## 3T Innovation in Tic-Tac-Toe

ECE 474 - Fall 2001

Victor Aprea
Paul Grzymkowski
Andre Kozaczka

## INTRODUCTION

Unlike many other project-oriented courses at Cornell University, the final project for ECE 474 - Fall 2001 gave us total liberty in choosing what we wanted to do within the realm of digital VLSI design. As a group we decided that we wanted to design something interesting, interactive, familiar and yet innovative. The project that surfaced from these modest criteria was not a processor, at least not in the conventional sense. The project was a hardware Tic-Tac-Toe game, fully capable of interfacing to an LED based game grid and array of push buttons. The game grid is shown in Figure 1.

The game is designed to be played player vs. computer opponent. The player uses one of nine pushbuttons to register a move on the board while observing the LEDs to determine where the computer has moved. One set of LEDs represents the player's pieces, while the other represents the computer's. The computer plays games non-deterministically and can be set to three levels of


Figure 1: Game grid with LEDs difficulty: easy, medium, and hard.

At first glance, the idea admittedly seems somewhat simplistic. After all, five year olds are capable of understanding the rules of Tic-Tac-Toe. But, in hardware, the game turned out to be an interesting, non-trivial implementation.

## DIGITAL HARDWARE DESCRIPTION

The hardware we designed was extremely unique and special purpose. The types of hardware we implemented were counters, registers, latches, multiplexors, some specialized combinational logic, and two PLAs. So in some sense it actually is a processor, albeit a highly specific, one-hot, 9-bit processor. We designed our circuits in CAST and simulated them in IRSIM.

## Datapath Design

Our datapath was designed with data flowing primarily in one direction, as show in Figure 2. In general, if the player is moving, desired positions on the board are latched into one end of datapath as inputs and stored into a register (POSO). If the computer is moving, desired positions are generated from a randomized or incrementable one-hot counter (PRONG or INC, respectively), or a corner counter (CCO) and stored in another register (POS1). This position information is used as an enable into the register file, in which pieces are written. The status of the board is checked for a win by a combinational unit (WC), and the board is latched as output once it is the next players' turn to move. Additionally, there is a unit that detects if a player occupies opposite corners of the board $(\mathrm{CCH}$, used by the computer for hard mode). If there is a win or all the board spaces are
full then a game over signal is output. The individual units of our datapath are described in the following sections.


Figure 2: Tic-tac-toe datapath. Thin units to the right of labeled functional blocks represent latches on the outputs of the blocks

## Register File

To save the status of the board we used nine special purpose two-bit registers, numbered zero through eight from the upper left, across rows. Each bit in a register represents the state of a player in that grid square. The internal state of the register is inverted. If a player owns a cell, that player's bit is set low. The allowable states for a register are then as follows:

- 01 implies that player one occupies the cell
- 10 implies that player two occupies the cell
- 11 implies a cell is not occupied

The state " 00 " is disallowed by the fact that two players may not occupy the same cell.
The register file takes a total of 13 inputs: a one bit input data wire, four global data input-output control wires, and nine enable wires. The register file produces 28 outputs: 18 LED outputs, nine data outputs, and one occupied bit. The global control inputs are read bit-zero (R0), write bit-zero (W0), read bit-one (R1), and write bit-one (W1). Whenever the read bit-zero signal is asserted, each of the nine registers passes its respective bit-zero to its data output. This output is in non-inverted form, so output high implies player one occupies a cell when read bit-zero is asserted.

It should be noted that the read bit-zero and read bit-one signals must be exclusive high. The reason for this constraint is that there is only one shared output bus, and applying the constraint allows for simplicity of data control. One can consider the constraint as multiplexing the output internally in the register file. The convention also makes the game control flow simple to implement as will be discussed in subsequent sections. The input data is distributed to each of the nine registers but it is only written to the register whose enable line is set (based on the position registers) and to the bit whose write line is set (depending on whose turn it is). The specific reasons for this convention will become clear in subsequent sections.

The registers contain built in combinational logic based on the internal state. Each register produces a bit which determines whether a cell is occupied by either player. In order for the occupied bit to be true a register must have its enable line set. The occupied bit is relatively easy to produce as it is simply the logical NAND of the two (inverted) bits in the register, as evidenced by the internal state encodings above. A global occupied signal is generated by the OR of the nine occupied bits for use by the computer opponent.

The 18 LEDs are driven directly by the internal state of each bit of the register file. Obviously because of the internal state encodings, the LEDs are intended to be active low and should be connected through pull-up resistors to $\mathrm{V}_{\mathrm{cc}}$ in the off-chip interface. Figure 3 below illustrates the schematic of a single register.


