Lab #8: Adding Branch Logic to the Datapath to Complete Your  
Computer

EECE 2323 – Prof. Xiaolin Xu

Date: 04/012/2022

Name: Fengyi Quan && Yichen Sun

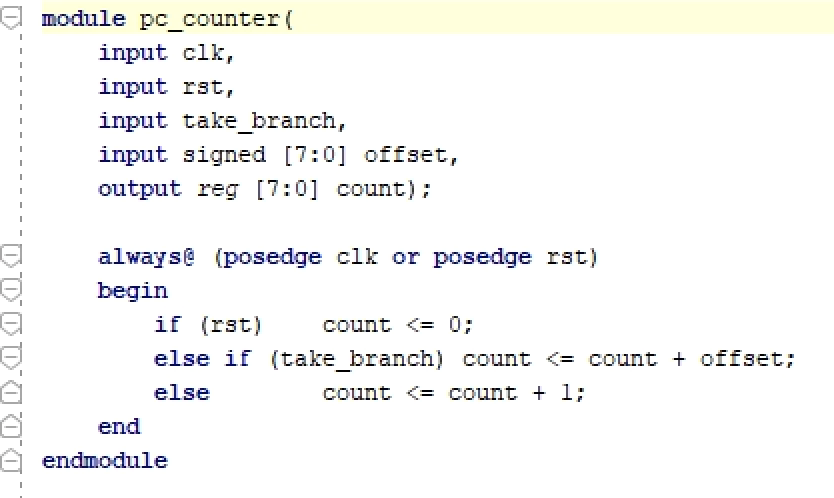
Northeastern University

1. **Background & Purpose**

The object of this lab is to make changes to the previous lab. It adds additional functionality to support branch. There are two steps to complete this lab. First is to rewrite PC counter to support offset jump. Second is to set SSD to the datapath to display the output of the ALU instructions. After that, it can demonstrate the feasibility of the system design by showing successfully running multiplier of two arbitrary number, manual single-instruction dispatch, and automatic, sequential, multiple-instruction, dispatch from instruction memory. It also supports executing arbitrary programs written in our MIPS-like assembly language.

1. **Prelab**

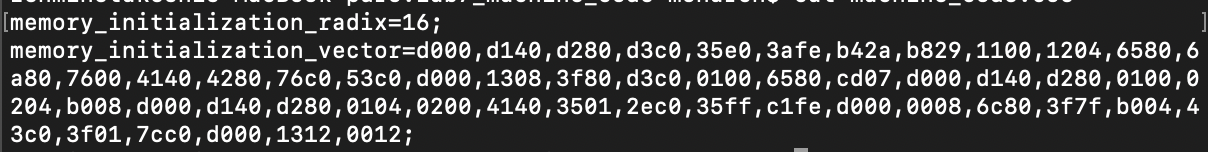
2.1.1 Design a PC logic with branching capability



2.1.2 A Signed Multiplier in Assembly

Please see Appendix B: Assembly Program

2.1.3 Generate the Machine Codes using the Assembler



1. **Design implementation**

To implement a PC which supports branching. All need to do is to add another take\_branch flag as input. When flag set, the result should increment or decrement by the corresponding offset which comes from the machine code. If reset signal is set, the pc counter will go to 0. In the event that neither reset nor take branch are on, pc will just simply be incremented by 1.

The multiplier design is to check if one of the numbers is not negative and use that number as counter. Then add another number to the result register that times. If both are negative, convert one to its two-complement positive number and use as counter. Then do the same steps above, after done, convert the result back to the negative, because if program goes into this step, it does not need to worried about xor the sign of two inputs. Because it already checked two signs when start.

