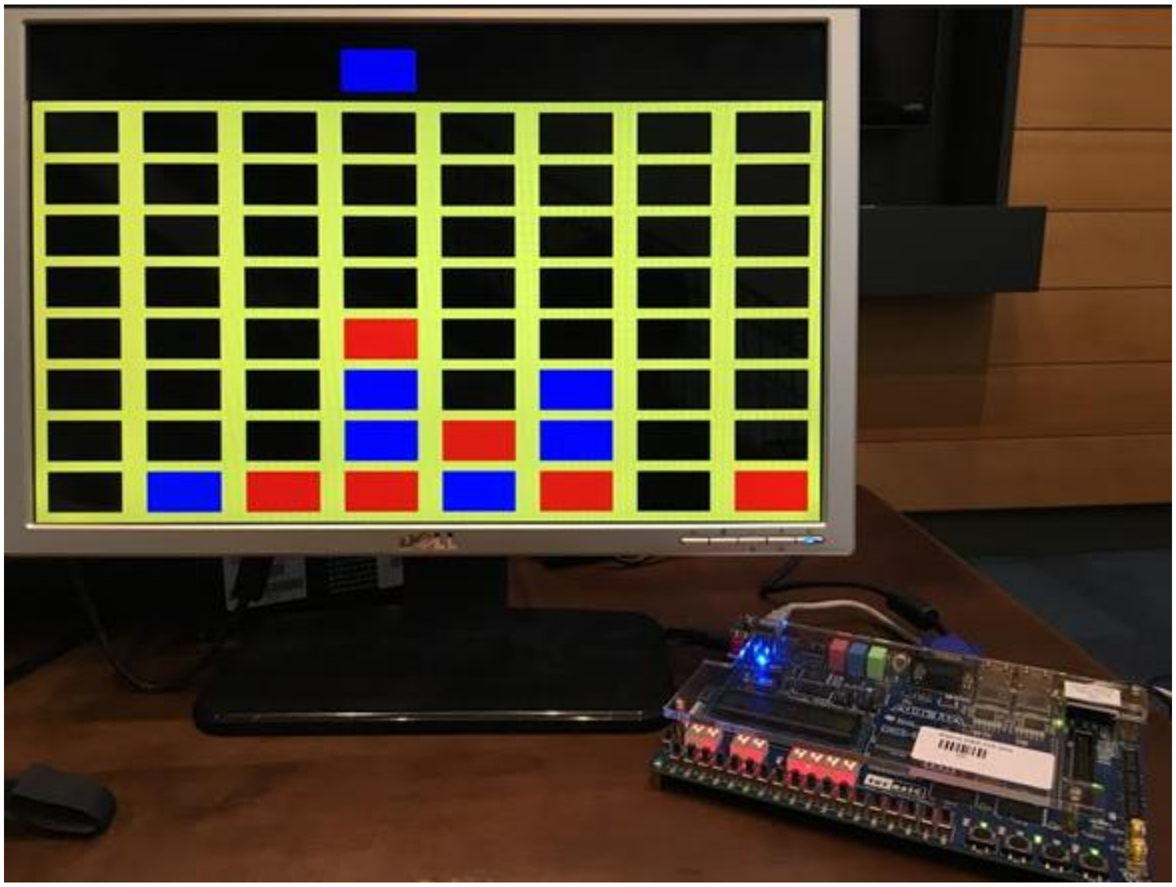
ECE 350 Final Project Report

Connect-Four using a Pipelined Processor



Austin Liu (abl17), Andrew Sun (as515), Kevin Liao (kl189)
Due: April 27, 2016

Note: DE2_115_Default.v is our top-level entity for our project

Introduction of Design:

For the final group project, we implemented an arcade-style game modeling the basic grid-based and turn-based genre. Our version of the project is a recreation of the common two-player board game “Connect-Four”, with slight modifications to the turn-based system and the board layout. As a generalized rule, we implemented the entire backbone of our project with structural Verilog, and the specific game logic through assembly code converted to a byte stream.

