# Input Output Requirements

Required to implement a simple calculator with 4 pre-defined operations: AND, OR, XOR, ADD. The type of operation is specified by a 2-bit input signal using switches (17:16), 00 for AND, 01 for OR, 10 for XOR, and 11 for ADD. The corresponding 2-bit value of the operator is represented with two red LEDs.

Operand 1 and Operand 2 are both 8-bit input signals, defined by switches (7:0) and switches (15:8), respectively. The hexadecimal values of the operands are each displayed on a pair of seven-segment blocks, hex5, hex4 and hex7, hex6, respectively.

The result of the operation is represented on the three seven segment displays (hex2, hex1, hex0). Since the addition of two 8-bit numbers can cause overflow, hex2 can display 1 when needed, or be blank otherwise.

# Design and Implementation

A multiplexer was implemented to select the correct operation depending on the specified operator input. This is intuitive to use, as opposed to an if-else statement structure, because of the familiarity and elegance of code writing.

However before computing the result, the two operands were concatenated with the 4-bit string “0000” to produce two 12-bit strings, with the four zeros being appended as the most significant bits so the value of the inputs are not actually affected. This ensures that the result from the any operation is also a 12-bit string. This is required because the integer addition of two 8-bit (binary) numbers can overflow and produce a carry bit, meaning at least 9 bits are required to represent the result. However, instantiating the SevenSegment components require a logic vector of length 4 to be mapped to port outputs. Since the result is to be displayed on three 7-segment displays, it only makes sense to have 12-bit results and therefore the modifications to the operand strings are necessary.

One core aspect of this project was to usage of mathematical operations on unsigned signals. This was how the ADD operation was implemented. After the modification, the operands were cast into unsigned vector signals and added together, before being cast back into logic vectors.

One detail to account for is the usage of the 7-segment display representing the most significant digit of the result. The only case when it displays something is when there is an overflow with the addition of the two operands, in which case, it displays “1”. Therefore by default it should be off and not display anything, because displaying a “0” as the most significant digit is redundant. This specification can be accounted for with the blanking input when instantiating the SevenSegment for the most significant bit of the result. By setting the blanking bit as “NOT result(8)”, the 7-segment is ensured to be blank when there is no carry bit.

# Problems Encountered

The debugging process was more lengthy than previous labs, the main issues were once again with syntax errors, and errors associated with vector length. For example trying to incorporate and if statement sequence was ineffective and not successful, as there wasn’t good documentation online to be found. Another nuance was the re-assignment of logic vectors. Caution is required as logic vectors must maintain the declared length throughout the program. This was discovered with the attempted concatenation of “0000”, and re-assignment of operands. In this case a new signal, or logic vector of length 12 was declared to represent the each modified operand.

4. Functional simulation waveform with coverage for critical cases. Mark your simulation waveforms explaining several different scenarios for each operation. You must prove that what you have designed is working. For instance, testing with the inputs set to 0 does not allow one to distinguish one operation from another or if any operation in particular is fully working. Often one tests critical cases - the limits where things may break (i.e. overflow). Do not print waveforms of all possible cases - give samples for critical cases of each operation and explain how you checked for proper operation. The goal is to have a simulation to prove that the circuit works; without doing a full exhaustive test of all possible inputs. It should prove that all operations work correctly for enough numbers to give confidence that the circuit is fully functional.

**library** ieee**;**

**use** ieee**.**std\_logic\_1164**.all;**

**use** ieee**.**numeric\_std**.all;**

--

-- 7-segment display driver. It displays a 4-bit number on 7-segments

-- This is created as an entity so that it can be reused many times easily

--

**entity** SevenSegment **is** **port** **(**

dataIn **:** **in** std\_logic\_vector**(**3 **downto** 0**);** -- The 4 bit data to be displayed

blanking **:** **in** std\_logic**;** -- This bit turns off all segments

segmentsOut **:** **out** std\_logic\_vector**(**6 **downto** 0**)** -- 7-bit outputs to a 7-segment

**);**

**end** SevenSegment**;**

**architecture** Behavioral **of** SevenSegment **is**

--

-- The following statements convert a 4-bit input, called dataIn to a pattern of 7 bits

-- The segment turns on when it is '0' otherwise '1'

-- The blanking input is added to turns off the all segments

--

**begin**

**with** blanking **&** dataIn **select** -- gfedcba b3210 -- D7S

segmentsOut**(**6 **downto** 0**)** **<=** "1000000" **when** "00000"**,** -- [0]

"1111001" **when** "00001"**,** -- [1]

"0100100" **when** "00010"**,** -- [2] +---- a ----+

"0110000" **when** "00011"**,** -- [3] | |

"0011001" **when** "00100"**,** -- [4] | |

"0010010" **when** "00101"**,** -- [5] f b

"0000010" **when** "00110"**,** -- [6] | |

"1111000" **when** "00111"**,** -- [7] | |

"0000000" **when** "01000"**,** -- [8] +---- g ----+

"0010000" **when** "01001"**,** -- [9] | |

"0001000" **when** "01010"**,** -- [A] | |

"0000011" **when** "01011"**,** -- [b] e c

"1000110" **when** "01100"**,** -- [c] | |

"0100001" **when** "01101"**,** -- [d] | |

"0000110" **when** "01110"**,** -- [E] +---- d ----+

"0001110" **when** "01111"**,** -- [F]

