Tyler Meier, Lauren Stiefel EECE251 – A51

Final Project Calculator Documentation 12/09/2022

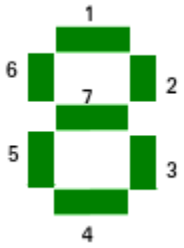
The Calculator project is written in Verilog with a source module wiring together 4 main submodules. The user can submit two 16-bit numbers and can choose to add, subtract, or multiply them to produce a 32-bit output.

Inputs:

- *btnU* – pressed state of the up button on the board
- *btnD* – pressed state of the down button on the board
- *btnC* – pressed state of the center button on the board
- *btnL* – pressed state of the left button on the board
- *btnR* – pressed state of the right button on the board
- *clk* – the 10 MHz clock from the board
- *sw[15:0]* – flipped-upward state for each of the 16 switches on the board. Note that *sw[0]* refers to the rightmost switch, and *sw[15]* refers to the leftmost switch.

Outputs:

- *led[15:0]* – controls the state of the 16 LEDS on the board. Note that *led[0]* refers to the rightmost LED and *led[15]* refers to the leftmost LED. A 1-state results in the LED being on, and a 0-state results in the LED being off.
- *seg[6:0]* – controls the 7 segments for the board's seven segment displays. Note that *seg[0]* controls segment 1 incrementing up to *seg[6]* controlling segment 7 as seen in the figure. A 1-state results in the segment being off, and a 0-state results in the segment being on. Note that the board wires control each corresponding segment together in each of the 4 seven-segment displays through a 7-wire cathode.

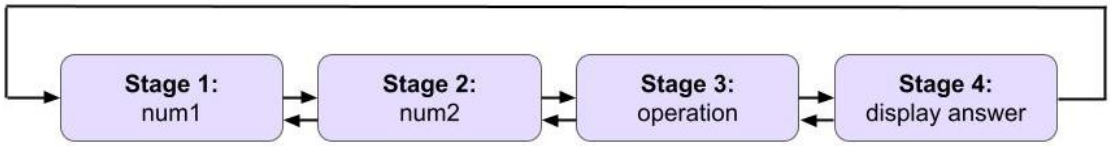


- *an[3:0]* – controls the board's 4 anodes which control the on and off state of the each of the 4 seven-segment displays in their entirety. Note that *an[0]* controls the rightmost seven-segment display, and *an[3]* controls the leftmost seven-segment display. A 1-state results in the seven segment display being off (all segments in that display are blank), and a 0-state results in the seven segment display being on (all segments in that display correspond to the state of *seg*).

Keywords:

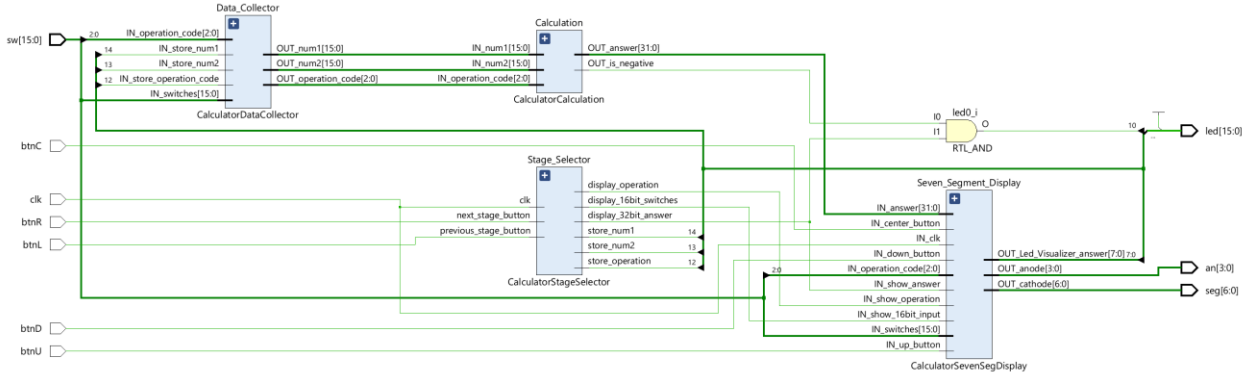
- **Calculator** – refers to the final project designed in its entirety as described in the Final Project Design Proposal.
- **Board** – refers to the physical Diligent BASYS 3 Artix-7 Field Programmable Grid Array (FPGA) circuit board in which the calculator is being implemented upon.

- **FSM** – acronym for finite state machine: refers to the sequential circuit controlled data flip flops and next-state combinational logic.
- **Source** – the Verilog file at the top of the module hierarchy. All modules in the calculator are contained within the source. All the source inputs and outputs are through the board.
- **Bus Multiplexer –** a combinational circuit component that functions similar to a regular multiplexer. The difference is instead of setting an output bit to mirror one of the inputs, the bus multiplexer sets an output bus to mirror one of the input buses of the same size. The selection input on a Bus Multiplexer is identical.
- **Data Flipflop –** a sequential circuit component that sets the output bit q to whatever is currently being sent to the input d on the rising edge of its clock input. The bit q is held constant until the next rising edge of the clock. A data flipflop is often referred to by its acronym dff.
- **Debouncer –** a circuit component used to synchronize the board's pushbuttons to the board's clock and to eliminate repeated inputs resulting from the metal contacts bouncing in and out of contact – thus the name debouncer.
- **Stages** – refers to the four different ways in which the user can interact with the calculator. The user can move in between stages on the board using *btnL* and *btnR*, the only exception is moving backwards to Stage 4 from Stage 1.

