ECE 350 Final Project: **Tic-Tac-Toe**

Anika Radiya-Dixit
Pritak Patel
Bruna Liborio

26 April 2016

Project Overview

For the final project, we implemented a tic tac toe game using our processor, MIPS code, the DE2 board, and a matrix of LEDs wired on a breadboard. We added additional features such as score tracking up to 15 points for each player (best out of 29 games), and a reset button to restart the game once a previous game finishes. The following sections describe our design in detail, including implementation, changes made to our processor, as well as difficulties we faced and how we solved problems that arose.

Design

We implemented a two player tic-tac-toe game. The game takes input from a computer keyboard, with the key {P} indicating "play again" once a game is over, and keys {Q, W, E, A, S, D, Z, X, D} representing the different tic-tac-toe slots. The output is displayed as either a red or green LED corresponding to the correct player and the slot indicated by the keyboard press. The program also keeps track of the turns and which turn corresponds to which player/color. Once an LED for a slot is used, the other LED in that slot is disabled since a player can no longer play in that spot. A single game ends when either all slots are filled or a player has gotten 3 in a row; this condition is checked after each turn. After each game, the players can press the P key to start another round. Players can enjoy up to 15 games each before the scores (displayed on the DE2 board's 7-segment displays) restart again from 0.

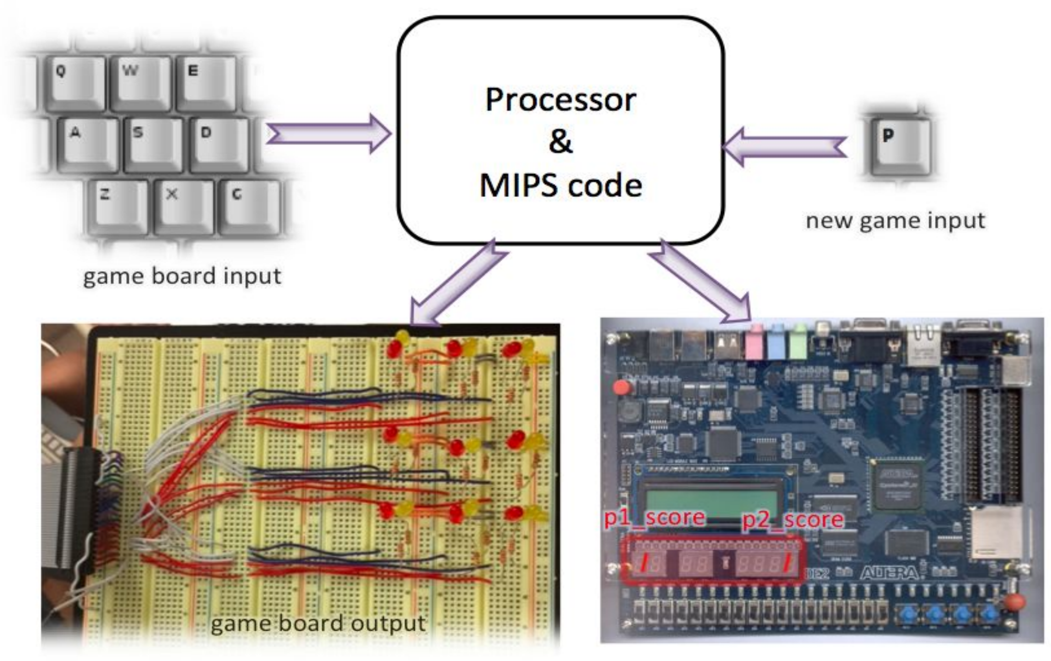


Figure 1: Flow of chosen inputs to chosen outputs (Note that here Player 2 LEDs are yellow, whereas the final product actually used green LEDs).

Input and Output

Keyboard Input:

“New Game” signal

```
P
```

Board: Each letter represents input to a tic-tac-toe slot.

```
Q    W    E
A    S    D
Z    X    C
```

Output: 3x3 array of [red, green] LEDs, each set corresponding to a tic-tac-toe slot. Player 1’s choices light up the red LEDs, and Player 2’s choices light up the green LEDs.

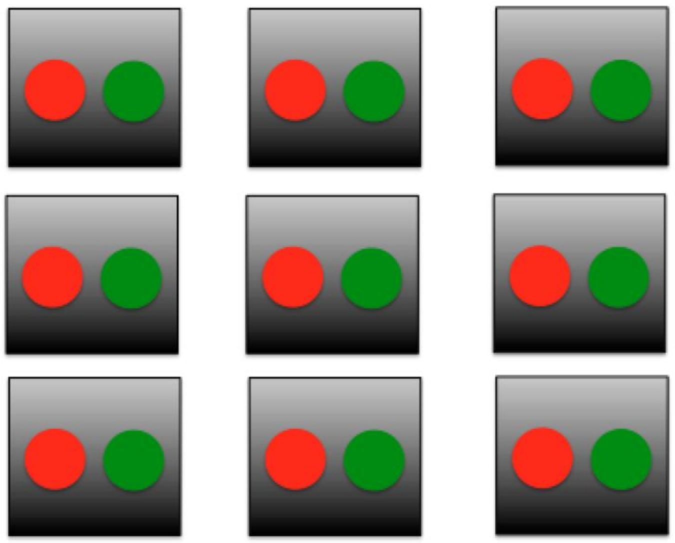


Figure 2: Basic board layout and concept.

