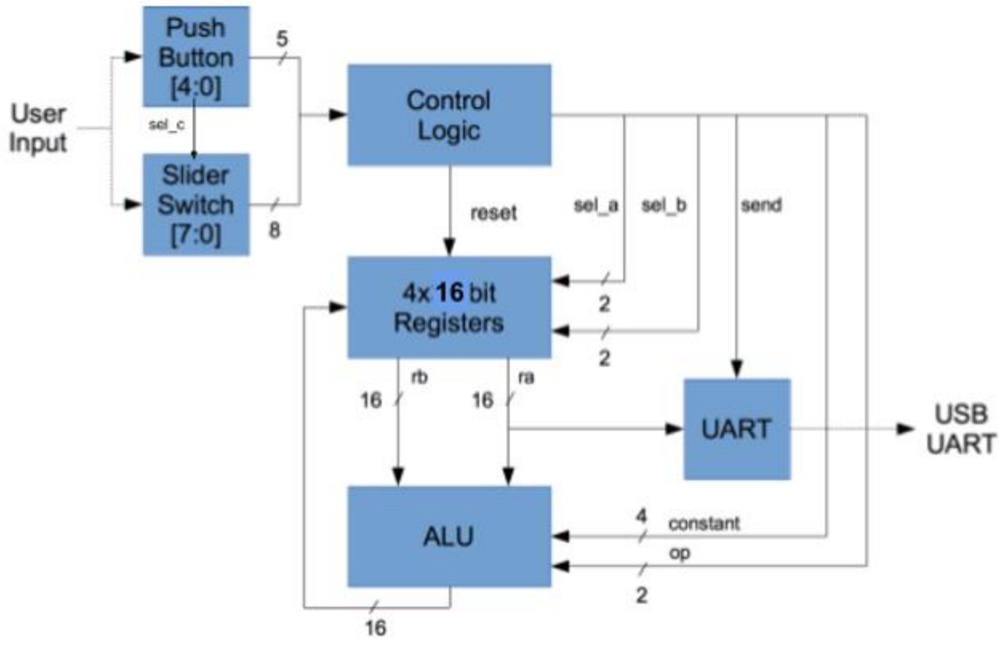
Lab 1 Report

Dennis Zang, 704766877
Partner: Hirday Gupta
Lab Section 5, October 27, 2019

Problem Statement:
The task at hand that was assigned is to modify an existing Verilog program which operates an arithmetic logic unit (ALU) (that originally only supports addition) through the use of 8 switches to encode and instruction and a switch to send the instruction. The modifications we were tasked with are as follows: to add a multiply function/submodule to the ALU, to add a separate button to replace the current send instruction, modify the "model_uart" submodule to format the simulation output into a single line, and modify the actual UART submodule "uart_top" to include the register number in the actual UART output.

Implementation/Algorithm:
The original sequential circuit works as follows: input is fed into the top-level module "nexys"; which includes 8 switches to encode instructions, a send button, a reset button, and a clock input; and outputs an 8-bit led for instruction count. In addition, an input RsRx (receiver) and output RxTx (transmitter) are used for UART input and output, respectively. "nexys3" would then operate an internal submodule "seq" to process the command, access internal 16-bit registers (submodule "seq_rf"), and perform the given arithmetic operations (submodule "seq_alu"). The submodule "uart_top" would be operating the actual interface for UART (the submodule "uart") using a queue to sequentially pass data for writing (using the submodule "uart_fifo"). In addition, for "nexys3", a submodule "model_uart" will be used for the testbench simulation to emulate UART.
The specific functions; PUSH (00), ADD (01), MULT (10), and SEND (11); are all found within the lab manual. Below is the given diagram showing how each I/O component and sub modules inside the "nexys3" module generally interact.



The details for how we modified the sequential circuit for each task are listed below.

("[Implementation] Missing Multiply Operation")

For this task, we basically had to make modifications to the submodule "seq_alu". Originally, "seq_alu" only consists of the submodule "seq_add" to support addition. So, to support multiplication, we create a copy of "seq_add" for multiplication, "seq_mult", which a near-identical module except for 1 line of code (it returns the product of inputs instead of sum, obviously). Then, to link up this new submodule, we needed to add and instance of "seq_mult" to "seq_alu" and add the multiply-command case for "seq_mult" to all switch cases that act as multiplexers for the return value, which was missing in the original.

("[Implementation] A Separate SEND Button")

For this task, to implement a separate SEND button, we had to begin our modification from the top-level module, "nexys3". Here, we added a new input variable, "btnD", to implement a new functionality that automatically performs the SEND function (11), which we will call DISPLAY. Like "btnS", we needed to buffer the previous values of "btnD" to detect a positive edge (on a delayed clock), and if a positive edge is detected, automatically replace the first two bits in the command buffer from the switches (the command type) to (11), which corresponds to SEND. The remaining existing parts of the "nexys3" module will then go through the usual order: passing the modified buffer to "seq" and then that output to "uart_top".

("[Simulation] Nicer UART Output")

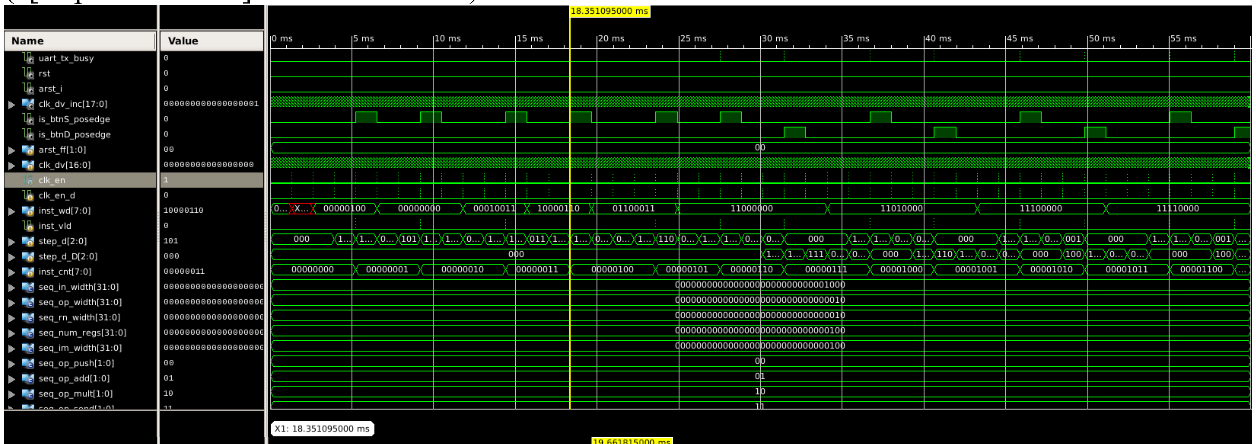
For this task, to output everything out in a single line, we needed to buffer the received bytes from the virtual UART receiver. So, in the module "model_uart", we added a buffer register that holds up to 7 bytes, or 56 bits (the reason for 7 bytes is for the next section). To use this buffer, we basically modified the existing subroutine that runs at every negative clock edge to put the current byte in the buffer if the current byte is not a carriage return '\r' or newline '\n', and if the current byte is a carriage return '\r', then display the current contents of the buffer as a string.