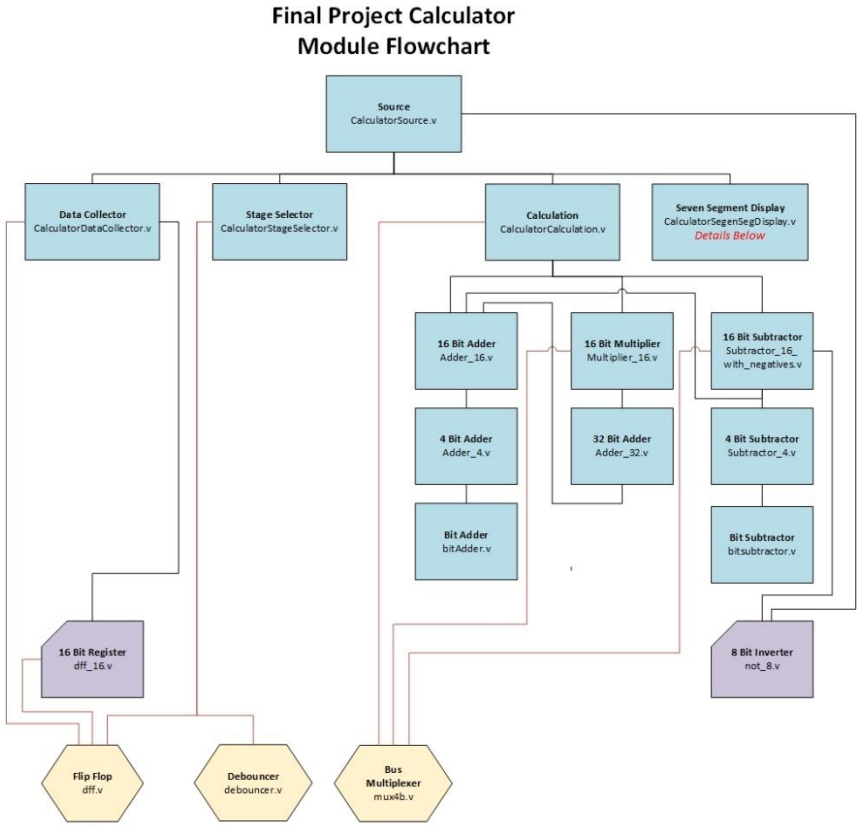


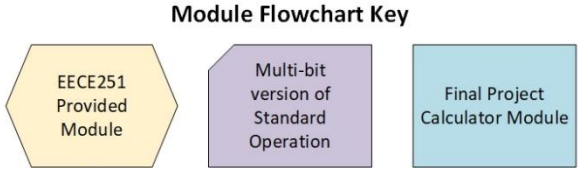
  - Stage 1: The user flips the 16 switches to correspond to the binary value of num1
  - Stage 2: The user flips the 16 switches to correspond to the binary value of num2
  - Stage 3: The user flips the 3 rightmost switches, *sw[2:0]*, corresponding to the desired operation performed on num1 and num2.
  - Stage 4: The user views the answer of the calculated operation performed on num1 and num2 on the seven-segment display, using *btnU, btnD*, and *btnC* to change which of the 8 possible digits of the answer are shown on the board's 4 seven-segment displays.
- **num1** – refers to the first 16-bit number the calculator will operate upon
- **num2** – refers to the second 16-bit number the calculator will operate upon
- **operation** – refers to the encoded mathematical operation, also labeled as "opcode" or operation code, which can represent addition, subtraction, and multiplication.
- **Hex Digit** – refers to the storage of a hexadecimal digit in 7-bit form which can directly output the digit on the board's seven-segment display. This is not the same as a Hexadecimal Value.
- **2's Compliment** – refers to the binary encoding method for numbers that can be both positive and negative.

Source Schematic:



Source Submodules:





1. Data Collector

The data collector pulls data from *sw* and stores the values of num1, num2, and the operation. The bits of num1, num2, and the operation are wired to the outputs of multiple data flipflops (dff), and the *sw* bits are wired to the inputs. The dff's clocks are controlled by wires which transition from 0-state to 1-state when data needs to be updated.

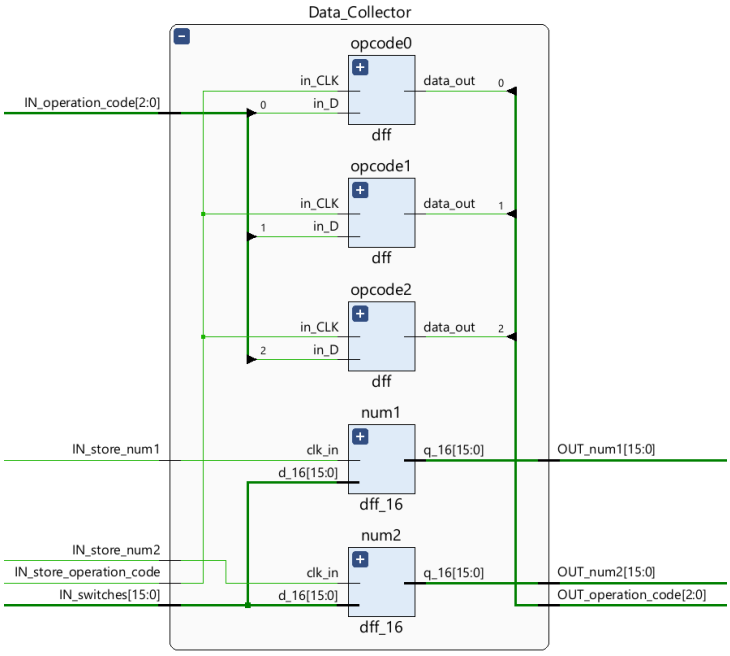
Inputs:

- IN_switches[15:0] – received directly from *sw*
- IN_operation_code[2:0] – received directly from *sw[2:0]*; a separate input for the operation code allows modularity for the source of those 3-bits.
- IN_store_num1 – received from the store_num1 output of the 'Stage Selector' module and transitions from a 0-state to a 1-state when the value of num1 needs to be updated.
- IN_store_num2 – received from the store_num2 output of the 'Stage Selector' module and transitions from a 0-state to a 1-state when the value of num2 needs to be updated.
- IN_store_operation_code – received from store_operation output of the 'Stage Selector' module and transitions from a 0-state to a 1-state when the value of the operation needs to be updated.

Outputs:

- OUT_num1[15:0] – holds the stored value of num1
- OUT_num2[15:0] – holds the stored value of num2
- OUT_operation_code[2:0] – holds the stored value of the operation code

Schematic:



Note the use of dff_16 which functions as a 16-bit register, making the 'Data Collector' module more compact compared to coding 16 individual dffs.

2. Stage Selector

The stage selector allows the user to navigate between the 4 stages using *btnL* and *btnR.* The stage selector has output wires cuing the 'Data Collector' module to update its stored values, and telling the Seven Segment Display module what to output.

Inputs:

- previous_stage_button – received from *btnL* to move backward through the stages.
- next_stage_button – received from *btnR* to move forward through the stages.
- clk – received from *clk* to clock the FSM and to synchronize the button inputs.

Outputs:

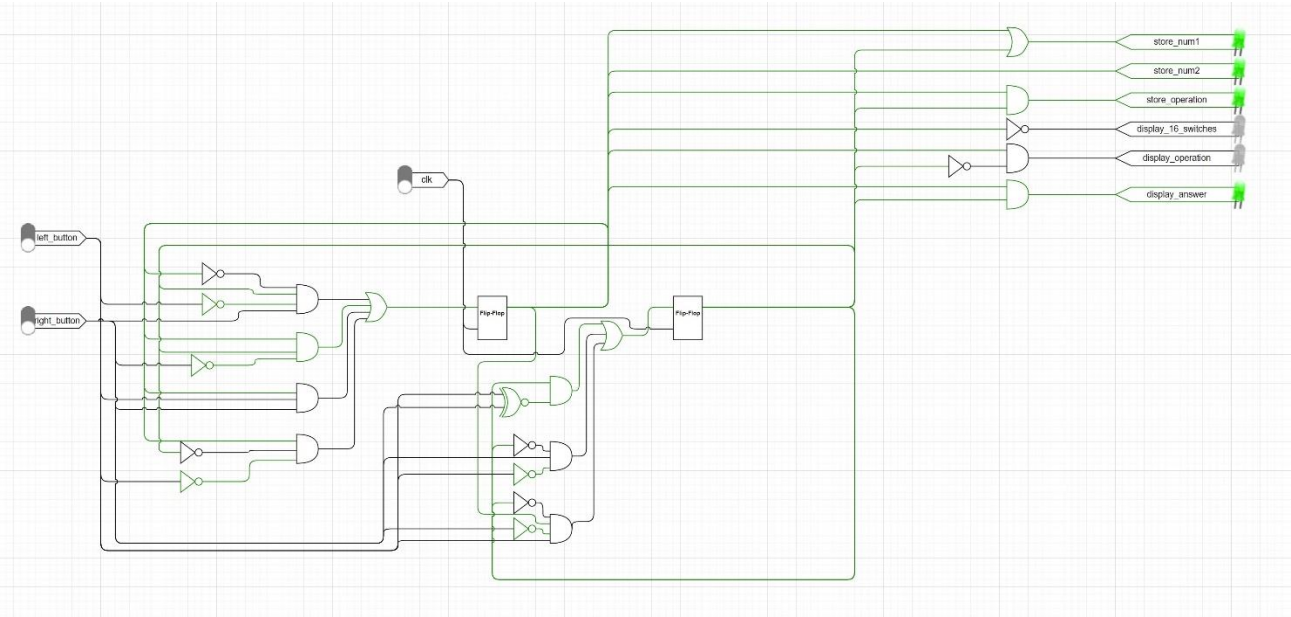
- store_num1 – sent to Data Collector module to trigger the storage of num1 at the transition from Stage 1 to Stage 2.
- store_num2 – sent to Data Collector module to trigger the storage of num2 at the transition from Stage 2 to Stage 3.
- store_operation – sent to Data Collector module to trigger the storage of the operation at the transition from Stage 3 to Stage 4.
- display_16bit_switches – sent to Seven Segment Display module to set the output on the seven-segment display to the 16-bit *sw* value Stages 1 and 2.
- display_operation – sent to Seven Segment Display module to set the output on the seven-segment display to the operation value on *sw[2:0]* during Stage 3.
- display_32bit_answer – sent to Seven Segment Display module to set the output on the seven-segment display to the 32-bit answer from the Calculation module.

FSM:

Present_Stage	left_button	right_button	Next_Stage	store_num1	store_num2	store_operation	display_switches	display_operation	display_answer
00	0	0	00	0	0	0	1	0	0
	0	1	01						
	1	0	00						
	1	1	00						
01	0	0	01	1	0	0	1	0	0
	0	1	10						
	1	0	00						
	1	1	01						
10	0	0	10	1	1	0	0	1	0
	0	1	11						
	1	0	01						
	1	1	10						
11	0	0	11	1	1	1	0	0	1
	0	1	00						
	1	0	10						
	1	1	11						

The binary encoding in Present_Stage and Next_Stage is 00 for Stage 1, 01 for Stage 2, 10 for Stage 3, and 11 for Stage 4. Note that if both or none of the buttons are pressed, the Stage will remain constant. Also note that when the right button is pressed at Stage 4, the user will move to Stage 1, however at Stage 1 the left button will keep the user at Stage 1.

FSM Implementation:



This schematic is implemented in the Stage Selector verilog module, except that the left_button and right_button in this schematic are the debounced previous_stage_button and debounced next_stage_button respectively.

3. Calculation

The Calculation module receives two numbers and an operation code from the 'Data Collector' module. Addition, subtraction, and multiplication is performed on the numbers by the three submodules. The operation code controls a Bus Mux which chooses which 32 bit sum, difference, or product will get sent out as the answer to be displayed.

The operation is also used to control the negative indicator output. When the operation is subtraction, and the subtraction difference is negative, then OUT_is_negative will be a 1-state.

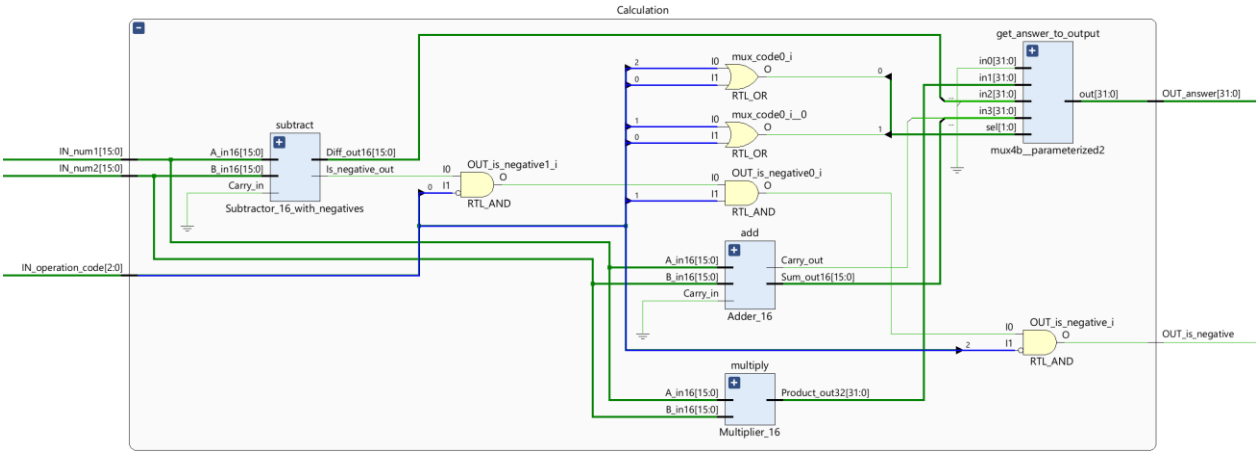
Inputs:

- IN_num1[15:0] – the first number to be operated on by the calculator received from num1 stored by the 'Data Collector' module
- IN_num2[15:0] – the second number to be operated on by the calculator received from num2 stored by the 'Data Collector' module
- IN_operation_code – the operation to be used by the calculator received from the operation stored by the 'Data Collector' module

Outputs:

- OUT_answer[31:0] – sent to the 'Seven Segment Display' module as the answer to be displayed
- OUT_is_negative – is wired to *led[10]* to inform the user if the answer is a negative value or not

Schematic:



Submodules:

a. 16 Bit Adder

Adds a 16-bit number to another 16-bit number resulting in a 16-bit sum and a Carry added onto the end as the 17th sum bit. Is composed of four '4 Bit Adder' modules linked together.

Inputs:

- A_in16[15:0] – received from IN_num1
- B_in16[15:0] – received from IN_num2; gets added to A_in16
- Carry_in – used when the 16-bit module is used in conjunction with other adder modules

Outputs:

- Sum_out16[15:0]- The 16-bit sum from adding A_in16 and B_in16, is sent to OUT_answer when the IN_operation code is addition.
- Carry_out- will be a 1-state if the sum of A_in16 and B_in16 can't be contained in a 16-bit number.