***NOTE** that below the Player 2 LEDs are shown below in Figure 3 are yellow. The final Player 2 LEDs were switched to be green to increase brightness; however, this image was created and formatted before this change was made.*

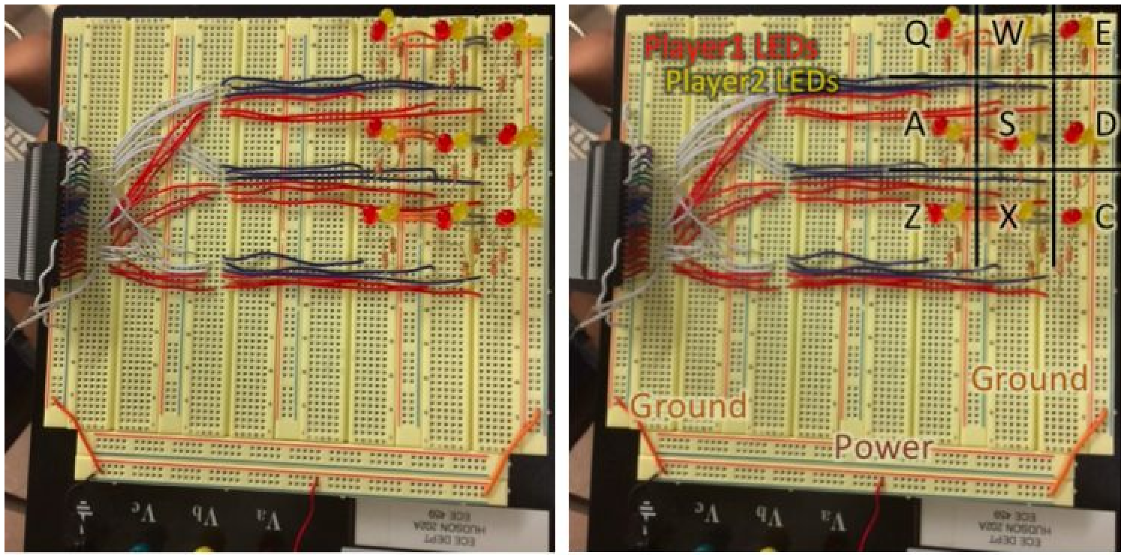


Figure 3: 3x3 array of red and yellow (later changed to green) LEDs for each tic-tac-toe slot. Only one LED per block will be lit per turn, indicating which player played in the slot. Code logic prevents two LEDs from lighting up at the same time.

Processor Changes

The project required more processor changes than originally foreseen. Since the game setup and MIPS code relied on setting aside registers to check against specific input and values as well as to take in specific values, many of the changes centered on the register file. In addition, the code used many ‘branch not equal’ and ‘branch equal’ instructions. While branch not equal was already implemented, branch equals had to be added and took the place of the blt instruction in the processor. Modules were also added for game logic, to output what was needed for the board based on processor module inputs and outputs, and for equal to 1 and equal to 2 checking since this comparison was used widely to produce final game logic output to control LEDs and the score 8 segment displays. In addition, the multiplication and division modules were added, as well as bex and setx.

List of Changes (which are described in detail below):

- Register file changes to provide direct inputs and outputs
- Blt instruction changed to beq instruction
- ‘Game logic’ module added
- ‘Equal to 1’ and ‘equal to 2’ modules added
- Mult/div added
- Setx and bex instructions added

Register File Changes

In order to implement the tic tac toe game simply and robustly, the MIPS game code was set up to reserve certain registers for certain values. Specifically, registers 6 to 14 were reserved as board slots, while register 15 was reserved for keyboard input. The slot registers were initialized with keyboard values which were always compared to register 15, the keyboard input. This made is to that whenever a slot register value matched keyboard input, it meant that the key was pressed and the value in that slot register was then almost immediately changed to indicate that the slot had been used and to prevent the slot from being used again.

Since this register system was fundamental to the game and to our MIPS code logic, most processor changes occurred within the register file so that register outputs could be read immediately and so that certain registers were always taking in certain values. In particular, register 15 was extracted from the general register file generation loop so that it could constantly take in keyboard input. Register 15's write data was wired directly to keyboard input so that this register constantly kept whatever value was the most recent keyboard press. The write enable bit was also set to the key pressed signal from the keyboard so that this register would always write the keyboard value pressed whenever a key was pressed thus keeping the keyboard input.

In addition, registers 6 to 14 in the code were reserved to represent board slots. In order to correctly and efficiently implement the game logic and check winning conditions, the values in these board slot registers had to be constantly read by the game logic module that was added to the processor. This required slight modifications to the register file. 9 output wires were added to the register file module which were each assigned to one of the slot register outputs. This then allowed the processor stage with the register file and in turn the processor module itself, which instantiates the stage, to have access to the values in these registers at all times. These register 'probing' wires were then passed into game logic which used the values in these registers to check for win conditions and to produce the correct game output at each clock cycle. This probing technique was also used for registers 4 and 5 which held the player's score and were also accessed by the processor which in turn sent it to game logic to produce final game output.

Instruction Changes

Besides changes to the register file, the code was written to use both 'branch not equals' and 'branch equals' instructions; since branch equals was not originally implemented, the processor had to be modified to support it. While bne already had processor hardware support and an opcode, beq did not. Since beq would have to be an I type instruction in order to jump to specific addresses/labels, one of the many available customizable R instructions could not be used, and the blt command that was originally implemented in the processor was converted to a beq command. This allowed existing muzes and hardware to be repurposed and allowed for the assembulator to continue being used to compile the written code. Had a new instruction with a completely new opcode been implemented, the assembulator would not have recognized it and those instructructions would have to have been added or modified by hand. Simply converted the processor blt logic to beq logic avoided this issue and by still using blt in the assembulator when it really meant branch equals produced the desired behavior without having to manually change/add binary instructions.

Game Logic Modules

A game logic module was added to handle game output. This module had the ‘probing’ wires from the slot registers, registers 6 to 14, and the ‘probing’ wires from the score registers, register 4 and 5, as inputs. The module outputs two arrays with 9 slots meant to be LED arrays, one controlling the red LEDs and one controlling the green LEDs. The slot registers work in that they hold a keyboard value when a keyboard key is pressed, the code detects if register 15’s value is equal to one of the keyboard values in the slot registers. If it is, the code then puts the player number who pressed the key, the player whose turn it is, into that register, so that the register will now hold a one or two indicating that the board slot has been played in and that the slot should now light up. Thus the game logic module checks all slot register values against one and two using ‘equal to one’ and ‘equal to two’ modules which were added with the sole purpose of comparing a number to one or to two respectively. If slot values match one, the LED 1 array slot corresponding to that board slot becomes high. If slot value match two, the LED 2 array slot corresponding to the board slot that got a match becomes high. These arrays are one of the final outputs of the game and processor and are used as DE2 board inputs which are connected to the final LED circuit representing the tic tac toe board and which light up the corresponding board slot LEDs when they are asserted/high in the correct colors based on the source array.

