) and \(R2 = a^2 + b^2\)).
- Set input \(X \leftarrow c\). Hold R2 output by disabling the register. Wait one clock cycle \((R1 = c\) and \(R2 = a^2 + b^2\)).
- Set up \(R1 \leftarrow R1 \cdot X\). Wait one clock cycle \((R1 = c^2\) and \(R2 = a^2 + b^2\)).
- Set up \(R2 \leftarrow R1 + R2\). Enable R2 so it can update. Wait one clock cycle \((R2 = a^2 + b^2 + c^2)\). Done!

Keep in mind that the R1 output is mapped to \(Z_1\) and R2 is mapped to \(Z_2\), so these are the signals you should observe in your simulation. For in-depth details on how this is implemented using the designed system, read through `testbench1.vhd`. Comments are provided for you to follow the test bench flow as a tutorial.
3.2 Series Calculation using a Test Bench

For a certain calculation, you need to compute the result of the following function:

\[
  f(n) = \sum_{k=0}^{n} k(k+1)(k+2)
\]  

(2)

However, you realize you can calculate the function much more efficiently by using the following series equivalence:

\[
  \sum_{k=0}^{n} k(k+1)(k+2) = \frac{n(n+1)(n+2)(n+3)}{4}
\]  

(3)

You now want to use your digital design to efficiently compute \( f(n) \) using the skeleton test bench testbench2.vhd, but due to resource constraints, you are given the following restriction: input \( X \) is hardwired to 1 and input \( Y \) is hardwired to \( n \); that is, in the test bench, the input \( X \leftarrow 1 \) and \( Y \leftarrow n \) assignments are held constant for the duration of the test bench, and they may not be modified.

Using equation 3 and your completed digital design, compute \( f(5) \) in as few clock cycles as possible using the skeleton test bench testbench2.vhd to write your code. Although you are only responsible for showing the \( f(5) \) simulation, you are free to try using different input values of \( n \) by modifying the constant assignment at line 30. Remember that you cannot modify \( X \) and \( Y \) after the initial \( X \leftarrow 1 \) and \( Y \leftarrow n \) assignments, but all other control signals (MUX select, ALU select, register enable, and reset) may be changed at any point, unrestricted (Hint: sketch out the data flow required for the calculation first, and write out what values are accumulated at each clock cycle. Then, translate it to test bench control signals to get the desired behavior). **Your result should be in \( \mathbb{Z}_2 \) (the output of R2) at the end of your computation.** It will help to first read through the testbench1.vhd code and comments to understand the PA1 test bench flow.
4 Deliverables

Please submit the following report saved as a single PDF:

1. Your code for each module and your second testbench. You can copy and paste the code into a Word document; make sure to clearly label each code block.
   - `alu.vhd`
   - `dff.vhd`
   - `pa1.vhd`
   - `testbench2.vhd`

2. Submit screenshots of the following:
   - Your output waveform of `testbench1.vhd` using \((x, y, z) = (5, 6, 7)\). Show the waveform from 0ns to 120ns. The clk, reset, control, input, and output signals should all be clearly visible. The bit vectors \(X\), \(Y\), \(Z1\), and \(Z2\) should also be unsigned decimal radix (check the adding signals to waveform guide if you need help formatting or changing the radix).
   - Your output waveform of `testbench2.vhd` computing \(f(5)\). Show the waveform from 0ns to 150ns. If it takes longer than 150ns to compute your output, show enough of the waveform to clearly show the result (multiple screenshots are fine). Format all the waveform signals the same way as the previous test bench.