Project 2 Report: Sequential 8-bit Multiplier using FPGA board Project

Andrew Antoine - Mostafa Elfaggal - Kirolous Fouty - Youssef Elhagg

The American University in Cairo

CSCE 230/2301 - DD1-SP23

Dr. Mohamed Shalan

27/05/2023

Author Note

This paper is prepared for the CSCE 2301 course instructed by Professor Mohamed Shalan

Mostafa B. Elfaggal
Department of Computer Science & Engineering, The American University in Cairo
CSCE 230/2301: Digital Design I
Dr. Mohamed Shalan
mostafaelfaggal@aucegypt.edu

Kirolous A. Fouty
Department of Computer Science & Engineering, The American University in Cairo
CSCE 230/2301: Digital Design I
Dr. Mohamed Shalan
kirolous_fouty@aucegypt.edu

Andrew A. Ishak
Department of Computer Science & Engineering, The American University in Cairo
CSCE 230/2301: Digital Design I
Dr. Mohamed Shalan
andrew625@aucegypt.edu

Youssef Elhagg
Department of Computer Science & Engineering, The American University in Cairo
CSCE 230/2301: Digital Design I
Dr. Mohamed Shalan
yousseframi@aucegypt.edu

OUTLINE

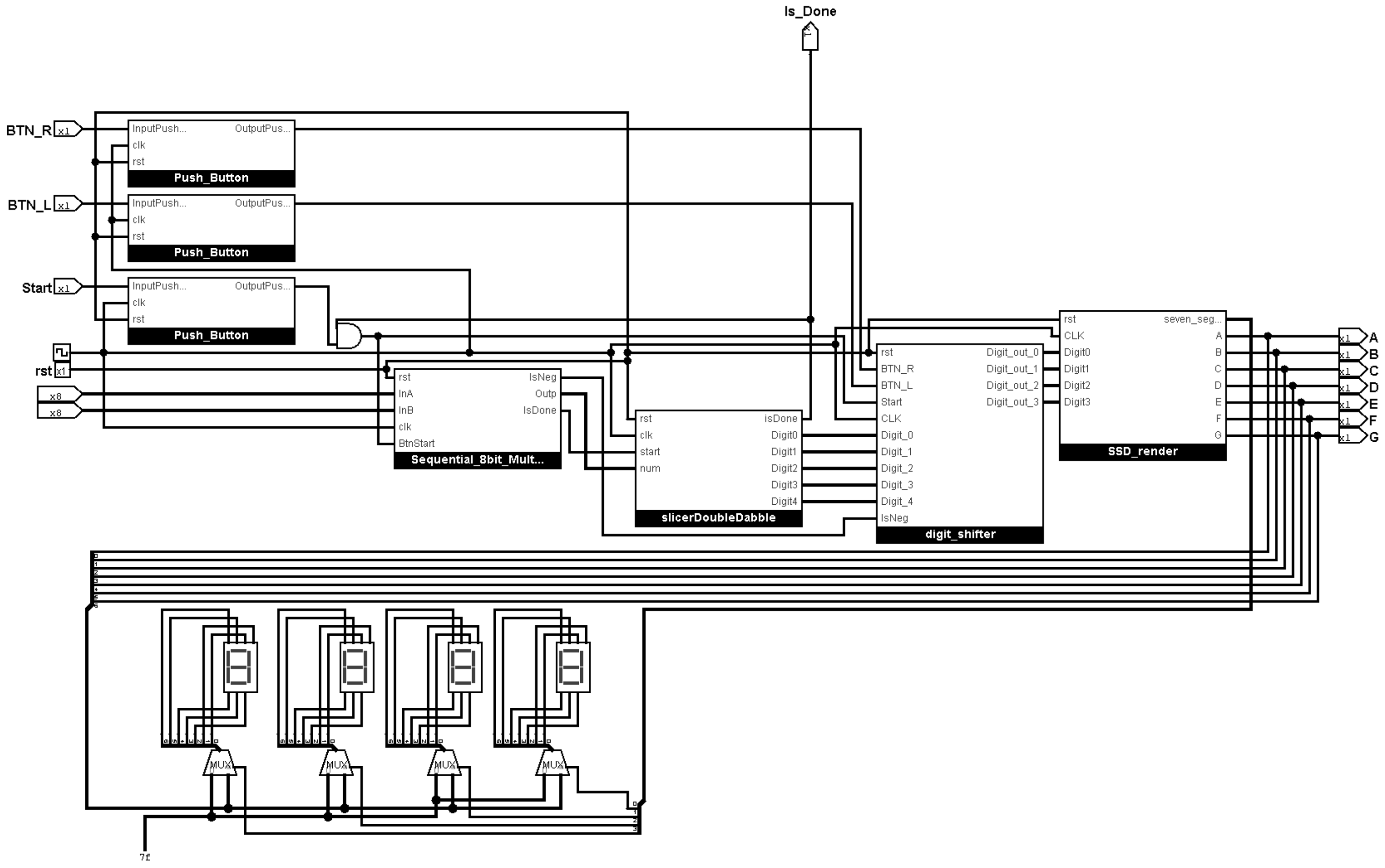
I - Introduction

This project is an implementation of a sequential 8-bit Multiplier using an FPGA board. It was implemented in a set of modules. The following program was divided into four main modules. The first module is the Signed Sequential 8-bit Multiplier; it is responsible for taking the two signed 8-bit inputs and outputs the 16-bit unsigned multiplication. The second module is the Binary to BCD module; it takes the 16-bit unsigned multiplication done by the multiplier and outputs five 4-bit digits to be displayed on the seven-segment display. The third module is the Digit Shifter module, it takes the five 4-bit digits provided from the Binary to BCD module, shifts left and right inputs and selects the four to be displayed. The fourth module is the Seven-Segment Display (SSD Render) module, it takes the four digits, already selected by the Digit Shifter module, and converts them to the necessary segments to be visually displayed on the Seven-Segment Display.

II - Outlining the Design & Work Structure

A - Project Top Module

Verilog File: Project.v

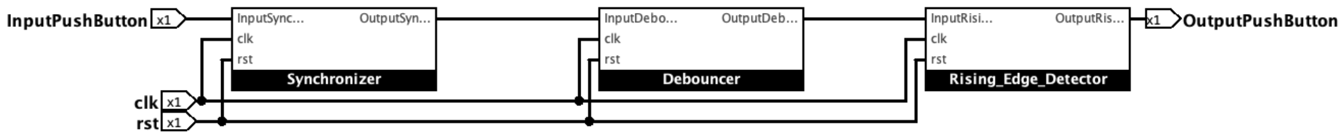


Description:

At the top level module, this is where everything comes together to complete the logic of the circuit. The top level module takes as inputs the two 8-bit signed binary numbers, the start signal coming from the button, the shift right and left signals also coming from the buttons, as well as the basic clock and reset signals. The circuit outputs the 7 signals for the Seven-Segment Display, and an IsDone signal indicating the end of the multiplication. Also, the start button signal is wired up with the IsDone signal, to ensure that no new multiplication will take place unless any ongoing one is complete.

