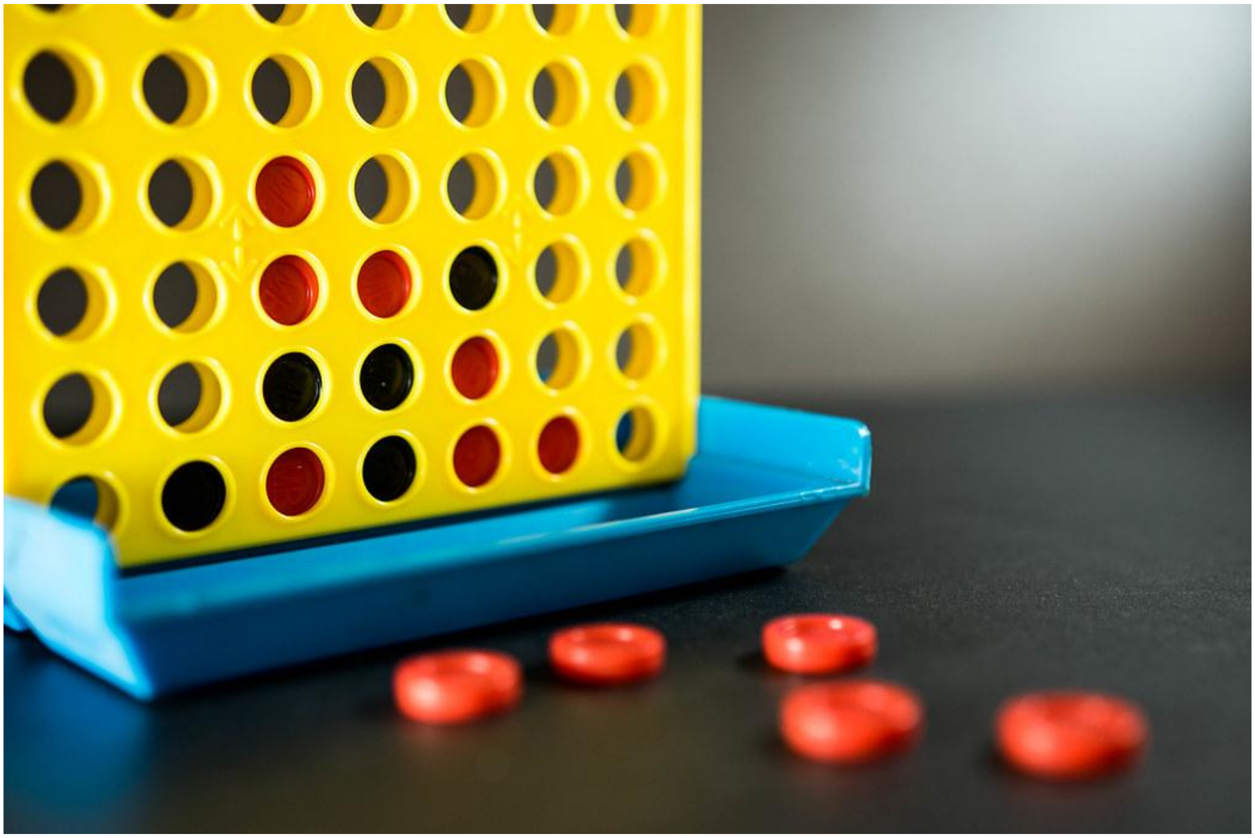
FINAL PROJECT: Connect 4



Created by: John ‘Jack’ Mismash, Andrew Porter, Vanessa Bentley, Zach Phelan
University of Utah
ECE 3710 Fall 2021 – Computer Design Lab

Abstract — **Our final project is the culmination of a semester's worth of work. We decided to specialize our general computer to create a 'connect four' digital game. Based off of the lab four CPUs, we utilize each lab and their respective modules to create a two player video game with a vga connection that displays the current game state. This report outlines the overview, design, synthesis and analysis of the final project of ECE 3710, Connect 4.**

I. Introduction/Game Specifications

The final project specifications for this computer application will be broken into a few segments: the general rules/structure of the game (the problem statement/objective), the hardware and software components of the project, and the interface and integration of peripherals. The rest of the report will focus on the conclusion of the project, as well as describe each team member's responsibilities and contributions.