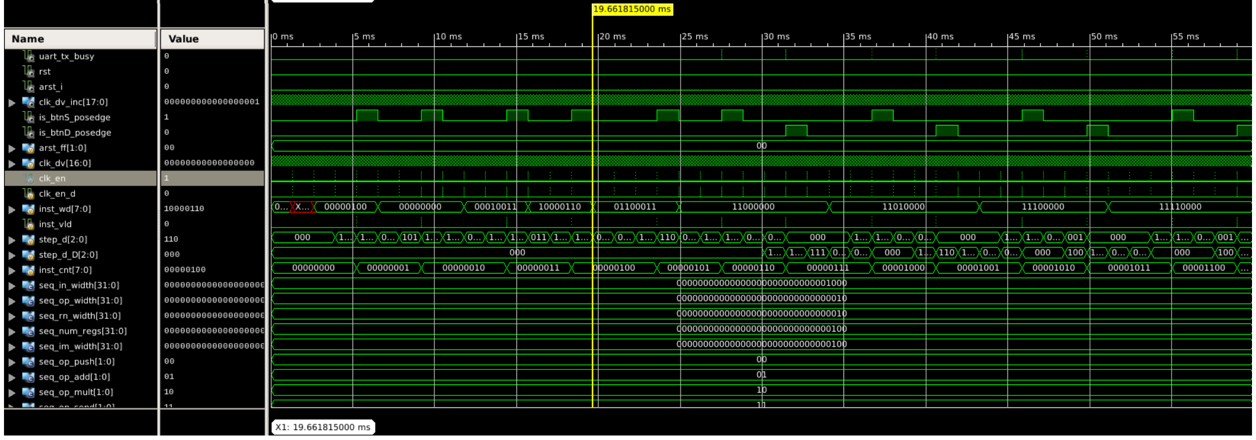
("[Implementation] Even Nicer UART Output")

For this task, we needed to include the register number of the value we want to display. So, to do this, we need to pass the register number encoded from the switches (position [5:4]) to the modified submodule "uart_top" from "nexys3". In "uart_top", since it is already implemented as a state machine that would sequentially choose what character to pass to the queue "uart_fifo" and the actual interface "uart", we added additional states to format the output into what we want (we added 4 additional states for the character 'R', one for ':', one for '\n', and one for '\r'). As per the original implementation, "uart_top" will cycle through each of the states in a particular order that follows the requested format ("R<register_number>:<value>").

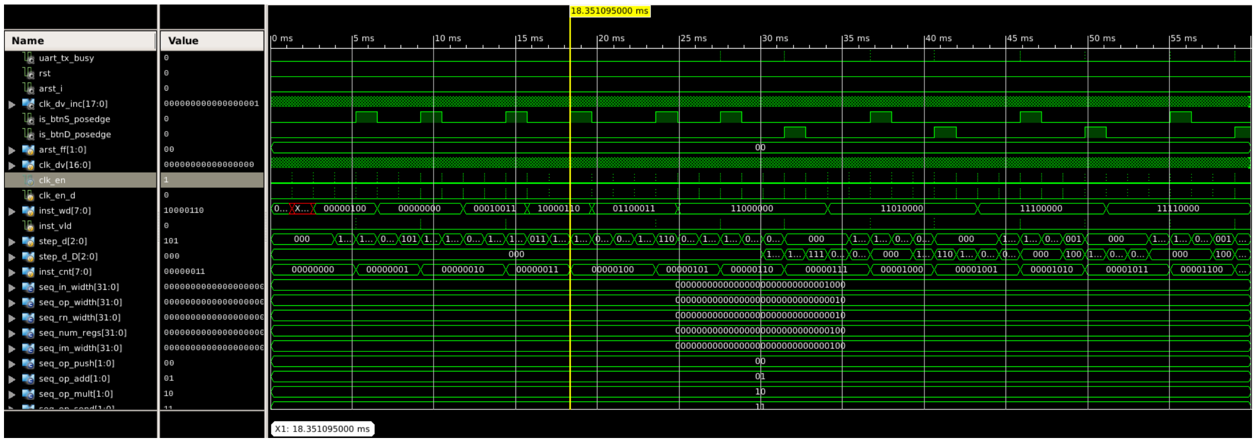
Simulation Screenshots:

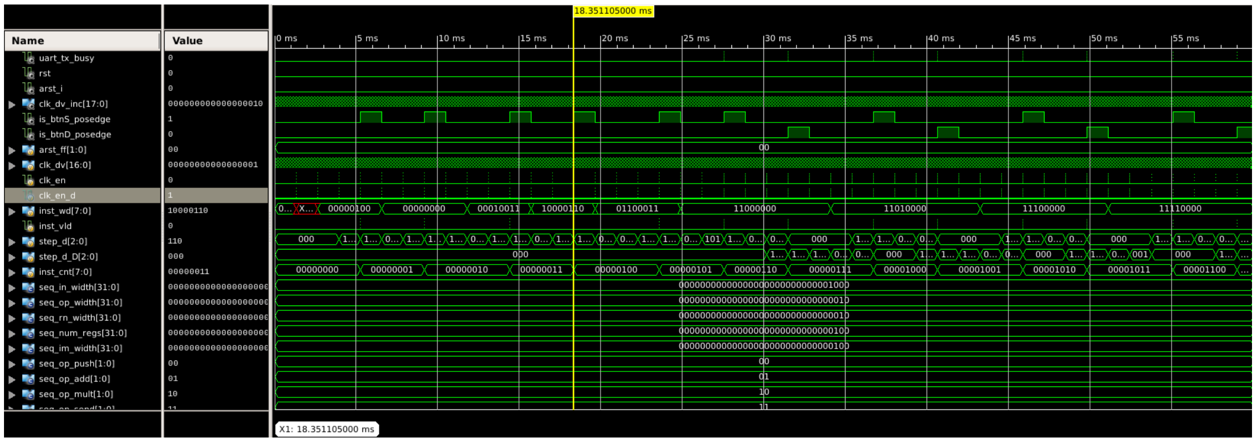
Below are screenshots of the requested waveforms for workshop 2.
("[Implementation] Clock Dividers")





