Implementation of Tic-Tac-Toe with Field Programmable Logic Arrays

11/29/2012

Lab Section: M008

By: Charles Young

Implementation of Tic-Tac-Toe with Field Programmable Logic Arrays

The basis for my failed University Honors Physics II construction project was intended to demonstrate a usage of electricity for logic circuit design. I intended to use the game of tic-tac-toe, as it is an easily understandable game, and it would have been fun to construct a device for one to play tic-tac-toe against. The device's responses could be programmed such that the end of the game could only result in a loss for the user, or a "cat", as ties are termed in tic-tac-toe.

To this end, I intended to use a Field Programmable Gate Array (FPGA) (Figure 1) to implement the circuitry necessary to produce a playable working tic-tac-toe machine. The circuitry would have been too extensive to wire physically, and FPGAs are reprogrammable and serve the purpose of testing and implementing specific circuit designs. FPGAs can be programmed either with programming code, or a schematic design file. A FPGA has assignable switches for inputs and LEDs for outputs. If the circuitry works on an FPGA, then it would work if I were using wires and integrated circuit chips by hand.

To implement this circuitry in the most basic way possible, on/off switches and lights could have been used. The switches would be for the user to make a move, and the lights for the device's response. Each bulb would be wired to be on depending on the combination of which switches are on and off, and which bulbs are already on. It would be necessary to devise a way to lock in the bulbs which are turned on, as moves could not be rescinded. This would definitely be the least optimized solution, but it could be made to work (Hillis 1998, 4-5).

To optimize the system, thus making designing the circuit more simply, logic gates need to be used. A logic gate is just a standardly used operation which has one or more inputs with one

output which is a function of the inputs. Seven simple gates that are AND gates, OR gates, XOR gates, NOT gates, NOR gates, NAND gates, and NXOR gates (Figures 2-8). AND gates output "1" when both inputs are "1". "1" denotes a state being on, high, or the Boolean state of TRUE, whereas "0" denotes a state of being off, low, or the Boolean state of FALSE. OR gates output "1" when any of the inputs equal "1". XOR gates output "1" when one, and only one, of the inputs equal "1". NOT gates output "1" when the single input equals "0", and output "0" when the single input is "1". NOR gates output "1" when both inputs are "0". NAND gates output "1" anytime either input equals "0" (Lewin 1974, xii). All gates can be constructed using OR gates and NOT gates (Hillis 1998, 24). The basics of the circuitry itself can be simple to grasp, but when considering all the various combinations that can be involved in a tic-tac-toe game, the circuitry will be extensive.

To correctly represent the moves which my system will make against its user, I used Boolean algebra to model when each light, which models the system's move, is to be turned on in the system. Boolean algebra is used to mathematically describe logic and reasoning. It is very different from general algebra. For instance "A+B" means "A" or "B", and "AB" means "A" and "B", not "A" multiplied by "B". (Brown 2005, 32). Each of the nine spaces will have a Boolean function which describes the conditions under which the system will make its move. The system's past moves will have to be accounted for along with the user's past moves.

FPGAs are not filled with these various gates, as that would be a waste of circuitry depending on how the FPGA was currently programmed. Instead, and FPGA is filled with three types of resources: logic blocks (from which the needed circuitry can be designed), I/O blocks (to connect the programmed circuitry to input and output pins), and Interconnection wires and

switches (for connecting between logic blocks) (Figure 9). FPGAs can be used to implement the equivalent of more than one million gates. The logic blocks are spaced out in a square 2-D arrangement with the I/O blocks boxing in all four sides. Interconnection wires run vertically and horizontally between the logic blocks, and interconnection switches are at the crossing of interconnection wires. The actual FPGA is only a chip which is inserted into a FPGA board. The board houses the various input and outputs that the chip may be programmed to take advantage of. When the FPGA is programmed, the interconnection wires are made into "routing channels" by allowing the FPGA programming application, Quartus II, connect and disconnect certain wires from other wires and from logic blocks. Within the logic blocks, the various resources are combined in the most efficient way to replicate the design schematic (Brown 2005, 109).