1. **Results and Analysis**

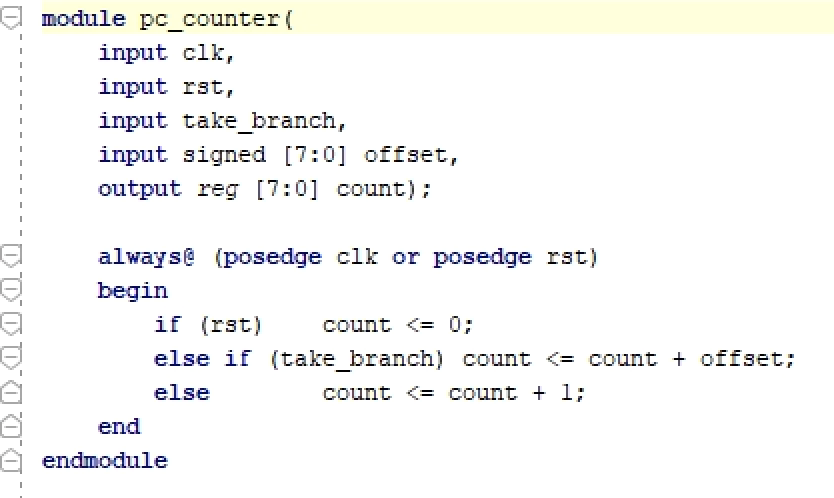
The multiplier and multiplicand should be set within the 8-bit data range with any sign. In the design, the baseline as two signed positives could be simply achieved by multiply them together with no further operations. This behavior also happens to two signed negatives since you do not need to consider whether the number is positive and negative. To encounter these two situations, the implementation requires at least four registers. The first register 0 is the initial count register as number zero, the register 1 and register 2 stores multiplier and multiplicand, and register 3 is used for product calculation. After assigning signed values to each register, the product register adds value in register 2 for every occurrence of value in register 1. This iteration ends as soon as counter register 0 is equal to register 1, then save and load data to retrieve the result output. However, the implementation of different signed numbers is slightly different. The assembly program should include guidelines to obtain the signs for the register and adapt to the final result.