Figure 3: Tic-tac-toe register high level design schematic. Subscripted signals are bit-specific, while non-subscripted are global to the register file.

## One-Hot Counter (PRONG and INC)

This unique counter is a state machine that generates a sequence of nine-bit one-hot outputs used to generate ENABLE signals for the register file. In essence, this component generates all the powers of two up to 512 in sequential order. On reset (MRST), the counter is initialized to $0 b 000000001$, and it then proceeds to increment through its designated sequence once per positive clock edge. Upon reaching the value $0 b 100000000$, the counter returns to its initial state and continues indefinitely (as long as the power is on). The device was implemented as a chain of D-flip-flops as see in Figure 4. Each flipflop's output is connected to the next flip-flop's input in the chain. The last flip-flop's output is connected to the first flip-flop's input. The master reset logic is not


Figure 4: One-hot counter schematic shown for simplicity. It entails a single combinational
gate with MRST and the D-input signals of each bit; bit-zero being an OR gate and the rest being AND gates.

The one-hot counter is intended to function in two ways. First, as a stand-alone unit it operates independently of any control logic aside from the clock. Second, the clock signal can be replaced by a control signal asserted

Under the clocked mode of operation the unit operates independently of the control logic, so it provides a pseudo-random index into the grid when the computer opponent begins considering its options. In order to maintain the "randomness" of this unit's output we instantiated two one-hot counters in our design: one clocked with the global clock (reset when the user hits START) and one precisely controlled. Incidentally, we gave our clocked unit the acronym PRONG, which stands for Pseudo-Random Obscure Number Generator, because it is used when the computer is going to make an entirely random move.

Under the second mode of operation a control signal pulse from the AI control causes the output of the counter to increment. This mode is used by the computer to systematically step through the grid to search for possible win or block positions. The reason for this is described in greater detail in later sections. In each mode of operation the unit is intended to generate a control signal, namely the enable line, when the computer opponent is playing. The control logic uses the output of the one-hot counter to access the register file for writing.

## Corner Counter (CCO)

The corner counter is essentially a subset of the one-hot counter. As if the one-hot counter was not specialized enough, this unit produces a one hot code sequence that selects corner cells of the grid. Recall that the grid numbering convention is zero through eight, starting at the upper-left corner and incrementing across rows, as illustrated in Figure 5. The corner counter state machine produces the repeating sequence: $0 b 100000000$, $0 b 001000000$, $0 b 000000100$, 0b0000000001. Additionally, given a special control signal (middle) the unit is able to override this output and assert 0b000010000, which allows the computer to move in the

| 0 | 1 | 2 |
| :--- | :--- | :--- |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

Figure 5: Grid Numbering Convention middle space. These alterations aside, the specifications of the unit are identical to the clocked one-hot counter in all other regards. Like INC, this counter is precisely stepped by the AI control. The general purpose of this unit is to allow the computer opponent to select a corner cell, usually for the first move. Reasons for this should become clear in later sections.

## Latching Scheme

As we have seen there are four ways of accessing the register file:

- Player one can pick a cell with pushbuttons
- The computer can pick a cell with PRONG
- The computer can pick a cell with INC
- The computer can pick a corner (or middle) with CCO

It will be shown that these three types of computer selection are sufficient for the computer opponent to be an impressive adversary. Each option constitutes a different enable line combination to access to the register file. Furthermore, each possible one-hot nine-bit output is generated by an independent hardware unit that would all like to write to the same enable lines of the register file. In order to select among these possible options using a single enable bus line for each bit, a distributed multiplexer strategy is necessary.


Figure 6: Read latch schematic


Figure 7: Enable bus control schematic

We use a number of 9-bit read latches to block the output of each ENABLE generating unit. The read latches then drive the nine-bit enable bus when enabled. These are not clocked latches, they merely sample and hold input when enabled. At any given time only one read latch is allowed to be enabled by the game control PLA, and thus we eliminate the condition of more than one unit trying to write to the enable bus at a given time. Figure 6 illustrates the transistor level implementation of a read latch bit, while Figure 7 illustrates how the read latches control bus usage.

A similar 18-bit read latch is used to latch out the current status of the register file to the LEDs.

The ENABLE BUS is then fed to position registers POS0, if it is the player's turn, or POS1, if it is the computer's turn. The registers are used to hold onto the current desired
position in the event that the player presses a different button, or the computer's counters are still incrementing while the enable signals are being used by the register file.