Our project involves usage of a pipelined processor that is based off the ECE350 Processor ISA. The processor acts as a structure to interpret core functionality, and much of the logic, data retrieval, and input detection, as well as arithmetic operations are handled by our processor.

Our project also implements specialized FSMs in a parallel Verilog module to the processor to determine game states such as *next-cycle* states and *next-drop* states, as well as to interface with our VGA output.

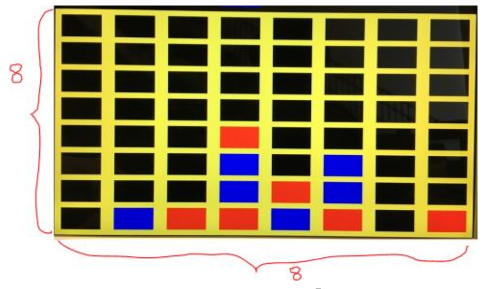


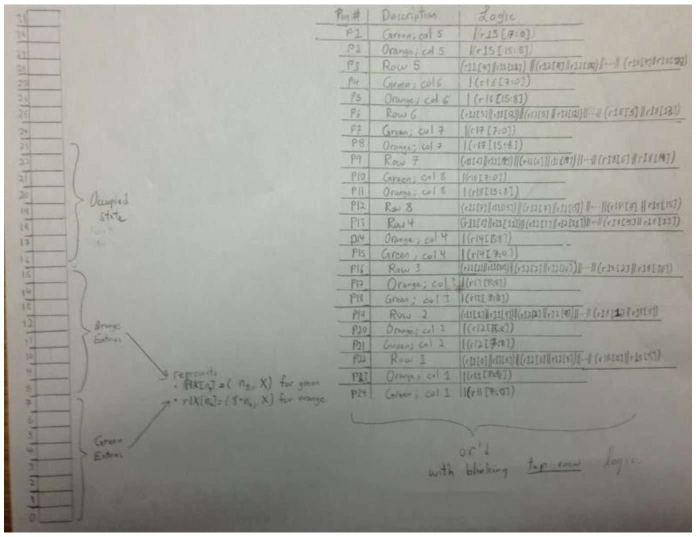
Figure 1: 8x8 Board Layout

Approach:

Our approach for this project was largely based on being able to represent the state of the board in a simple way. Storing board state in *dmem* contributed many challenges and issue mainly due to the fact that to represent 2-dimensional arrays in assembly code required more complex data structures. In order to simplify our design, we reserved key registers (*$r16 to $r23*) for board state and register *$r24* for metadata state. These registers are the core interface for communication between the processor and the *vga_controller*, and these are directly outputted from the processor as wires.

Due to the behavior of Connect-Four, our group realized that for any one state, there cannot exist two entries of the 64 in one input space. This meant that we could organize each register to represent either a row or a column. We decided to organize the register to represent an individual column because Connect-Four implements more column operations than row operations.

For occupation of columns, we separated the register such that the least significant 8 bits represent the occupation of blue pieces while the next 8 bits represent the red pieces. Bits 16 through 23 of a particular register represent whether a state is occupied or not, and is essentially metadata. A representation of the register format can be seen below:



Pin #	Description	Logic
P1	Green; col 5	|(r15[7:0])
P2	Orange; col 5	|(r15[15:8])
P3	Row 5	(r11[4] || r11[12]) || (r12[4] || r12[12]) || ··· || (r18[4] || r18[12])
P4	Green; col 6	|(r16[7:0])
P5	Orange; col 6	|(r16[15:8])
P6	Row 6	(r11[5] || r11[13]) || (r12[5] || r12[13]) || ··· || (r18[5] || r18[13])
P7	Green; col 7	|(r17[7:0])
P8	Orange; col 7	|(r17[15:8])
P9	Row 7	(r11[6] || r11[14]) || (r12[6] || r12[14]) || ··· || (r18[6] || r18[14])
P10	Green; col 8	|(r18[7:0])
P11	Orange; col 8	|(r18[15:8])
P12	Row 8	(r11[7] || r11[15]) || (r12[7] || r12[15]) || ··· || (r18[7] || r18[15])
P13	Row 4	(r11[3] || r11[11]) || (r12[3] || r12[11]) || ··· || (r18[3] || r18[11])
P14	Orange; col 4	|(r14[15:8])
P15	Green; col 4	|(r14[7:0])
P16	Row 3	(r11[2] || r11[10]) || (r12[2] || r12[10]) || ··· || (r18[2] || r18[10])
P17	Orange; col 3	|(r13[15:8])
P18	Green; col 3	|(r13[7:0])
P19	Row 2	(r11[1] || r11[9]) || (r12[1] || r12[9]) || ··· || (r18[1] || r18[9])
P20	Orange; col 2	|(r12[15:8])
P21	Green; col 2	|(r12[7:0])
P22	Row 1	(r11[0] || r11[8]) || (r12[0] || r12[8]) || ··· || (r18[0] || r18[8])
P23	Orange; col 1	|(r11[15:8])
P24	Green; col 1	|(r11[7:0])

or'd with blinking top row logic.

Occupied state
Orange Extras
Green Extras
represents
• r1X[n] = (n_x, X) for green
• r1X[n] = (8-n_x, X) for orange

Figure 2: Output register representation

From the outputted registers, basic game logic can determine winning conditions and next states through synchronous FSMs. This is described in a later section.

Input and Output:

In this project, we decided to go with VGA output. We were initially going to use an 8x8 LED matrix with two colors of LED to display the board state. We did initial tests trying to get the entire matrix displaying red and green. We spent approximately 6 hours trying to figure out how the matrix worked before we realized that it was probably going to be simpler just to use a VGA output since the LED matrix would require lots of multiplexing in order to get the correct lights to light up when they were supposed to be lit. This is due to the fact that the LEDs are wired by column. So it would be easy to get single LEDs to light up but it would be much much harder to get LEDs to light up in a display necessary to show a Connect Four board. After this realization, we quickly switched gears to trying to get a VGA display working. In order to do this, we used the software that came on the CD accompanying the DE2 board. DE2_Default included all the pins already mapped to their respective outputs. There were also structures in the vga_controller module that gave us insight into how VGA worked.

From this project and inspecting the modules, we learned that VGA relies on several things. There are front and back porches which are like “overshoots” for pixel output. These outputs are not visible since they don’t actually appear on screen but they are important to align the horizontal lines as well as the vertical position. Between the front and back porches, the RGB values are then relevant since they enable the monitor to paint the picture onto the screen. The default modules allowed for the displaying of a default picture that contained many colors and designs. In order to bend the vga_controller to our will, we added new inputs and also bypassed some of the memory modules to simplify the circuitry necessary for the purposes of creating Connect Four. The new inputs we passed into the vga_controller module were 9 registers, specifically registers 16-24. This was important since registers 16-23 stored the board columns and register 24 stored other relevant information such as which column the coin being prepared to drop was sitting above and also what color the prepared coin was.

The top 70 rows of pixels were assigned to be black unless it was covered by a blue or red coin and the yellow grid was designed to be static except for the open coin holes. The colors of the coin openings depended solely on the data in registers 16-23. Each column was represented by one register as follows: bits 31-24 of the register were zeroes; bits 23-16 showed a 1 if there was a coin of any color in each slot with bit 23 being the bottom-most slot and bit 16 being the top most slot; bits 15-8 represented true when a red coin existed in each slot; bits 7-0 showed a 1 if a blue coin existed in that slot. The VGA colors were then assigned using these register bits to find the colors.

We were able to replace the module that assigned each pixel to a color by using some simple math such as modulo for horizontal position on the screen and a general greater than or less than for the vertical position on the screen. Using several layers of logic, we were able to assign each of the coins states to display black if there was no coin in that position, red if there was a coin and the corresponding red coin bit was also true and a blue coin if there was a coin but it was not red. Since we only needed 4 colors for this project, black, blue, yellow and red, we were able to simplify the .mif file containing all the color codes to only contain 4 spots: 00, 01, 10 and 11 in binary. This made it very easy to determine what color went where since the left bit would be true if there was a coin present and the only two options for when a coin was present would be 10 which was blue or 11 which was red.