Below is the general rule set of the Connect 4 game, as well as any specifications that change the way the system must be designed:

1. Connect 4 is played by taking turns and selecting a column to place a piece (a red token for Player 1, and a yellow token for Player 2) at the lowest position relative to the bottom of the game board, and with an opening row/column position that is not already occupied by another piece.
2. After the total number of turns is higher than six, then there is a chance that a player can win the game.
3. The objective of the game is to connect four tokens belonging to the same player that lines up horizontally, vertically, or diagonally.
4. Once a player gets 'connect four' or there are no spaces on the game board, the game is over.
5. The game board consists of a six by seven block grid where each block represents a position in which a player may possibly place a piece.
6. The game may be played with one (human vs. computer) or two players (human vs. human), although our design focused on a two-player mode.

II. Design Overview

A. Hardware Components

The hardware components of this final project consist of the following:

1. FPGA: The FPGA board allows us to contain the logic of the circuit for our game and holds values in both memory and registers that are used to play and display the game.
2. VGA Monitors/TV: This component displays the gamboard by connecting with the FPGA board.
3. SNES Controller: This component allows for the user to input data to the game about where to place the token.

The final portion of the lab is to connect the physical VGA connector to the FPGA. This physical connection takes place when the VGA is both connected to a VGA compatible monitor and connected to the VGA port on the FPGA at the same time. The VGA has 15 pins for connections. These VGA connections can be used for various projects such as low level video game development, image processing, and even a terminal window for custom processors. [1] VGA connections with older models of monitors and TV's used cathode ray tubes (CRT's) in order to display RGB values on a screen, and the specific connection that our group had was with a 16 bit color depth capability.

This connection requires two specific signals that interface with the FPGA, known as *hsync* and *vsync*, and these signals help determine what position on the screen and what pixel row/column to consider. The resolution that was defined, 640 x 480 pixels, allowed us to further help our design in deciding how we wanted to display our game. For further details on the design of the bit generation and signal handling for the VGA connection, adhere to section III and the appropriate subsection *VGA Connection*.

This project assumes the reader is familiar with the architecture of previous labs. As such, we will not discuss the data flow of our computer. However, a block diagram is provided for reference in Figure 1.

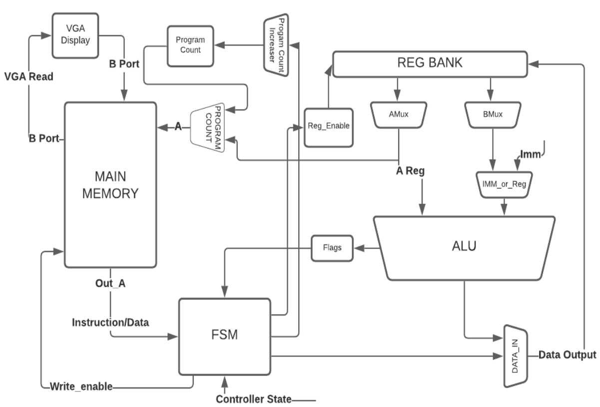


Figure 1: Block Diagram of Final Project

B. Software Design Components

Our computer has a two-port read/write capability and we will use this to our advantage in the system. While normal operations are handled through port A, such as instructions and writes to memory, port B is used by the VGA display to get the information needed for displaying the pixel by pixel signals sent.

Instructions that are used in the design of this game can be seen in Table 1. Our FPGA game design is able to accept many other instructions besides those seen in the table. However, these other instructions are not used in the logic of the game and as such are not included in this report. The other instructions not included are bitwise operations. One instruction that is not part of the standard instruction set, but is relevant to our game design is CTLST. This instruction is short for Controller Status and takes the input from the user and savis it directly to R0. This was important for our program as we needed a way to capture our user input.