## Win Checker (WC)

The win checker is a basic circuit block that detects if there is a win anywhere on the game board, and it is comprised of two main parts. The first part of the win checker detects if there is a three-in-a-row combination for any of the eight possible combinations. The second part of the win checker takes these eight outputs and OR's the result. The inputs to the win checker are read out from the register file. The control signals $r 0$ and $r l$ for the register file choose the player's moves or the AI's moves to be read.

## STATE MACHINE DESIGN

The main flow of our Tic-Tac-Toe design is controlled by a gamePLA, while the moves of the computer opponent, the AI, are controlled by another PLA. Both PLAs were written using PEG and generated as layout for simulation with our datapath in IRSIM.

## Game Control

A game control FSM exists to direct the flow of the game and check to see if the human player has made a valid move (refer to Appendix A for state diagram of game control). The game control state machine (GCSM) first debounces the start button and then resets the registers and the game board. Depending on the state of the clock when the push button is pressed, either the AI or human player goes first. When it's the player's move, the GCSM waits for a ready bit to be set high. The ready bit is the ORed inputs of the player and indicates that the human player had made a move. The GCSM writes the player's move into the pos0 register and then checks to see if the move is valid (i.e. not in the same spot as any previous moves). If valid, the player's move is written by asserting the $r_{-} p o s 0$ and $w 0$ signals. The signal $r_{-} p o s 0$ controls the read of the pos0 register while $w 0$ is the write control for the human player's move in the register file. Next, the GCSM refreshes the LEDs by latching out the outputs of the register file. If there is a win, the game ends and the player wins. However, if there is neither a win nor a tie, the GCSM enables the AI and waits for its move to be ready (like a handshake protocol). Once the ready bit from the AI is high, the GCSM writes the move by setting $r_{-}$posl and wl high.

The tie bit is set high when the "tie counter" reaches nine (i.e. nine moves have been made, all spots on the board should be full). When this occurs, as when there is a win, the game is over.

## AI Strategy

Anyone who has played Tic-Tac-Toe knows a thing or two about how to win at this simple game. The main goal of any Tic-Tac-Toe strategy is to setup a double win situation in which case the opponent can only block one win. Being very knowledgeable in the art of Tic-Tac-Toe strategy, we played some test games against ourselves and began writing down some of the "unwritten" tactics. Interestingly enough, a flow diagram began to emerge and we came up with two nearly flawless strategies: one for when a player has the first move, and a second for when a player has the second move. The strategies not only try to force double wins, but also pick moves that increase the chances of winning when there is no double win possible. These strategies became the basis of our "hard mode" setting.

However, the main challenge was to implement this strategy using a finite state machine. An obvious solution would simply be making a FSM that knew where every ' X ' and ' O ' was on the game board, and then could proceed making the best move possible. This solution requires many inputs and many states, making the FSM bloated and unpractical. Our solution, instead, does not need to know the exact location of each move, but rather knows general information about each move (i.e. corner, side, or middle move). The FSM we implemented knows if a spot is occupied, the status of each side spot, if there is a win, and if a corner move is opposite of another previous move.

To make a move, the AI has to its disposal three one-hot counters: one counter stepping through each spot on the board (INC), one counter stepping through the corner spots (CCO), and one randomly choosing spots (PRONG). These three counters are instrumental in keeping the FSM relatively small.

The AI finite state machine can be broken down into three main stages, which follow from basic tic-tac-toe game play: check for a possible win, check for a possible block, and then make an appropriate move. For easy mode, the AI simply makes a random move using PRONG and does not first check for a win or a block. For medium and hard, however, the first priority is always to check for a possible win-move. To find one, the AI steps through each board location using INC and if the position is available, it moves its piece there. If a win is not detected, the AI will simply delete that move and go onto the next possible move. If a win is detected, the AI leaves its piece in that spot and wins the game. If no win-move is detected after moving through the entire board, the AI moves onto the second priority, preventing the opponent from winning. To block a possible win by the opponent, the AI does the exact same thing as when it is searching for a win move but this time the AI moves the player's piece into each unoccupied spot. If a win is detected (meaning the opponent would win if it went there during its next move), the AI moves into that spot thereby blocking the possible win. However, if neither a winmove nor a block-move is possible, the AI must decide where to move its piece. For medium mode this just means randomly picking a spot with PRONG, but for hard mode there is another weapon -- the smart move.