While again, these are enough tools to implement the tic-tac-toe machine, the FPGA allows for higher capabilities for the user to take advantage of. Larger units, such as flip-flops (Figure 10), are able to be dropped onto the block schematic design file since designing one with wires and gates would be highly redundant over time. A flip-flop would be beneficial for the tic-tac-toe design. A flip-flop is essentially a unit with a standard design of gates and connections with input "D" and output "Q" and " Q` ". If Q = 1, then Q` = 0, and vice versa. A flip-flop serves to save a value and to leave the ability to change that value. The flip-flop sets Q to the value of D based on certain extra input conditions, such as the clock time and optional inputs such as preset and clear commands. It could be useful to have a flip-flop to save the value of the current state of each of the nine cells. D could be connected to the pin of a physical switch on the FPGA. By having a flip-flop, there exists one of two outputs that is "1" regardless of whether the input is "0" or "1". This is a feature that is simple, but makes implementation much easier. For

instance, the condition may exist where the system's first move will be the fifth cell, which means that before the move occurs, all cells are empty. With the added benefit of flip-flops, the Boolean expression will simply be reduced to:

$$f_5 = Q`_0Q`_1Q`_2Q`_3Q`_4Q`_5Q`_6Q`_7Q`_8$$

In this expression, f_5=1 only when all the Q`s equal "1", or when all the Qs equal "0". Given that there are many possible instances where the fifth cell will be selected by the system as a next move, the Boolean expression for when the system will use the fifth cell will have many expressions combined by OR("+") expressions.

Now, with all necessary elements introduced, a more comprehensive plan for designing this machine can be laid out. As shown in Figures 11 & 12, which are the first and second half of a block schematic diagram for the FPGA, there are nine input switches for the user, labeled s_0-s_8. It is simply convention in programming that numbering start with "0" instead of "1". There are nine output LEDs, l_0-l_8, which signify the system's moves. There are nine flip-flops, 0-8. Each of the nine will work the same, such that the following Boolean equations apply to all of them, and are relatively simple as shown below:

$$D_N = s_N + f_N$$

$$l_N = F_N$$

For all nine flip-flops, D equals "1" when either the corresponding switch turned to the on ("1") position, or when the corresponding function "f" equals "1". f_0-f_8 are the functions that dictate what cell the system will choose to mark next. "f" equals "1" when that cell is to be taken by the system. To display the system's choice, each LED lights up (l_N="1") when its corresponding function "f_N" equals "1".

Devising the f_N functions is what gets tricky. By using a strictly "blunt force" method of writing these nine functions, each function will have many "base" terms with a standard form combined with OR statements. The "base" terms are individual single cases for which f_N can equal "1". The "base" terms are possible cases of the state of the board. All variables in these terms are combined with AND statements so that every variable must equal "1" so that the term, as a whole, will equal "1". This will indicate that the term actually reflects the current state of the board. Since the multiple terms are combined with OR statements, only one term must equal "1" for f to equal "1". In fact, only one term should equal "1" at a time using this blunt force approach since each term defines every cell. It would not make sense for the possible same state of the board to be present in more than one function, as that would mean that the system is making two moves in one turn. Each of the "base" terms systematically describes the condition of each cell. If cell N should be unoccupied, then Q`N will be in the sequence. If cell N should be occupied by the user, then sN will be in the sequence. If cell N should be already occupied by the system, then QNs`N should be in the sequence. If cell 2 should be taken by the system when the user has cells 0, 1, and 8 and the system has cells 3 and 4, then the function would be as follows:

$$f_2 = s_0 s_1 Q`_2 Q_3 s`_3 Q_4 s`_4 Q`_5 Q`_6 Q`_7 s_8 + (.....)$$

At this point, one of the key features of the flip-flop can be appreciated. It can be noted by some that if f_2 depends on f_1 and f_1 depends on f_2, as soon as one of them changes state that the second may change state since the second function would only be in its previous state when the first function was in its previous state. This seems as if it would keep going back and forth, which is a problem. This would mean that the system could take back a move that it had

previously made. That would never be acceptable. To circumvent this problem, no function will reference another function directly. Instead, the flip-flop outputs, Q and Q`, will be referenced. The reason that this fixes the problem is that the flip-flop can be programmed to only propagate D through the flip-flop and into Q once, which would be when either the user or the system selects that cell. After that time, it no longer matters what f equals, since Q is set for the duration of the game. This eliminates the circular referencing and the ability of the system to "take back" moves. To start a new game, one physical button on the FPGA can be linked to the reset on all of the flip-flops so that Q can change again.

While still utilizing the "blunt force" method of writing the "f" functions, they can be optimized somewhat. It only makes sense that with so many terms for so many possibilities combined by OR statements, that the functions are not optimized. There are many method used to help visualize what can be optimized within a function. These methods include sum-of-products, product-of-sums, truth tables, Karnaugh maps, and the tabular method for minimization (Brown 2005, 33-213). The method I would implement would be using the Boolean algebra properties with the sum-of-products and product-of-sums rules. The following properties of Boolean algebra apply and can allow for the "f" functions to be simplified:

$xy = yx$	Commutative
$x+y = y+x$	
$x(yz) = (xy)z$	Associative
$x + (y + z) = (x + y) + z$	
$x(y + z) = xy + xz$	Distributive
$x + yz = (x + y)(x + z)$	

x + xy = x	Absorption
x(x + y) = x	
xy + xy` = x	Combining
(x + y)(x + y`) = x	
(xy)`= x` + y`	DeMorgan's Theorem
(x+y)` = x`y`	
x + x`y = x + y	
x(x` + y) = xy	
xy + yz + x`z = xy + x`z	Consensus
(x + y)(y + z)(x` + z) = (x + y)(x` + z)	

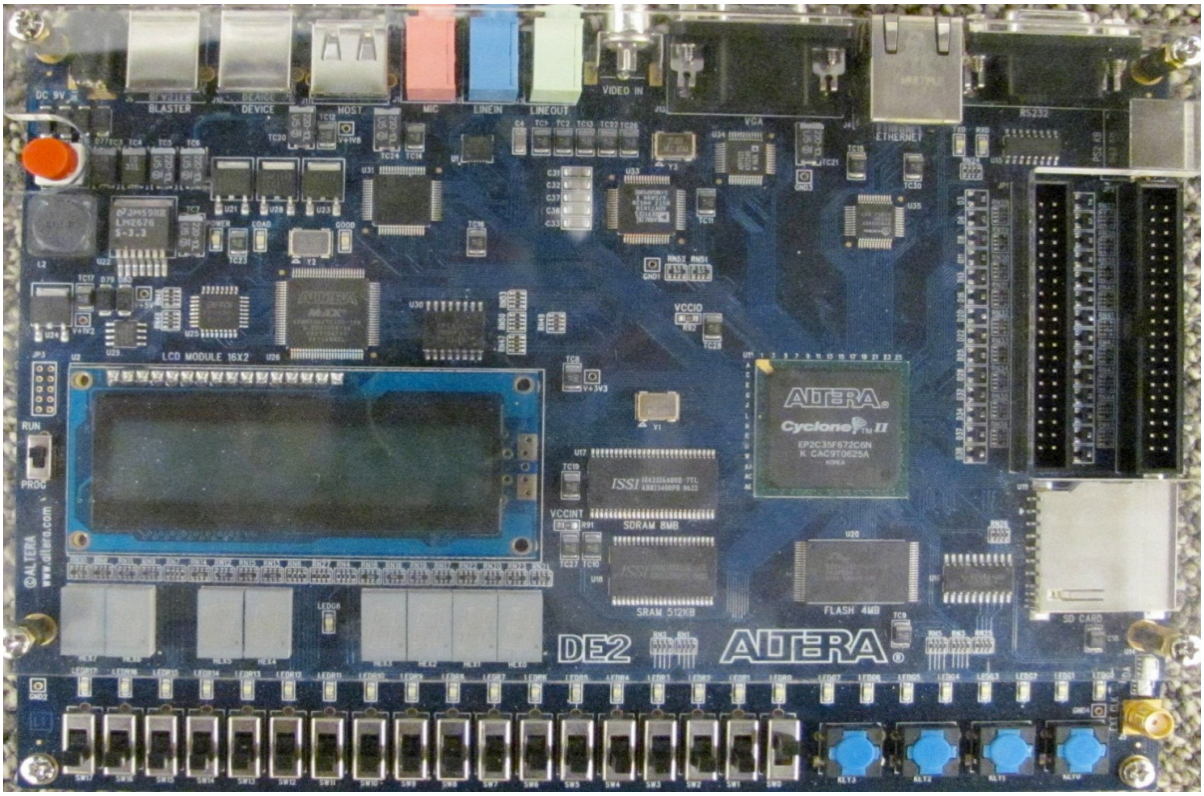
To the best of my knowledge, all of the design knowledge previously presented is accurate and could be easily implemented. I could have done so, I believe, but one obstacle stood in my way that I did not know how to overcome in a manageable amount of time. The one problem is that of deciding what terms use for the "f" functions. What cells should the system pick and when? As previously stated, it is possible to have the right response for every move so that the user cannot win. A tic-tac-toe tree can be drawn out, just as Hillis did (Hillis 1998, 5) (Figure 13), but I did not feel that I would be able to do that by hand without mistake. Instead, I wrote a computer program in C++, which I thought would ingeniously solve my problem. My program, attached as Appendix A, has nine nested FOR loops which serve the purpose of running through every possible action and response between user and system, stopping when one

opponent would win and outputting “WIN”, “LOSE”, or “DRAW”. The first few lines of output are as follows:

```
0
0 1
0 1 2
0 1 2 3
0 1 2 3 4
0 1 2 3 4 5
0 1 2 3 4 5 6
LOSE
0 1 2 3 4 5 7
0 1 2 3 4 5 7 6
0 1 2 3 4 6
0 1 2 3 4 6 5
0 1 2 3 4 6 5 7
0 1 2 3 4 6 7
0 1 2 3 4 6 7 5
0 1 2 3 4 7
0 1 2 3 4 7 5
0 1 2 3 4 7 5 6
0 1 2 3 4 7 6
LOSE
0 1 2 3 5
0 1 2 3 5 4
0 1 2 3 5 4 6
0 1 2 3 5 4 6 7
WIN
```