Table 1: Instructions Used

Instruction	**Opcode**	**Instruction Type**	**Operands**
ADD	00000000	R-type	Rsrc, Rdest
ADDI	00000001	I-type	Imm, Rdest
SUBI	00001001	I-type	Imm, Rdest
CMP	00001010	R-type	Rsrc, Rdest
CMPI	00001011	I-type	Imm, Rdest
XOR	00001111	R-type	Rsrc, Rdest
JMP	01000000	Rel-type	Rtarget
JGE	01000011	Rel-type	Rtarget
JE	01000101	Rel-type	Rtarget
LOAD	10011001	Ind-type	RDst, RAddr

CTLST	10000000	Ind- Type	Imm, Rdest
STORE	11011010	Ind-type	RSrc, RAddr

Although 16-bit word sizes were recommended for this project, they also were a good design choice. These values gave us enough space to have individual codes for each instruction, and leave enough bits to do operations with sufficiently large immediate values. This was important as our assembly instructions grew because the amount of lines needed to jump increased.

C. Assembly Instruction Design

For the assembly design of this project, we divided the assembly code into three sections:

1. User Input
2. Token Placement
3. Check Win

Our first section is to update user input to the FPGA. This is done once at least every 12 clock cycles and is under the label Check_input in our assembly code in Appendix A. This loop continually updates user input until the user decides to move or place a token.

Token placement first begins by moving to a corresponding column. In Table 2, it can be seen that register R1 is the designated register for column selection. In our 7x6 game board, each of the six columns was assigned a number zero to six, left to right. A player input indicating a selection to a different column would manipulate the value in R1. R1 was also a register that was connected to the VGA display through our FSM. In the final design, the VGA logic would look to this register to know where to place the column indicator on the screen.

Once a column was selected by the user, the assembly code would go into .Select and .Token_Loop sections of the assembly code. .Select initializes R14 to point to the lowest row in the desired column, while also setting R15 to 0. By using load instructions, we were able to compare the R14 pointer location to see if a token had been placed there already. If so, we would point R14 to the next row up in the same column. In this way, we simulated the physical game by having the token occupy the lowest spot possible. At the end of this section, R15 holds the row number (0-5, bottom to top) and R14 holds the address in memory of the newly placed token.

At this point, we begin the third section: checking for a win. The check for win uses a host of temporary variables that can be seen in Table 2. These temporary variables are place holders that are used to check data for a win after a token has been placed. Systematically, we check to the right of the placed token for a horizontal win. If we come across an empty token spot, or the other player's token, then we begin checking on the left side. With each matching token, we increment register R5. When R5 reaches 3, we know that four tokens were placed in a row, and the player won. If a horizontal check does not result in a win, then we check the vertical direction. After vertical direction, we check for the diagonals, resetting R5 when we change directions. If a win is detected, our game logic enters a loop that keeps the game in a winning state until the player presses ‘enter’ on the game controller to start a new game. If a turn does not result in a win, the player number in register R2 is switched and a new turn takes place.

D. Game Board in Memory

For our design of the game, we found it most convenient for the game to be stored in memory. We decided that with the limited amount of registers needed for our game logic, this would be a better course of action.

Row							
5	00	00	00	00	00	00	00
4	00	00	00	00	00	00	00
3	00	00	00	00	00	00	00
2	00	00	00	00	00	00	00
1	00	00	00	00	00	00	00
0	00	00	00	00	00	00	00
Column	**0**	**1**	**2**	**3**	**4**	**5**	**6**

Figure 2: Game Board in Memory

In Figure 2, we can see the layout of this game board. Each pair of zeroes refers to an abbreviated 16-bit memory block. The two least significant bits determine if the token spot is occupied: 00 for empty, 01 for player 1, 10 for player 2. Register R3 will always point to the address of token spot (0,0). From that point, we can reference every other token spot. For example if R3 contains the memory address 0x800, then token spot (0,0) is at memory address 0x800, and token spot (0,5) is at memory address 0x805. This decision was made to ease the load of the assembly code to check for a win.

As an example of the process, if a token was put in token place (2, 6), then checking the tokens underneath required only subtracting an immediate 7 from the temporary address pointer in R10. Likewise, checking the bottom right diagonal or top right diagonal only necessitated subtracting 6, or adding 8 to R10, respectively. Edge cases where these checks would “go off” the board were handled by updating R6 and R15 and checking their bounds. In this way, tokens that would appear in the upper right of the placed token in memory that are actually on the same row, are disregarded.