## Computer Moves First - Hard Mode

As mentioned above, we developed two general strategies for the smart move based on whether a player moved first or second. Please refer to Appendix B for a flow chart of this strategy. If the AI moves first, we decided the first move would be a random corner. Initially, we thought about having the AI always move to the middle, but that would be boring. Instead, the AI moves to a random corner, (based on the value in the corner counter) thereby adding some variability to the game. The second move of this strategy is based on where the player moves (corner, side, or middle spot). If the player moves into one of the sides, we get a situation like in Figure 8. The best move would to go into the middle, setting up a double win situation. The player needs to go into the bottom left corner in order to block the AI's possible win. According to the strategy, the next move is in a corner not


Figure 8: Corner-SideMiddle Combination next to the player (i.e. a corner that has sides that are not occupied by the player). From Figure 9, we see that the only corner that meets this criterion (and is not occupied) is the bottom right corner. This move is the best possible move because it gives the AI a double win (as indicated by the squares in Figure 9). In fact, using this strategy after the player moves into any side results in a win every time. The FSM is able to pick the correct corner by cycling through each corner and checking the side spots to see if the player occupies them.

So what if the player's second move is the middle square? The AI will take the corner opposite of its first move. The FSM is able to find the opposite corner by stepping through the corners with CCO, putting its piece in the available space, and observing if this causes CCH to go high, indicating if the current corner is opposite a previous move. If the player's next move is one of the two free corners, a double win situation occurs when the AI takes the other free corner. If the player's next move is a side, there will be a tie (see Appendix D for an illustration of these types of


Figure 9: Double Win Situation scenarios).

There are two other possible player moves to consider. The player can move in either the opposite corner the AI went in or a different corner. If the player goes in the corner opposite the AI, the AI's next move will be in the middle square, issued by a special signal to CCO. There will be a double win situation for the AI if the player's next move is not in a corner. If the player's second move is a corner not opposite the AI, the game is basically over because the AI will take the opposite corner of its previous move and set up a double win.

## Player Moves First - Hard Mode

When the player moves first, we took a more defensive angle when determining what type of strategy to implement. Please refer to Appendix C for a flow chart of this strategy. There are three possible first moves to defend against: corner, side, and middle. If the player moves in the corner, the AI chooses to move in the opposite corner of the player using CCO and CCH as before, but this time writing the player's piece to the board. If the player moves in the side square, the AI moves in the middle and sets a flag high. The flag will indicate what corner the AI should take for its second move. If the player moves into the middle, the AI moves in a random corner. The AI's second move is any free corner, except if the player's first move was a side (i.e. flag set high). When the flag is set high, the AI will only take a corner that is next to the player, preventing a possible double win for the player (see Appendix $\mathbf{E}$ for a sample game). The third move, if possible, is to move into the middle square. Everything after that is either a win, or a block, or just filling in remaining spots in the board randomly.

We found this second-move strategy to work fairly well and we initially thought the strategy was flawless. However, we did find a fault in our strategy and we were able to beat the AI. Can you figure out how to beat the AI? See Appendix F for the answer!

## LAYOUT \& SIMULATIONS

We used Magic to layout our design, and LVS and IRSIM to confirm that it worked correctly according to our specifications.

## Layout

For the most part, our layout followed directly from our datapath design, as shown in Appendix G. For the global game control and the AI state machine we used PLA generation tools, but for the rest of the blocks we created the layout from scratch. Since many of our blocks in our datapath were the same on the bit level we were often able to simply array a unit cell nine times and connect additional wires as necessary. Some blocks, however (like the win checker) did not lend themselves well to such a technique and required added attention.

In general we conformed to the standard of having the metall layer used for local connections and run vertically for shared signals, and using metal2 for horizontal connections. Metal3 was then used sparingly; mostly for large horizontal busses that many blocks would share (like the ENABLE BUS). The bit-pitch we chose (and made every effort to conform to) was $100 \lambda$ which was spacious for some components and quite restrictive for others. In general, transistors were sized minimally, except where LVS indicated that outputs required staticizers, and then we painstakingly increased the size of the required transistors, often disrupting the rest of our layout in the process (as is typically the case in VLSI, we imagine). In the end our datapath ended up consuming
roughly $1,127 \times 2,003=2,257,381 \lambda^{2}$. The AI PLA, even when minimized rivaled our datapath at $874 \times 1,757=1,535,618 \lambda^{2}$, which was considerably larger than our gameplay control PLA at $650 \times 612=397,800 \lambda^{2}$.