B - Push Button Module

Verilog File: push_button.v

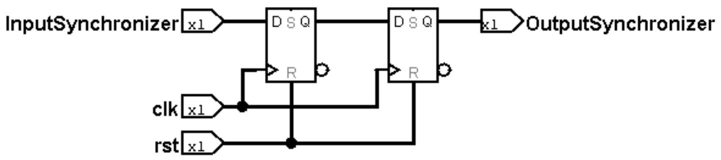


Description:

The push button module is a rising edge detector that converts human input to a tick pulse that is in sync with the clock. It does this through 3 phases. Phase 1 is the synchronizer. Then phase 2 is the debouncer. Before the final phase of the rising edge detector. The first phase is needed whenever any two systems of different clocks interact, such as in this case. The physical button has the human clock, while the multiplier system has its own clock. The second phase is needed since the button is a physical component which requires debouncing before being used in a structured system. The final phase implements the core functionality of a push button which reduces the human push down to a one-tick pulse.

1) Synchronizer

Verilog File: synchronize.v

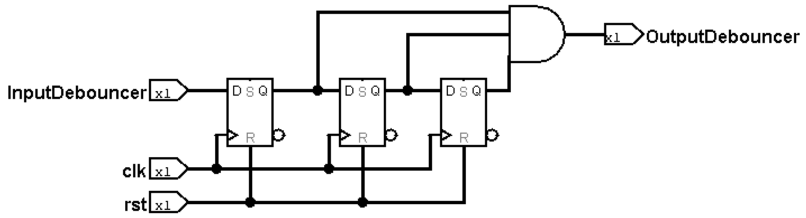


Description:

The Synchronizer module is composed of two D-flip flops that takes an asynchronous input signal, and a clock signal and outputs a synchronized signal. This system is needed to unify the two clocks: the human input and the clock of the circuit. The first flip-flop samples the signal at the clock rising edge and the output may be metastable if the signal is changing at the moment. The second flip-flop samples the meta signal that is stable, and produces the unified output.

2) Debouncer

Verilog File: debouncer.v

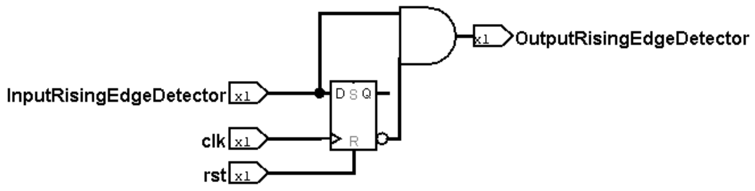


Description:

The Debouncer module is composed of three flip-flops that take the input signal and stabilize it. In the FPGA board, the switches are mechanical components that when pressed they might move back and forth several times after being pressed and before becoming stationary. Thus, the debouncing circuit is designed to eliminate these glitches brought on the switch transitions. It does this by adding an AND gate to the output of the three flip-flops to ensure that the signal is high in the three flip-flops; therefore, it ensures that the signal is stable.

3) Rising Edge Detector

Verilog File: rising_edge_detector.v

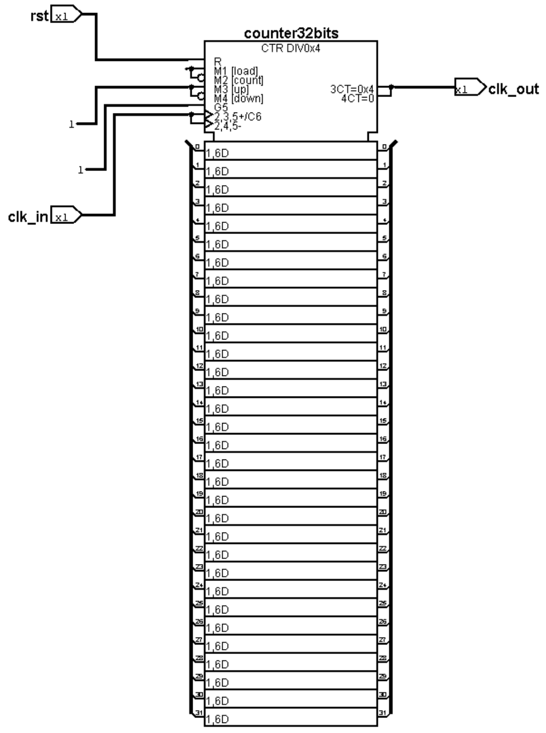


Description:

The Rising Edge Detector module is composed of a single flip-flop and an AND gate that takes the initial input and the output of the flip-flop inverted. When the input signal changes from 0 to 1, the rising-edge detector produces a brief one-clock-cycle tick. It is typically used to signal the beginning of a slowly changing input signal.

C - Clock Divider Module

Verilog File: clock_divider.v, mod_n_counter.v

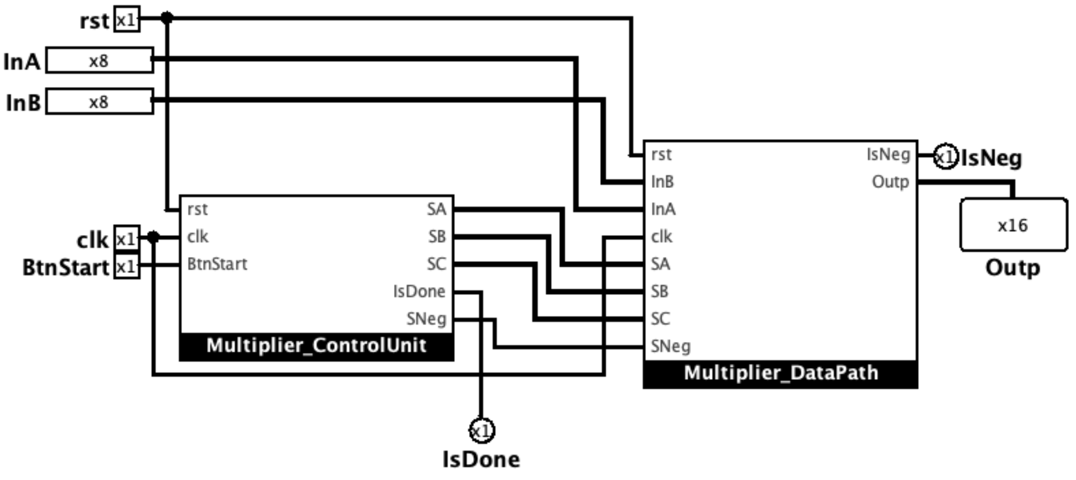


Description:

The Mod-n Counter module is used as a clock divider that counts from 0 to 31 before returning to 0. It has 32 states and it requires a flip flop for each bit of the binary representation of the 32 bits when implemented using a synchronous binary counter. The comparator output permits a synchronous reset when the counter reaches the N-1 state, setting the counter back to 0 on the subsequent clock cycle. In Verilog, the Mod-n counter is created in an Always block and using If conditions. The always block is configured to run whenever the clock cycle's positive edge occurs, mimicking a synchronous circuit. In this block, the count increases by 1 unless it reaches N-1, at which point it is reset to 0.