The game logic module also contains input for the player score wire probes and directs these inputs to a ‘hex to binary’ module which converts the value to 8 binary bits, a set of 8 for each score. These bits are outputs of the game module into the processor and eventually into the DE2 board where they are assigned to pins for eight segments displays.

Basic Processor Updates

Single cycle multiplication and division modules were added to the processor and incorporated to be useable instructions. This involved adding opcode recognition as well as muxes to control processor flow. Bex and setx were also implemented with added muxes, logic, and status bit checks.

Challenges

We faced many challenges throughout the course of this project which we had to address one by one:

1. We were unable to add beq as an instruction due to the way the instruction architecture was set up as we described in the previous section.

    To address this problem we changed the blt instruction to perform the action of a beq instruction so that our assembulator would work and minimal changes would be required from the processor. As a result, we have submitted two versions of our verilog code - one that works with our project, and one that preserves the original blt instruction.

2. Our correctly working bypass network was making sequential add instructions perform the wrong calculations because we actually wanted the value to come out of the original register, but instead it was coming from an intermediate value.

Because the bypass network was working correctly, we decided to modify our mips code to fix this problem instead of breaking the processor. We changed all of the problematic add instructions to addi instructions to prevent the bypass network from being used and this solved the problem.

3. At the end of a single game, the next game would automatically place an input for the first players' turn because when we held down the last move on the previous game, the code would automatically restart itself and the keyboard key that was being pressed would quickly be put into the register when we didn't want it to.

   Instead of having the game restart itself automatically, which also didn't allow players to see how they won/lost (because the board would reset quickly), we implemented a button to reset the board and start the next game. The "P" button on the keyboard forced the game to go into a constant loop until the first player for the next game pressed their input.

4. We tested our code thoroughly on the waveform editor with various inputs and outputs and everything seemed to be working, but the DE2 board wasn't functioning correctly.

   We realized the problem may have been with the clock being too fast for the board/processor to run properly, which was in fact the problem. We slowed down the clock by using the PLL provided in the original skeleton, which ran the processor at 10 Mhz instead of 50Mhz and our board worked correctly.

Circuits

Construction Description

The main circuit built was to represent a tic tac toe board with LEDs; it was basically a personalized LED matrix. The circuit consisted of 18 total LEDs, 9 red and 9 green. The LEDs were arranged in pairs on the board so that, for example, the top left slot in the board had one green and one red LED. 9 such pairs were created to mimic a tic tac toe board. 220 KOhm resistors were connected between each LED and it's wire connection to the DE2 board. The sets of LEDs were all connected to the board through resistors and were turned on and off by board output. The code ensured that only one LED in each slot would be on at the same time. The circuit was built with wires, resistor, and LEDs all of which were trimmed to the specific sizes needed so that all the resistors, LEDs, and wires would lay close to the board. The layout was planned so that the circuit would look neat and presentable as a final product.

Rationale

An LED matrix was chosen as the main output because a large easy to see grid was desired. While the LED matrix could have provided the functionality needed, the display would have been smaller. Also, there was little to no good resources found online to help wire up an LED matrix and so it was determined that a personalized circuit would better meet what was desired out of the game. Although the wiring of

LEDs is not impressive, the circuit was built and wired logically and neatly in order to show effort and end with a good, finished product. Once the processor and code started working fully, the chosen output circuit did not disappoint and provided all the needed functionality for a tic-tac-toe game.

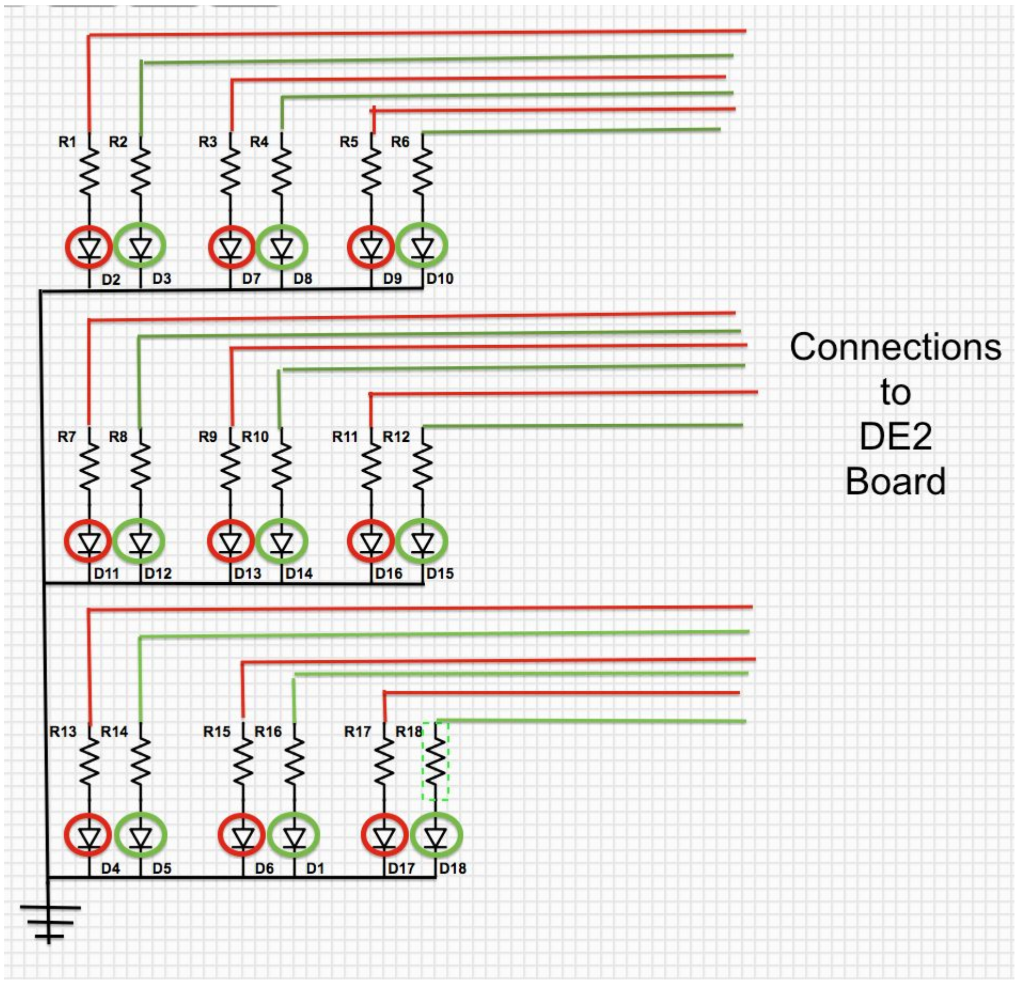


Figure 4: Circuit schematic for the tic-tac-toe board built with red LEDS and green LEDS respectively circled in red and green.