Given we were working within a 40-pin pad frame it became fairly straightforward what signals needed to be carried on or off-chip. Our input pins consisted of 21 pins: 2 GND, $2 \mathrm{~V}_{\mathrm{DD}}$, MRST, START, 2 PHI0, 2 PHI1, DIFF0, DIFF1, and the 9 inputs (IN) from the push buttons. Our outputs were the remaining 19 pins: GAMEOVER and the 18 outputs (OUT) to the LEDs. The padframe pinout can be seen along with the package pinout and some test results in Appendix J. PHIO and PHI1 were locally inverted using larger inverters (around $130 \lambda$ wide). We chose to put the inputs on the left and top of our chip, keeping all the IN signals together. This left the right side and bottom of our chip for output which minimized the amount of "crazy wires" we had spanning across our entire chip. All in all our design fit comfortably in the pad frame (as seen in Appendix H), taking up about $3,292 \times 3,249=10,695,708 \lambda^{2}$.

## Simulation Results

Simulations of our CAST files and layout in IRSIM indicate that our design works perfectly from within the pad frame. The inputs are received correctly and the outputs are clearly present and correct. When a game has ended the GAMEOVER value is properly asserted high. The three difficulty levels behave exactly as expected, and the computer's moves seem somewhat random as desired. Easy mode is clearly easy and the computer often loses. You have to be tricky to beat the computer on medium, but it is very possible and can happen often once you figure out how the AI works. Beating the computer on hard is nearly impossible, and from our extensive testing, we found the AI to never lose when it went first. The AI also took advantage of any mistake we made and won the game. But when the player goes first in hard mode we have found an exceptional case in which the computer can be fooled and can lose. This is probably because the computer's goal in hard mode isn't merely to tie (that would be too boring), but to actively try to defeat the user. It has a small bag of tricks it tries to pull from, which apparently leaves it slightly vulnerable in one obscure case, which can be exploited. In a way, this makes our game more interesting, because if you can never win on hard mode, why bother playing? But if it is possible to win, just very difficult, then it makes the game more challenging and fun.

Timing is not a particularly relevant issue because our design operates on a human scale (on the order of 30 ms or more). We merely need to clock it in such a way that everything still works. With this objective, we found that our design was able to simulate correctly with a step size of 20 ns with a PHI1/PHI0 clocking scheme of $0 / 0,0 / 1,0 / 0,1 / 0$.

We initially found that simulating our design from outside the pads did not work, but we later discovered that if we manipulated the input clocks (inserting additional $0 / 0$ states into the sequence) we could get it to work from outside the pads as well.

## TESTING OUR DESIGN

Our design was fabricated by MOSIS in a $.5 \mu \mathrm{~m}$ CMOS, 5 V process. We received 5 chips back in a standard 40-pin DIP package. A discussion of the testing procedure we used on our chips and our observations follow.

## Test Setup

As mentioned earlier, we designed the chip with the intention of using pushbutton inputs and LED outputs. For a diagram of the testing setup please refer to Appendix I. The supply voltage was obtained from an adjustable Tektronix power supply. The LEDs were active-low by design because they were driven by internally inverted register file values. They were driven through $1.5 \mathrm{~K} \Omega$ resistors from the LED pins to $\mathrm{V}_{\mathrm{cc}}$. The exception is the game-over LED, which is active-high. The pushbuttons were active-high by design. They were driven low through pull-down resistors when the pushbuttons were open. No hardware de-bouncing was incorporated into the pushbuttons because our game-control state machine was intended to enforce a form of software de-bouncing. The LEDs were laid out in a way that visually reflected an actual game of Tic-Tac-Toe, and pushbuttons were laid out as intuitively as possible.

We generated the two-phase non-overlapping clock signals using the Atmel AT90S8515 micro-controller, operating at 4 MHz or 8 MHz , and programmed with simple C programs whose sources can be found in Appendix K. The first, more complicated, program allowed for user input through MS Windows Hyper-Terminal in order to adjust the various properties of the clock signals (e.g. pulse-width, non-overlap period, etc.), and was able to generate clock signals with frequencies on the order of 1 KHz . The second, much simpler, program used assembly code to generate non-overlapping clock signals on the order of the micro-controller crystal frequency, with parametric adjustments enforced through the insertion or deletion of no-ops in the code. This version of the clockgeneration produced clocks with frequencies upwards of 1 MHz .

Using a Tektronix digital oscilloscope we were able to probe individual pins to establish their voltages when the chip is not being clocked (see Appendix J), and find their voltages at various states of a game. We further used the oscilloscope to establish the correctness of intended frequency characteristics of the two-phase non-overlapping clock signals. The most important testing output we had at our disposal, however, was the LEDs, as they monitored the state of the register file and the state of the game itself.