D - Signed Sequential Multiplier Module

Verilog File: signed_sequential_multiplier.v

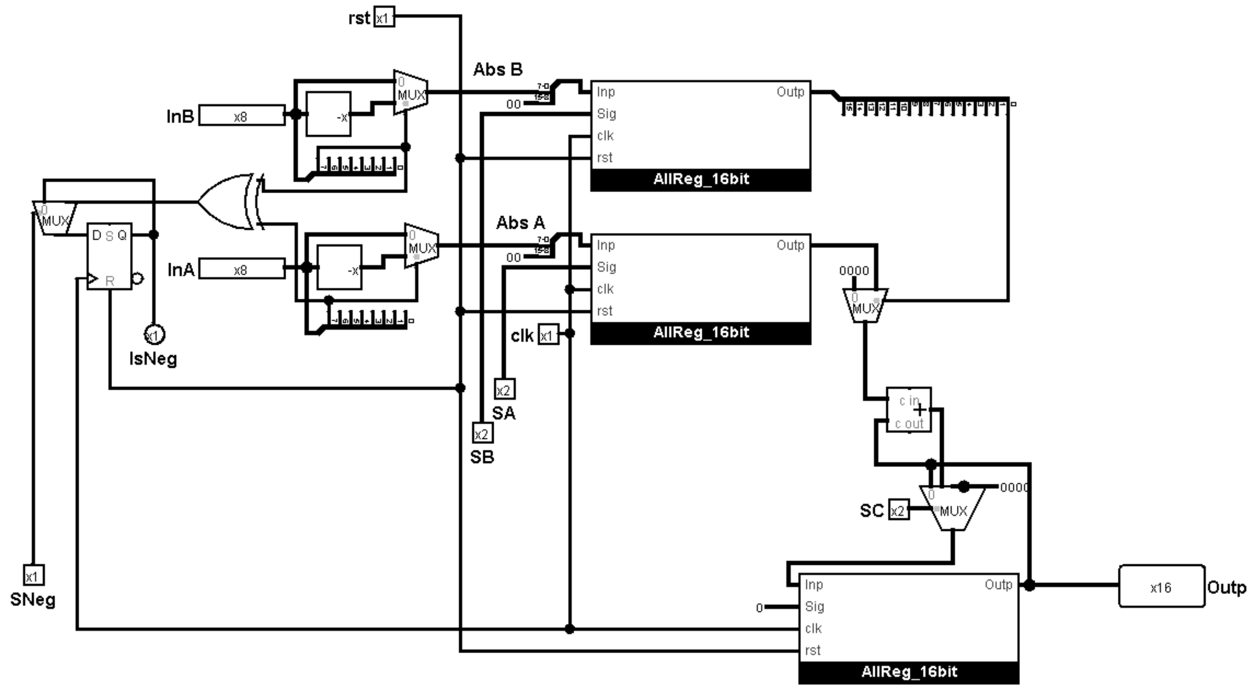


Description:

The signed sequential multiplier is a block which multiplies two signed 8-bit binary numbers (InA, InB). Designed in RTL design, it is broken down into a Control Unit and a Datapath. The inputs for the multiplier module in general are the two signed binary numbers, the clock (since it is a sequential module), and BtnStart which is a signal from the external to recognize the start of a computation (it is also to give time for the user to put the A, B input numbers). The module also has a reset signal like all other sequential modules, to reset all internal flip flops. The module outputs the output of the multiplication which is unsigned 16-bits (Outp), and a negative flag (IsNeg) which is true when the output is supposed to be negative, and an IsDone flag to represent whether the a calculation is taking place at the given moment or not. The IsDone seems always turned on when viewed on the FPGA, as the calculation process takes up 8 clock cycles, which is very fast for the human to see; hence it seems like the led never turns off. The RTL design breaks off the module into two parts, where the datapath does the actions of the algorithm, while the control unit supplies the right sequence of signals to perform the actions.

1) Multiplier Datapath

Verilog File: multiplier_DP.v



Logisim:

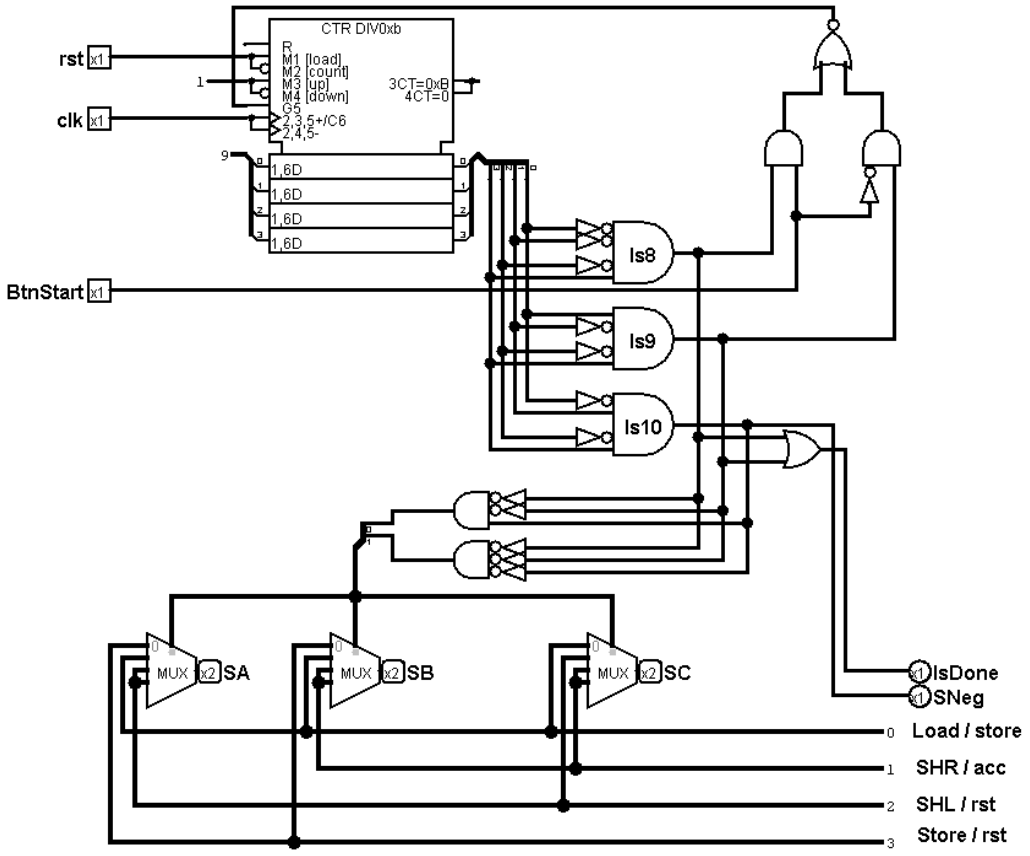
The datapath follows a template similar to that on the slides. It makes use of a module named AllReg_16bit, which is a 16 bit register that can perform one of 4 actions at a time: load, store, shift left, shift right. The action to perform is based on the signal provided. It also takes in the clock, reset, a load value as inputs. The datapath takes InA, InB as inputs from the multiplier top level module. It also takes in the reset, and clock signals. It also takes in a bunch of signals to control its actions, such as SA, SB, SC, SNeg. The datapath outputs the negative flag of the multiplier module, and the product (Outp). The process of the datapath starts at the entry of the two signed 8-bit numbers, which then goes through an absolute computation. That computation converts the inputs into a positive value, by another passing it as if it is already positive, or by computing its 2's complement and passing that if it is negative. We can identify if the value is positive or negative based on the most significant bit. Also the product's negativity can be computed by XOR the negativity of the two operands. However, that process is computed when the signal SNeg is provided, as the IsNeg shouldn't be computed combinationally since the inputs can change after a calculation is done. At which case, the IsNeg flag should retain its value until the new calculation is started. The Abs values of the operands are passed into 16-bit registers by padding them with 8 zeros to the left. After which the reg for operand A shifts left 8 times, while the reg for operand B shifts right 8 times. With every shift an accumulation in the product register takes place, where based on the least significant bit of regB, either we add the new value of regA to the product or maintain the value as is by accumulating zero onto the product. After the computation is complete the product can be found from the reg dedicated for it (far bottom left corner).

