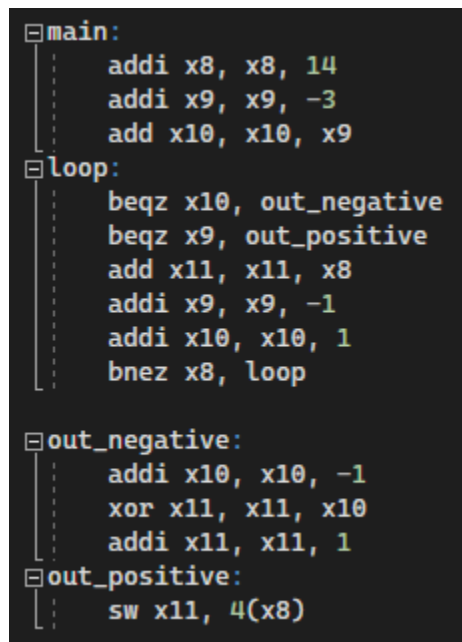


Background

Branching is a fundamental component of every computer and allows for much more sophisticated programs to be run. The current processor does not have this capability and can only run sequential instructions with no ability to jump to various instructions severely limiting the overall functionality. The purpose of this lab is to implement this branching functionality and test it with a simple multiplication program using the integrated RISC-V instructions built into the processor.

Pre-Lab

To multiply two signed numbers, two values must be added together continuously until we have reached the desired number of repetitions. The following screenshot of RISC-V assembly code solves this problem:



```
main:
    addi x8, x8, 14
    addi x9, x9, -3
    add x10, x10, x9
loop:
    beqz x10, out_negative
    beqz x9, out_positive
    add x11, x11, x8
    addi x9, x9, -1
    addi x10, x10, 1
    bnez x8, loop
out_negative:
    addi x10, x10, -1
    xor x11, x11, x10
    addi x11, x11, 1
out_positive:
    sw x11, 4(x8)
```

Figure 1: Screenshot of RISC-V Assembly Code to Multiply Two Values

Here, we are multiplying the value 14 by -3. To do this, we are adding 14 to itself continuously until -3 is equal to 0. At this point, we branch to out and apply a twos complement if the value is negative or only store the value if it is positive.

The code generated the following .coe file of instructions:

```
memory_initialization_radix=16;  
memory_initialization_vector=0439,14f5,9526,c511,c881,95a2,14fd,0505,f87d,157d,8da9,0585,c04c;
```

Figure 2: Screenshot of RISC-V Compressed Instructions

Design Implementation

2.1 – Designing the Top Level

After uploading the top-level file provided on Canvas, several changes were made to match naming conventions in the various project files. After this, the VIO was generated for the lab. The VIO consists of 19 input probes for all important information such as the current instruction, data entering and leaving the memory and registers, and alu operations.

2.2 – Program Counter

For the program counter designed in Lab 6, one change was made to introduce the branching functionality to the processor. This is done by checking the `take_branch` flag and applying an operation to the current instruction such as addition or subtraction to reach our desired instruction. The SystemVerilog code for this module can be found in Appendix A labeled as figure 3.

2.5.3 – Testing your Design

Using the generated .coe file for the RISC-V assembly code seen in Figure 1, everything was tested in hardware. The purpose of the RISC-V assembly code is to multiply 14 by -3. This will return a value of -42. Images of the output of the code working in Hardware can be found in Appendix B labeled as figures 4 through 23.

One thing to note is that due to the number of operations, the two `beqz` operations will not be shown after the first cycle of operations. Figures 4 through 12 will show all the operations from `addi x8, x8, 14` to `bnez, x8, loop. 13` through 18 will show the addition operations and 19 through 22 will show the two's complement and storing operations.