## Testing Parameters

The main parameters that we tweaked extensively were the clock characteristics and the operating voltage. The first time we inserted a chip into our test setup, it did little more than heat up. This problem seemed to be a syndrome among chips being tested by other
groups in the lab, and upon extensive examination of the input pads by a team of professors the suggested solution, that turned out to cure the overheating problem, was to operate the chips at four volts or lower. As it turned out, our chips performance was greatly enhanced by lowering the voltage well beyond four volts.

The more volatile parameter was the clocking characteristics. One can consider twophase non-overlapping clocks to have the following four parameters associated with them:

- Phase one pulse-width
- Time between end of phase one and beginning of phase two
- Phase two pulse-width
- Time between end of phase two and beginning of phase one

By painstakingly adjusting these four parameters, varying degrees of functionality were achieved. As a result of possible on-chip clock skew we found that we needed to increase the space between the two clock phases (the same phenomenon found during our simulations from outside the pads). The most interesting observation that we made was that, counter-intuitively, the slower we ran the clock the less functional our chip became. Ultimately, that fact proved to be the death of the complicated C-coded clocking scheme, and the triumph of the simple assembly coded clocking scheme. The reason which came to surface later was that the automatically generated layout for the PLAs was not internally staticized, so the slower the clock was the more leakage resulted in the dynamic nodes of the PLA.

We wondered if there was a problem with driving LEDs straight off the chip, but even when we detached the LEDs and measured the voltages directly we got the same results. We also wondered how our pulldown resistors on our inputs were affecting our outputs so we reduced them all from $1.5 \mathrm{k} \Omega$ down to $150 \Omega$ but found that this made all the output garbage (similar to our one "bad" chip). Conversely, we increased them to around $10 \mathrm{k} \Omega$ with no noticeable improvement in results over our best configuration.

## Observations

When we finally settled on a reasonable clocking scheme and operating voltage we found that our chip could indeed reset itself and startup a new game. However, it would not permit us to view the first four spaces of the board, and the player was not able to put their pieces there. The computer clearly could put its pieces in those spaces, though, because it would win even though there were no visible wins on the board. We thought we might be dealing with a bad chip but found that four of our five chips behaved this way (the fifth had bad output all the time and was probably defective). We wondered if there was a problem with the lower bits of our datapath (specifically with the register file) until we started messing around with our supply voltage again. When we lowered the supply voltage from 4 V to around 3.25 V all of the computer's moves became visible on the LEDs, and the player could move to additional spaces (but still not all of them). If we
lower the supply too much, of course, then all the LEDs turn on dimly and the output is impossible to read.

## RESULTS AND CONCLUSIONS

Among other things, this project has been a great learning experience for us. We learned how to design a CMOS "microprocessor" from the transistor-level up. We got a chance to see our own design fabricated and available for testing, and we got to experience the (often-harsh) difference between simulation and reality.

If we were to change our design and fabricate our chip again we would do a few things differently. It is clear to us now that using flip-flops for our counters greatly complicates the clocking scheme we need to use, and next time we would only use latches and staticizers. Second, we wouldn't latch our LED outputs before they go off-chip. For our purposes we really didn't need to latch them. We did it solely to make the output prettier, removing the indeterminate board states. If we could get a direct view into the register file when we stepped the input slowly we would be able to see its status at all times. This change would greatly help the debugging process and allow us to see if the computer is actually writing and removing pieces from the board in the way we had intended.

While disappointed with our initial results (when the chip would not even reset), we were quite pleased that our game would play, even though not all the spaces were functioning. The three difficulty levels were all apparent, and the resulting actions of the computer (of which we could see) were consistent with the AI we had designed for it. Whether the player or computer would go first appeared to be random (as designed), and the spot the computer would choose first was also random as to our specifications. When a win is visible to the user the GAMEOVER light is clearly illuminated, so win checking appears to be working.

The most recent observations we made when lowering the supply voltage would seem to indicate there isn't necessarily a defect with our design, but that we still haven't gotten the test setup right yet. Clearly, the clocking strategy is finicky as a result of using flip-flops with only one phase of the clock, but it's not entirely clear if this will prevent our chip from functioning. Ever optimistic, we believe that if we finally get the supply voltage and clocking scheme right we may, someday, have a $100 \%$ fully functional chip.