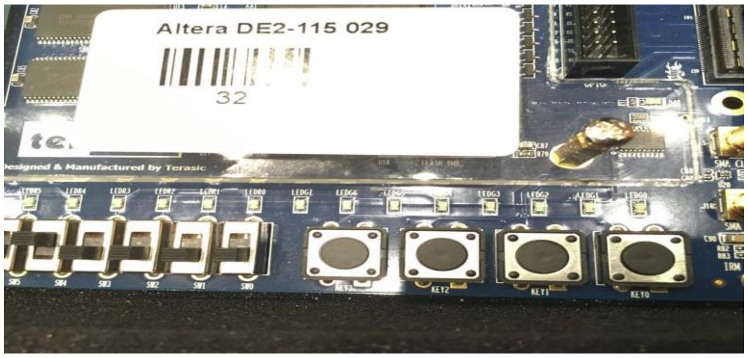


Figure XX: Keys used for physical inputs

The physical input to our game was through three buttons on the DE2 board: KEY0, KEY2, and KEY3. All three of these buttons were passed into our processor. KEY0 served as the “reset” key. Upon pressing this button, the entire board is cleared and the position of the piece is placed at the top left hand corner. This was easily implemented as we simply assigned the KEY0 pin assignment to our processor’s reset input. KEY2 and KEY3 were “cycle” and “drop” respectively.

Likewise, we assigned both the pins of these keys to the input of our processor. Upon pressing KEY2 (and letting go), the current piece at the top of the board will cycle to the right. Upon reaching the end of the grid, the next cycle will begin back at the beginning of the board. Upon pressing KEY3 (and letting go), the current piece at the top of the board will drop at its corresponding column, and fill the bottom-most unfilled slot of the column. Immediately thereafter, the current piece above the board will switch to the other color to signal that it is now the next player’s turn.

Specifications:

The specifications for our project are relatively straightforward. The rules are standard Connect Four rules. It is a simple 8x8 rectangular grid, in which the goal of the game is to place your pieces such that you form “four in a row”. Valid “four in a row” combination include four pieces horizontally, vertically, or diagonally in any direction. Upon winning, the game automatically resets the board. The players can reset the board at any time with a click of a button. The game is turn based, so upon dropping a piece, the game automatically switches the column select piece to the other color.

Modifications to Processor Design:

The basic changes to the processor were catered specifically for design practicality. The table below shows our modified ISA including the additional commands.

Instruction	**Opcode (ALUOp)**	**Type**	**Usage**	**Operation**
cyc	00000 (01000)	R	cyc $rd, $rs, $rt	Cycles through register $rs ($r24) and changes metadata based on FSM logic, storing into $rd
drop	00000 (01001)	R	drop $rd, $rs, $rt	Performs a drop operation on $rs and modifies the corresponding metadata structure for $rXX ($r16 to $r23) based on the current state of the board. This interfaces directly with the Drop FSM module.
cwc	00000 (01010)	R	cwc $rd, $rs, $rt	Performs a conditional check based on $rs, and stores the condition (win or loss) in the 9th bit of $rd.
chb	00000 (01011)	R	chb $rd, $rs, $rt	Performs a bit-flip operation on $rs and storing into $rd. This is mainly aimed towards the metadata structure $r24, flipping the 8th bit. Every drop instruction should be accompanied by a chb instruction.
rdin	00000 (01100)	R	rdin $rd, $rs, $rt	Performs a user-input check. Specifically checks to see whether the user has activated the external input (button in our case, but can be any generic input). Asserts bits 1 or 2 or both of $rd based on which button is activated at the time.

Table 1: Additions to ECE350 Processor ISA

We modified the I/O wires of the processor to accept two additional inputs (one each for the buttons), and 9 additional outputs: one register for each column and a register for metadata. These directly interface with the FPGA VGA display. The below image shows a detailed version of our pipelined processor.