1. **Conclusion & Recommendations**

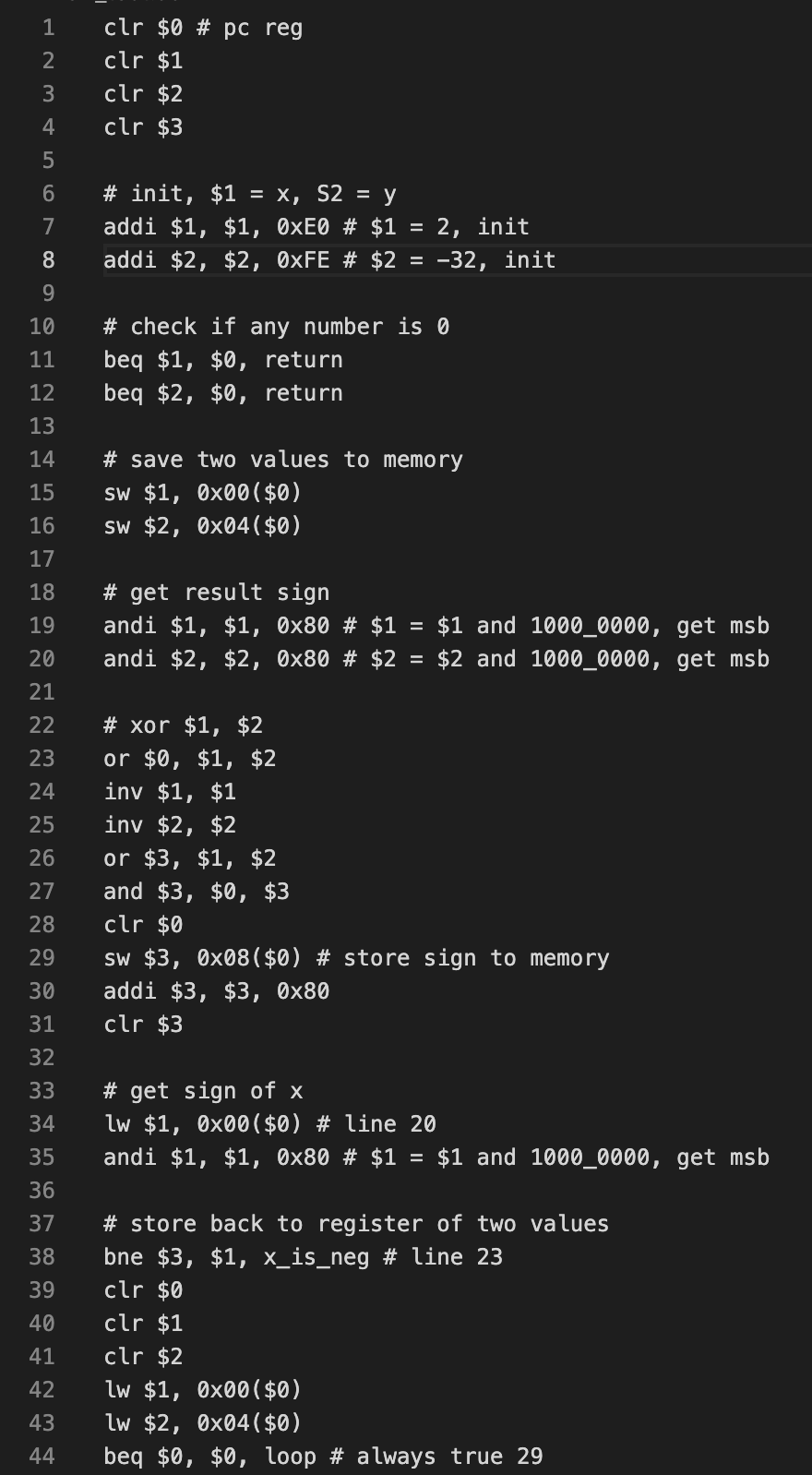
The objective of this lab is to complete our previous design in lab 7 by rewriting the Program Counter (PC) logic and set the seven-segment display (SSD) to the datapath. In the prelab, the signed multiplier is successfully implemented by an assembly program which performs multiplication operation with two signed numbers (in two’s complement system). The range of multiplier and multiplicand is designed to be within the 8-bit data range. After generating the machine codes using the assembler, the program runs correctly in the FPGA with hardware Manager. The VIO and the SSD test the design and step each line in assembly to obtain the final result. However, our design fulfills the multiplication operation with two negatives and two positives but has mistakes in operating one positive signed number and one negative signed number.