My intention was to use this to trace the game choices from the last choice made back to the first, so that the right choice could be made first. By systematically going through this output and crossing out any system move that can possibly result in defeat, a feasible strategy could be worked out and written into the nine “f” functions. If the user makes the first move, the system has to have eight different responses for each of the user’s different possible choices. Next, it has six different responses within each of the first eight to cover all possible user moves. It will have to have four responses and then two responses in the same manner as before. Using the pure

"blunt force" method for writing the "f" functions, there would then have to be 384 terms between all nine functions to account for all of the choices that the system would make in all of those various eventualities. In a game, there are 9 first choices, 8 second choices, 7 third choices, and so on. Therefore, there are 9! (362,880) ways for all of the cells to be filled presuming that the order of filling the cells should be considered a difference between two sets of cells filled the same way. Obviously, there are not 9! ways for the game to end, as many games are won before all tiles are filled, and my program reflected that. Unfortunately, I did not grasp how extensive the list of possible outcomes would still be. With single-spaced 11 point type, my output file has a length of 3793 pages. I believe this data to be correct, and given time and more knowledge I believe I could develop another program to sort through the data into something which I could manageably work with. Even if that were to happen, there still is the issue of 384 terms in my "f" functions. Perhaps there is another approach than bluntly accounting for every possible case. Given that many of the possible eventualities are copies of each other but rotated by some multiple of 90 degrees, there may be ways to minimize the functions. Nothing in my project happened which made my project fail besides my approach being so unwieldy that there was no way to continue forward. Many times in getting to the point I did, I tried and was able to find a more efficient path forward. My understanding and skills are greater than they were before I began this project, because the path I took was not the path I intended to travel but a path I found while in the midst of working.