E. Register Designations

Table 2 gives a description of each register’s designated purpose. Registers for general-purpose use are not bolded and are not occupied in the course of the game. Register R9 is an optional-use token that was implemented to reduce the amount of time to check for wins. If R9 is less than 7, it is impossible for a winning situation to occur, so there is no need to check for a win. This reduces time taken by the FPGA logic during the game.

Table 2: Registers in Use

Register name	Use
R0	**User Input**
R1	**Column Number**
R2	**Player (A or B)**

R3	**GAME PTR**
R4	**R14 Game Value**
R5	**Win Count**
R6	**Temp Col Number**
R7	**Temp Game Value**
R8	**Temp Row Number**
R9	**Token Counter**
R10	**Temp Address Current Token Spot**
R11	Local variable
R12	Local Variable
R13	Local variable
R14	**Address Current Token Spot (Check_win)**
R15	**Row Number**

III. Tools Used

Below are the tools or software design specifications we used.

A. General Overview

1. CPU: The CPU controls the overall structure of the system that the game uses to run all instructions given by the user, as well as does any calculations for the game logic.
2. ALU: The ALU is contained within the CPU, and is responsible for any arithmetic calculations needed by instructions.
3. Assembler: The assembler essentially deconstructions instructions into bits for the computer to read.
4. FSM(s): Also known as Finite State machines, these code structures allow for sequencential logic for games.
5. Bit Generation: This module was important in specifying pixels and their respective RGB values for the VGA connection.

B. VGA Connection

To further explain the process of sending the *hsync* and *vsync* signals and the proper RGB values, we first start with our *top_level_counter* module. This module was responsible for handling the proper signals and controls the timing of the vga connection. More specifically, this module assigned the *hsync* and *vsync* values based on the current position on the screen, and had two main variables to maintain what each x and y position we were currently scanning across, named *hcount* and *vcount*.

These two counters were regulated by whether or not they were within the screen resolution size. The *top_level_counter* module also contains a state conversion module that translates our input from the user into a three-bit format to use within our project. These two variables, *hcount* and *vcount* were used within our *bitgen* module, which was responsible for outputting our specific 16 bit RGB value for each position on the screen, as well as our memory lookup address needed for checking the current game state. We also forwarded the column number and player identification (essentially whose turn it was) to this *bitgen* module as well to draw a column indicator for the player to make their selection. See *Figure 3* for a block diagram of the *top_level_counter* module.

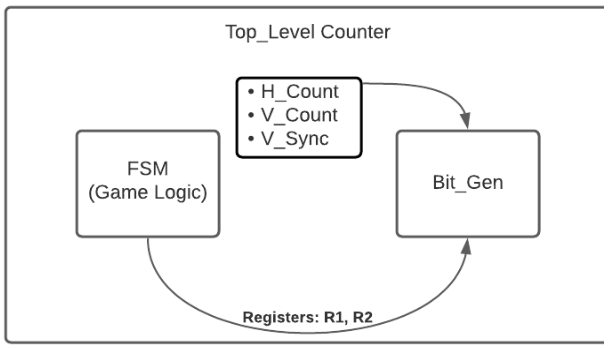


Figure 3: VGA Connection Block Diagram

Overall, our modules were integrated together with the *top_level_counter* being set as our top level entity, and this module calls upon the *FSM*, *Bitgen*, *Memory*, and *State Conversion* modules.

C. Python Assembler

In order to use the assembly instructions with our CPU, we would have to compile the instructions into a language the CPU would understand: machine code. Rather than doing this by hand, we decided to create an assembler in Python to perform this task.

The assembler would take in a text file containing the instructions and output the equivalent machine code instructions into a different text file. In order to be more confident and prevent recurring bugs, we also wrote many tests to verify behaviors.

The overview of the program is as follows: First, scan for labels and keep track of which line number the label will correspond to at any given time. Then, go through each assembly instruction, line by line, transforming the assembly into the machine code as specified in Table 1.