Verilog:

The datapath Verilog describes the wiring of the datapath, while also describing the AllReg_16bit module implicitly. It begins by computing AbsA and AbsB, which checks if it is positive or negative (most significant bit), and either assigns the input or its negative value. After which the always block describing A and B, case over their signals and perform the appropriate action based on the signal (load / shift right / shift left / store). When loading it takes in the input and concatenates 8 zeros to it on the left. For the product block with signal SC, the register either accumulates onto the product (add A if B[0] is 1 or 0 if B[0] is 0), or clears the product (load 0) which is needed during the init state. The signal can also dictate to store the value, which is the default case. The last always block computes the IsNeg when the SNeg signal is on. To compute IsNeg, we XOR the negativity of the two operands. Further we noticed a corner case, which is when one of the operands is 0, so to handle the IsNeg accordingly, we check if either of the operands is 0 or not.

2) Multiplier Control Unit

Verilog File: multiplier_CU.v



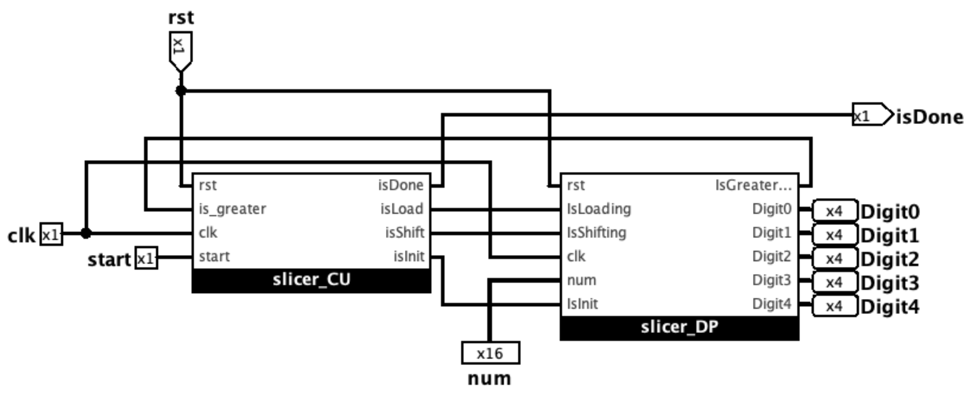
Logisim:

The control unit is responsible for producing the correct signals in order to perform the algorithm as needed. It was designed as an FSM, and the diagram for which was posted on github. The states are separated into 11 states, where the first 8 (0-7) are used for the 8 times in which the datapath shifts and accumulates. State 9 is the one where the multiplier waits for the input signal to come in from the outside to begin the calculation. State 10 is the init state, where the multiplier initializes all needed processes to begin its calculation (load the operands into their registers, clear out the old product, set the new IsNeg flag). State 8 is the end state, at which the multiplier waits for the start signal to return back to off so as to account for the human holding down the signal. This also is needed since the input start signal comes in with a slowed down clock, and thus its one clock cycle is many cycles in the multiplier's clock. The end state ensures that the multiplier wouldn't compute many iterations. Furthermore, the multiplier cannot use the button's clock since that would lead the multiplier to slow down, which is not a desired outcome. The state is computed using a counter, which can be reseted into 9 (wait for start). Some combinational logic is applied to compute to correct signals to pass to the datapath at every clock cycle. This logic wasn't optimized using k-maps, rather it makes use of some very easy to read and understand logic using multiplexers.

Verilog:

For the control unit, we use the standard structure for FSM. We start by an always block to compute the nextState. Then an always block that sets the presentState to the nextState with every clock cycle. Finally we compute the outputs (control signals). We assign the isDone and SNeg signals to their appropriate states. Then we break the signals SA, SB, SC into always blocks, each of which uses a case statement to describe the mapping of the state to their values.

E - Binary to BCD Module (Double Dabble Algorithm)



Description:

The Binary to BCD module is responsible for dividing the 16-bit result of the multiplication into 5 4-bit digits, so that they could be later on displayed on the Seven-Segment Display. For example, if the result of the multiplication is $(169)_{10}$, which in binary is $(10101001)_2$, the output of the binary to BCD converter would be as follows: Digit0 = $(1001)_2$, Digit1 = $(0110)_2$, Digit2 = $(0001)_2$, and Digit3 = Digit4 = $(0000)_2$, where they represent the 9, the 6, the 1, and the Zeros respectively. Our first approach to solving this problem was to implement a Mod-10 algorithm.

Mod-10 algorithm: This algorithm utilizes the modspace of 10 to compute the 5 digits. The proof of concept has been uploaded on github. The concept relies on the fact that the only part of the 16 bit number that could affect the units digit for example, are the units part of every bit of the 16 summed up. For instance, take the number 23 = $(0000000000010111)_2$. The units digit of this number is 3, which is the sum of (6 + 4 + 2 + 1) mod 10, where these values are the unit part of each bit valued 1. To do this process the algorithm needs to make use of addition in the modspace of 10, So we created a module which adds up to numbers in the modspace of 10, and produces an outcome also in the modspace of 10. So (8 + 7) mod 10 = 5. Using this module, we structured an array where we sum up the units for each bit if it is 1, or sum up a zero. The carry outs are passed to the next level. So the carry out of the units is passed as carry in to the tens and so on and so forth. We manually set up the operands as constant representing the mod 10 portion of the bit. So at the adder representing the tens bit of 32, the constant operand is 3 as 4-bits $(0011)_2$.

However, upon further research, we learned about a much more efficient algorithm called Double Dabble.