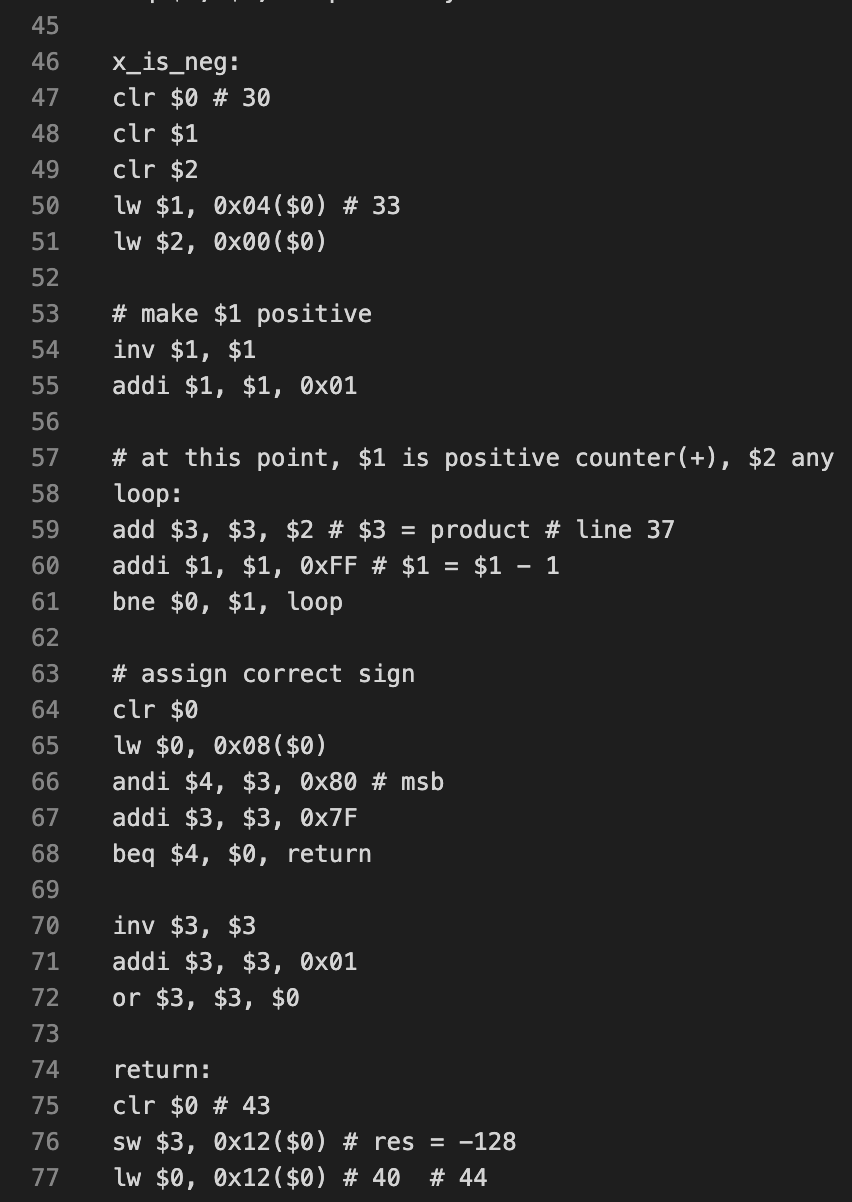
1. **Appendices**

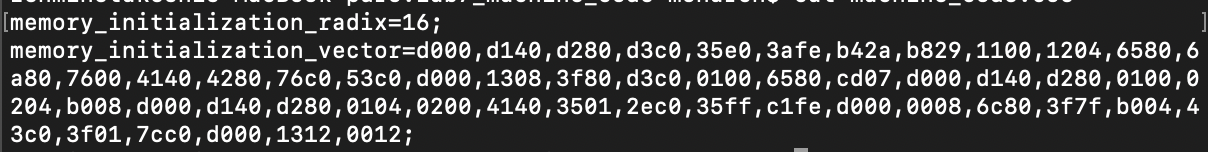
Appendix A: PC counter Module



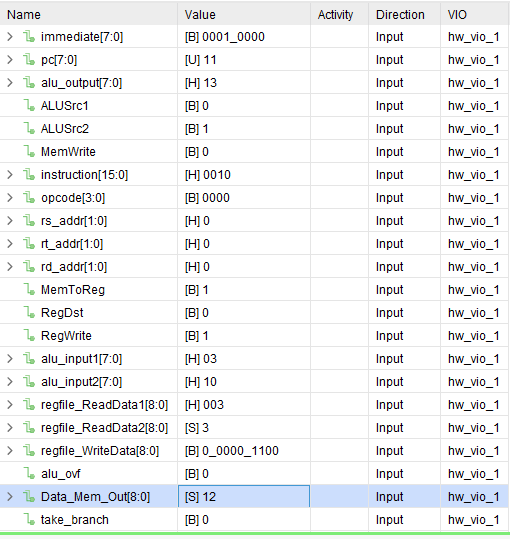
Appendix B: Assembly Program (2 \* -32)





Appendix C: Machine Code (2 \* -32)

Appendix D: Output screenshots (3 \* 4)



Appendix E: Output screenshots (2 \* -32)