While there was an assembler provided, creating our own allowed for full flexibility to meet our programs needs. So when we needed to add the CTLST instruction, it was painless to accomplish. It was definitely preferable to write our own assembly than to translate all the assembly instructions by hand.

V. Project Experience

The initial project idea seemed feasible in the amount of time left. We were confident that we would be able to get a working Connect 4 game with a controller and potentially a single-player mode with an AI. But small issues piled up to the end, becoming too large to fix before presentations.

Often, we would be confident in the design and that it was working, but had not explicitly tested it. This led to assumptions being built off of assumptions that become difficult to untangle. It would have been more sustainable to create tests progressively, testing individual components to ensure that they work as expected.

While we were able to rectify most issues that we discovered, the memory interfacing with the VGA was the most difficult for us to solve. We did not manage to make it work, but we tried different methods. One such method was using a “generate” block to create one register for each piece on the game board. This piece would be updated when the cathode for the VGA display is going back to the top (*vsync* pin goes high). Then, each register would be read to determine the color for the corresponding block. While this should work in theory, the memory was not being read to these registers correctly. There was a small logic issue, but there is still an underlying issue that is preventing memory from being accessed by the VGA module.

Overall, we have learned to better set our expectations and to create smaller, more robust test benches. The key to building a working project is to verify and polish the components that make it up.

VI. Team Member Contributions

Each of four team members were each essential in our overall design and completion of many components within this final project. Below are some of the specific contributions each team member made, as well as how their strengths played into their role.

1. Andrew Porter - Andrew spent a significant amount of time designing the overall structure of the CPU and ALU, and debugging. He used his knowledge of CS 4400 to accurately create accurate assembly code for the game to run. He also helped with various portions of the VGA and internal connected components. *Appendix A* shows the specific code he contributed to.
2. Vanessa Bentley - Vanessa helped create and design the block diagrams for our final project and previous labs, and she also helped Andrew write and test the assembly code. *Appendix A* shows the specific code she contributed to.
3. John 'Jack' Mismash - Jack spent his time on the final project with the VGA connection, and testing the output of the RGB values propagated to the monitor, as well as integrating all the components needed from our previous labs into the *top_level_counter* and *bitgen* modules. He and Zach both were able to draw the Connect 4 game board and respective pieces on the monitos successfully.
4. Zach Phelan - Zach helped alongside Jack in debugging and testing the VGA connection, but also contributed by creating our Python Assembler file that helped our CPU read all of the instructions in binary.

VII. Conclusion

In conclusion, this report has provided an overview for setting up a Connect 4 digital logic game using the DE1-SoC programmable logic board (FPGA) and using the Quartus and ModelSim software for the design and simulation. The circuit functionality could have only been possible with each step of the digital logic design process, and each part of the design is crucial to providing the correct simulation. These results were able to be verified using the testbench and verifying the output on the FPGA board and monitor display as well. We can see that this design helps build the skills of designs with sequential and combinational logic/circuits as well as the use of clock timing.

Unfortunately, this project was not completed in its entirety, since we were unable to get a functioning connection to the memory module and update the display as needed. For further revisions of this project, we recommend further testing and design of the integration of the game state into the bit generation for the VGA output.

We also had hoped to use a SNES controller to retrieve input from the user, but we had to resort to only using the input buttons from the FPGA itself, as well as a reset switch to control the input.

VII. Bibliography

[1] Embedded Thoughts, J. (2016, December 30). *Driving a VGA monitor using an FPGA*. Embedded Thoughts. from https://embeddedthoughts.com/2016/07/29/driving-a-vga-monitor-using-an-fpga/