The additional instructions added to the original ISA are all R type instructions. The main purpose of implementing R type instructions was to simplify the design and reduce debugging, as the additional commands exhibit many characteristics that normal R type instructions exhibit, and follow the same bypassing and similar control logic. Hence, most of the computation of the $rd for the additional instructions are resolved in the *execute* stage. The additions can be see in the figure below, and the individual block modules can be seen in the “Specific Circuital and FSM Diagrams” section.

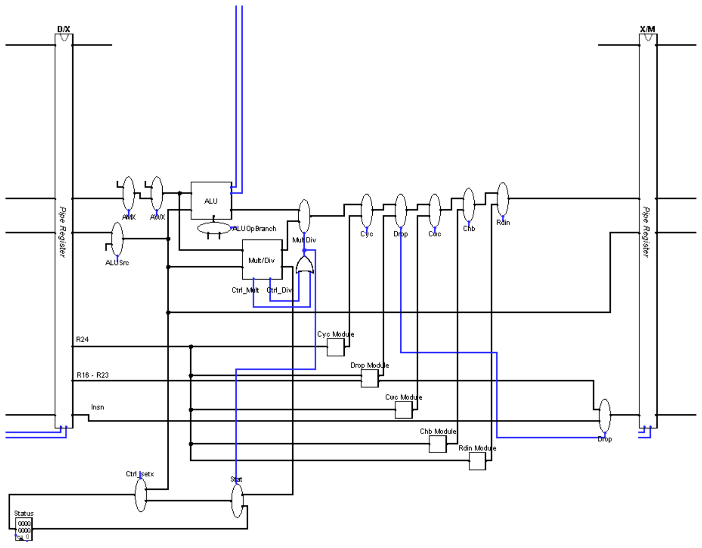


Figure 3: Modified Execute Stage

The specific usage of the additional instructions are explained further in the “Assembly Programs/Code Description” section.

Procedure to solve Challenges:

There were several challenges involving the input/output of this project. Firstly, as described in the input/output section above, we had major issues getting the LED matrix to work, as we did not understand how multiplexing worked. We solved this challenge by simply bypassing it and used the VGA instead, where we based our implementation off of the given VGA code from the given CD.

Another major challenge we had was that when we pressed a button, the board would perform many of the corresponding operations. For example, when we pressed KEY3 to drop a piece, the game would immediately fill up the entire column and switch colors many times. Likewise when pressing KEY2 to cycle. The problem was easily identified: since the clock of the processor and the DE2 board is around 50 MHz (which translates to 20ns per cycle), a key press would go through several hundred cycles, and our program would process each cycle as an individual key press. We solved this problem through the assembly code. Upon detecting a button press, the assembly code would branch to another loop that waited for all buttons to be unpressed before continuing to perform the operation. This ensured that the processor would only need to process one button press at a time.

Another issue that we had was implementing our custom instructions. Building another assembulator was a daunting task so instead we bypassed the assembulator by simply writing our custom instructions as something similar and editing the resulting imem.mif file. For example, our custom cycle instruction was an R type instruction, and we coded that as a “div” instruction since we didn’t use div in our code. Then in imem.mif, we edited “div” to our custom instruction’s opcode.

Testing:

Testing for the DE2 board was relatively straightforward. It mainly consisted of compiling, blasting, yelling F#@%!, making some edits and recompiling. In all honesty, it involved a lot of trial and error since our code was very rarely perfect the first time that we compiled. We ended up compiling and blasting an innumerable amount of times for innumerable amounts of time. Testing little by little paid off since we were able to detect small problems early and isolate them in short segments of code. Debugging wise we made, when things didn’t work, we used vector waveforms to help give us more information about what went wrong.

Specific Circuital and FSM Diagrams:

There were three very specific circuits that were added to our processor to help us with the display and logic elements of creating Connect Four. These circuits are the drop module, the cycle module and the wincon module.

