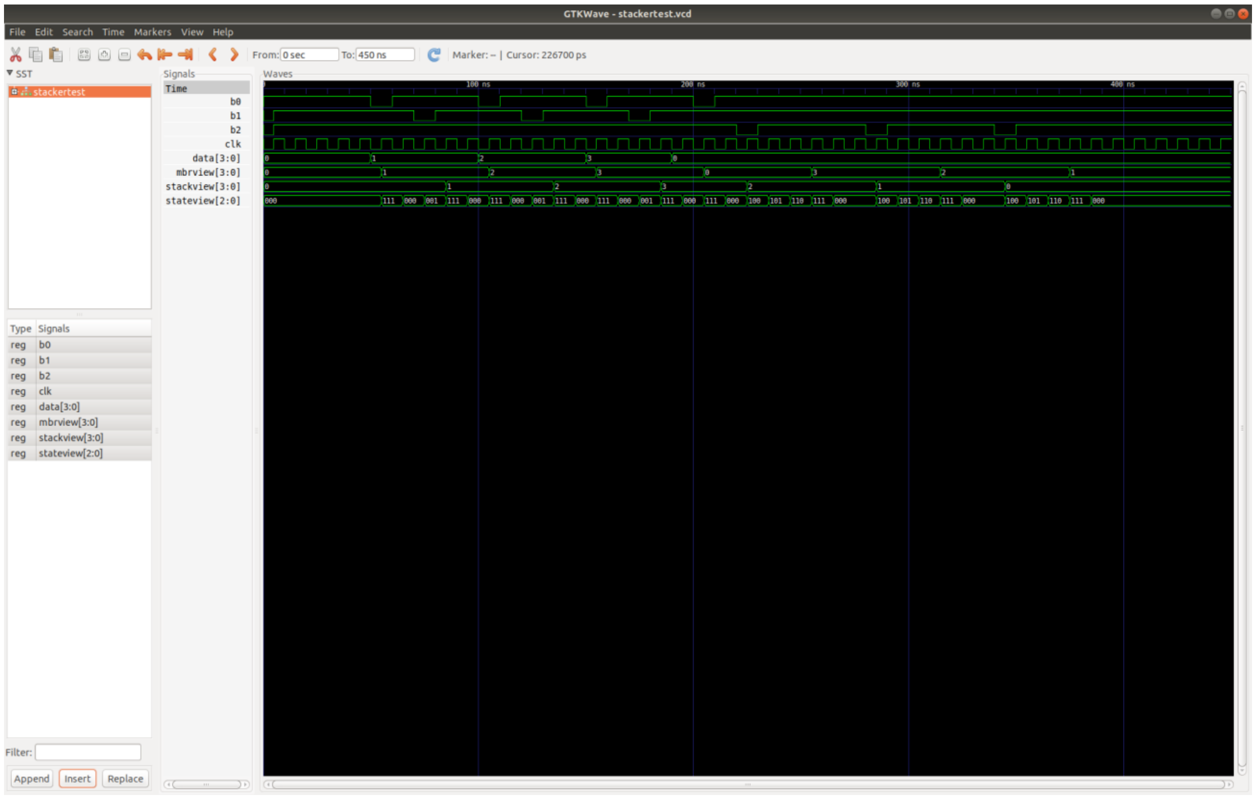
Stack-Based Calculator

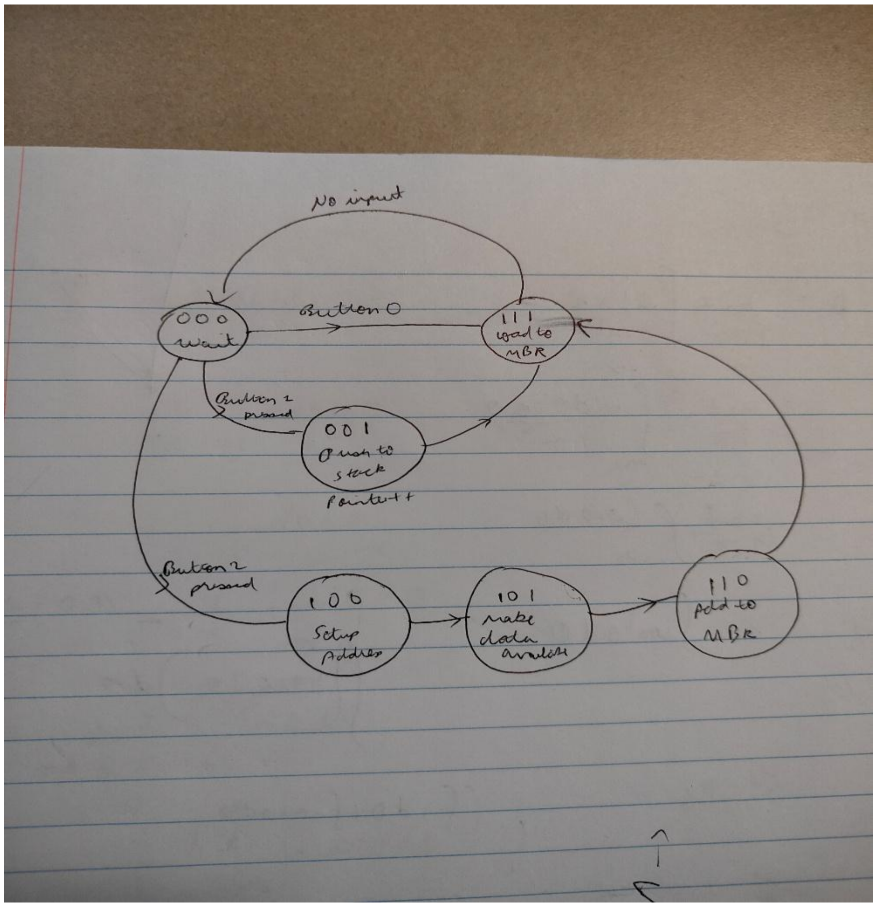
This project implements a stack-based calculator using VHDL hardware description language. The calculator utilizes random-access memory (RAM) along with registers to create a stack data structure, allowing users to perform basic arithmetic operations (+, -, *, /) through a sequence of button presses and switch configurations. By designing a state machine to control data flow between memory and registers, the project illustrates principles such as memory addressing, register transfers, stack operations, and instruction execution cycles. The implementation demonstrates how complex computational tasks can be broken down into simpler sequential operations managed by a finite state machine, providing insight into how actual processors handle mathematical operations using memory and registers as temporary storage mechanisms.

Description of Top level file:

Required GTKwave of stacker:



The top-level design of the Project 6 calculator revolves around a finite state machine that coordinates stack operations, arithmetic logic, and input/output handling using RAM and a memory buffer register (MBR). Users interact with the calculator through three buttons: Capture (b0), Enter (b1), and Action (b2) alongside data and operation switches. Pressing Capture stores the current 8-bit switch value into the MBR, which functions as both the working display and the operand register. Enter pushes the MBR onto a RAM-based stack, incrementing a 4-bit stack pointer unless the stack is full. When Action is pressed, the calculator pops the top value from the stack (if not empty), performs an arithmetic operation (add, subtract, multiply, or divide) with the MBR, and stores the result back in the MBR. The state machine ensures correct sequencing: capturing values, handling memory latency during pops, executing arithmetic, and preventing overflow or division by zero. A 7-segment display driver converts the 8-bit MBR into a three-digit decimal output for user-friendly visualization. This design maintains clarity and robustness while allowing for future enhancements, such as overflow detection, expanded operations, or fixed-point arithmetic.



The state machine in the image represents the control logic for the stack-based calculator. It begins in state 000 (wait), where it monitors button inputs. If **Button 0** is pressed, it moves to