APPENDIX A: Assembly Code

```
.START
        XOR R2, R2   //Initialize Game Variables
        XOR R1, R1
        XOR R8, R8
        XOR R9, R9    //Game token counter
        ADDI 1,  R2   //Start with player 1
        //Set memory pointer R3 to address of game block

.Check_input

        CTLST         //Get the state of the controller

        CMPI 0, R0            //if R0 == 00 continue to check
        JE .Check_input

        CMPI 1, R0            //if R0 == 01 , move right
        JE .Move_right

        CMPI 2, R0                    //if R0 == 10 , move left
        JE .Move_left

        CMPI 3, R0                    //If R0 == 11 , selected
        JE .Select

        JUMP .Check_input    //Repeat the Loop

//Moves the cursor one position to the right. R1 is where the current
column is kept
.Move_right

        ADDI 1, R1
        CMPI  7, R1  //If R1 == 7, set it equal to 0
        JE .Start_over
        JUMP .Check_input


//Moves the cursor one position to the left. R1 is where the current
column is kept
.Move_left

        SUBI 1, R1
        CMPI  -1, R1  //If R1 == -1, set it equal to 6
        JE .Wrap_around
        JUMP .Check_input

//User select position. Now the token will be placed in the lowest
spot of the column(R1)
.Select

        XOR R15, R15          //Initialize R15 Loop Var
        XOR R14, R14          //Initialize R14 Displacement
        ADD R1, R14           //Getour column number/
displacement
        ADD R3, R14           //Points to lowest row position in
column in memory (i, j)

.TOKENLOOP
        CMPI 6, R15           // Check if we have iterated through
all rows
        JGE  .Check_Input     //Exit the loop
        LOAD R4, R14                  //Load Address Value of
R14 into R4
        CMPI 0, R4            //Compare value at location
        JE .ChangeTokenVal  //If spot not occupied by token,
change the value
        ADDI 1, R15                   //Increment the loop
        ADDI 7, R14                   //Point to the j+1th row
in the ith column
        JUMP  .TOKENLOOP


.ChangeTokenVal
        STORE R2, R14       //Store the player value (01 or 10)
at address pointed by R14
        ADDI 1, R9     // Increment token counter
        CMPI 7, R9
        JL   .Switch_player //Not possible to win with less than 7
tokens on the board
        JUMP .Check_win


.Start_over                                   //Sets column to 0
        XOR R1, R1
        JUMP .Check_input


.Wrap_around                       // Sets column to 6
        XOR R1, R1
        ADDI 6, R1
        JUMP .Check_input

.Check_win              //Check horizontally, vertically, and diagonally
for 4 in a row of one 'color'

.Horizontal_right//check horizontal to the right
        XOR R6, R6   //Clear column Variable
        ADD R1, R6   // Load Column into R6
        XOR R5, R5    // Clear Win Count
        XOR R10, R10  //Clear out R10
        ADD R14, R10  //MOV R14 to R10
        ADDI 1, R10   //Get address to the right
        ADDI 1, R6 //add to column number
        CMPI 7, R6 //Check if out of bounds
        JE .Horizontal_left  //Jump if out of bounds
        LOAD R7, R10 // get value stored in R10
        CMP R2, R7   //Compare value of current token, to the
value to the right
        JE .Horizontal_right_again

.Horizontal_left //Check horizontal to the left
        XOR R6, R6  //Clear column Variable
        ADD R1, R6  // Load column of piece
        XOR R10, R10  //Clear out R10
        ADD R14, R10  //MOV R14 to R10
        SUBI 1, R10   //Get value to the left
        SUBI 1, R6   //Move column left
        CMPI -1, R6   //check bounds
        JE .Vertical
        LOAD R7, R10 // get value stored in R10
        CMP R2, R7   //Compare value of current token,
        JE .Horizontal_left_again
        JUMP .Vertical

.Horizontal_right_again
        ADDI 1, R5     // ADD 1 to win count
        CMPI 3, R5    // Check if 4 in a row
        JE .Game_over
        ADDI 1, R10   //Get value to the right
        ADDI 1, R6 //add to column number
```

```
        CMPI 7, R6 //Check if out of bounds