Sample game flow and progress of game outputs:

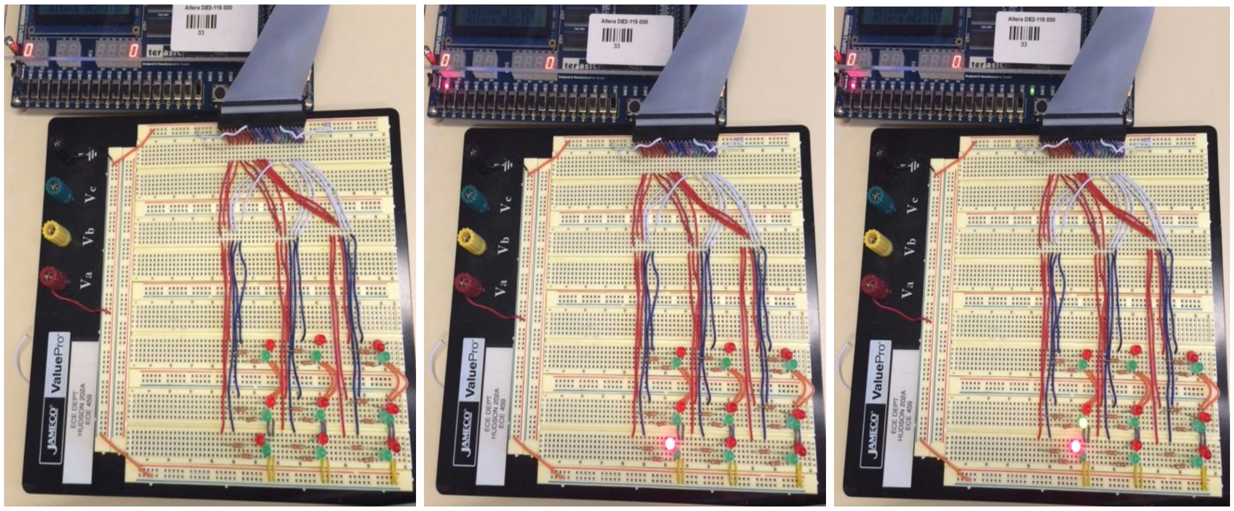


Figure 5.1: GAME 1: Initial Board → GAME 1: first player 1 turn → GAME 1: second player 1 turn →

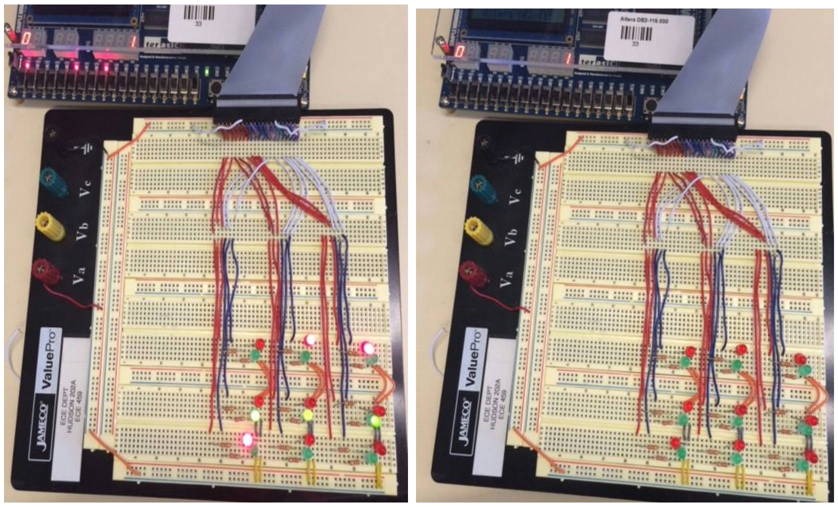


Figure 5.2: GAME 1: player 2 wins (score changes) → 'P': play again is pressed after the win →

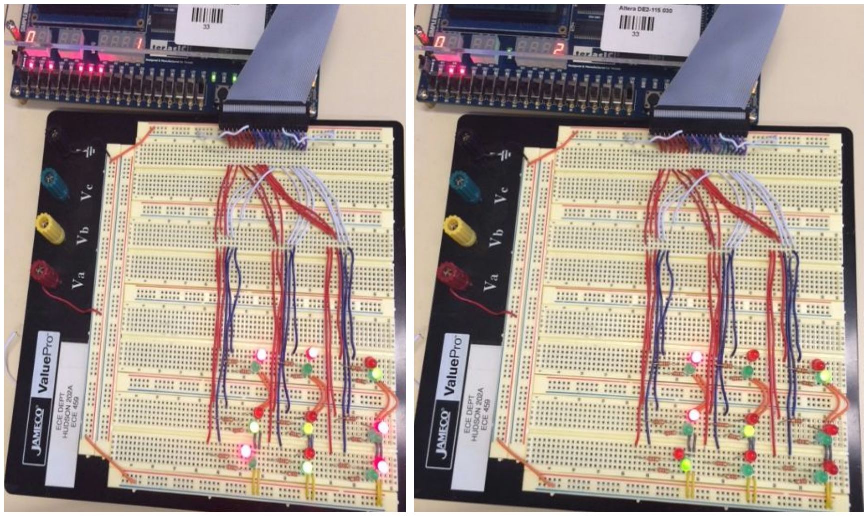


Figure 5.3: GAME 2: ends in a draw (score is unchanged) → GAME 3: player 2 wins (score changes) →

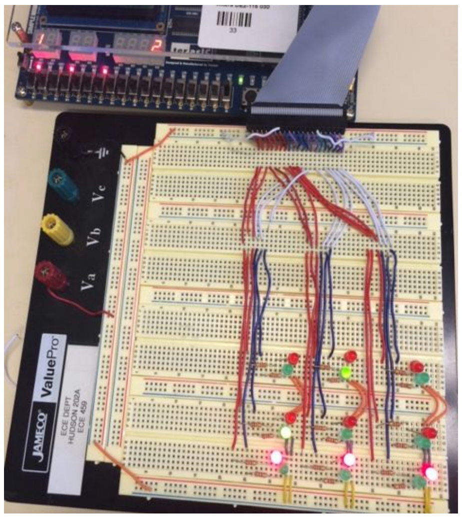


Figure 5.4: GAME 4: player 1 wins (score changes)

Testing

We used several methods to test our project:

1. To test our initial MIPS code, we used QTspim and tested it thoroughly with various combinations. Because the assembulator doesn't like comments and certain formatting, we then had to convert the working MIPS code in QTspim to a format that the assembulator would work with. In the conversion process, some of our code was changed by accident which was eventually fixed throughout the other testing processes.