Above are two consecutive occurrences of "clk_en". As we can see, the two occurrences are at 18.351095000ms and 19.661815000ms, respectively. The period for "clk_en" would be **<u>1.31072ms</u>**.

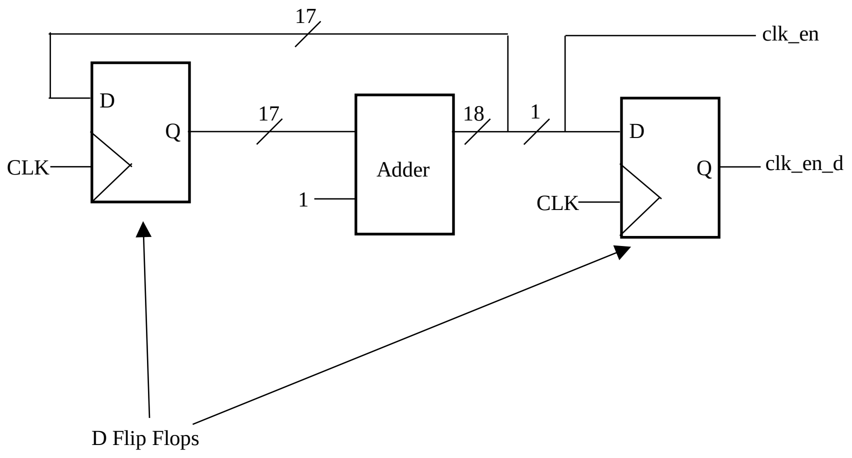




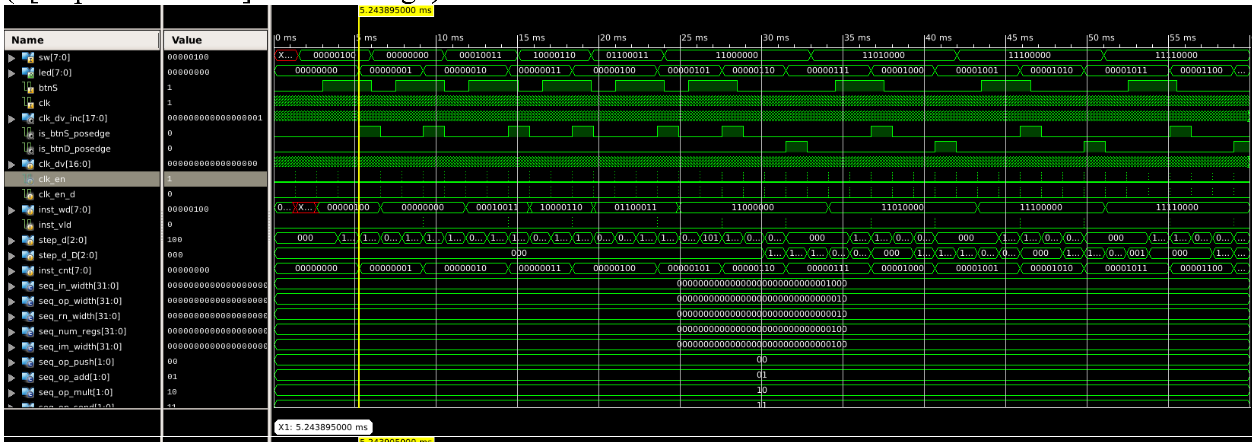
Name	Value
uart_tx_busy	0
rst	0
arst_i	0
clk_dv_inc[17:0]	000000000000000010
is_btnS_posedge	1
is_btnD_posedge	0
arst_ff[1:0]	00
clk_dv[16:0]	00000000000000001
clk_en	0
clk_en_d	1
inst_wd[7:0]	10000110
inst_vld	0
step_d[2:0]	110
step_d_D[2:0]	000
inst_cnt[7:0]	00000011
seq_in_width[31:0]	0000000000000000000
seq_op_width[31:0]	0000000000000000000
seq_rn_width[31:0]	0000000000000000000
seq_num_regs[31:0]	0000000000000000000
seq_im_width[31:0]	0000000000000000000
seq_op_push[1:0]	00
seq_op_add[1:0]	01
seq_op_mult[1:0]	10

From the above, we can see an instance of "clk_en" = 1. It then immediately becomes 0 when "clk_en_d" becomes 1 exactly 1 "clock_dv" increment later. Given that "clk_en" is set to 1 for 18.351095000ms to 18.351105000ms, or for a duration of 0.00002ms, the duty cycle of "clk_en" would be (0.00002/1.31072) = **<u>0.00153%</u>**. In addition, during the clock cycle that "clock_en" is high, the value of "clock_dv" is 00000000000000000.

Given that "clk_en" and "clk_en_d" are set to 1 only for 1 "clk_dv", which counts from 0 to 11111111111111111 (in binary), the circuit corresponding to these 3 values should look something like the figure below.