Conclusion

In this lab, branching was successfully incorporated into the RISC-V processor allowing for sophisticated operations to be executed and significantly increasing the functionality of the processor. As the instruction memory serves as a queue executor for instruction stored within, the program counter now allows for any inputted program to jump from one instruction to another. With this being the final step in the design process of the RISC-V processor, our knowledge of processor have significantly increased as each component was built off of the previous and served as one sequential circuit where the importance of each component was emphasized and highlighted by the next component.

Appendix A – System Verilog Code

```

15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 ///////////////////////////////////////////////////////////////////
21
22
23 module program_counter(
24     input clk, rst,
25     input logic take_branch,
26     input [8:0] offset,
27     output logic [8:0] pc
28 );
29
30 always @(posedge rst, posedge clk) begin
31     if(rst)
32         pc <= 0;
33     else if(take_branch)
34         pc += offset;
35     else
36         pc += 1;
37 end
38
39 endmodule
40

```

Figure 3: Program Counter with Branching Functionality

Appendix B – Images of Code Working on Hardware

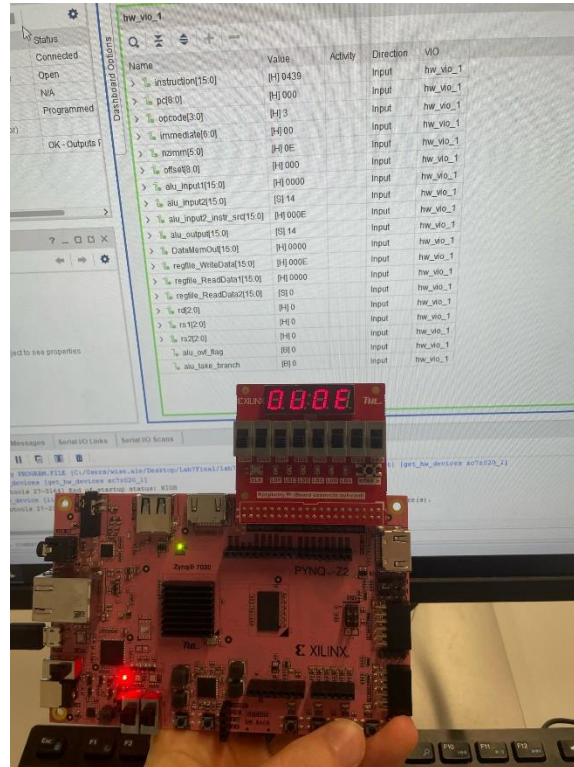


Figure 4: Setting Register x8 to 14

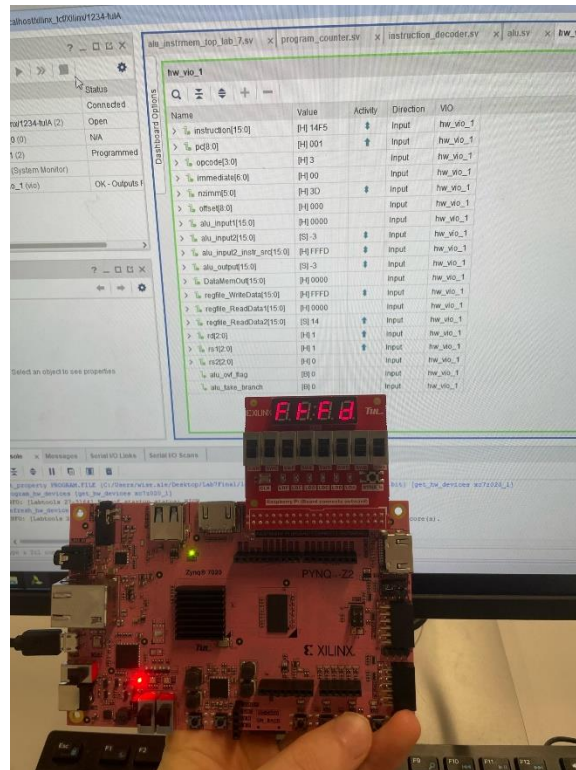


Figure 5: Setting Register x9 to -3

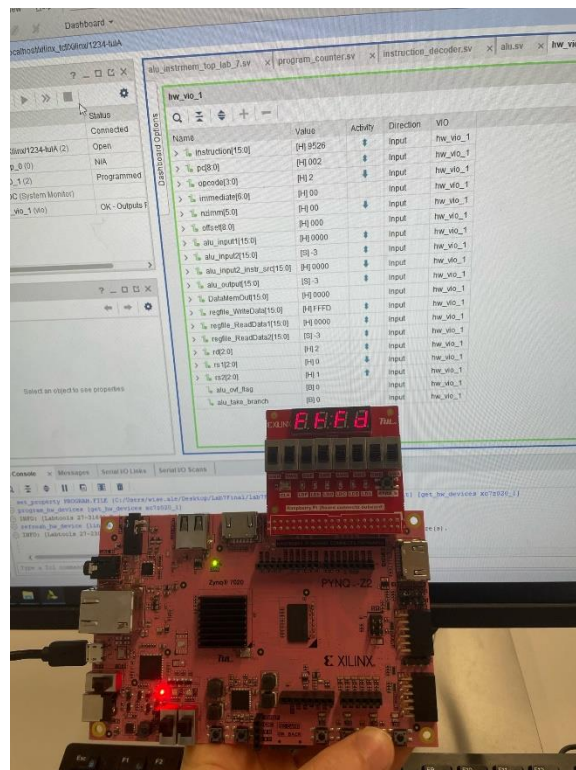


Figure 6: Copying Register x9 into x10

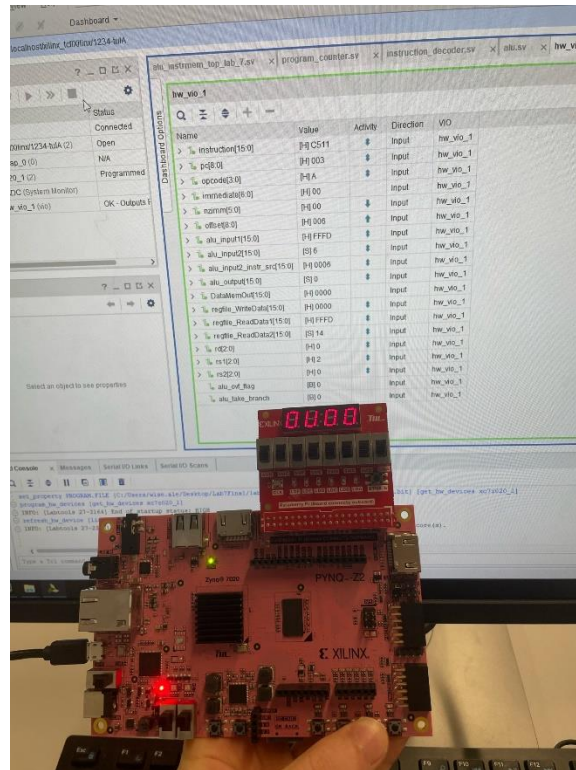


Figure 7: Checking if x10 is 0

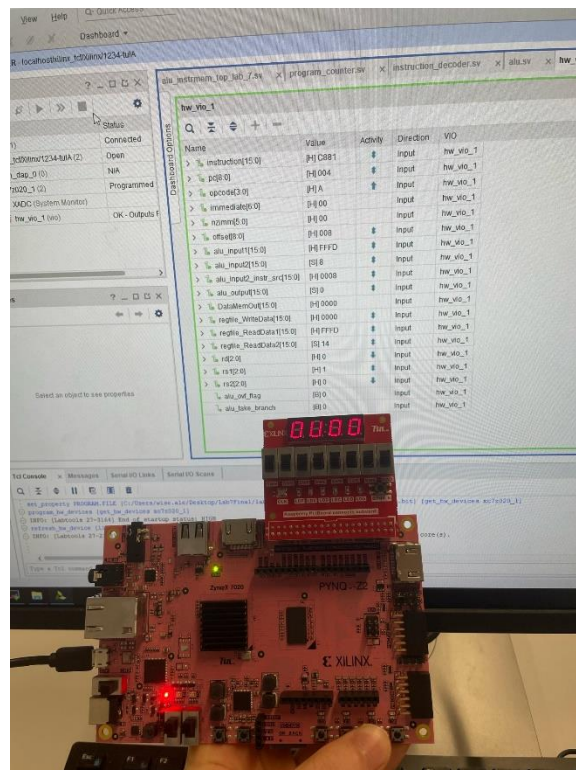


Figure 8: Checking if x9 is 0

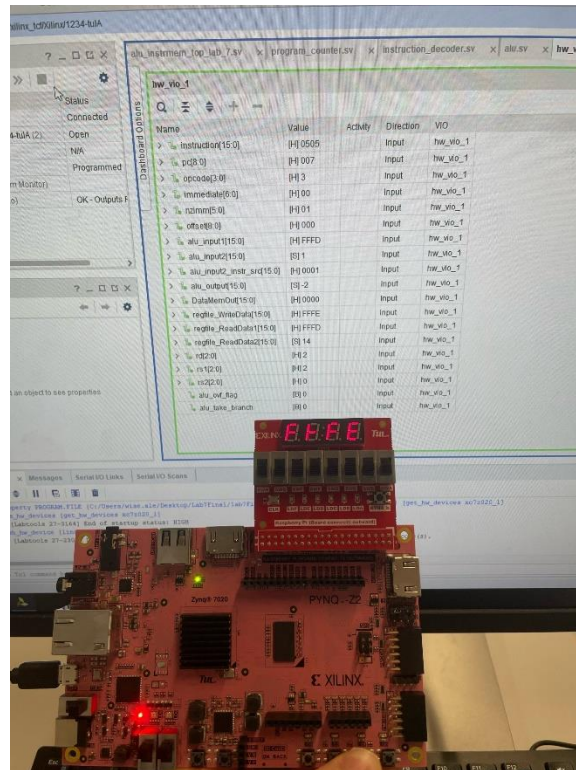


Figure 11: Incrementing x10 by 1 (-2)