How it works:

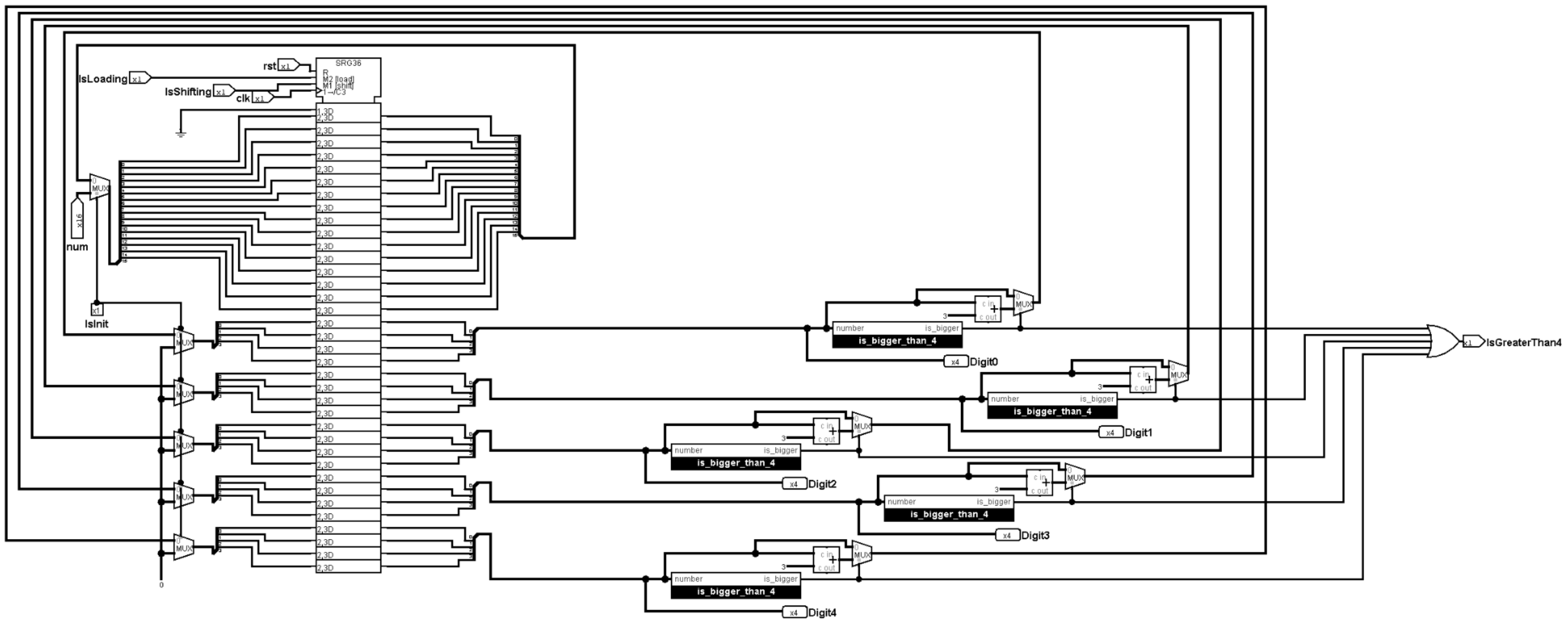
Each bit of the binary number is shifted to the left, causing the most significant bit (MSB) to be transferred to the least significant bit (LSB) of the accumulating BCD (Binary Coded Decimal) number. Following each shift, all BCD digits are checked, and if any digit is 5 or greater, 3 is added to it. We continue this process until all bits of the binary number are shifted.

Why it works:

This approach works because a left shift doubles the value of all BCD digits. Since BCD digits cannot exceed nine, if a digit is already five or more before the shift, the resulting post-shift digit would be ten or more, which is not valid in BCD representation. By adding three to any BCD digit greater than five, two things happen: first, during the next shift, the added 3 becomes 6, accounting for the difference between BCD (which uses 10 binary codes) and binary (which uses 16); second, adding 3 causes the MSB of the BCD digit to become 1, effectively carrying it over to the next digit.

Inputs:
start: The start signal of the BCD module is a signal which turns on after the multiplier is done, so that the BCD is chained after the multiplication.
num: the 16-bit product from the multiplier
clk, rst are the standard clock and reset signals
Outputs:
isDone: whether or not the module has completed the process
Digitx: the 4-bit number representing the Digit

1) Datapath

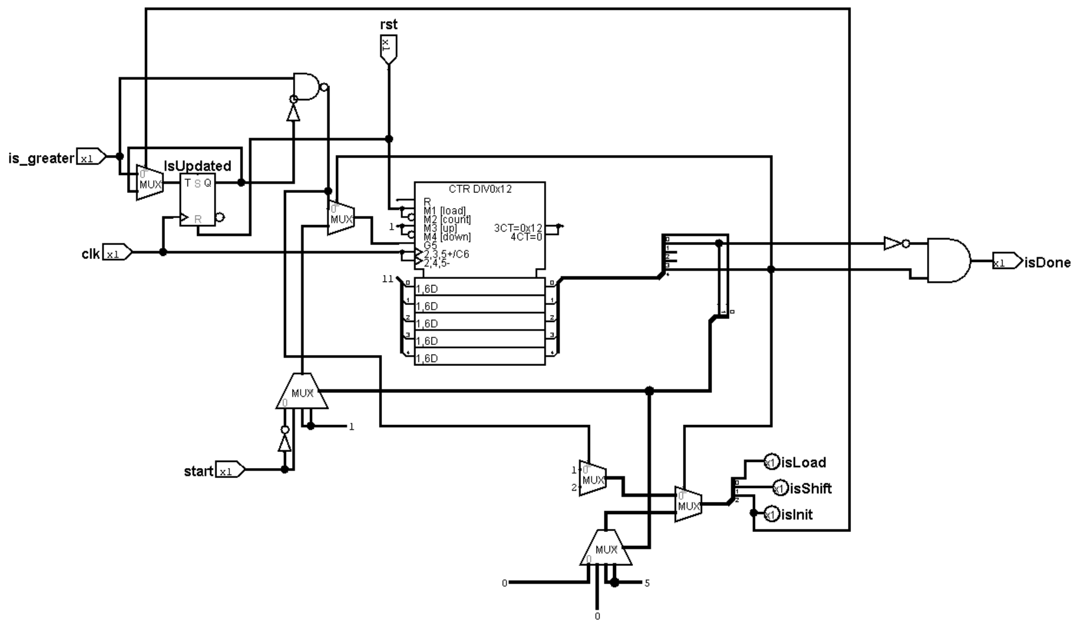


To implement the Datapath of the Binary to BCD converter, we used a 36-bit shift register, responsible for shifting the 16-bit binary input from its MSB, and into the LSB of the BCD digits. The other 20 bits are the 5 4-bit BCD digits, where each 4 bits of each digit are checked to see if they are bigger than 4 after every shift. If any digit is greater than 4, 3 is added to it. Additionally, the ORing of the BiggerThan4 signal for each digit is an output signal for the control unit, to determine whether to stop or continue shifting. The output of each BCD digit after being added 3 (or not), is re-inputted to the shift register. The process continues until all 16 bits are shifted.