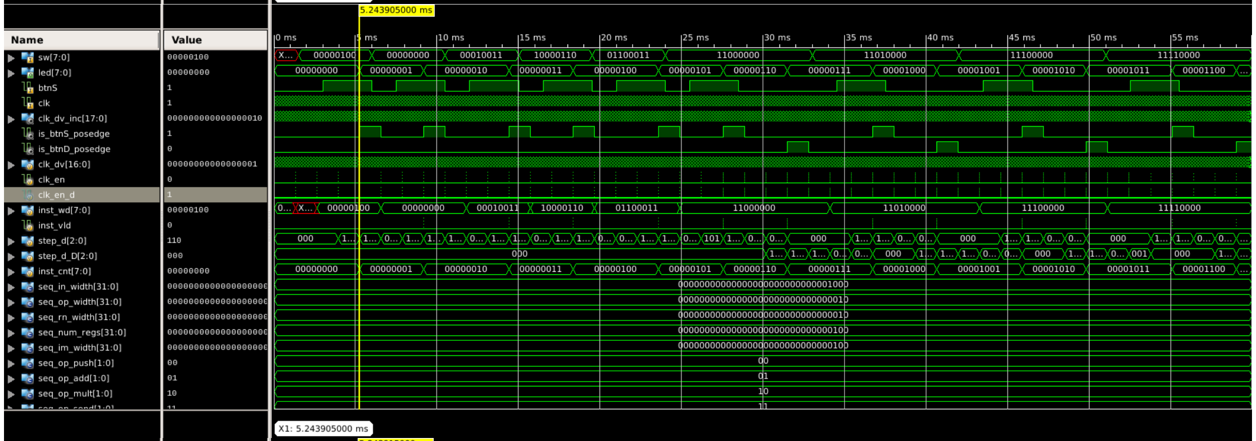


(“[Implementation] Debouncing”)



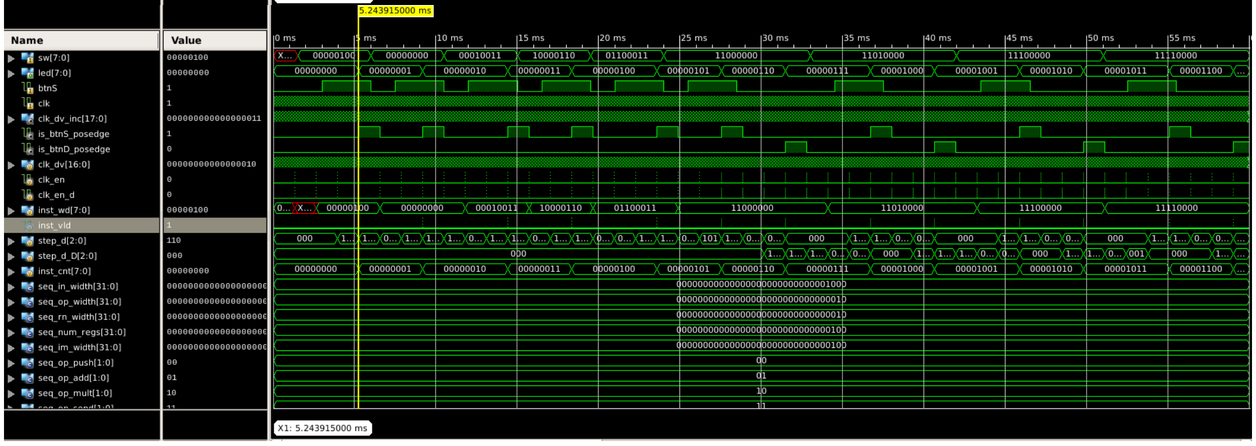
Name	Value
sw[7:0]	00000100
led[7:0]	00000000
btnS	1
clk	1
clk_dv_inc[17:0]	000000000000000001
is_btnS_posedge	0
is_btnD_posedge	0
clk_dv[16:0]	00000000000000000
clk_en	1
clk_en_d	0
inst_wd[7:0]	00000100
inst_vld	0
step_d[2:0]	100
step_d_D[2:0]	000
inst_cnt[7:0]	00000000
seq_in_width[31:0]	00000000000000000000
seq_op_width[31:0]	00000000000000000000
seq_rn_width[31:0]	00000000000000000000
seq_num_regs[31:0]	00000000000000000000
seq_im_width[31:0]	00000000000000000000
seq_op_push[1:0]	00
seq_op_add[1:0]	01
seq_op_mult[1:0]	10

X1: 5.243895000 ms



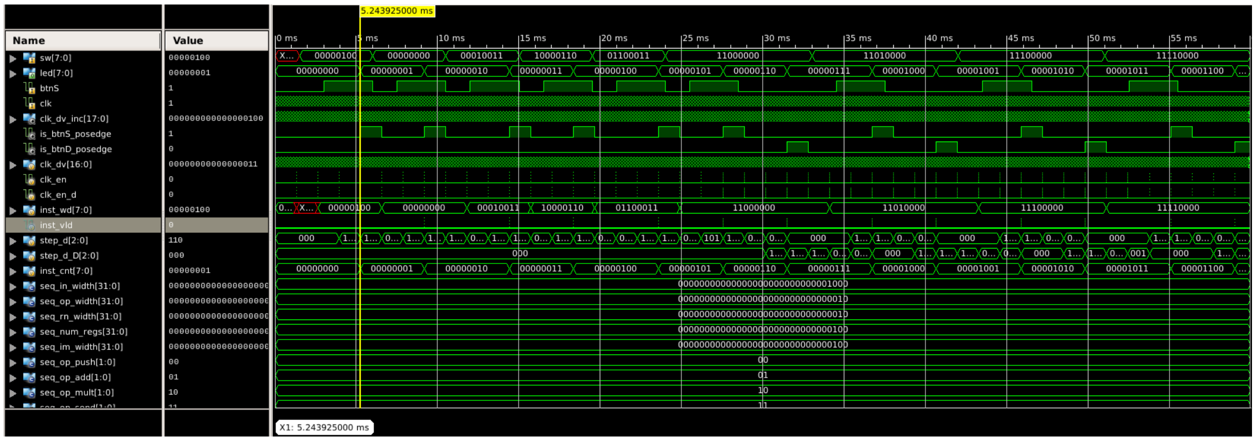
Name	Value
sw[7:0]	00000100
led[7:0]	00000000
btnS	1
clk	1
clk_dv_inc[17:0]	000000000000000010
is_btnS_posedge	1
is_btnD_posedge	0
clk_dv[16:0]	00000000000000001
clk_en	0
clk_en_d	1
inst_wd[7:0]	00000100
inst_vld	0
step_d[2:0]	110
step_d_D[2:0]	000
inst_cnt[7:0]	00000000
seq_in_width[31:0]	00000000000000000000
seq_op_width[31:0]	00000000000000000000
seq_rn_width[31:0]	00000000000000000000
seq_num_regs[31:0]	00000000000000000000
seq_im_width[31:0]	00000000000000000000
seq_op_push[1:0]	00
seq_op_add[1:0]	01
seq_op_mult[1:0]	10