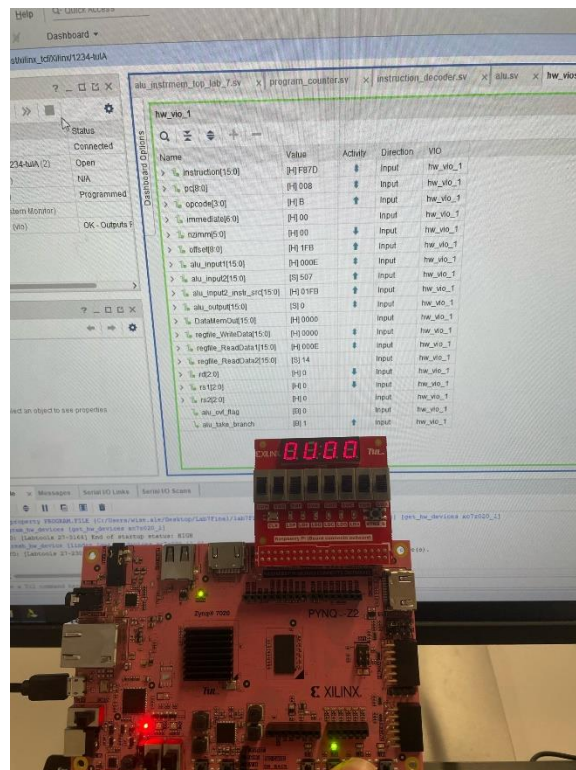


Figure 12: Branching Back to top of Loop

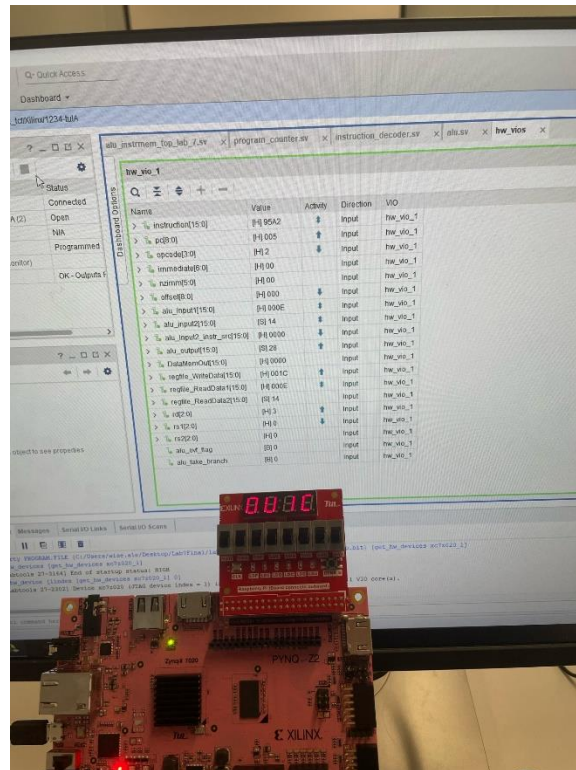


Figure 13: Adding x8 to x11 a Second Time (28)

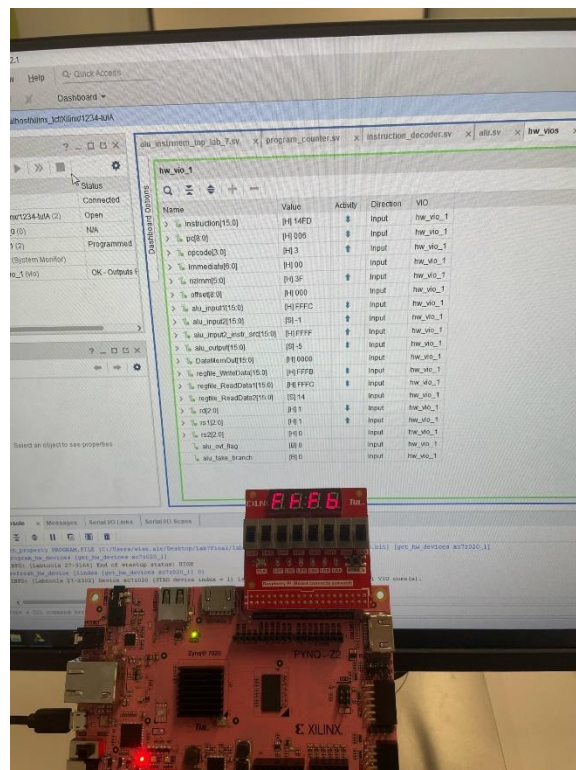


Figure 14: Decrementing x9 by 1 (-5)