Submodules:

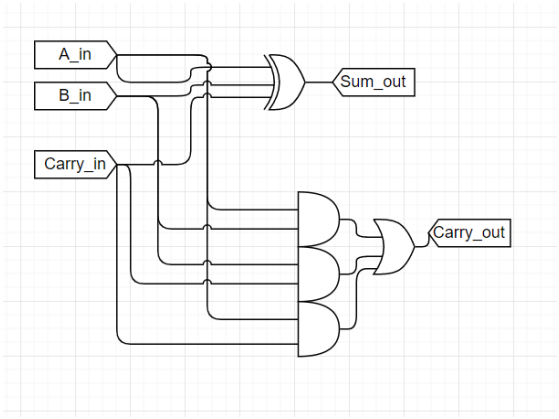
i. bitAdder

The bitAdder adds a 1-bit number to another 1-bit number.

Truth Table:

A_in	B_in	Carry_in	Sum_out	Carry_out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Truth Table Implementation:



Sum_out=A_in ⊕ B_in ⊕ Carry_in

Carry_out= (A_in*B_in) + (A_in*Carry_in) + (B_in*Carry_in)

Note that for the rightmost bit Adder of num1 and num2, the Carry_in is set as 0. That is because that bitAdder doesn't have any carry from a different bitAdder. Any other bitAdder will take the Carry_in from the bitAdder directly to the right of it.

Inputs:

- A_in- First 1-bit number
- B_in- Second 1-bit number to be added to A_in
- Carry_in- Received from A_in and B_in of digit directly to right of current A_in and B_in. Will carry a 1 if the previous digits calculated a 1+1.

Outputs:

- Sum_out- The 1-bit sum of A_in and B_in
- Carry_out- Sends 1 to the next bitAdder in succession if it calculates 1+1.

ii. 4-Bit Adder

Combines four 1-bit adders to create a 4-bit adder module. This module can add two 4-bit numbers together and is the building block for the 16-bit adder.

Inputs:

- A_in4[3:0]- First 4-bit number
- B_in4[3:0]- Second 4-bit number to be added to A_in4
- Carry_in- Received the Carry_out of the preceding 4-bit adder. Will carry a 1 if the previous digits summed to larger than a 4-bit number.

Outputs:

- Sum_out4[3:0]- The 4-bit sum of A_ in4[3:0] and B_ in4[3:0]
- Carry_out- Sends 1 to the next 4-bit Adder in succession if Sum calculates 1+1.

b. 16 Bit Subtracter

Subtracts a 16-bit number from another 16-bit number. All previous subtractor modules stored their outputs using 2's compliment. This module uses 4-bit subtractors to get the difference stored in 2's compliment. Afterwards, the difference is converted to represent the binary value of the difference's absolute value. The final carry of the subtractor modules indicate if the overall difference is negative, with a 1-state when the difference is negative and is in 0-state when the difference is positive.

Inputs:

- A_in16[15:0] – received from IN_num1
- B_in16[15:0] – received from IN_num2; gets subtracted from A_in16
- Carry_in – comes from any previous subtraction modules; will always be 0-state in this use case.

Outputs:

- Diff_out[15:0] – the absolute value of A_in16 minus B_in16, is set to OUT_answer when the IN_operation code is subtraction.
- Is_negative_out – is 1-state when the difference is negative and is 0-state when the difference is positive. Will be sent to negative indicator LED when the chosen operation is subtraction.

Two's Complement Implementation:

This is how the 16-bit subtractor originally stores the difference: using two's complement. To convert the 2's compliment into the magnitude, when the negative bit is in the 1-state, the 16 2's compliment bits will be inverted, and then 1 will be added to that new value.

Signed Hexadecimal	Signed Value	Unsigned Value	Negative Bit (17th bit)	2's Complement (16 bits)
+FFFF	+65535	65535	0	1111 1111 1111 1111
+FFFE	+65534	65534	0	1111 1111 1111 1110
+FFFD	+65533	65533	0	1111 1111 1111 1101
…	…	…	…	….
3	3	3	0	0000 0000 0000 0011
2	2	2	0	0000 0000 0000 0010
1	1	1	0	0000 0000 0000 0001
0	0	0	0	0000 0000 0000 0000
-1	-1	131072	1	1111 1111 1111 1111
-2	-2	131071	1	1111 1111 1111 1110
-3	-3	131070	1	1111 1111 1111 1101
…	…	…	…	….
-FFFD	-65533	65539	1	0000 0000 0000 0011
-FFFE	-65534	65538	1	0000 0000 0000 0010
-FFFF	-65535	65537	1	0000 0000 0000 0001

Submodules:

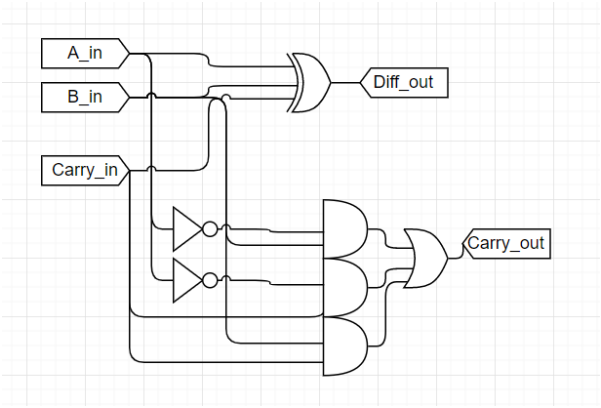
i. *bitSubtractor*

The purpose of the bitSubtractor is to subtract a 1-bit binary number B_in from another 1-bit binary number A_in. This bitSubtractor will subtract these two numbers according to the following truth table:

A_in	B_in	Carry_in	Diff_out	Carry_out
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

The borrow element is used when a 1 is subtracted from a 0, because the first number is unable to be subtracted by the second number and return a positive number. The borrow takes a 1 from the number directly to the left of it.

This is how 1-bit subtraction is logically implemented:



Diff_out=A_in ⊕ B_in ⊕ Carry_in
Carry_out= (A_in'*B_in) + (A_in'*Carry_in) + (B_in*Carry_in)

Note that for the rightmost bitSubtractor of num1 and num2, the Carry_in is set as 0. That is because that bitSubtractor doesn't have any carry from a different bitSubtractor. Any other bitSubtractor will take the Carry_in from the bitSubtractor directly to the right of it.

Inputs:
- A_in- First 1-bit number
- B_in- Second 1-bit number to be subtracted from A_in
- Carry_in- Received from the Carry_out of the preceding bitSubtractor. Will carry a -1/ borrow a 1 from next digit in succession if the previous digits calculated a 0-1.

Outputs:
- Diff_out- The 1-bit difference of A_in and B_in
- Carry_out- Sends -1 to the next bitSubstractor in succession if Diff calculates 0-1.

ii. 4-Bit Subtractor

The 4-bit subtractor combines four 1-bit subtractors to subtract a 4-bit number B_in4 from another 4-bit number A_in4. Four of the 4-bit subtractors combine to make the 16-bit subtractor

Inputs:
- A_in4[3:0]- First 4-bit number
- B_in4[3:0]- Second 4-bit number to be subtracted from A_in4
- Carry_in- Received from the Carry_out of the preceding 4-bit subtractor. Will carry a -1/ borrow a 1 from the number directly to the left if the previous digits calculated a 0-1.