X1: 5.243905000 ms



Name	Value
sw[7:0]	00000100
led[7:0]	00000000
btnS	1
clk	1
clk_dv_inc[17:0]	000000000000000011
is_btnS_posedge	1
is_btnD_posedge	0
clk_dv[16:0]	00000000000000010
clk_en	0
clk_en_d	0
inst_wd[7:0]	00000100
inst_vld	1
step_d[2:0]	110
step_d_D[2:0]	000
inst_cnt[7:0]	00000000
seq_in_width[31:0]	00000000000000000000
seq_op_width[31:0]	00000000000000000000
seq_rn_width[31:0]	00000000000000000000
seq_num_regs[31:0]	00000000000000000000
seq_im_width[31:0]	00000000000000000000
seq_op_push[1:0]	00
seq_op_add[1:0]	01
seq_op_mult[1:0]	10

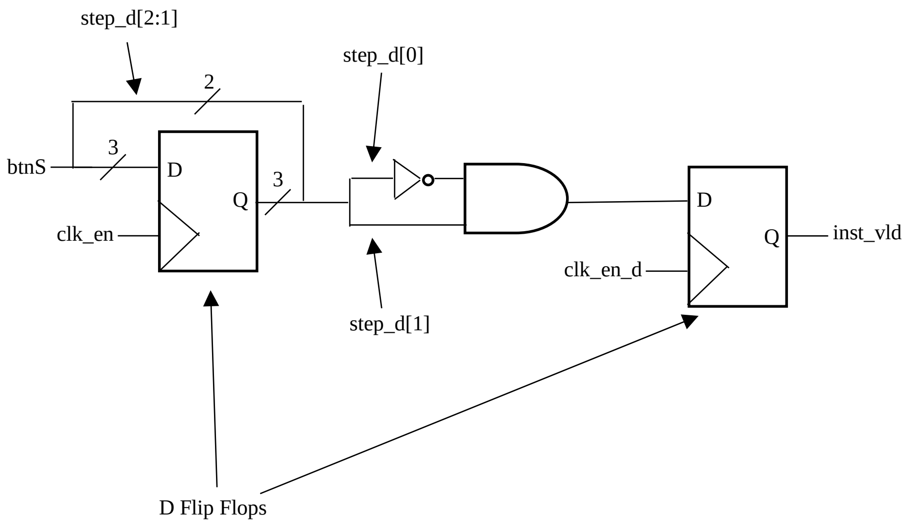
X1: 5.243915000 ms



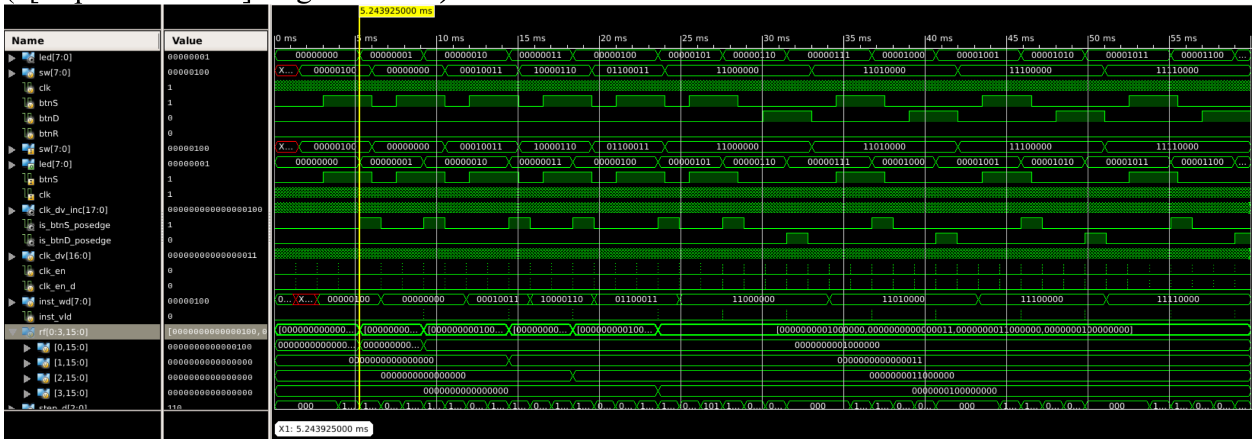
Name	Value
sw[7:0]	00000100
led[7:0]	00000001
btnS	1
clk	1
clk_dv_inc[17:0]	000000000000000100
is_btnS_posedge	1
is_btnD_posedge	0
clk_dv[16:0]	00000000000000011
clk_en	0
clk_en_d	0
inst_wd[7:0]	00000100
inst_vld	0
step_d[2:0]	110
step_d_D[2:0]	000
inst_cnt[7:0]	00000001
seq_in_width[31:0]	0000000000000000000
seq_op_width[31:0]	0000000000000000000
seq_rn_width[31:0]	0000000000000000000
seq_num_regs[31:0]	0000000000000000000
seq_im_width[31:0]	0000000000000000000
seq_op_push[1:0]	00
seq_op_add[1:0]	01
seq_op_mult[1:0]	10

As we can see in the pictures above, when “clk_dv” periodically becomes 0, “clk_en” becomes 1, and in the immediate next “clk_dv”, “clk_en” becomes 0 and “clk_en_d” becomes 1 for one “clk_dv”. Then, in the immediate next “clk_dv”, “inst_vld” is set to 1 for one clock divider signal (as a positive edge has been detected in the previous clock divider period).