ALTERA DE2 FPGA Board

Figure 1

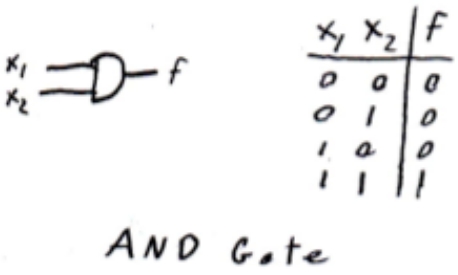


Figure 2

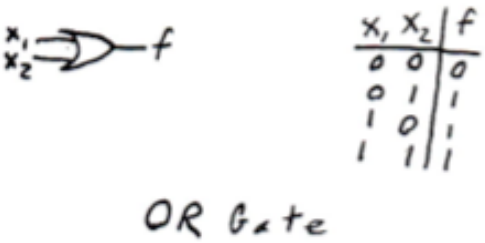


Figure 3

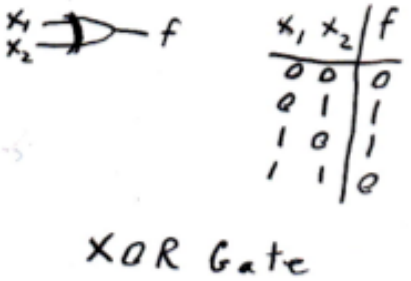


Figure 4

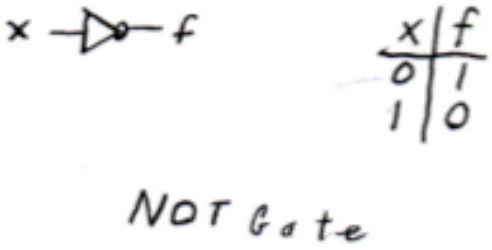


Figure 5

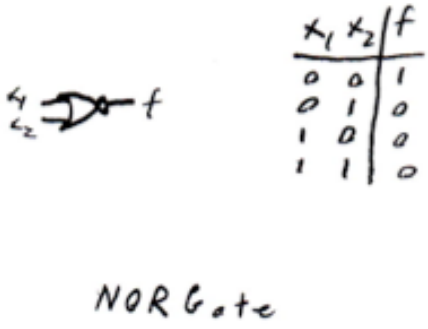


Figure 6

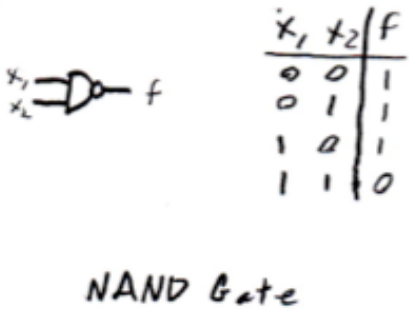


Figure 7

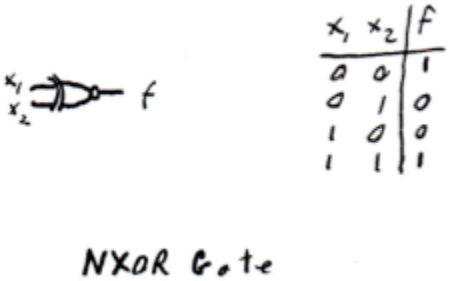
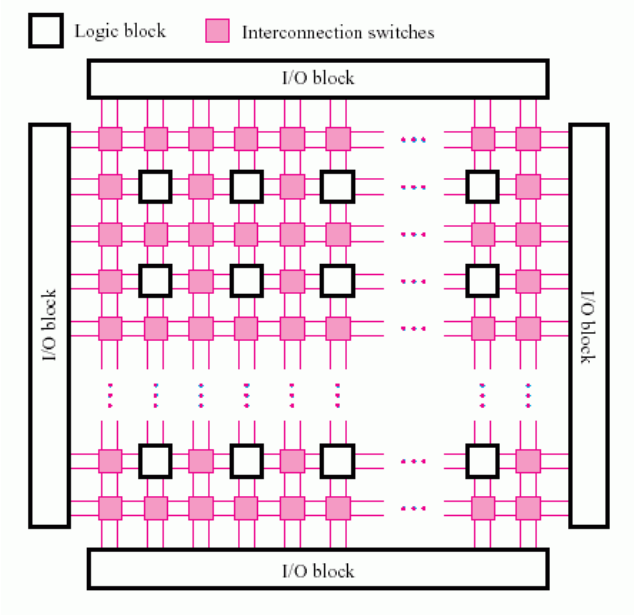
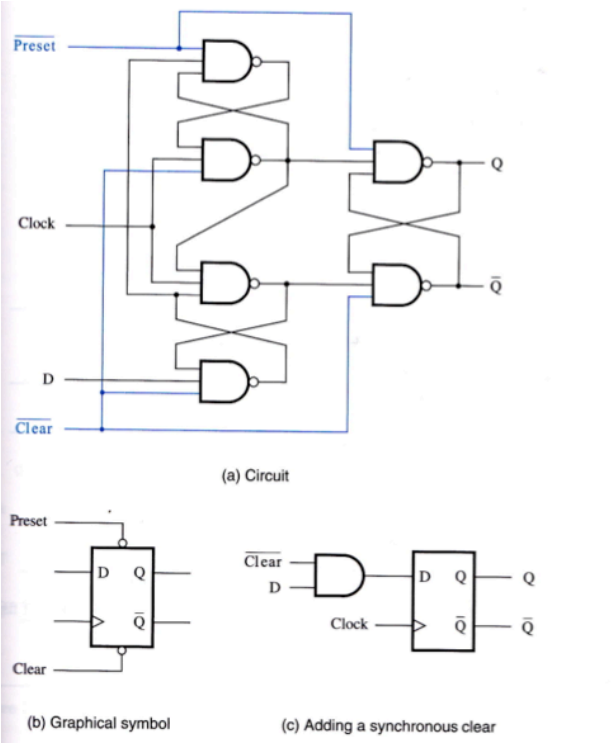