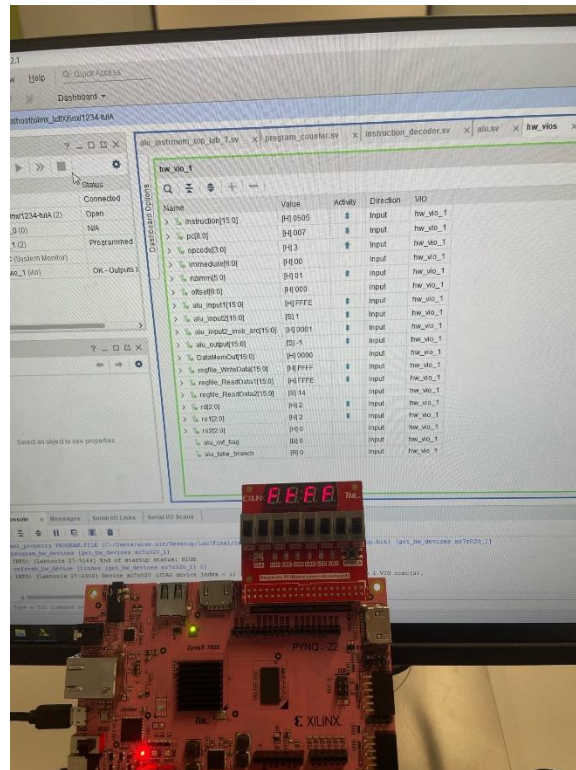


Figure 15: Incrementing x10 by 1 (-1)

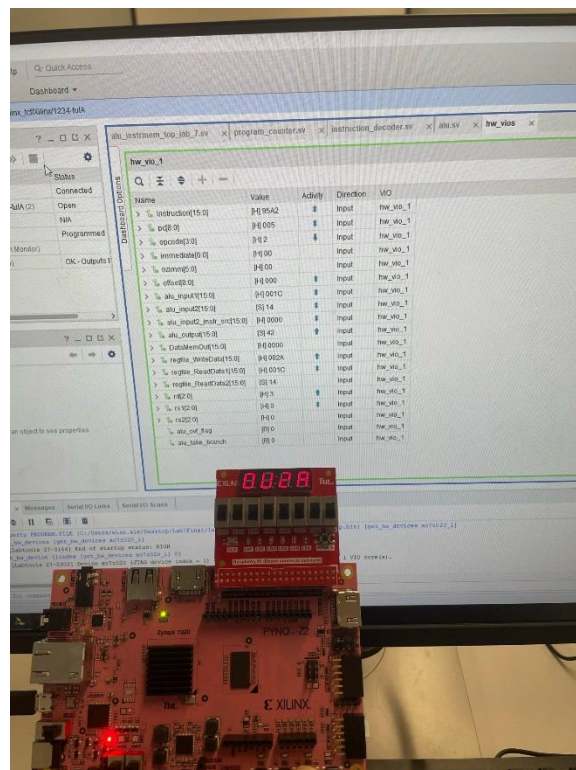


Figure 16: Adding x8 to x11 a Third Time (42)

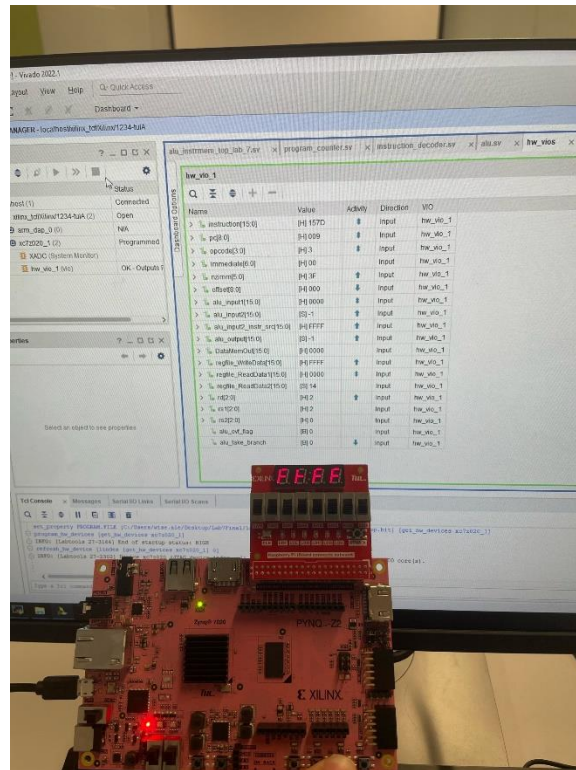


Figure 19: Branched to out_negative, Setting x10 to -1