        JE .Horizontal_left  //Jump if out of bounds
        LOAD R7, R10 // get value stored in R10
        CMP R2, R7   //Compare value of current token, to the
value to the right
        JE .Horizontal_right_again
        JUMP .Horizontal_left

.Horizontal_left_again
        ADDI 1, R5     // ADD 1 to win count
        CMPI 3, R5    // Check if 4 in a row
        JE .Game_over
        SUBI 1, R10   //Get value to the right of
        SUBI 1, R6   //Move column left
        CMPI -1, R6   //check bounds
        JE .Vertical
        LOAD R7, R10 // get value stored in R10
        CMP R2, R7   //Compare value of current token
        JE .Horizontal_left_again

.Vertical
.Vertical_down
        XOR R8, R8   //Clear row Variable
        ADD R15, R8   // Load row into R8, R15 is our loop
variable but also shows the row we selected
        XOR R5, R5    // Clear Win Count
        XOR R10, R10  //Clear out R10
        ADD R14, R10  //MOV R14 to R10
        SUBI 7, R10   //Get address below
        SUBI 1, R8 //sub to row number ////
        CMPI -1, R8 //Check if out of bounds
        JE .Diagonal //Jump if out of bounds
        LOAD R7, R10 // get value stored in R10
        CMP R2, R7   //Compare value of current token, to the
value below
        JE .Vertical_down_again
        JMP .Diagonal

.start_ladder_2
        JUMP .Start

.check_input_ladder_2
        JUMP .Check_Input

.Vertical_down_again
        ADDI 1, R5     // ADD 1 to win count
        CMPI 3, R5    // Check if 4 in a row
        JE .Game_over
        SUBI 7, R10   //Get value below
        SUBI 1, R8 //row number
        CMPI -1, R8 //Check if out of bounds
        JE .Diagonal //Jump if out of bounds
        LOAD R7, R10 // get value stored in R10
        CMP R2, R7   //Compare value of current token
        JE .Vertical_down_again

.Diagonal

.Diagonal_up_left
        XOR R6, R6   //Clear column Variable
        XOR R8, R8   //Clear row Variable
        ADD R15, R8
        ADD R1, R6   // Load Column into R6
        XOR R5, R5    // Clear Win Count
        XOR R10, R10  //Clear out R10
        ADD R14, R10  //MOV R14 to R10
        ADDI 6, R10   //Get address diagonal up left
        SUBI 1, R6 //sub to column number
        ADDI 1, R8  //ADD row number
        CMPI 6, R8 //Check if out of bounds
        JE .Diagonal_down_right //Jump if out of bounds
        CMPI -1, R6 //Check if out of bounds
        JE .Diagonal_down_right //Jump if out of bounds
        LOAD R7, R10 // get value stored in R10
        CMP R2, R7   //Compare value of current token
        JE .Diagonal_up_left_again
        JUMP .Diagonal_down_right

.Diagonal_up_left_again
        ADDI 1, R5     // ADD 1 to win count
        CMPI 3, R5    // Check if 4 in a row
        JE .Game_over
        ADDI 6, R10   //Get value below
        SUBI 1, R6 // sub to column number
        ADDI 1, R8  //ADD row number
        CMPI 6, R8 //Check if out of bounds
        JE .Diagonal_down_right //Jump if out of bounds
        CMPI -1, R6 //Check if out of bounds
        JE .Diagonal_down_right //Jump if out of bounds
        LOAD R7, R10 // get value stored in R10
        CMP R2, R7   //Compare value of current token
        JE .Diagonal_up_left_again

.Diagonal_down_right
        XOR R6, R6   //Clear column Variable
        XOR R8, R8   //Clear row Variable
        ADD R1, R6   // Load Column into R6
        ADD R15, R8
        XOR R10, R10  //Clear out R10
        ADD R14, R10  //MOV R14 to R10
        SUBI 6, R10   //Get address diagonal up left
        ADDI 1, R6 //add to column number
        SUBI 1, R8  //sub row number
        CMPI -1, R8 //Check if out of bounds
        JE .Diagonal_up_right //Jump if out of bounds
        CMPI 7, R6 //Check if out of bounds
        JE .Diagonal_up_right //Jump if out of bounds
        LOAD R7, R10 // get value stored in R10
        CMP R2, R7   //Compare value of current token
        JE .Diagonal_down_right_again
        JUMP .Diagonal_up_right

.start_ladder_3
        JUMP .start_ladder_2

.check_input_ladder_3