2. To test that our processor was reading and executing instructions correctly we used the waveform extensively with up to 25 different outputs at once. We simulated entire games by manually inputting values for ps2_out, which would have normally been inputs from the keyboard. Once we had our code completely working on the waveform processor, we began testing using our actual outputs.

3. To test that our processor and code was working on the DE2 board, we hooked up 9 red LEDs on the board and 9 green LEDs. We tested on the DE2 itself before extending to the breadboard just to make sure every connection along the way was correct and that we could pinpoint the exact error.

4. Once we had everything working on the DE2 board LEDs, including every combination of possible keyboard inputs and victories, we reassigned the pins to the corresponding pins on our breadboard design and tested all of the LEDs on the breadboard using the same process.

Game Logic Code Description

Blocks of instruction (in order of execution):

- **Constants:** Register $r0 holds the value 0000, $r1 holds the value 0001 = 1, and $r2 holds the value 0010 = 2. After each player presses a key, a 1 or 2 is stored in the register representing the appropriate board slot. To light up the red or yellow LED in that slot, we compare each register value to 1 or 2. Since this is a frequent operation, we found it useful to reserve two registers $r1 and $r2 to hold the values of the constants.

- **Player initialization:** Register $r3 is designated as the "which turn" register, switching between 1 and 2 after each player presses a valid key and the LED lights up. $r3 = $r1 if it is player 1's turn, and $r3 = $r2 if it is player 2's turn.

- **Board initialization:** To implement the game simply and robustly, we reserve certain registers for certain values. Specifically, 9 registers, r6 to r14, are reserved as board slots as shown in the image below, and r15 is reserved for keyboard input. The slot registers are initialized with keyboard values, which are continuously compared to the input into register 15. As a result, a slot register value matching a keyboard input indicates that the key was pressed. The value in that slot register is then immediately changed to 1 or 2 - depending on whose turn it is - to indicate that the slot had been used, preventing the slot from being changed again, since the same board slot cannot be played twice.

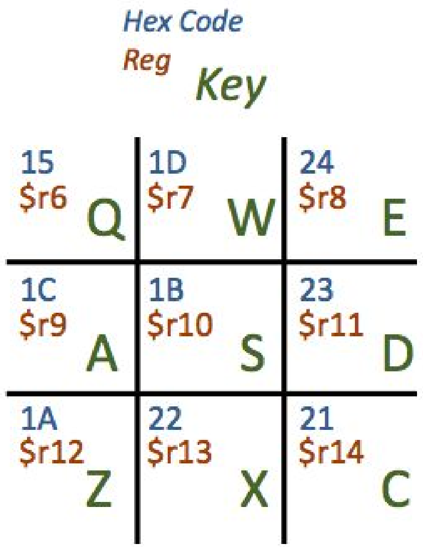
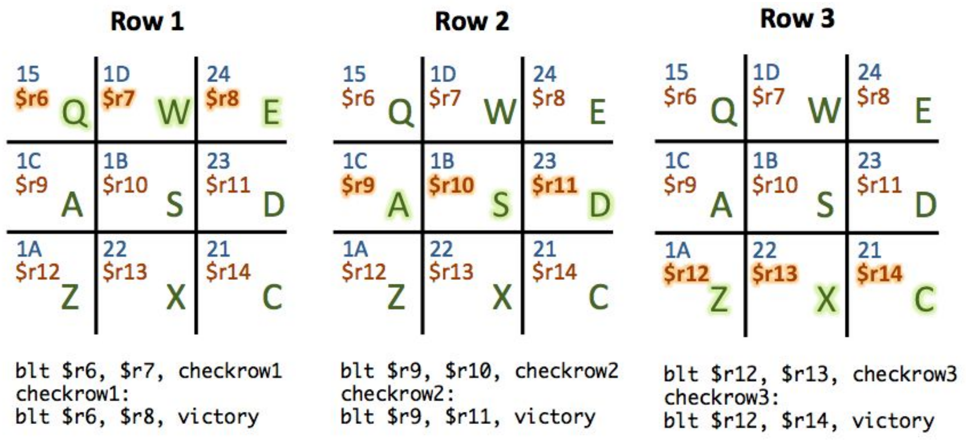


Figure 6: Registers designated for game board slots.

- **Logic for taking inputs:** The slot registers initially hold a keyboard value. When a keyboard key is pressed, the code detects if register 15's value is equal to one of the keyboard values in the slot registers. If so, the code then puts the player number who pressed the key, the player whose turn it is, into that register, so that the register will now hold a one or two indicating that the board slot has been played in and that the slot should now light up. Thus the game logic module checks all slot register values against one and two using 'equal to one' and 'equal to two' modules which were added with the sole purpose of comparing a number to one or to two respectively. If slot values match one, the LED 1 array slot corresponding to that board slot becomes high. If slot value match two, the LED 2 array slot corresponding to the board slot that got a match becomes high. These arrays are one of the final outputs of the game and processor and are used as DE2 board inputs which are connected to the final LED circuit representing the tic tac toe board and which light up the corresponding board slot LEDs when they are asserted/high in the correct colors based on the source array.

- **Logic for switching turns:** After a player has pressed a valid key and the appropriate LED lights up, the *changeturn* logic starts executing. If the value in register 3 is 1, then it is currently player 1's turn, so the logic checks this condition and then changes the value in register 3 to 2. Likewise, if the value in register 3 is 2, then player 2's turn has just completed, so our logic then changes the value in register 3 back to 1. At the end of each game, the MIPS code checks the winning player based on who played last, and initializes the turn register to hold the number of the losing player. For example, if player 1 won, then player 2 goes first in the next round. In the event of a draw, the player who started the game is also the first player in the next game, so register 3 remains unchanged.