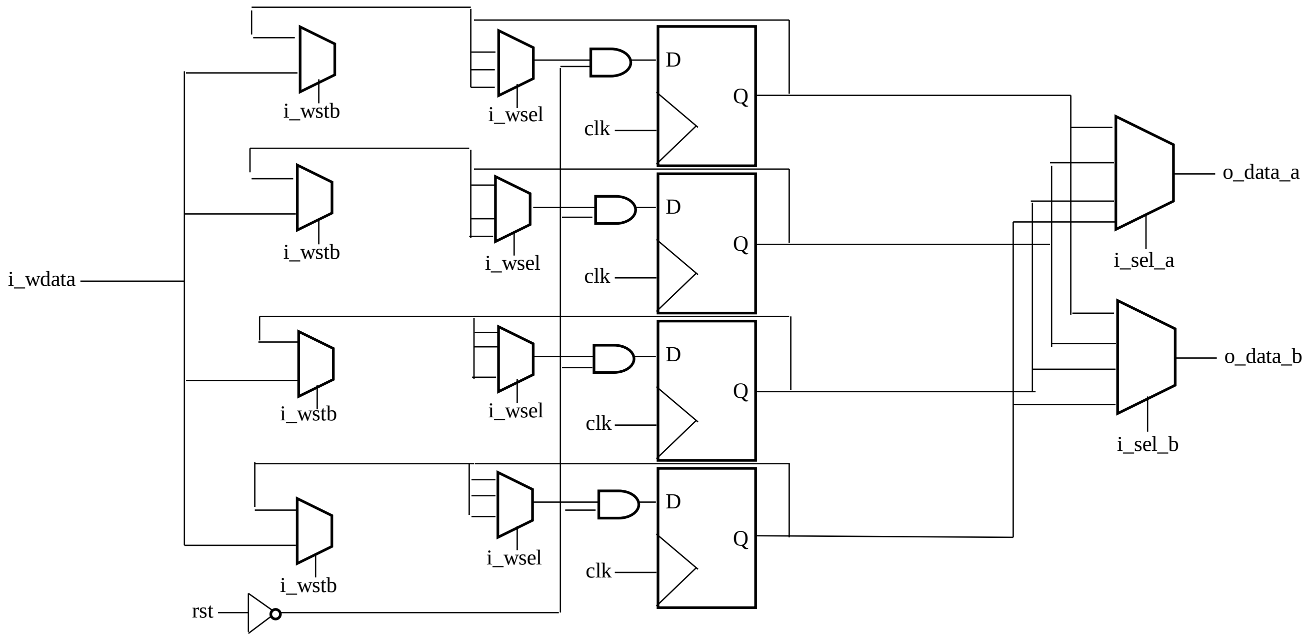
In the original code, if we are to use “clk_en_d” instead of “clk_en” in line 102 of “nexys3.v”, then we would not be able to detect the positive edge of “btnS”, as we need the previous value of “btnS” (queued in array “step_d”). More specifically, the next always block in line 99 with the conditional for “clk_en_d” as well cannot access the “previous” “btnS” value, as it is currently being queued in the same time frame (and no other always block will catch this positive edge in the next time frame).
If we are to switch “clk_en <= clk_dv_inc[17]” in line 76 with “clk_en <= clk_dv [16]”, then we will have a 50% duty cycle instead of a single pulse that lasts only one “clk_dv”. This would mean that the two always blocks at lines 84 and 99 would always trigger at each undivided “clk” edge, meaning the module would sample at each “clk” frequency half the time (for each value of “clk_en” from 10000000000000000 to 11111111111111111), which would defeat the purpose of metastability and noise cancelation by sampling at a lower frequency.
Shown below is a diagram corresponding to the values clk_en, step_d[1], step_d[0], btnS, clk_en_d, and inst_vld.



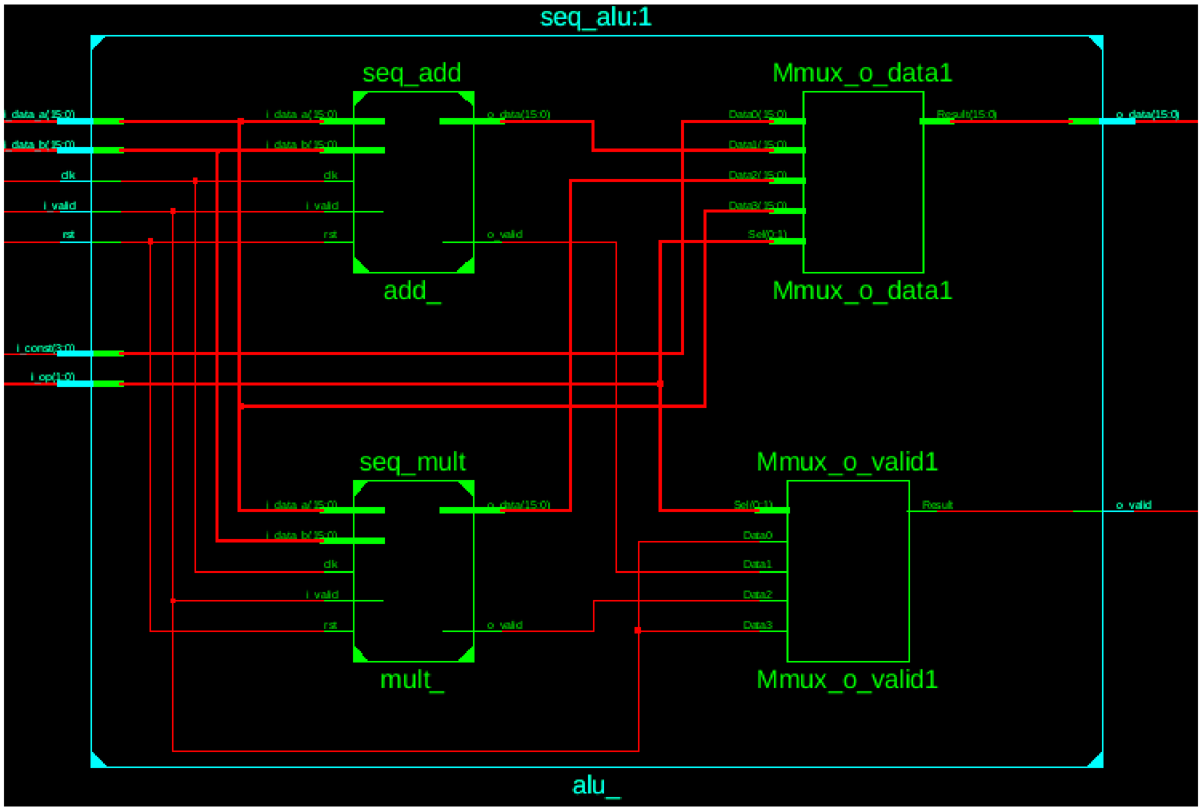
(“[Implementation] Register File”)