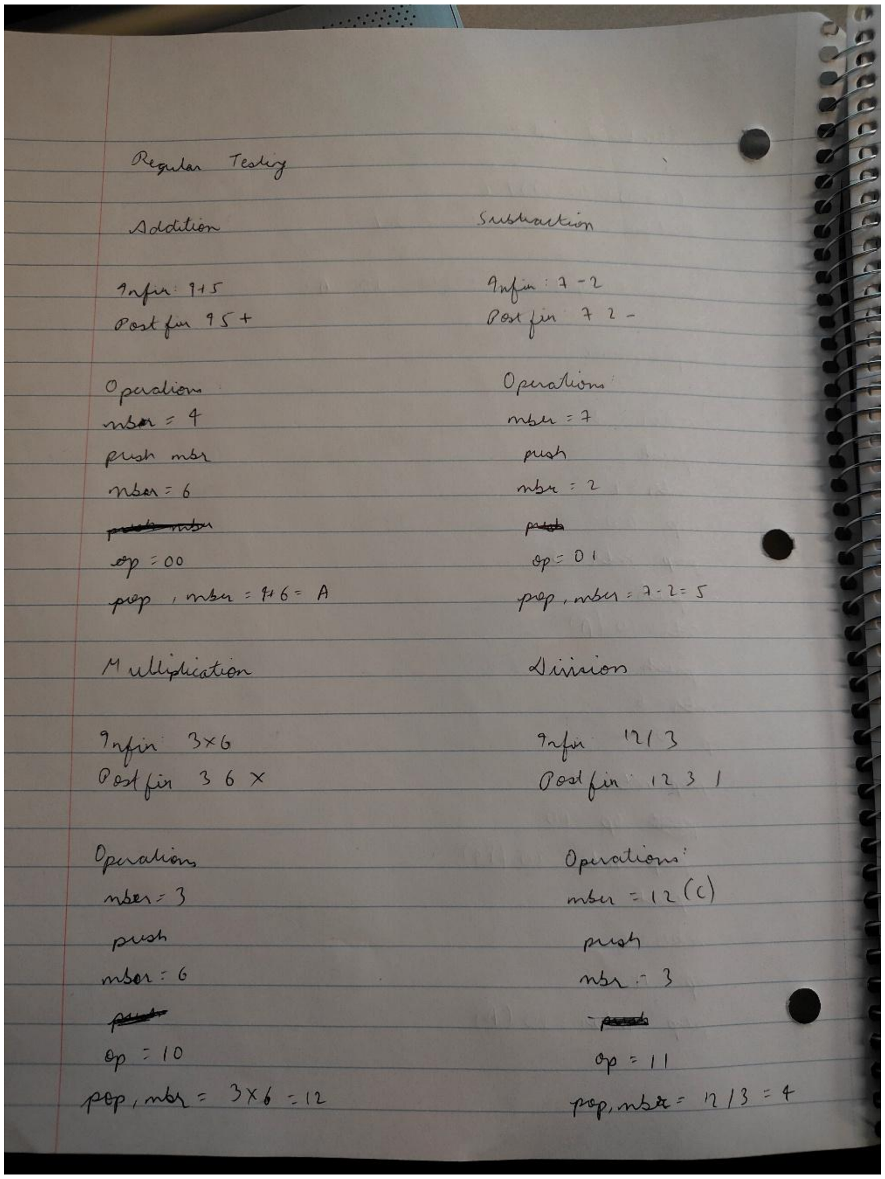
111, where the switch value is loaded into the Memory Buffer Register (MBR). If **Button 1** is pressed, it transitions to 001, where the MBR is pushed onto the stack and the stack pointer is incremented. If **Button 2** is pressed, indicating an arithmetic operation, it transitions to 100, where it sets up the address for the stack pop. Then it moves through 101 and 110—a delay sequence allowing memory output to stabilize—before executing the arithmetic operation and storing the result back into the MBR. Finally, it loops back to 111 to wait for button release before returning to 000. Each transition ensures synchronization with memory and avoids unintended operations, preserving correctness in both stack manipulation and computation.

Testing strategy used:

I used simple arithmetic calculations to test the calculator.

https://drive.google.com/drive/folders/1zYPis5rZkYX9sZ9qHeGK1mOmma0kCKLN?usp=sharing

This drive contains all video clips of the tests performed which are also described below:



Regular Testing

Addition

Infix: 4+5
Postfix 4 5 +

Operations
mbr = 4
push mbr
mbr = 6
~~push mbr~~
op = 00
pop , mbr = 4+6 = A

Subtraction

Infix : 7 - 2
Postfix 7 2 -

Operations
mbr = 7
push
mbr = 2
~~push~~
op = 01
pop , mbr = 7 - 2 = 5

Multiplication

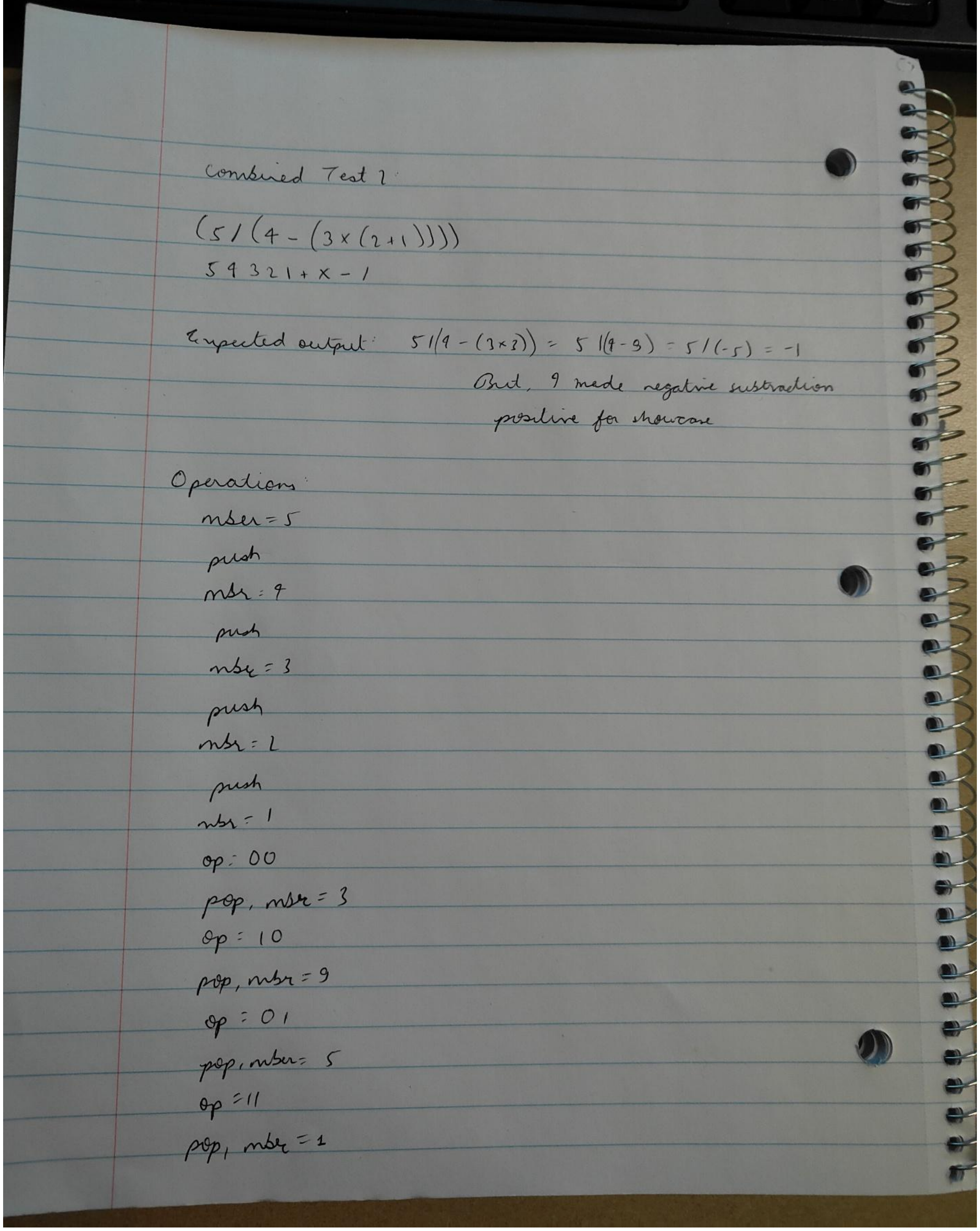
Infix 3×6
Postfix 3 6 ×

Operations
mbr = 3
push
mbr = 6
~~push~~
op = 10
pop , mbr = 3×6 = 12

Division

Infix 12/3
Postfix 12 3 /

Operations:
mbr = 12 (C)
push
mbr = 3
~~push~~
op = 11
pop , mbr = 12/3 = 4


Combined Test 2:

$(5/(4-(3\times(2+1))))$

5 4 3 2 1 + x - /

Expected output: $5/(4-(3\times3)) = 5/(4-9) = 5/(-5) = -1$

But, I made negative subtraction positive for showcase

Operations:

mber = 5

push

mbr = 4

push

mbr = 3

push

mbr = 2

push

mbr = 1

op: 00

pop, mbr = 3

op = 10

pop, mbr = 9

op = 01

pop, mber = 5

op = 11

pop, mbr = 1

Combined Test 1

((3+2) x (10-6)) / 2

Postfix = 3 2 + 10 6 - x 2 /

Expected output = (5 x 4) / 2 = 10 = A

Operations:

mbr = 3

push

mbr = 2

op = 00

pop, mbr = 5

push

mbr = 10 (A)

push

mbr = 6

op = 01

pop, mbr = 4

~~push~~ op = 10

pop, mbr = 20 (14)

push

mbr = 2

op = 11

pop, mbr = 10 (A)

Extensions:

Firstly, I utilized the wider display we worked on project 2 to make the output screen wider. It also allowed me to display the values in decimal as I had programmed it in the past. I present it in the extension.mp4 video in the drive.

Next, for additional extension I tried using safe stack push and stack division but I did not get to finish the extension due to some issue causing the output to start at 128 instead of 0. Regardless of my efforts, I was unable to solve this and finally stopped working on it. It is present in the ExtensionPlus folder.

Acknowledgements:

For completing this project, I took help from articles online at Stackoverflow, reddit, intel's forums and took ideas from classmates whom I am immensely grateful for.