- **Conditions for player wins or draw:** A player wins if the values of the registers corresponding to each row (Figure 7.1), column (Figure 7.2), or diagonals (Figure 7.3) are equal, as described in the boards below. The code then updates the score for the winning player, and switches the first turn to the losing player. A draw is registered if the game has completed nine turns, indicated that the victory condition has not yet been executed, and all the board slots are filled.

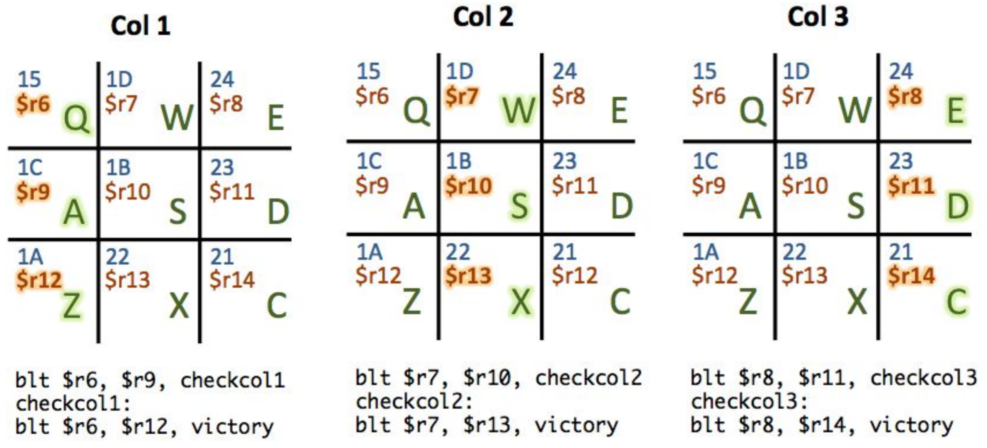


```
blt $r6, $r7, checkrow1
checkrow1:
blt $r6, $r8, victory
```

```
blt $r9, $r10, checkrow2
checkrow2:
blt $r9, $r11, victory
```

```
blt $r12, $r13, checkrow3
checkrow3:
blt $r12, $r14, victory
```

Figure 7.1: Victory Conditions: Checking Rows.

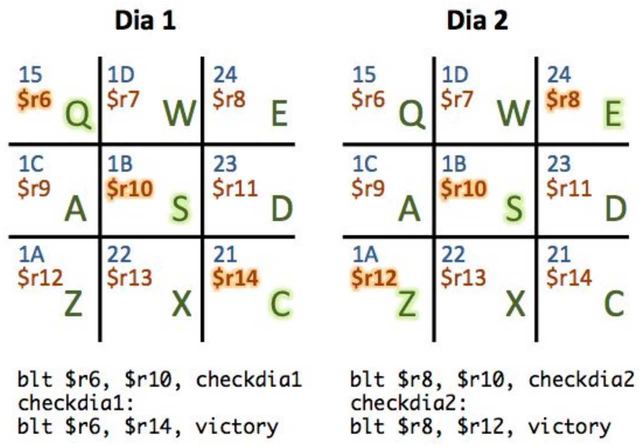


```
blt $r6, $r9, checkcol1
checkcol1:
blt $r6, $r12, victory
```

```
blt $r7, $r10, checkcol2
checkcol2:
blt $r7, $r13, victory
```

```
blt $r8, $r11, checkcol3
checkcol3:
blt $r8, $r14, victory
```

Figure 7.2: Victory Conditions: Checking Columns.



```
blt $r6, $r10, checkdia1
checkdia1:
blt $r6, $r14, victory
```

```
blt $r8, $r10, checkdia2
checkdia2:
blt $r8, $r12, victory
```

Figure 7.3: Victory Conditions: Checking Diagonals.

- **Logic to reset board for new game:** We went beyond our proposal to implement a “new game” button - key P. When P is pressed, the value of P is inputted into register 15, and the board resets to allow the players to play another round. The logic to clear all the board registers to their initial keyboard values takes place after the *reset* label.

Conclusions & Comments

Overall, our project was quite successful and we were able to implement all of the features we described in our original project proposal and some additional features as well. If we were to implement additional features besides the ones we already discussed, we could have added a turn indicator and a mid-game reset button. We were almost able to implement the turn indicator in our project demo but weren’t able to debug it quickly enough. Implementing both of these additional features would be relatively easy as it would just require a register value to be outputted or be controlled by a button on the keyboard - both things we already did throughout the project extensively.

By working on this project we learned how difficult it was to actually get the interface between a programming code such as MIPS and a processor to work correctly initially as well as a great deal about how the Altera DE2 functioned. Ultimately, we all would agree that it was a fitting end of the year project to put the knowledge and skills we learned throughout the year to test.

ASSEMBLY CODE

NOTE that blt in the first column below means branch equals. The logic for instruction blt was replaced for the final project in order to have a branch equals instruction which was needed for the code. In the assembulator all beq instructions were written as blt in order for the assembulator to provide the correct format and opcode. The logic and hardware for blt in the processor however was converted to branch equals logic. The second column has the beq as branch equals.

NOTE the some padding instructions (add $r0, $r0, $r0)were added in order to ensure the processor ran the logic correctly.