The drop module enabled for the dropping of individual coins into a column. This was done by passing in all of the board state registers as well as the single metadata register. From the metadata, the module was able to determine the column above which the next coin would be dropped and then the color of the coin that would be dropped. It then selected the column that would have a new coin added by the use of a 8-to-1 multiplexer that singled out the column having a coin added. After this was determined, each occupied bit was determined through combinational logic depending on the state of the column passed in to have a coin dropped. For example, if the column was originally empty, the 7th occupied bit representing whether there is a coin in the lowest possible slot would be changed from 0 in the original state to a 1 in the next state. When any occupied bit is changed, the corresponding red and blue representing bits would be changed as well if the corresponding occupied bit is changed, the bit directly under the current bit is already occupied and the color is the correct color for the bit it represents. The drop module outputs a 32 bit output that are to be written to the register that was originally passed in to be modified. The figure below shows a circuit diagram of the module.

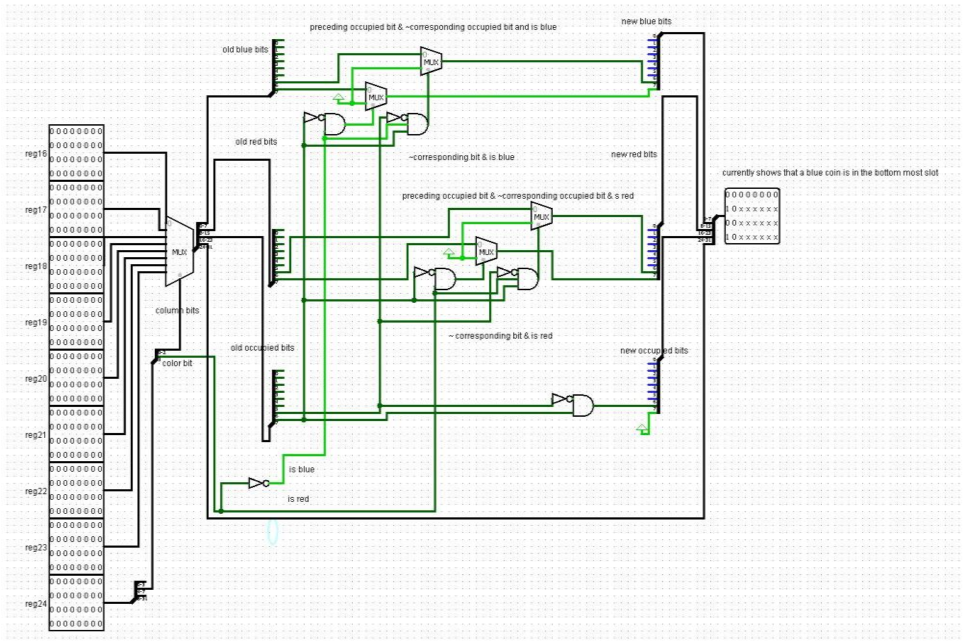

preceding occupied bit & ~corresponding occupied bit and is blue
new blue bits
old blue bits
MUX
old red bits
~corresponding bit & is blue
new red bits
currently shows that a blue coin is in the bottom most slot
preceding occupied bit & ~corresponding occupied bit & s red
reg16
reg17
reg18
reg19
reg20
reg21
reg22
reg23
reg24
column bits
color bit
old occupied bits
~ corresponding bit & is red
new occupied bits
is blue
is red

Figure 4: Drop module sample diagram. Note: some bits were left unconnected to not clutter the diagram. All unconnected bits mimic the logical behavior of the bit immediately inferior to it.

The cycle module enabled the dropping coin to change between different potential columns by scrolling through the columns from left to right, 000 to 111. Cycling only requires 3 bits in the register 24 metadata and thus all the operations from cycling are the result of logical operations on these 3 bits in one register. The module modifies these three bits by adding one until the input becomes 111 which causes an output of 000 and thus the entire loop begins again. This is done with combinational logic for each of the individual bits of the output as functions of the input. The following figure shows a diagram of the FSM states.