Outputs:
- Diff_out4[3:0]- The 4-bit difference of A_ in4[3:0] minus B_ in4[3:0]
- Carry_out- Sends -1 to the next 4-bit substractor in succession if calculates 0-1.

c. 16 Bit Multiplier

Multiplies a 16-bit number by another 16-bit number. The method is the same way that multiplication would be done by hand on a binary number. The first bit of B_in16 is observed, and if it's a 1-state then A_in16 gets added to the total, otherwise nothing gets added. Then the second bit of B_in16 is observed, and if it's a 1-state then A_16 shifted one bit to the left gets added to the total, otherwise nothing gets added. This process repeats and A_16 keeps getting shifted. When the last bit of B_in16 is observed, if it's a 1 state then A_16 shifted 15 bits to the left gets added to the total. The addition of these numbers requires fifteen iterations of the 32-bit adder module.

Inputs:

- A_in16[15:0] – received from IN_num1
- B_in16[15:0] – received from IN_num2; gets multiplied by A_in16

Inputs:

- Product_out32[31:0] – the 32-bit product from multiplying A_in16 by B_in16 and is sent to OUT_answer when the IN_operation code is multiplicaiton.

Submodules:

i. 32-bit Adder

The 32-bit Adder module links two 16-bit adder modules together for the sole purpose of adding the larger addends required by the '16 Bit Multiplier' module.

Inputs:

- A_in32[31:0] – first 32-bit number
- B_in32[31:0] - second 32-bit number which is added to A_in32
- Carry_in – used for linking together adder modules and will always be 0-state in this use case.

Outputs:

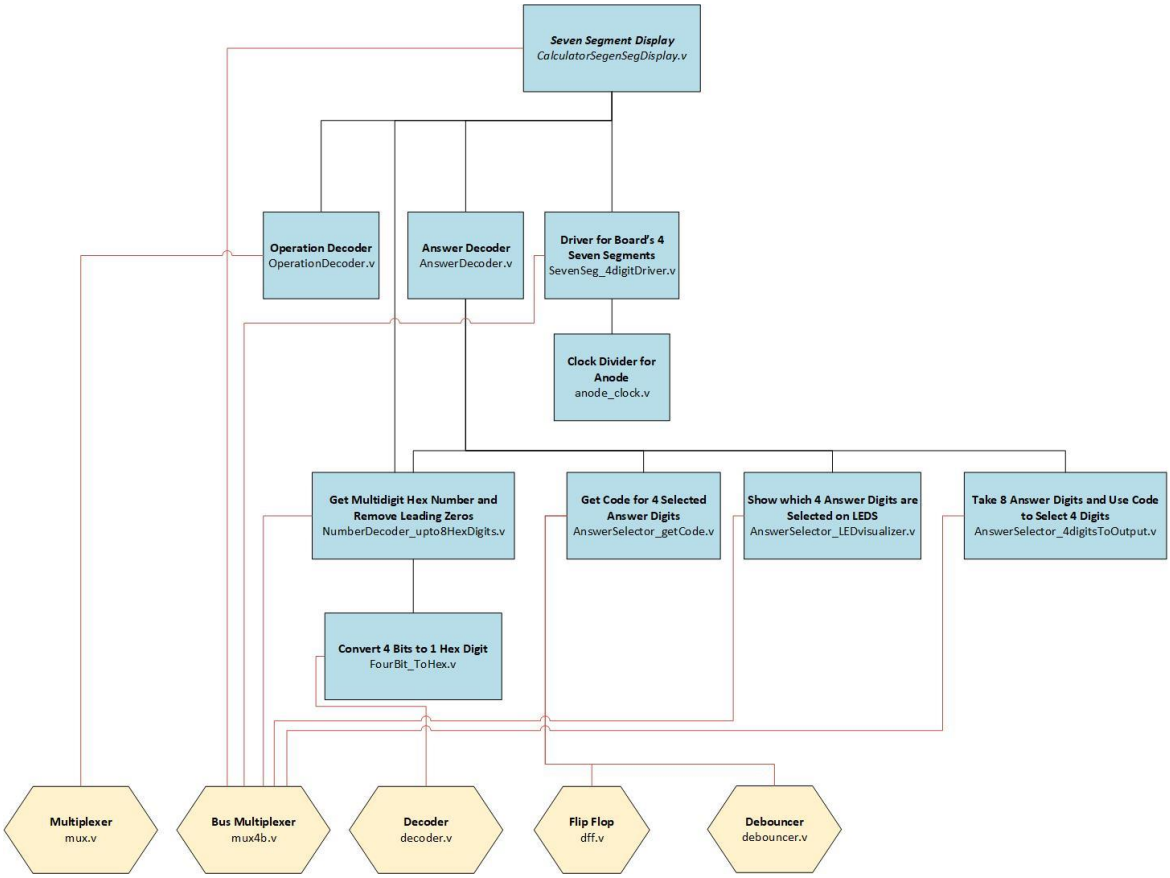
- Sum_out32[31:0] – sum of A_in32 and B_in32
- Carry_out – the carry from adding A_in32 and B_in32, will never be needed by the '16 Bit Multiplier' since the addends can't get large enough

4. Seven Segment Display

The Seven Segment Display module is responsible for displaying the correct data to the seven-segment display at the correct time, as well as driving the board's four seven-segment displays. Four Bus Multiplexers are used for each of the 4 seven-segment displays to choose which data is output based on the stage (seen in the column of four mux4b modules in the schematic controlled by show_code[1:0]).

In Stage 1 and 2 when IN_show_16bit_input is in 1-state, the binary value on the switches will be converted to a 4 hex digits using the Get Multidigit Hex Number and Remove Leading Zeros module. In Stage 3 when IN_show_operation is in 1-state, the operation from IN_operation_code[2:0] will be decoded in the Operation Decoder module to seven-segment letters ADD, SUB, MUL. In Stage 4, when IN_show_answer is in 1-state, the 32-bit answer will be converted to 8 hex digits in the Get Multidigit Hex Number and Remove Leading Zeros module. Four digits of the 8-digit hex answer will be shown, and the user can use the up, down, and center buttons to navigate through which digits will be displayed. 8 LEDS will also be used to show which 4 of the 8 answer digits are currently on the seven-segment displays to avoid ambiguity using 'Show which 4 Answer Digits are Selected on LEDS' module.