Uncommented Version Used in Assembulator	**Commented Version To Explain Logic**
	##
.text	# TIC-TAC-TOE GAME
main:	# ECE 350 Final Project
add $r0, $r0, $r0	# Pritak Patel, Anika Radiya-Dixit, Bruna Liborio
addi $r21, $r0, 77	# How it works: Each player takes turns pressing one of the keys on the
addi $r1, $r0,1	keyboard
addi $r2, $r0,2	# (q w e, a s d, z x c) corresponding to a spot on the TIC-TAC-TOE
addi $r17, $r0,9	# board. LEDs that are linked to the FPGA and our processor will
addi $r6, $r0, 21	# light up correspondingly with each players input (red for player 1,
addi $r7, $r0, 29	# and yellow for player 2). When either a player wins or the board
addi $r8, $r0, 36	is
addi $r9, $r0, 28	# fully filled (draw), the game restarts and a point is added to a
addi $r10, $r0, 27	player's
addi $r11, $r0, 35	# win total. The win totals are displayed on 2 seven segment
addi $r12, $r0, 26	displays.
addi $r13, $r0, 34	#
addi $r14, $r0, 33	##
addi $r16, $r0, 0	# INITIALIZE REGISTER VALUES
bne $r3, $r1, setturn1	# $r0 = 0
bne $r3, $r2, setturn2	# $r1 = 1 // constants representing player 1 and 2
reset:	# $r2 = 2
add $r0, $r0, $r0	# $r3 = whose turn it is, 1 or 2 (starts at 1 by assignment)
addi $r21, $r0, 77	# $r4 = number of wins for player 1 (P1) (starts at 0 by default)
blt $r21, $r1, reset	# $r5 = number of wins for player 2 (P2) (starts at 0 by default)

```
addi $r1, $r0,1
addi $r2, $r0,2
addi $r17, $r0,9
addi $r6, $r0, 21
addi $r7, $r0, 29
addi $r8, $r0, 36
addi $r9, $r0, 28
addi $r10, $r0, 27
addi $r11, $r0, 35
addi $r12, $r0, 26
addi $r13, $r0, 34
addi $r14, $r0, 33
addi $r16, $r0, 0
bne $r3, $r1, setturn1
bne $r3, $r2, setturn2
setturn2:
add $r0, $r0, $r0
addi $r3, $r2, 0
j player2turn
setturn1:
add $r0, $r0, $r0
addi $r3, $r1, 0
j player1turn
player1turn:
add $r0, $r0, $r0
blt $r15, $r6, qslot
blt $r15, $r7, wslot
blt $r15, $r8, eslot
blt $r15, $r9, aslot
blt $r15, $r10, sslot
blt $r15, $r11, dslot
blt $r15, $r12, zslot
blt $r15, $r13, xslot
blt $r15, $r14, cslot
j player1turn
player2turn:
add $r0, $r0, $r0
blt $r15, $r6, qslot
blt $r15, $r7, wslot
blt $r15, $r8, eslot
blt $r15, $r9, aslot
blt $r15, $r10, sslot
blt $r15, $r11, dslot
blt $r15, $r12, zslot
blt $r15, $r13, xslot
blt $r15, $r14, cslot
j player2turn
```

```
#       $r6-14 = board spots, start with characters' ASCII values
#       $r15 = stores the input character
#       $r16 = number of turns passed
#       $r17 = 9 (for comparison purposes)
#
#       Board:
#       $r6  (q)    $r7  (w)    $r8  (e)
#       $r9  (a)    $r10 (s)    $r11 (d)
#       $r12 (z)    $r13 (x)    $r14 (c)
.text
main:

   # initialize constants
   addi $r21, $r0, 77 #set reg 21 to “p” value
   addi $r1, $r0, 0x00000001 #set reg 1 to 1
   addi $r2, $r0, 0x00000002 #set reg 2 to 2
   addi $r17, $r0, 0x00000009 #set reg 17 to 9

   # initialize game board
   addi $r6, $r0, 0x00000071 #set reg 6 to "q"
   addi $r7, $r0, 0x00000077 #set reg 7 to "w"
   addi $r8, $r0, 0x00000065 #set reg 8 to "e"
   addi $r9, $r0, 0x00000061 #set reg 9 to "a"
   addi $r10, $r0, 0x00000073 #set reg 10 to "s"
   addi $r11, $r0, 0x00000064 #set reg 11 to "d"
   addi $r12, $r0, 0x0000007A #set reg 12 to "z"
   addi $r13, $r0, 0x00000078 #set reg 13 to "x"
   addi $r14, $r0, 0x00000063 #set reg 14 to "c"

   # initialize number of turns passed
   addi $r16, $r0, 0x00000000 #set reg 16 to 0

# initialize player turn based on previous turn

bne $r3, $r1, setturn1  #branch when player turn is not 1 to set the turn to
one
                          #this will happen when the turn
                          #is 2 or when initializing with
                            #turn equal to 0)
bne $r3, $r2, setturn2  #branch when player turn is
                            #not 2 to set the turn to two
                            #this will happen when the turn is 2
setturn2:
add $r3, $r2, $r0
j player2turn

setturn1:
```

```
qslot:
add $r0, $r0, $r0
addi $r6, $r3, 0
j checkback
wslot:
add $r0, $r0, $r0
addi $r7, $r3, 0
j checkback
eslot:
add $r0, $r0, $r0
addi $r8, $r3, 0
j checkback
aslot:
add $r0, $r0, $r0
addi $r9, $r3, 0
j checkback
sslot:
add $r0, $r0, $r0
addi $r10, $r3, 0
j checkback
dslot:
add $r0, $r0, $r0
addi $r11, $r3, 0
j checkback
zslot:
add $r0, $r0, $r0
addi $r12, $r3, 0
j checkback
xslot:
add $r0, $r0, $r0
addi $r13, $r3, 0
j checkback
cslot:
add $r0, $r0, $r0
addi $r14, $r3, 0
j checkback
checkback:
add $r0, $r0, $r0
checkvictory:
add $r0, $r0, $r0
blt $r6, $r7, checkrow1
row2check:
add $r0, $r0, $r0
blt $r9, $r10, checkrow2
row3check:
add $r0, $r0, $r0
blt $r12, $r13,
```

```
add $r3, $r1, $r0
j player1turn

##########################################
#   COMPLETE ENTIRE PLAYER 1 TURN
#   Checks input against slots, checks victory conditions, sets player turn to 2
#
#       * Not sure how to make this part more efficient. Currently it checks each
slot,
#       and even when one gets filled, it keeps checking the rest still. Also not
sure
#       how to implement it so that if someone presses an already taken slot it
wont
#       override it.
player1turn:

    # compare input to all slots
    beq $r15, $r6, qslot
    beq $r15, $r7, wslot
    beq $r15, $r8, eslot
    beq $r15, $r9, aslot
    beq $r15, $r10, sslot