The lines of code where a register is first written a non-zero value are lines 68 and 88, where the “seq_rf” and “seq_alu” are wired to sequentially evaluate the instructions to perform. “seq_alu”, when performing a PUSH instruction, would simply perform assignments to “o_data” and “o_valid” in separate switch cases in separate always blocks (in lines 32 and 40 in “seq_alu.v”, respectively). One thing to note is that although the assignments are blocking, there is only a single line performed in each, so the underlying execution is still sequential. Finally, the output would be passed to “seq_rf” to update the register files.
The values that “seq_alu” gets from “seq_rf” are also received through sequential execution. “seq_rf” would return 2 values from two register numbers, which it receives as input). The output values are wired to “seq_alu” to perform the computation, and the return values is passed back to “seq_rf” on the next clock cycle.
If I were to implement the readout logic, I would use flip-flops (particularly D flip-flops) to cache the previous value in the previous time frame to keep particular input values constant as they are not done being loaded (ie when “seq_rf” needs the output of “seq_alu” to update registers, I would use D flip-flops to keep the previous input loaded to keep the “seq_alu” module on standby). In addition, for the actual registers, D flip-flops can allow for persistent storage (until the next write-in) one can continuously perform read-out with.
Below is a simple circuit diagram of the register file block.

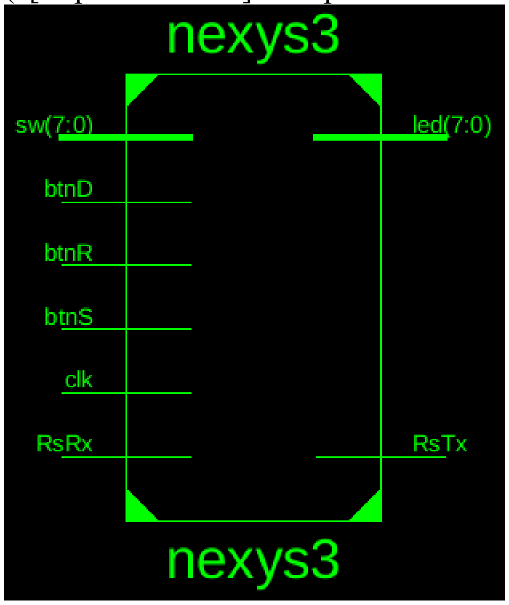


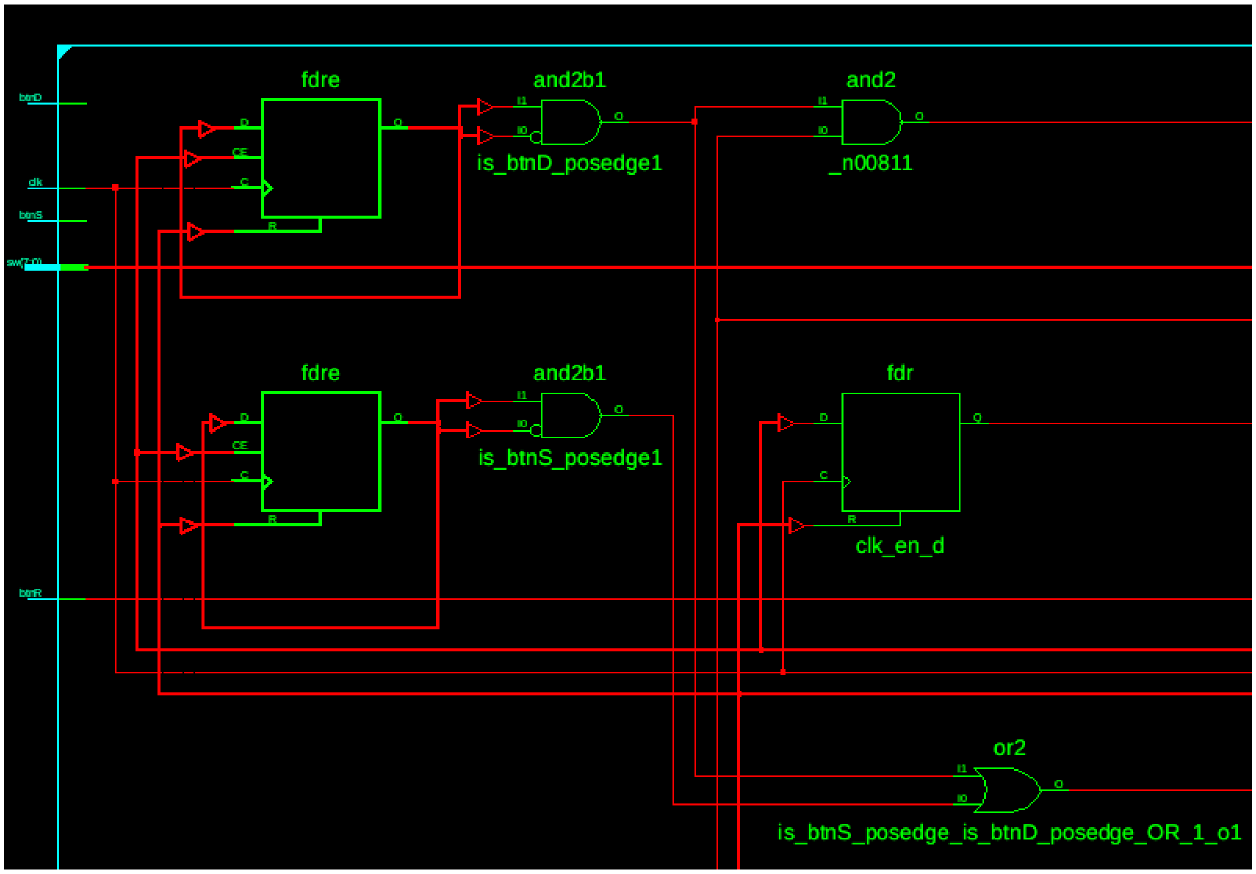
Netlists:
("[Implementation] Missing Multiply Operation")



As we can see above, when we added "seq_mult" to "seq_alu", a new module is instantiated, and a new case is considered by the multiplexers (case statements in Verilog code).