<u>Seven Segment Display Module Structure:</u>



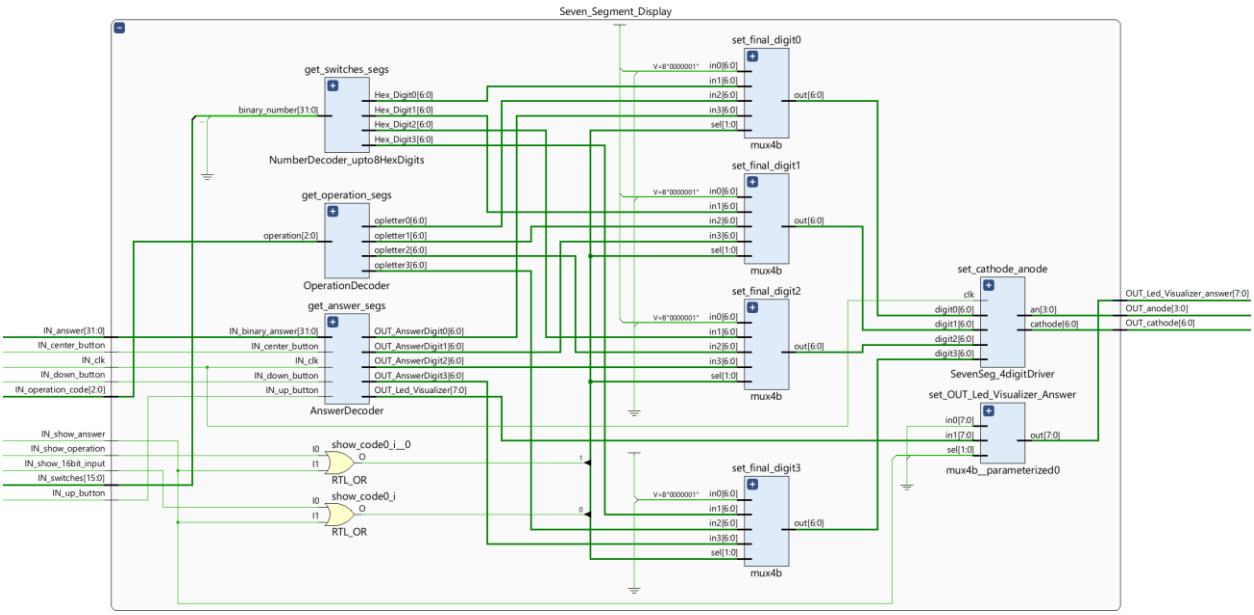
Inputs:

- IN_switches[15:0] – received from *sw*
- IN_operation_code[2:0] – received from *sw[2:0]*, allows modularity in what data can be used to control the operation
- IN_answer[31:0] – received from the answer output of the 'Calculations' module
- IN_show_16bit_input – received from the display_16bit_switches output of the 'Stage Selector' module; when in a 1-State, whatever is on the 16-bit switches will be displayed on the seven-segment display in hexadecimal.
- IN_show_operation – received from the display_operation output of the 'Stage Selector' module; when in a 1-State, whatever is on the 16-bit switches will be displayed on the seven-segment display in hexadecimal.
- IN_show_answer – received from the display_32bit_answer output of the 'Stage Selector' module; when in a 1-State, the 32-bit answer will be displayed on the seven-segment display as 8 hexadecimal digits the user can scroll through.
- IN_up_button – received from *btnU* to cycle through the 8 answer hex digits one digit at a time in the direction of the least-significant digit
- IN_down_button – received from *btnD* to cycle through the 8 answer hex digits one digit at a time in the direction of the most-significant digit
- IN_center_button – received from *btnC* and to toggle between showing the 4 most-significant and 4 least-significant answer digits
- IN_clk – comes directly from *clk* and is used to drive the Clock Divider for Anode as well as the FSM for the answer digit selection

Outputs:

- OUT_Led_Visualizer_answer[7:0] – sent to *led[7:0]*. All 8 bits are 0-state when the answer is not being shown. When the answer is shown, 4-bits are 1-state corresponding to the 4 hex answer digits being shown on the 4 seven-segment displays.
- OUT_cathode[6:0] – sent to *seg[6:0]* to control the cathodes of the board's seven-segment displays
- OUT_anode[3:0] – sent to *an[3:0]* to control the anodes on the board's seven-segment displays.

Schematic:



Submodules:

a. Get Multidigit Hex Number and Remove Leading Zeros

This module takes in a 32-bit binary value and outputs 8 hex digits. It is used for the 16-bit switch value and the 32-bit answer value. It operates by dividing up the 32-bit binary_number input into groups of 4 bits and sending them to the 'Convert 4 Bits to 1 Hex Digit' module 8 times.

As stated in the name, this module also removes leading zeros from the hex digits outputted. So, if the hex digits would output on the seven-segment display 0059 for example, it instead outputs 59 and leaves the rightmost 2 seven-segment displays blank.

To remove leading zeros, the module keeps track of if there was a nonzero value behind each digit from digit 0 to digit 6 (there will never be a value behind digit 7) in the bus called is_value_behind[6:0]. It does this by ORing together each group of 4 bits which represent one of the 8 digits. For example, is_value_behind[3] will be true if any binary bit correspoinding to digits 4, 5, 6, 7 is 1-state. This would indicate that there is a nonzero value behind digit 3.

Lastly, the is_value_behind[6:0] is used to drive 7 Bus Muxes which choose if the original hex value of the digit, or 7'b1111111 (resulting in a blank seven-segment display) will be shown. For example, digit 5 will be shown if is_value_behind[4] is true, otherwise it will be blank. Note the least-significant digit will always be shown, because even if that value is zero, we want to show "0" on the right-most seven-segment display.

Inputs:

- binary_number[31:0] – any 32 bit value representing a number that needs to be converted to hex digits.

Outputs:

- Hex_Digit0[6:0] – hex digit converted from binary_number[3:0]
- Hex_Digit1[6:0] – hex digit converted from binary_number[7:4]
- Hex_Digit2[6:0] – hex digit converted from binary_number[11:8]
- Hex_Digit3[6:0] – hex digit converted from binary_number[15:12]
- Hex_Digit4[6:0] – hex digit converted from binary_number[19:16]
- Hex_Digit5[6:0] – hex digit converted from binary_number[23:20]
- Hex_Digit6[6:0] – hex digit converted from binary_number[27:24]
- Hex_Digit7[6:0] – hex digit converted from binary_number[31:28]

Submodules:

i. Convert 4 Bits to 1 Hex Digit

Takes in a 4-bit binary number and outputs its hex digit. 7 Decoders are used for each seven-segment display segment. When the particular segment is supposed to be set to 1-state for a particular number, that number's decoder pin is added to the OR gate. The result of the OR gate is wired to the Hex_Digit's segment output.

Inputs:

- four_bit[3:0] – 4-bit binary number
- en – enable for decoders; is always set to 1-state

Outputs:

- [6:0]Hex_Digit – hex digit decoded from four_bit input.