    beq $r15, $r11, dslot
    beq $r15, $r12, zslot
    beq $r15, $r13, xslot
    beq $r15, $r14, cslot
    j player1turn #if input matched no slots, wait for other input from this player

##########################################
#   COMPLETE ENTIRE PLAYER 2 TURN
#   Checks input against slots, checks victory conditions, sets player turn to 2
#
#       * Not sure how to make this part more efficient. Currently it checks each
slot,
#       and even when one gets filled, it keeps checking the rest still.
#       Also not sure
#       how to implement it so that if someone presses an already taken slot it
wont
#       override it. // Anika: added logic to (q/w/e/...)slot to ingore invalid input

player2turn:

    # compare input to all slots
    beq $r15, $r6, qslot
    beq $r15, $r7, wslot
    beq $r15, $r8, eslot
    beq $r15, $r9, aslot
```

```
checkrow3
col1check:
add $r0, $r0, $r0
blt $r6, $r9, checkcol1
col2check:
add $r0, $r0, $r0
blt $r7, $r10, checkcol2
col3check:
add $r0, $r0, $r0
blt $r8, $r11, checkcol3
dia1check:
add $r0, $r0, $r0
blt $r6, $r10, checkdia1
dia2check:
add $r0, $r0, $r0
blt $r8, $r10, checkdia2
j changeturn
checkrow1:
add $r0, $r0, $r0
blt $r6, $r8, victory
j row2check
checkrow2:
add $r0, $r0, $r0
blt $r9, $r11, victory
j row3check
checkrow3:
add $r0, $r0, $r0
blt $r12, $r14, victory
j col1check
checkcol1:
add $r0, $r0, $r0
blt $r6, $r12, victory
j col2check
checkcol2:
add $r0, $r0, $r0
blt $r7, $r13, victory
j col3check
checkcol3:
add $r0, $r0, $r0
blt $r8, $r14, victory
j dia1check
checkdia1:
add $r0, $r0, $r0
blt $r6, $r14, victory
j dia2check
checkdia2:
add $r0, $r0, $r0
```

```
    beq $r15, $r10, sslot
    beq $r15, $r11, dslot
    beq $r15, $r12, zslot
    beq $r15, $r13, xslot
    beq $r15, $r14, cslot
    j player2turn #if input matched no slots, wait for other input from this player

###########################################
#   ADD VALUE TO CORRECT SLOT
#       Based on the input value by the player, places the player turn value into
#       the slot register to keep track of the board state.
qslot:
    addi $r6, $r3, 0 #add the player turn value to the slot
    j checkback
wslot:
    addi $r7, $r3, 0
    j checkback
eslot:
    addi $r8, $r3, 0
    j checkback
aslot:
    add $r9, $r3, 0
    j checkback
sslot:
    add $r10, $r3, 0
    j checkback
dslot:
    add $r11, $r3, 0
    j checkback
zslot:
    add $r12, $r3, 0
    j checkback
xslot:
    add $r13, $r3, 0
    j checkback
cslot:
    add $r14, $r3, 0
    j checkback

checkback:

###########################################
#   CHECK VICTORY CONDITIONS
#       Since you can only compare two values at once, it checks them and
then checks
#       the last value if needed to see if a victory condition is met.
checkvictory:
```

```
blt $r8, $r12, victory
changeturn:
add $r0, $r0, $r0
addi $r16, $r16, 1
blt $r16, $r17, winlogic
toturn2:
add $r0, $r0, $r0
bne $r3, $r1, toturn1
addi $r3, $r2, 0
j player2turn
toturn1:
add $r0, $r0, $r0
bne $r3, $r2, toturn2
addi $r3, $r1, 0
j player1turn
victory:
add $r0, $r0, $r0
blt $r3, $r1, player1wins
blt $r3, $r2, player2wins
player1wins:
add $r0, $r0, $r0
addi $r4, $r4, 1
j winlogic
player2wins:
add $r0, $r0, $r0
addi $r5, $r5, 1
j winlogic
winlogic:
add $r0, $r0, $r0
blt $r15, $r21, reset
j winlogic
add $r0, $r0, $r0
add $r0, $r0, $r0
add $r0, $r0, $r0
add $r0, $r0, $r0
.data
```

```
beq $r6, $r7, checkrow1
row2check:
beq $r9, $r10, checkrow2
row3check:
beq $r12, $r13, checkrow3
col1check:
beq $r6, $r9, checkcol1
col2check:
beq $r7, $r10, checkcol2
col3check:
beq $r8, $r11, checkcol3
dia1check:
beq $r6, $r10, checkdia1
dia2check:
beq $r8, $r10, checkdia2
j changeturn #no slots are matching, no possibility of winning, change turn

checkrow1:
beq $r6, $r8, victory #other slots match, have a victory
j row2check
checkrow2:
beq $r9, $r11, victory
j row3check
checkrow3:
beq $r12, $r14, victory
j col1check
checkcol1:
beq $r6, $r12, victory
j col2check
checkcol2:
beq $r7, $r13, victory
j col3check
checkcol3:
beq $r8, $r14, victory
j dia1check
checkdia1:
beq $r6, $r14, victory
j dia2check
checkdia2:
beq $r8, $r12, victory

changeturn:
add $r16, $r16, $r1
beq $r16, $r17, winlogic #reached 9 turns, board is full with no victory; wait
for play again signal

toturn2: #if turn 1, jump to player2 turn
```

```
bne $r3, $r1, toturn1 #if not player1 turn, change to turn 1
add $r3, $r0, $r2
j player2turn

toturn1: #if turn 2, jump to player1 turn
bne $r3, $r2, toturn2 #if not player2 turn, change to turn 2
add $r3, $r0, $r1
j player1turn

##########################################
#   WHEN A PLAYER IS VICTORIOUS
#       Checks which player has won, and appropriately gives them a point
and resets the game.
victory:
    beq $r3, $r1, player1wins #check who has won
    beq $r3, $r2, player2wins

player1wins:
    add $r4, $r0, $r1 #add a point for player 1
    j winlogic

player2wins:
    add $r5, $r0, $r1 #add a point for player 2
    j winlogic
##########################################
# WHEN A GAME IS OVER
#          Keeps looping until key ‘p’ is pressed
winlogic:
add $r0, $r0, $r0
blt $r15, $r21, reset
j winlogic
add $r0, $r0, $r0
add $r0, $r0, $r0
add $r0, $r0, $r0
add $r0, $r0, $r0

.data
```