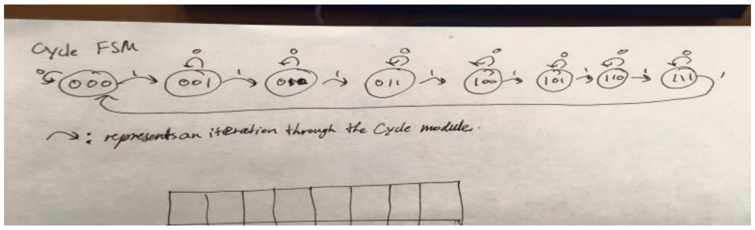


Figure 5: Cycle module FSM diagram.

The wincon module enabled for the detection of a winner whether that winner was red or blue. This was determined by assessing all the possible combinations and locations that were possible four-in-a-rows. The four main ways of winning include vertically, horizontally, diagonally up and diagonally down. For the purposes of this report, they will be referred to as up, right, right up and right down. There are actually many different ways to describe the possible ways of winning since you can interpret the four points in two different directions even though they have the same positions. The wincon module basically wires four of the bits in the register that form a straight line on the board into an AND gate and then repeats this for every possible combination of win conditions. All of these AND gates are then hooked into an OR gate such that if any four are in a row, the module signals that there was a win. The following figure shows how no win direction needed to be tested 64 times since it is not possible to win with 3 or less coins in a direction. The diagonals were tested a total of 25 times each where the horizontal and verticals were tested a total of 40 times each.

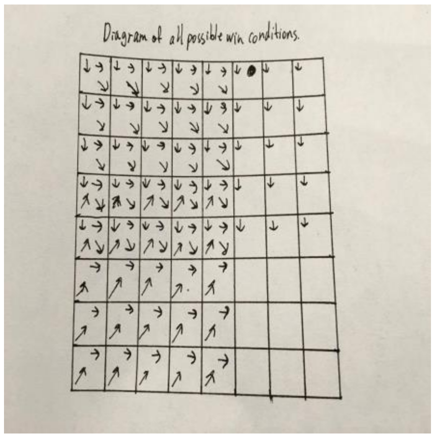


Figure 6: Diagram of all possible win conditions and directions tested in Wincon module

Assembly Programs/Code Description:

The assembly code for the program consists of three main sections: the main game loop, the button checks, and procedure calls for when a button check is passed.

The game loop section is quite simple: we read the input from user using the rdin custom command, then do the corresponding action based on what button was pressed. This was implemented using a while-if-then structure implemented using bit masks, blt, and jump, as seen below:

```
game_loop:
       rdin $r1, $r1, $r30                  # read input from user
       and $r2, $r1, $r14                   # r2 = r1 AND 3 (create bit mask)
       blt $r2, $r5, game_loop              # if $r2 < 1, jump back to game_loop
       blt $r2, $r3, cycle_button_check     # if $r2 < 2, cycle
       blt $r2, $r4, drop_button_check      # else if $r2 < 3, drop
       j game_loop                          # else restart game loop
```

The button checks section were necessary because of the problem described in the challenges section of this report: when we pressed a button, the board would perform many of the corresponding operations. The button checks would continuously check whether or not the button is still being pressed, and only allows the procedure to continue when the button is let go. An example, the drop button check, is seen below:

```
cycle_button_check:
       rdin $r9, $r9, $r30   # read input from user
       and $r10, $r9, $r14   # create bit mask; $r14 = 3
       blt $r10, $r5, cycle  # branch to cycle if $r10 < $r5
       j cycle_button_check  # redo cycle
```

The final section consisted of actually calling the custom instructions involved with the function. The simple example is the cycle function where we simply just call the cyc custom instruction then return to the game loop:

```
cycle:
       or $r24, $r30, $r30   # call cycle function
       j game_loop           # jump back to game_loop
```

Conclusion:

In conclusion, the integration of Connect Four was a great success. This project involved using many concepts from both class and lab all integrated together. Some of these concepts include creating a pipelined processor programmed with verilog, creating assembly code to integrate our processor into an arcade game, and implementation of specialized FSMs . There were many hardships along the way, but we managed to solve our challenges by fully understanding the concepts the greatest extent.