Figure 8



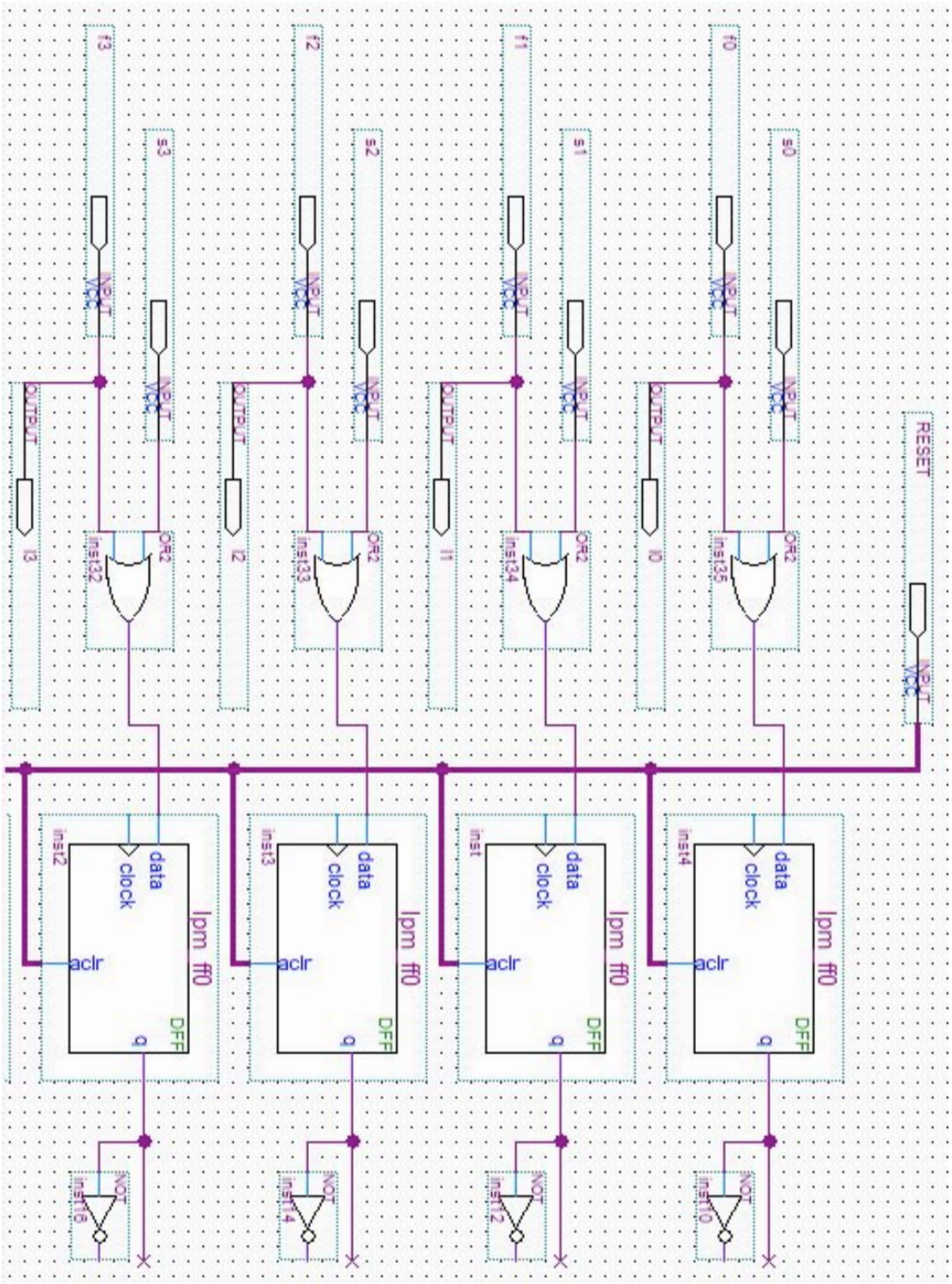
Basic FPGA Layout (Brown 2005, 110)

Figure 9



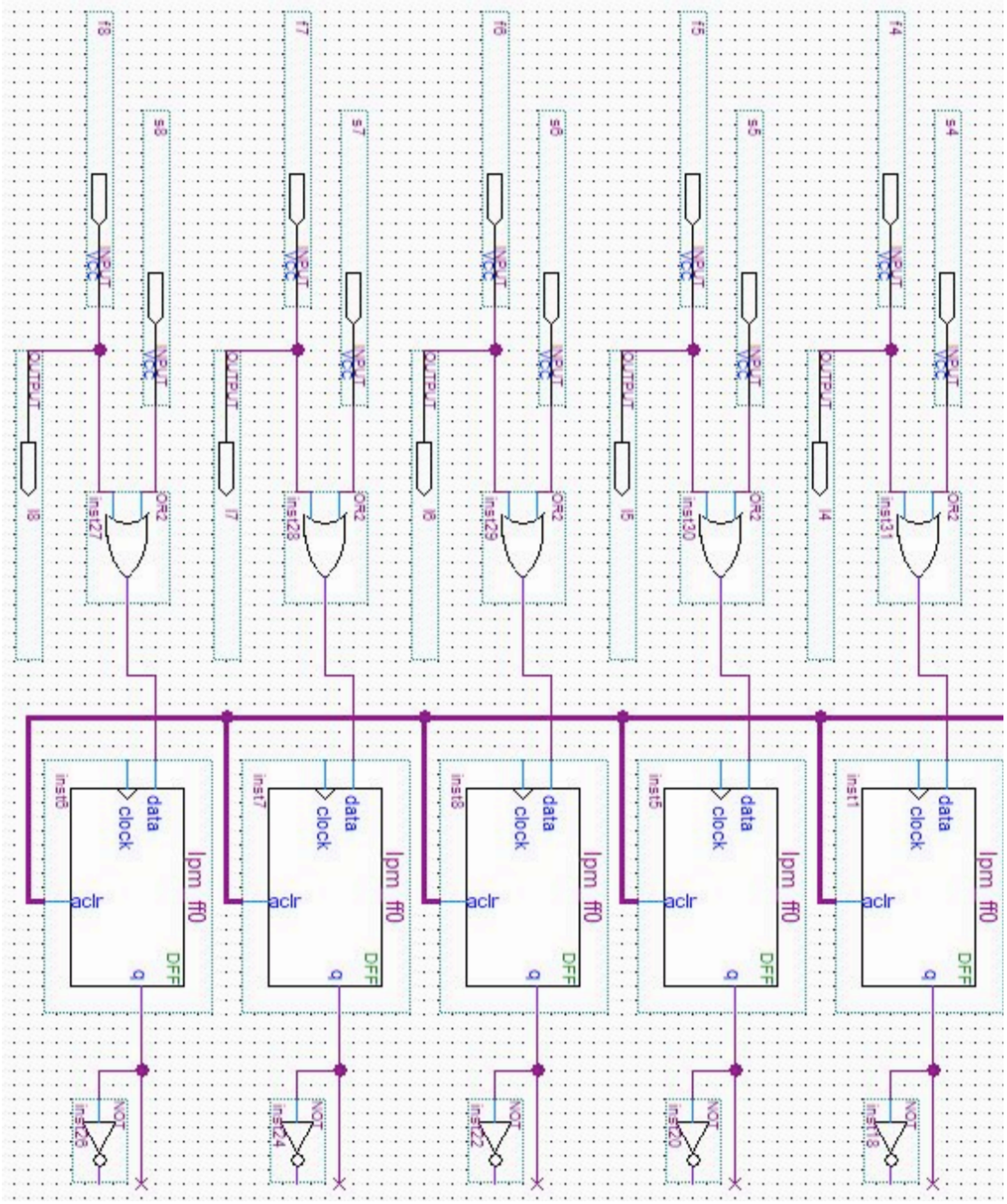
Positive Edge Triggered D Flip-Flop with Clear and Preset (Brown 2005, 397)