Truth Table:

four_bit[3]	four_bit[2]	four_bit[1]	four_bit[0]	CA	CB	CC	CD	CE	CF	CG	Hex value
0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	1	1	0	0	1	1	1	1	1
0	0	1	0	0	0	1	0	0	1	0	2
0	0	1	1	0	0	0	0	1	1	0	3
0	1	0	0	1	0	0	1	1	0	0	4
0	1	0	1	0	1	0	0	1	0	0	5
0	1	1	0	0	1	0	0	0	0	0	6
0	1	1	1	0	0	0	1	1	1	1	7
1	0	0	0	0	0	0	0	0	0	0	8
1	0	0	1	0	0	0	0	1	0	0	9
1	0	1	0	0	0	0	0	0	1	0	a
1	0	1	1	1	1	0	0	0	0	0	b
1	1	0	0	0	1	1	0	0	0	1	C
1	1	0	1	1	0	0	0	0	1	0	d
1	1	1	0	0	1	1	0	0	0	0	E
1	1	1	1	0	1	1	1	0	0	0	F
			Hex_Digit	[0]	[1]	[2]	[3]	[4]	[5]	[6]	

b. Operation Decoder

Takes in the operation code, and outputs the 4 letters that it encodes (for the seven-segment display). Operation code 001 outputs the letters ADD, operation 010 outputs SUB, operation 100 outputs MUL. If the operation code is invalid, the output will be four blank values. The module only needs 7 regular multiplexers since the bit patterns repeat regularly as seen in the truth table.

Inputs:

- operation[2:0] – the encoded operation from IN_operation_code

Outputs:

- [6:0]opletter0 – an encoded letter for a seven-segment display
- [6:0]opletter1
- [6:0]opletter2
- [6:0]opletter3

Truth Table:

operation	CA0	CB0	CC0	CD0	CE0	CF0	CG0	Letter
000	1	1	1	1	1	1	1	off
001	1	0	0	0	0	1	0	d
010	1	1	0	0	0	0	0	b
100	1	1	1	0	0	0	1	L
opletter0	[0]	[1]	[2]	[3]	[4]	[5]	[6]	

operation	CA1	CB1	CC1	CD1	CE1	CF1	CG1	Letter
000	1	1	1	1	1	1	1	off
001	1	0	0	0	0	1	0	d
010	1	0	0	0	0	0	1	U
100	1	0	0	0	0	0	1	U
opletter1	[0]	[1]	[2]	[3]	[4]	[5]	[6]	

operation	CA2	CB2	CC2	CD2	CE2	CF2	CG2	Letter
000	1	1	1	1	1	1	1	off
001	0	0	0	1	0	0	0	A
010	0	1	0	0	1	0	0	S
100	0	0	0	1	1	0	1	M
opletter2	[0]	[1]	[2]	[3]	[4]	[5]	[6]	

operation	CA3	CB3	CC3	CD3	CE3	CF3	CG3	Letter
000	1	1	1	1	1	1	1	off
001	1	1	1	1	1	1	1	off
010	1	1	1	1	1	1	1	off
100	0	0	1	1	0	0	1	M
opletter3	[0]	[1]	[2]	[3]	[4]	[5]	[6]	

001 = Add	010 = SUb	100 = MUL
Note: all other operation combinations will result in the letter being "off"		

c. Answer Decoder

The Answer Decoder first uses the 'Get Multidigit Hex Number and Remove Leading Zeros' module on the 32-bit answer from IN_answer. The purpose of the rest of the 'Answer Decoder' module is select which 4 hex digits of the 8-hex-digit answer will be shown on the 4 seven-segment displays on the board. The 'Get Code for 4 Selected

Answer Digits' module will get the answer_select_code[3:0]. The 'Show which 4 Answer Digits are Selected on LEDS' module will use the code to show the selected digits. The 'Take 8 Answer Digits and Use Code to Select 4 Digits' module will take the 8-hex digits for the answer and output 4 of those hex digits to be displayed on the board's seven segment display all based on answer_select_code.

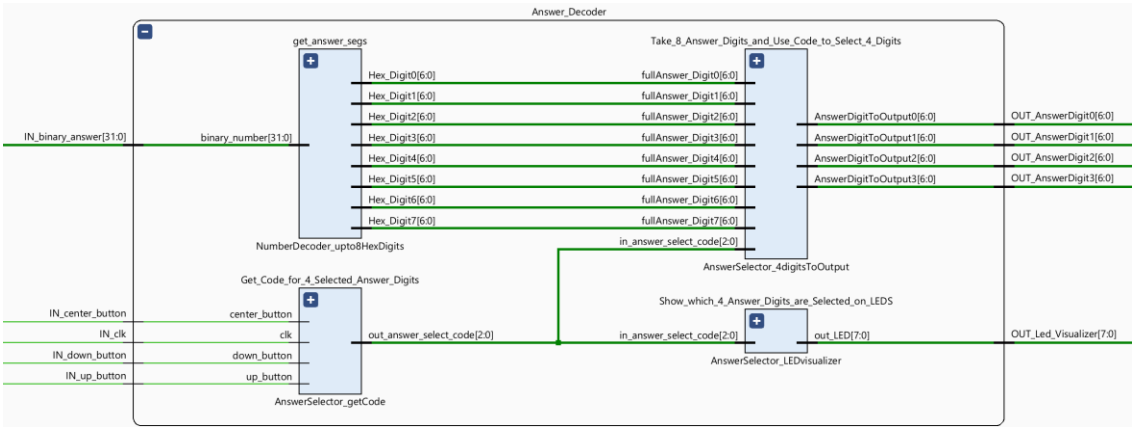
Inputs:

- IN_binary_answer[31:0] – answer value from IN_answer
- IN_up_button – from IN_up_button
- IN_center_button – from IN_down_button
- IN_down_button – from IN_down_button
- IN_clk – from IN_clk

Outputs:

- OUT_Led_Visualizer[7:0] – goes to OUT_Led_Visualizer_answer
- OUT_AnswerDigit0[6:0] – chosen answer hex digit to be on board's rightmost seven-segment display
- OUT_AnswerDigit1[6:0] – chosen answer hex digit to be on board's center-right seven-segment display
- OUT_AnswerDigit2[6:0] – chosen answer hex digit to be on board's center-left seven-segment display
- OUT_AnswerDigit3[6:0] – chosen answer hex digit to be on board's leftmost seven-segment display

Schematic:



Submodules:

i. Get Code for 4 Selected Answer Digits

The purpose of this module is to get a code which will be saved to answer_select_code[2:0] in the 'Answer Decoder' module. This answer code can range from 0 to 4 in binary. It directly controls which 4 of the 8 answer digits are shown on the board's seven segment display as shown below:

Usage of answer_select_code

	answer hex digit							
answer_select_code	7	6	5	4	3	2	1	0
000					X	X	X	X
001				X	X	X	X	
010			X	X	X	X		
011		X	X	X	X			
100	X	X	X	X				

answer digits with an X will be outputted to the seven-segment display for the corresponding answer_select_code

The up, down, and center buttons let the user change the answer_select_code.