## APPENDIX

## Appendix A: Finite State Machine of Game Control



## Appendix B: Flow Chart of Hard AI (going first)



## Appendix C: Flow Chart of Hard AI (going second)



Appendix D: Corner - Middle - Corner (AI going first)


Appendix E: Avoiding the Double Win - Player going first



DOUBLE WIN - Al loses



TIE - Double win prevented


Appendix F: How to Beat AI - Player must go first

Player


## Appendix G: TTTPATH - Datapath Layout



Appendix H: TOP - Padframe Layout


Appendix I: Test Setup


Appendix J: Pinouts and Unclocked Test Results

| Pad | Package | Pin | Vfloat |
| :---: | :---: | :---: | :---: |
| p14 | pin 1 | OUT04 | 0.0 |
| p13 | pin 2 | OUT03 | 0.0 |
| p12 | pin 3 | OUT02 | 0.0 |
| p11 | pin 4 | OUT01 | 0.0 |
| p10 | pin 5 | OUT00 | 0.0 |
| p9 | pin 6 | START | 2.5 |
| p8 | pin 7 | VDD | 5.0 |
| p7 | pin 8 | DIFF1 | 2.5 |
| p6 | pin 9 | DIFF0 | 2.5 |
| p5 | pin 10 | PHI1 | 2.5 |
| p4 | pin 11 | PHI1 | 2.5 |
| p3 | pin 12 | PHIO | 2.5 |
| p2 | pin 13 | PHIO | 2.5 |
| p1 | pin 14 | GND | 0.0 |
| p0 | pin 15 | MRST | 2.5 |
| p39 | pin 16 | IN8 | 2.5 |
| p38 | pin 17 | IN7 | 2.5 |
| p37 | pin 18 | IN6 | 2.5 |
| p36 | pin 19 | IN5 | 2.5 |
| p35 | pin 20 | IN4 | 2.5 |
| p34 | pin 21 | IN3 | 2.5 |
| p33 | pin 22 | IN2 | 2.5 |
| p32 | pin 23 | IN1 | 2.5 |
| p31 | pin 24 | IN0 | 2.5 |
| p30 | pin 25 | VDD | 5.0 |
| p29 | pin 26 | OUT17 | 0.0 |
| p28 | pin 27 | OUT16 | 0.0 |
| p27 | pin 28 | OUT15 | 0.0 |
| p26 | pin 29 | OUT14 | 0.0 |
| p25 | pin 30 | OUT13 | 0.0 |
| p24 | pin 31 | OUT12 | 0.0 |
| p23 | pin 32 | OUT11 | 0.0 |
| p22 | pin 33 | OUT10 | 0.0 |
| p21 | pin 34 | OUT09 | 0.0 |
| p20 | pin 35 | GOVER | 0.0 |
| p19 | pin 36 | GND | 0.0 |
| p18 | pin 37 | OUT08 | 0.0 |
| p17 | pin 38 | OUT07 | 0.0 |
| p16 | pin 39 | OUT06 | 0.0 |
| p15 | pin 40 | OUT05 | 0.0 |