Figure 10



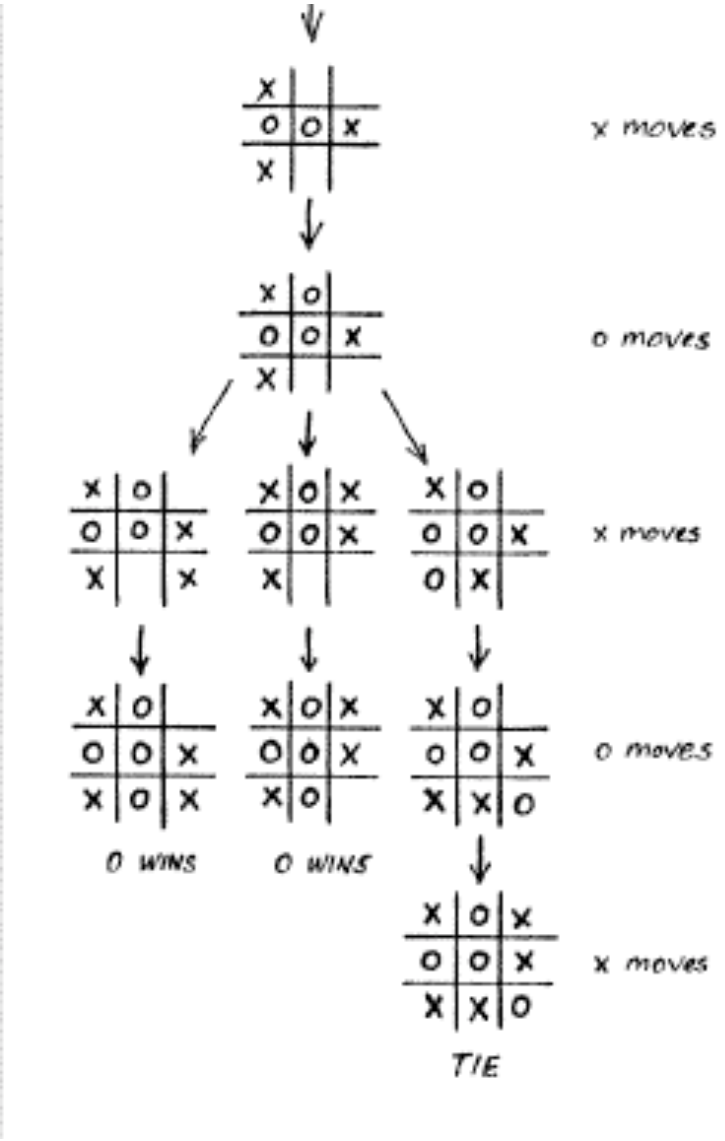
First Half of the Block Flow Diagram for the FPGA

Figure 11



Second Half of the Block Flow Diagram for the FPGA

Figure 12



Part of a Game Tree for Tic-Tac-Toe (Hillis 1998, 5)

Figure 13

Works Cited

Brown, Stephen D., and Zvonko G. Vranesic. 2005. *Fundamentals of digital logic with VHDL design*. Dubuque, IA: McGraw-Hill Companies.