Inputs:

- up_button – from IN_up_button; decreases answer_select_code by 1 for every press
- down_button – from IN_down_button; increased answer_select_code by 1 for every press
- center_button – from IN_center_button; toggles between answer_select_code being 000 and 100.
- clk – from IN_clk, drives button debouncer and the FSM

Outputs:

- out_answer_select_code[2:0] – sent to the 'Show which 4 Answer Digits are Selected on LEDS' and 'Take 8 Answer Digits and Use Code to Select 4 Digits' modules

FSM

answer_code q[2:0]	up button	down button	next_code d[2:0]
000	0	0	000
	0	1	001
	1	0	000
	1	1	000
001	0	0	001
	0	1	010
	1	0	000
	1	1	001
010	0	0	010
	0	1	011
	1	0	001
	1	1	010
011	0	0	011
	0	1	100
	1	0	010
	1	1	011
100	0	0	100
	0	1	100
	1	0	011
	1	1	100

Implemented Logic from FSM

Below is the sequential logic implemented in the final Verilog module. The ub, db, and cb stand for up button, down button, and center button respectively. These buttons are debounced before being wired to the sequential logic. Note that to make the problem easier to solve, the center button was able to be placed on the outside of each of the entire expressions. Pressing the center button should set d0 and d1 to 0-state, and the center button should toggle d2 on each press as defined in Inputs.

d0 = [q0(ub⊕db)' + q1q0'(ub⊕db) + q2'q1'q0'(ub'*db) + q2q1'q0'(ub*db')] * cb'
d1 = [(q1⊕q0)(ub' * db) + q1(ub⊕db)' + (q2+(q1q0))(ub * db')] * cb'
d2 = [q1q0(ub' * db) + q2 (ub * db')'] ⊕ cb

ii. Show which 4 Answer Digits are Selected on LEDS

Uses the answer_select_code to turn on the rightmost 8 LEDS in accordance with the selected answer digits. This is done with Bus Multiplexers where the LED patterns are the inputs, the answer_select_code is the select, and the output for the LEDs is the output. The LED's 1-state exactly matches the 'Usage of answer_select_code table'.

Inputs:

- in_answer_select_code[2:0] – received from 'Get Code for 4 Selected Answer Digits' module

Outputs:

- out_LED[7:0] – sent to OUT_Led_Visualizer_answer

iii. Take 8 Answer Digits and Use Code to Select 4 Digits

Uses exclusively Bus Multiplexers to select 4 answer digits to output from the 8 total answer digits. The answer_select_code goes to the select lines.

Inputs:

- fullAnswer_Digit0[6:0] – 0th answer hex digit received from "Get Multidigit Hex Number and Remove Leading Zeros" module
- fullAnswer_Digit1[6:0] – 1st answer hex digit received from "Get Multidigit Hex Number and Remove Leading Zeros" module
- fullAnswer_Digit2[6:0] – 2nd answer hex digit received from "Get Multidigit Hex Number and Remove Leading Zeros" module
- fullAnswer_Digit3[6:0] – 3rd answer hex digit received from "Get Multidigit Hex Number and Remove Leading Zeros" module
- fullAnswer_Digit4[6:0] – 4th answer hex digit received from "Get Multidigit Hex Number and Remove Leading Zeros" module

- fullAnswer_Digit5[6:0] – 5th answer hex digit received from “Get Multidigit Hex Number and Remove Leading Zeros” module
- fullAnswer_Digit6[6:0] – 6th answer hex digit received from “Get Multidigit Hex Number and Remove Leading Zeros” module
- fullAnswer_Digit7[6:0] – 7th answer hex digit received from “Get Multidigit Hex Number and Remove Leading Zeros” module
- in_answer_select_code[2:0] – answer select code received from “Get Code for 4 Selected Answer Digits” module

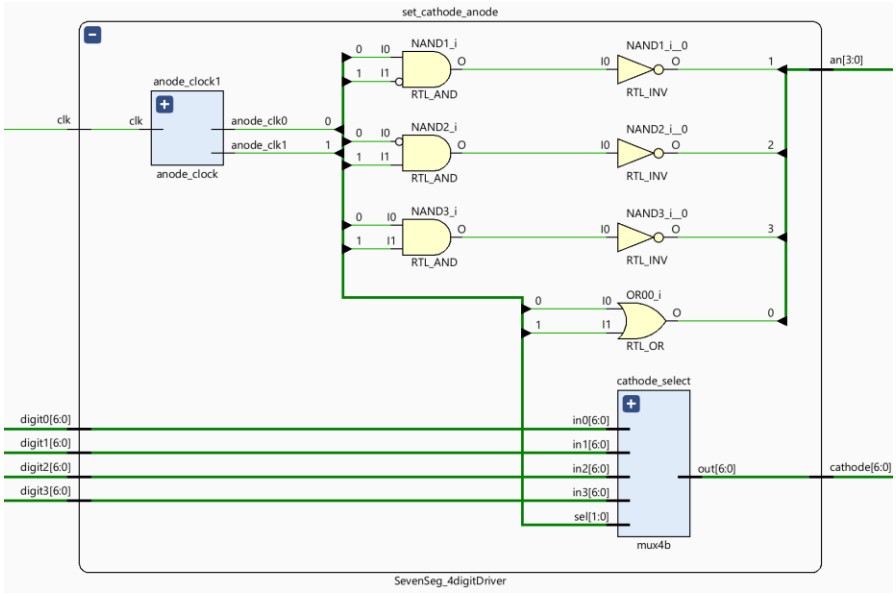
Outputs:

- AnswerDigitToOutput0[6:0] – answer hex digit to be shown on rightmost seven-segment display
- AnswerDigitToOutput1[6:0] – answer hex digit to be shown on center-right seven-segment display
- AnswerDigitToOutput2[6:0] – answer hex digit to be shown on center-left seven-segment display
- AnswerDigitToOutput3[6:0] – answer hex digit to be shown on leftmost seven-segment display

d. Driver for 4 Seven Segment Display Digits on the Board

This module completes the final step in outputting data on the board’s seven-segment displays. It takes in 4 buses called digits, each corresponding to one of the board’s seven-segment display. A 2-bit bus called anode_clk cycles from 00 to 01 to 10 to 11 back to 00, transitioning every 2.6ms coming from the ‘Clock Divider for Anode’ module. These binary values correspond with which anode will be activated, and during that time the 3 other anodes will be turned off. At the same time, the anode_clk is used in a Bus Multiplexer to controller which of the 4 digits is sent to the cathode.

Schematic:



Inputs:

- clk – received from IN_clk
- digit0[6:0] – controls what is displayed on rightmost seven-segment display
- digit1[6:0] – controls what is displayed on center-right seven-segment display
- digit2[6:0] – controls what is displayed on center-left seven-segment display
- digit0[6:0] – controls what is displayed on leftmost seven-segment display

Outputs:

- cathode[6:0] – sent to OUT_cathode
- an[3:0] – sent to OUT_anode

Submodules:

i. Clock Divider for Anode

Divides the board's 10 MHz clock into a 2 bit cycler that transitions every 2.6ms. This is accomplished by counting from 0 to (2^19)-1 in a 19-bit number and cycling back to 0 afterwards. The 19-bit number is incremented by 1 every 10 MHz clock rising edge. Next, bits 19 and 18 of that 19-bit number are sent as the anode_clock.

Inputs:

- clk – from IN_clk, runs at 10 MHz oscillations

Outputs:

- anode_clk[1:0] – sent to 'Driver for 4 Seven Segment Display Digits on the Board' as a 2 bit cycler controlling which of the 4 anodes is turned on.