The input control signals:

- isInit: responsible for signaling if the whole process needs to be initialized. When the isInit signal is on, the new 16-bit input is loaded into the shift register, and all of the 20 bits for the BCD digits become Zeros.While isInit, the shifting is disabled.
- isLoading: responsible for signaling if the shift register needs to load new inputs. Whether a new 16-bit input needs to be inputted to the register, or a BCD digit is bigger than 4 and thus needs to be re-inputted after adding 3.
- isShifting: responsible for signaling to the shift register whether or not to keep shifting. The isShifting signal is off if new input needs to be loaded to the register.
- Clock and reset signals

2) Control Unit



The control unit is designed as an FSM. States 0-15 handle shifting the bits into the 20-bit outcome. During the shifting process, the counter might hold it incrementing, if an addition of 3 has to be done. This is identified using the DFF named isUpdated and the is_greater flag from the datapath. The control unit sends the signal to load (isLoading) which leads the datapath to update the digits which are greater than 4 in the following cycle. After which the control unit using the logic surrounding the is_greater and isUpdated ensures not to re-update and get back to incrementing the counter so as to continue shifting. State 16 is the end state which awaits the start signal to turn off (accounting for human input, similar to that explained in the Multiplier). State 17 is the waiting for the start signal or the idle state, at which all registers hold their values until the start signal is passed, and the process begins. State 18 is the init state, where the registers are loaded and cleared out to the needed values before the process begins (loading the

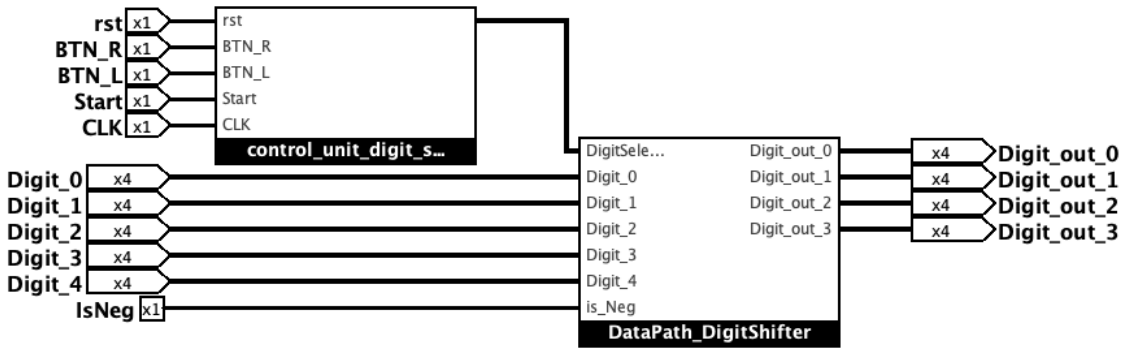
20-bits with zero, and the product into the first 16-bits. The isDone flag is only on during the end and wait for start states. The rest of the signals are produced in their corresponding states, such that during the incrementing of the counter for the bits, the isShifting is turned on. While if is updating as there is a set of 4-bits greater than 4, then the isLoading signal is outputted. And the isInit signal is outputted during the init state.

3) Verilog Combinational implementation

As mentioned above, we used a sequential circuit to implement the double dabble algorithm on Logisim. However, when implementing the algorithm in Verilog, we just used a for loop, to loop over the 16 bits from left to right. In every iteration, we check if any of the 5 BCD digits are bigger than 4, if that is the case, we add 3 to its value and re-assign it using normal blocking assignment. The shifting of the BCD digits is implemented by concatenating the MSB of the binary input to the LSB of the BCD digits, excluding the MSB of the BCD, and re-assigning the result to the BCD using blocking assignment. Following this method, the algorithm is implemented combinationally rather than sequentially, which is much more time efficient.

F - Digit Shifter Module

Verilog File: DigitShifter.v



Description:

The digit shifter, as the name suggests, is responsible for shifting the digits right and left, selecting which four digits should be displayed on the Seven-Segment Display. When multiplying 8 bits by 8 bits, the maximum number of bits for the output would be 16 bits. The 16 bits in binary could be represented at maximum as 5 digits in decimal. For example, multiplying $(-128)_{10}$ x $(-128)_{10}$ yields $(16,384)_{10}$. Moreover, as another example, the result of the multiplication could be a negative number, i.e. $(-239)_{10}$. Since we only have 4 displays, it remains to figure out how we could display these digits on the Seven-Segment Displays. We then use the digit shifter in order to shift left and right between the digits so that the user can see the whole result.

<u>How it works:</u>

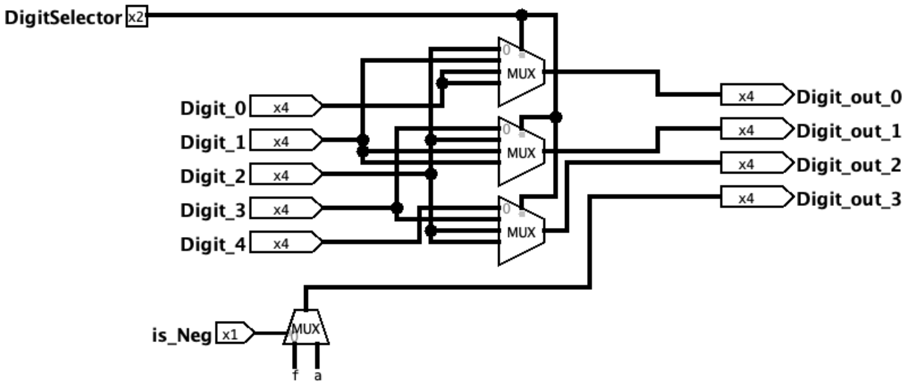
Let us take as an example the number $(-12345)_{10}$:

At the initial state, the first 3 digits from the right are displayed, the leftmost display is always used to display the sign. This is what the user sees on the Seven-Segment display:

Initial State => -345
Shift left => -234
Shift left => -123
Shift left => -123 (reached the end so stays the same)
Shift right => -234
And so on…