## Appendix K: 85152 Phase Clock Generation Source Code

```
/*
/ Generates 2 phase non-overlapping clocks based on 4 events
/ Event1 corresponds to the duration of silence between end
/ of phase2 and beginning of phase1
/ Event2 is the duration of the first phase (edge to edge)
/ Event 3 is the silence between end of phase one and start of phase2
/ Event 4 is the duration of the second phase (edge to edge)
/
/ The user interface uses Hyperterminal
/ When prompted for a command the following options are valid
/ a, b, c, or d followed by an integer
/ the letter designates which event you wish to modify
/ a is event1, b is event2, etc...
/ the integer corresponds to the new time for the given event
/ all times are relative to the beginning of the 2-phase period
/ you can also type e to see the current time values of
/ the four events in hyperterminal
*/
#include <90s8515.h>
#include <stdio.h>
#include <stdlib.h>
#define t1 1
#define t2 250
#define t3 1500
#define increment 5
#define begin {
#define end }
void task1(void);
void task2(void);
void gets_int(void);
void initialize(void);
//RXC ISR variables
unsigned char r_busy; //recieve ISR is running
unsigned char r_index; //current string index
unsigned char r_buffer[16]; //input string
unsigned char r_ready; //flag for receive done
unsigned char r_char; //current character
unsigned char reload, pulse;
int time1, time2, time3;
int event1, event2, event3, event4;
interrupt [TIMO_OVF] void timer0_overflow(void)
begin
    //reload to force 1 mSec overflow
    TCNT0=reload;
    //Decrement the three times if they are not already zero
    if (time1>0) --time1;
    if (time2>0) --time2;
    if (time3>0) --time3;
end
interrupt [UART_RXC] void uart_rec(void)
begin
    r_char=UDR; //get a char
    UDR=r_char; //then print it
    //build the input string
    if (r_char != '\r') r_buffer[r_index++]=r_char;
    else
    begin
        putchar('\n'); //use putchar to aoid overwrite
```

```
        r_buffer[r_index]=0x00; //zero terminate
        r_busy=0; //and clean up
        r_ready=1; //signal cmd processor
        UCR.7=0; //stop rec ISR
        end
end
void main(void)
begin
            initialize();
            //main task scheduler loop
            while(1)
            begin
                if (time1==0) task1();
                if (time2==0) task2();
            end
end
void task1(void)
begin
    time1 = t1;
    //phase one goes high @event1 ms
    if(pulse==event1)begin
            PORTC = 1;
            pulse++;
        end
        //phase 1 goes low at event2 ms
        else if (pulse == event2)begin
            PORTC = 0;
            pulse++;
        end
        //phase 2 goes high at event3 ms
        else if (pulse == event3)begin
            PORTA = 1;
            pulse++;
        end
        //phase 2 goes low at event4 ms
        else if (pulse == event4)begin
            PORTA = 0;
            pulse = 1;
        end
        //always increment the phase counter
        else pulse++;
        PORTB=~pulse;
end
void task2(void)
begin
    int num;
    time2 = t2;
    //change the rising edge of phase 1
    if(PIND == ~0x04 && event1<(event2-increment))
        event1+=increment;
    if(PIND == ~0x08 && event1 > increment)
        event1-=increment;
    //change the falling edge of phase 1
    if(PIND == ~0x01 && event2 < (event3-increment))
        event2+=increment;
    else if(PIND == ~0x02 && event2 > (event1+increment))
                event2-=increment;
    //change the rising edge of phase 2
    if(PIND == ~0x80 && event3 < (event4-increment))
                event3+=increment;
    else if(PIND == ~0x40 && event3 > (event2+increment))
            event3-=increment;
    //change the falling edge of phase 2
```

```
    if(PIND == ~0x10)
        event4+=increment;
    else if(PIND == ~0x20 && event4>(event3+increment))
        event4-=increment;
    if(r_ready)
    begin
        num = atoi(&r_buffer[1]);
        //printf("\nInput = %d\n\rType=%c\n\r", num,r_buffer[0]);
        switch (r_buffer[0])
        begin
            case 'a': if(num > 0 && num < event2) event1 = num;
                        else putsf("Input Out of Bounds\n\r");
                        break;
            case 'b': if(num > event1 && num < event3) event2 = num;
                        else putsf("Input Out of Bounds\n\r");
                break;
            case 'c': if(num > event2 && num < event4) event3 = num;
                else putsf("Input Out of Bounds\n\r");
                break;
                if(num > event3) event4 = num;
                else putsf("Input Out of Bounds\n\r");
                break;
                    case 'e':
                printf("event1=%d\n\revent2=%d\n\revent3=%d\n\revent 4=%d\n\n\r", eve
            nt1,event2,event3,event4);
                                    break;
                                    printf("bad command line argument\n\r");
            end
            gets_int();
        end
end
void gets_int(void)
begin
    r_busy=1;
    r_ready=0;
    r_index=0;
    UCR.7=1;
    printf("Enter Command: ");
end
void initialize(void)
begin
    DDRC=0xff;
    DDRA=0xff;
    DDRB=0xff;
    DDRD=0x00;
    PORTB = 0xff;
    PORTA = 0;
    PORTC = 0;
    //3.69 MHz
    //UCR=0x18;
    //UBRR=0\times17
    //8 MHz
    //UBRR=51;
    //UCR=0x18;
    //4 MHz
    UBRR=25;
    UCR=0x18;
    putsf("INITIALIZED\n\r");
    //set up timer 0
    reload=256-62; //value for 1 Msec
    TCNT0=reload;
    TIMSK=2; //turn on timer 0 overflow ISR
    TCCR0=3; //prescalar to 64
```

```
    time1=t1;
    time2=t2;
    event1 = 20;
    event2 = 30
    event3 = 50;
    event4 = 60;
    pulse = 1;
    gets_int();
    //crank up the ISRs
#asm
#endasm
end
// 2-phase non-overlapping clock generator using embedded assembly
#include <90s8515.h>
void main(void) {
    DDRC=0xff;
    DDRA=0xff;
    PORTC=0;
    PORTA=0;
    #asm
                .equ PORTC = 0x15
                .equ PORTA = 0x1b
                clr R16
                set R17
    #endasm
    #asm
    loop:
                nop
                out PORTA, R17 // phi0 on
                nop
                out PORTA, R16 // phi0 off
                nop
                nop
                nop
                out PORTC, R17 // phil on
                nop
                out PORTC, R16 // phil off
                rjmp loop
    #endasm
}
```