"1111111" **when** **others;** -- [ ]

**end** Behavioral**;**

--------------------------------------------------------------------------------

-- Main entity

--------------------------------------------------------------------------------

**library** ieee**;**

**use** ieee**.**std\_logic\_1164**.all;**

**use** ieee**.**numeric\_std**.all;**

**entity** Lab2 **is** **port** **(**

ledr **:** **out** std\_logic\_vector**(**17 **downto** 0**);** -- displays the operator and result on the other end

sw **:** **in** std\_logic\_vector**(**17 **downto** 0**);** -- 18 dip switches

hex6**,**hex7 **:** **out** std\_logic\_vector**(**6 **downto** 0**);** -- display for operand 2

hex4**,**hex5 **:** **out** std\_logic\_vector**(**6 **downto** 0**);** -- display for operand 1

hex0**,** hex1**,** hex2 **:** **out** std\_logic\_vector**(**6 **downto** 0**)** -- display for the result

**);**

**end** Lab2**;**

**architecture** SimpleCircuit **of** Lab2 **is**

--

-- In order to use the "SevenSegment" entity, we have to use this declaration

-- It's signals have to correspond to the entity declared above

--

**component** SevenSegment **port** **(**

dataIn **:** **in** std\_logic\_vector**(**3 **downto** 0**);**

blanking **:** **in** std\_logic**;**

segmentsOut **:** **out** std\_logic\_vector**(**6 **downto** 0**)**

**);**

**end** **component;**

-- Create any signals, or temporary variables to be used

-- Unsigned is a signal which can be used to perform math operations such as +, -, \*

-- std\_logic\_vector is a signal which can be used for logic operations such as OR, AND, NOT, XOR

--

**signal** operand1**,** operand2**:** std\_logic\_vector**(**7 **downto** 0**);** -- 8-bit intermediate signals (wires)

**signal** operand1\_mod**,** operand2\_mod**:** std\_logic\_vector**(**11 **downto** 0**);**

**signal** result**:** std\_logic\_vector**(**11 **downto** 0**);** -- output signal for the three 7segment displays

**signal** operator**:** std\_logic\_vector**(**1 **downto** 0**);** -- input signal represents operator

**begin**

-- Intermediate signal assignments

operand1 **<=** sw**(**7 **downto** 0**);** -- connect the lowest 8 switches to operand1

operand2 **<=** sw**(**15 **downto** 8**);** -- connect the highest 8 switches to operand2

operator **<=** sw**(**17 **downto** 16**);** -- connect 2 switches to input operator signal

-- concatenate for 12 bits

-- apparently you can't just re-assign the signal variable in this case because the signal vector

-- is declared to be a fixed length

-- implementing a multiplexer, dependent on the operator input signal

**with** operator **select**

result **<=** operand1\_mod AND operand2\_mod **when** "00"**,**

operand1\_mod OR operand2\_mod **when** "01"**,**

operand1\_mod XOR operand2\_mod **when** "10"**,**

std\_logic\_vector**(**unsigned**(**operand1\_mod**)** **+** unsigned**(**operand2\_mod**))** **when** "11"**;**

-- note that the + operator only supports unsigned vector types

-- the input signals are cast to type, then cast back to std\_logic\_vector

-- light up LED to display operator

ledr**(**17 **downto** 16**)** **<=** operator**;**

-- light up the 9 red LEDs to display the result

ledr**(**8 **downto** 0**)** **<=** result**(**8 **downto** 0**);**

-- Instantiate instants of each SevenSegment components

-- Think of the instantiation as a constructor that takes in signal inputs and maps it to the corresponding

-- "member" signals within that component

Operand1\_MSD\_display**:** SevenSegment **port** **map(**operand1**(**7 **downto** 4**),** '0'**,** hex5**);**

Operand1\_LSD\_display**:** SevenSegment **port** **map(**operand1**(**3 **downto** 0**),** '0'**,** hex4**);**

Operand2\_MSD\_display**:** SevenSegment **port** **map(**operand2**(**7 **downto** 4**),** '0'**,** hex7**);**

Operand2\_LSD\_display**:** SevenSegment **port** **map(**operand2**(**3 **downto** 0**),** '0'**,** hex6**);**

result\_MSD**:** SevenSegment **port** **map(**result**(**11 **downto** 8**),** not result**(**8**),** hex2 **);** --

result\_2ndD**:** SevenSegment **port** **map(**result**(**7 **downto** 4**),** '0'**,** hex1 **);** -- display the second digit

result\_LSD**:** SevenSegment **port** **map(**result**(**3 **downto** 0**),** '0'**,** hex0 **);** -- dispaly the least significant digit

**end** SimpleCircuit**;**