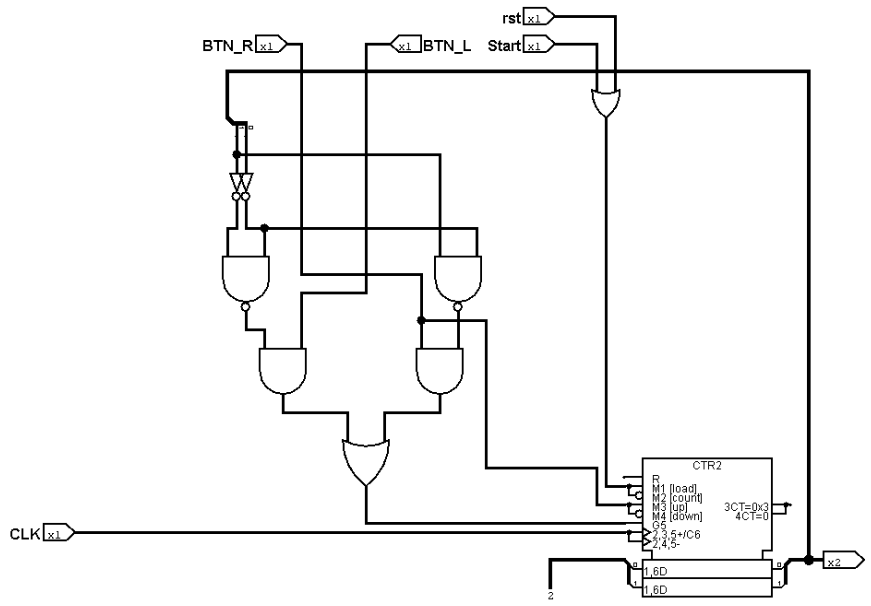
<u>Logisim:</u>

1) Datapath



The Datapath of the digit shifter we implemented is combinational. It takes as an input the 5 4-bit digits coming from the binary to BCD converter, as well as the is_Neg signal which indicates whether the product is negative or positive (1 if negative 0 if positive). We used three 4-bit 4x1 MUXs, one for each digit to be displayed at every state. The selection line of all the MUXs (input from the Control Unit) corresponds to the state the shifter is currently in, or in other words, it corresponds to how many times the user pressed to shift right or left. There are three states in total: at the initial state, the DigitSelector is $(00)_2$ and the MUXs output digits 0,1, and 2. The second state, the DigitSelector is $(01)_2$, the MUXs output digits 1, 2, and 3. The Third state, the DigitSelector is $(10)_2$, the MUXs output digits 2, 3, and 4. Concerning the sign, the Is_Neg signal is the selection line of a 4th 4-bit 2x1 MUX, which outputs $(1010)_2$ if Is_Neg is 1 or $(1111)_2$ if Is_Neg is 0.

2) Control Unit



The control unit of the digit shifter is responsible for setting the state for the datapath. The control unit takes as inputs the shift right and left signals, the start signal, as well as the basic clock and reset signals. Simply enough, we used a mod-2 counter that counts up or down depending on the shift right and left signals. When the start signal is 1 (or the reset signal is 1), the counter gets loaded with 2 (opposite from Verilog). For the shift right and left signals, if the shift right signal is 1, the counter counts up to 2, or if the shift left signal is 1, the counter counts down to 0. The important thing here was to implement some combinational logic to stop the counter when cycling back from 0 after decrementing, or from 2 after incrementing. Meaning that we want the counter to stay at 2 even after signaling to shift right; or to stay at 0 even after signaling to shift left. The combinational logic was easily done using K-maps.

Verilog:

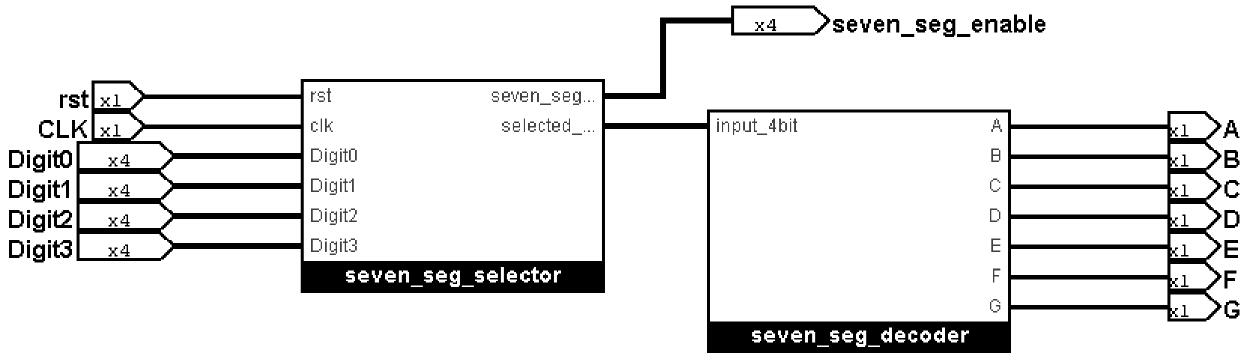
To implement the digit shifter in Verilog, we used a simple FSM to represent each state. For the shift right and left signals, if the shift right signal is 1 and the state is not $(00)_2$, we decrement the state by 1. On the other hand, if the shift left signal is 1 and the state is not $(10)_2$, we increment the state by 1. For each state, we concatenate the sign with the corresponding three digits of each state, and assign it to the output. We used a 2D array as a wire for the concatenation of the digits inside the module, and assigned it to a 16-bit output (four 4-bit digits). This is because Verilog does not allow for 2D inputs or outputs.

G - SSD Render Module

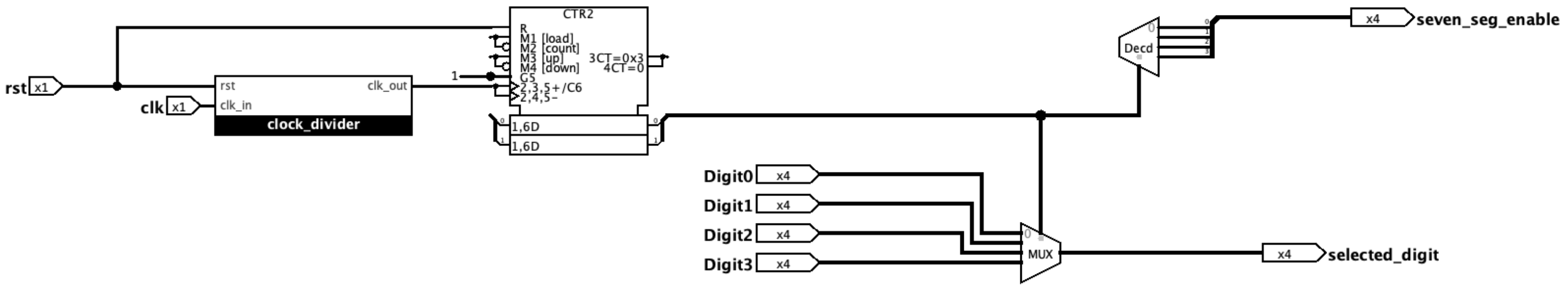
Verilog File: SSD_render.v

Description:

The SSD Render module consists of the Seven-Segment Selector module and the Seven-Segment Decoder module. This SSD Render takes in rst, CLK, Digit0, Digit1, Digit2, and Digit3 as input, and it passes them to the Seven-Segment Selector as shown in the diagram below. Then, after acquiring the outputs of the Seven-Segment Selector, which are the enable output and the selected digit output, it passes the selected digit to the Seven-Segment Decoder to acquire its output, which is the seven enable's of the seven segments of the display (the segments a,b,c,d,e,f, and g).