        JUMP .check_input_ladder_2

.Diagonal_down_right_again
        ADDI 1, R5     // ADD 1 to win count
        CMPI 3, R5    // Check if 4 in a row
        JE .Game_over
        SUBI 6, R10   //Get value below to the right
        ADDI 1, R6 //add to column number
        SUBI 1, R8  //sub row number
        CMPI -1, R8 //Check if out of bounds
        JE .Diagonal_up_right //Jump if out of bounds
        CMPI 7, R6 //Check if out of bounds
        JE .Diagonal_up_right //Jump if out of bound
        LOAD R7, R10 // get value stored in R10
        CMP R2, R7   //Compare value of current token
        JE .Diagonal_down_right_again
```

```
.

.Diagonal_up_right
        XOR R6, R6   //Clear column Variable
        XOR R8, R8   //Clear row Variable
        ADD R1, R6   // Load Column into R6
        XOR R5, R5    // Clear Win Count
        XOR R10, R10  //Clear out R10
        ADD R14, R10  //MOV R14 to R10
        ADD R15, R8 // MOV R15 to R8
        ADDI 8, R10   //Get address diagonal up right
        ADDI 1, R6 //ADD to column number
        ADDI 1, R8  //ADD row number
        CMPI 6, R8 //Check if out of bounds
        JE .Diagonal_down_left //Jump if out of bounds
        CMPI 7, R6 //Check if out of bounds
        JE .Diagonal_down_left //Jump if out of bounds
        LOAD R7, R10 // get value stored in R10
        CMP R2, R7   //Compare value of current token
        JE .Diagonal_up_right_again
        JUMP .Diagonal_down_left

.Diagonal_up_right_again
        ADDI 1, R5     // ADD 1 to win count
        CMPI 3, R5    // Check if 4 in a row
        JE .Game_over
        ADDI 6, R10   //Get value below
        ADDI 1, R6 // ADD to column number
        ADDI 1, R8  //ADD row number
        CMPI 6, R8 //Check if out of bounds
        JE .Diagonal_down_left //Jump if out of bounds
        CMPI 7, R6 //Check if out of bounds
        JE .Diagonal_down_left //Jump if out of bounds
        LOAD R7, R10 // get value stored in R10
        CMP R2, R7   //Compare value of current token
        JE .Diagonal_up_right_again

.Diagonal_down_left
        XOR R6, R6   //Clear column Variable
        XOR R8, R8   //Clear row Variable
        ADD R1, R6   // Load Column into R6
        XOR R10, R10  //Clear out R10
        ADD R14, R10  //MOV R14 to R10
        ADD R15, R8 // MOV R15 to R8
        SUBI 8, R10   //Get address diagonal down left
        SUBI 1, R6 //sub to column number
        SUBI 1, R8  //sub row number
        CMPI -1, R8 //Check if out of bounds
        JE .Switch_player //Jump if out of bounds
        CMPI -1, R6 //Check if out of bounds
        JE .Switch_player //Jump if out of bounds
        LOAD R7, R10 // get value stored in R10
        CMP R2, R7   //Compare value of current token
        JE .Diagonal_down_left_again
        JUMP .Switch_player

.Diagonal_down_left_again
        ADDI 1, R5     // ADD 1 to win count
        CMPI 3, R5    // Check if 4 in a row
        JE .Game_over
        SUBI 8, R10   //Get value below the left
        SUBI 1, R6 //add to column number
        SUBI 1, R8  //sub row number
        CMPI -1, R8 //Check if out of bounds
        JE .Switch_player //Jump if out of bounds
        CMPI -1, R6 //Check if out of bounds
        JE .Switch_player //Jump if out of bounds
        LOAD R7, R10 // get value stored in R10
        CMP R2, R7   //Compare value of current token
        JE .Diagonal_down_left_again
        JUMP .Switch_player

.Switch_player                  //change color
        CMPI   1, R2                    //Check If we are player
1 or player 2
        JE      1               //Skip over the next instruction
        XOR   R2, R2                    //R2 = 0 in case that R2
== player 2 so that adding 1 gets us to 1
        ADDI 1,  R2                     //If we are player 1,
increments to 2, or if at 2 goes to 0 then to 1
        JUMP .check_input_ladder_3

.Game_over              //Ended with winner or tie
        CTLST      //wait for enter button to start new game
        CMPI    3, R0
        JE       .start_ladder_3
        JUMP  .Game_over
```