("[Implementation] A Separate SEND Button")

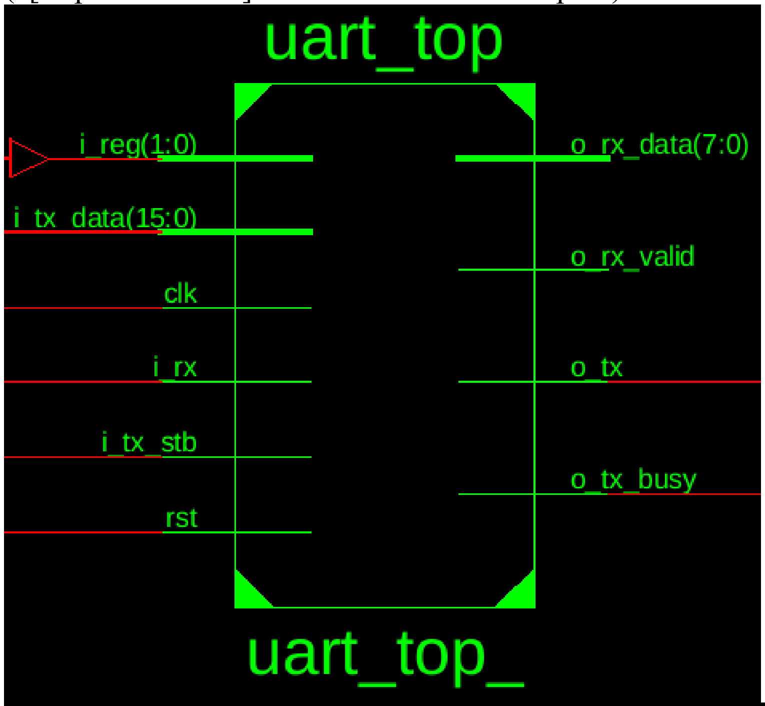


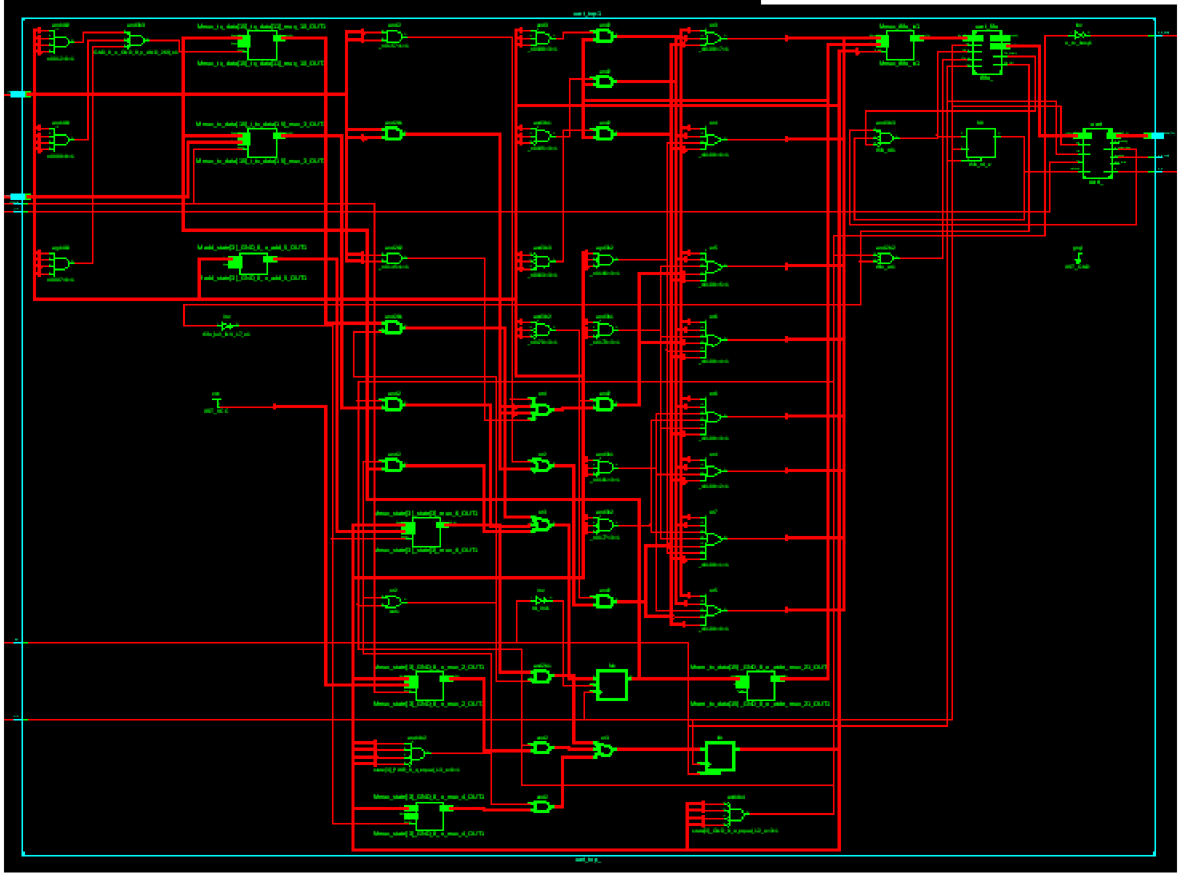


Tthe above 2 pictures show the implementation changes from adding a new button for DISPLAY ("btnD"), like changing the conditional for when a command (SEND or DISPLAY) button is pressed and adding an input to a multiplexer to manipulate variable "inst_wd[7:0]" (not shown).

("[Simulation] Nicer UART Output")
We cannot actually generate any netlists for "model_uart".

("[Implementation] Even Nicer UART Output")





The above would be the effects of adding a new input "i_reg" (to display register number) and adding extra states to "uart_top".

Inferences:

(Challenges)

The main challenge in our case would be learning how to read the logic flow sequential logic through each submodule in the project. In addition, we needed to know how each submodule connected to each other, and where we can make changes to the existing code to implement new functionalities (ie adding new states to “uart_top”).

Also, this lab provided us with a good basis for experimenting how to use Xilinx ISE for debugging and testing purposes.

(Mistakes)

We definitely spent more time that needed in modifying “uart_top”, especially with caching characters to send over UART and the state machine implementation (which took longer than it should have).