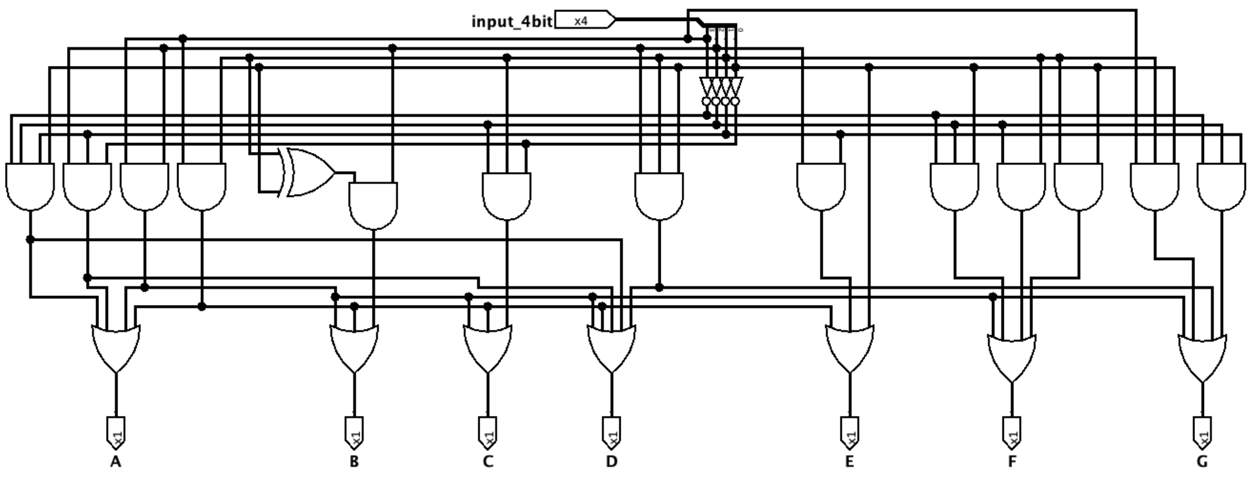
1) Seven-Segment Selector



Description:

The job of this module is to match each digit with the seven-segment unit intended for its display. In this circuit, the Seven-Segment Selector takes in as input: rst, clk, Digit0, Digit1, Digit2, and Digit3. Then, using a binary counter whose maximum value is set to $(4)_{10}$, we keep switching between the four given digits and at the same time switching the enabled seven segment unit. Thus, we can display each specific digit on the desired seven segment that we specified for it, and, using the clock divider, we do that at a suitable frequency such that the human eye cannot notice the flickering or turning off of any seven segment display.

2) Seven-Segment Decoder



Description:

After selecting the digit that we desire to display, which is taken here as input, using the Seven-Segment Selector, we use the Seven-Segment Decoder to turn on or off, which is the output of this module, each segment of the seven segments of the unit, (segments a,b,c,d,e,f, and g), according to the selected digit.

III - Implementation Issues

- None

IV - Validation and How to Use

Verilog File: Project_tb.v
Constraint File: project_const.xdc
Demo 3 Video:
https://drive.google.com/file/d/1uljTUnwgsDP7TtkrhU7ftT0M9LzmvWRW/view?usp=sharing

For validation, we used a testbench for simulation testing, which was done using iVerilog compiler (but can also be run on Vivado), and a constraint file to implement the Verilog code onto the FPGA board. You can see the final outcome on the board using the video linked above. You can also find a screenshot of the simulation wave at "/sources/Design Sources and Simulations/Simulation/Simulation Wave.png"

To display the simulation pre-created:

- Run "/sources/Design Sources and Simulations/Simulation/simulation_gtkwave.gtkw"

- This should open up gtkwave with the pre-defined configuration and pre-computed simulation that appears in the screenshot provided

To run your own simulation using iVerilog:

- Head to "/sources/Design Sources and Simulations"
- Ensure that you have iVerilog and gtkwave installed
- In the terminal type "iverilog -o simulation.out *.v"
- Type "vvp simulation.out"
- Head to "Simulation"
- From which in the terminal type "gtkwave simulation.vcd"
- Gtkwave should open up showing the wave

To create and add source files into a new Vivado project:

- Open up Vivado and start a new project by click "Create Project"
- Press Next
- You may rename or relocate the project folder if needed, otherwise press Next
- Continue Pressing Next, until prompted about the Default Part (board)
- Select Boards at the top left
- Search and select basys 3, then press Next
- Finally press Finish
- Once the project is created, click on "Add Sources" under the flow navigator on the left
- Choose "Add or create design sources"
- In the window that follows, choose "Add Files"
- Head to "/sources/Design Sources and Simulations"
- Excluding the testbench (Project_tb.v), select all .v files (
  - Binary_to_BCD.v
  - clock_divider.v
  - debouncer.v
  - DigitShifter.v
  - mod_n_counter.v
  - multiplier_CU.v
  - multiplier_DP.v
  - Project.v
  - push_button.v
  - rising_edge_detector.v
  - signed_sequential_multiplier.v
  - SSD_render.v
  - synchronize.v

)
- After selecting the files, click Finish

To simulate the testbench using Vivado:

- Make sure to have created a Vivado project using the steps mentioned above
- Choose “Add sources”
- Choose Add or create simulation sources”
- Choose “Add Files”
- Head to “/sources/Design Sources and Simulations”
- Choose “Project_tb.v”
- Click Finish
- Under “Simulation Sources/sim_1”, Select “Project_tb” as top (if it is already top, ignore this step)
- Choose “Run Simulation” in the “Flow Navigator” on the left, and “Run Behavioral Simulation” from the popup menu
- The simulation should run and produce the same results as that of iVerilog and gtkwave

To implement the Verilog code onto an FPGA (Basys 3):

- Make sure to have created a Vivado project with the Basys 3 board selected
- Add the constraint file to the project by clicking “Add sources”
- Choose “Add or create constraints”
- Choose “Add Files”
- Head to “/sources/Constraints”
- Select “project_const.xdc”
- Click Finish
- Finally, to run the project choose “Generate Bitstream”
- Choose “Yes” on the popup that follows about running synthesis
- And Choose “Ok” on the popup to launch runs
- Wait until, you get “Bitstream Generation Completed”, and then click “Cancel”
- Make sure your board is properly connected to the PC
- From the “Flow Navigator” on the left, under “PROGRAM AND DEBUG”, under “Open Hardware Manager”, click on “Open Target”, and choose “Auto Connect”
- Once connected, click on “Program Device” and choose the first item from the popup menu
- In the dialog which appears accept the dialog, and the board will be programmed
- You may begin using the board\

To flash the FPGA board (Basys 3):

- Follow the following tutorial (https://sites.google.com/a/umn.edu/mxp-fpga/home/vivado-notes/programming-the-basys3-boards-non-volatile-flash-memory-through-vivado?pli=1)

V - Contribution

- Multiplier: Mostafa & Andrew
- Binary_to_BCD: Youssef & Kirolous
- Digit Shifter: Andrew & Youssef
- SSD Render: Kirolous & Mostafa