Hillis, W. Daniel. 1998. *The pattern on the stone: the simple ideas that make computers work.* New York: Basic Books.

Lewin, Douglas. 1974. *Logical Design of Switching Circuits*. New York: American Elsevier Publishing Company, Inc.

Appendix A

C++ Code Used to Generate the 3793 Page List of
All Possible Tic-Tac-Toe Outcomes

```
#include <iostream>
using namespace std;

int test(int array[]);

int main () {

 int board[8] = {0};
 int w = 0;

 for (int a=0;a<9;a++) {
  board[a]=1;
  cout << a << endl;
  for (int b=0;b<9;b++){
   if (board[b]==0){
    board[b]=2;
    cout << a << " " << b << endl;
    for (int c=0;c<9;c++){
     if (board[c]==0){
      board[c]=1;
      cout << a << " " << b << " " << c << endl;
      for (int d=0;d<9;d++){
       if (board[d]==0){
        board[d]=2;
        cout << a << " " << b << " " << c << " " << d << endl;
        for (int e=0;e<9;e++){
         if (board[e]==0){
          board[e]=1;
          cout << a << " " << b << " " << c << " " << d << " " << e << endl;
          w = test(board);
          if (w == 0){
           for (int f=0;f<9;f++){
            if (board[f]==0){
             board[f]=2;
             cout << a << " " << b << " " << c << " " << d << " " << e << " " << f << endl;
             w = test(board);
             if (w == 0){
              for (int g=0;g<9;g++){
               if (board[g]==0){
```

```
            board[g]=1;
            cout << a << " " << b << " " << c << " " << d << " " << e << " " << f << " " << g <<
endl;
            w = test(board);
            if (w == 0){
             for (int h=0;h<9;h++){
              if (board[h]==0){
               board[h]=2;
               cout << a << " " << b << " " << c << " " << d << " " << e << " " << f << " " << g <<
" " << h << endl;
                w = test(board);
                if (w == 0){
                 for (int i=0;i<9;i++){
                  if (board[i]==0){
                   board[i]=1;
                   cout << a << " " << b << " " << c << " " << d << " " << e << " " << f << " " << g
<< " " << h << " " << i << endl;
                    w = test(board);
                    board[i]=0;
                  }
                 }
                }
                board[h]=0;
               }
              }
             }
             board[g]=0;
            }
           }
          }
          board[f]=0;
         }
        }
       }
       board[e]=0;
      }
     }
     board[d]=0;
    }
   }
   board[c]=0;
  }
 }
```

```
    board[b]=0;
   }
  }
  board[a]=0;
 }


return 0;
}




int test(int array[]){
 if ((array[0]==array[1]) && (array[1]==array[2]) && (array[2]==1)){
  cout << "LOSE1" << endl;
  return 1;
 }
 if ((array[3]==array[4]) && (array[5]==array[4]) && (array[5]==1)){
  cout << "LOSE2" << endl;
  return 1;
 }
 if ((array[6]==array[7]) && (array[7]==array[8]) && (array[8]==1)){
  cout << "LOSE3" << endl;
  return 1;
 }
 if ((array[0]==array[3]) && (array[3]==array[6]) && (array[6]==1)){
  cout << "LOSE4" << endl;
  return 1;
 }
 if ((array[1]==array[4]) && (array[4]==array[7]) && (array[7]==1)){
  cout << "LOSE5" << endl;
  return 1;
 }
 if ((array[2]==array[5]) && (array[5]==array[8]) && (array[8]==1)){
  cout << "LOSE6" << endl;
  return 1;
 }
 if ((array[0]==array[4]) && (array[4]==array[8]) && (array[8]==1)){
  cout << "LOSE7" << endl;
  return 1;
 }
```

```
 if ((array[2]==array[4]) && (array[4]==array[6]) && (array[6]==1)){
  cout << "LOSE8" << endl;
  return 1;
 }
 if ((array[0]==array[1]) && (array[1]==array[2]) && (array[2]==2)){
  cout << "WIN1" << endl;
  return 2;
 }
 if ((array[3]==array[4]) && (array[4]==array[5]) && (array[5]==2)){
  cout << "WIN2" << endl;
  return 2;
 }
 if ((array[6]==array[7]) && (array[7]==array[8]) && (array[8]==2)){
  cout << "WIN3" << endl;
  return 2;
 }
 if ((array[0]==array[3]) && (array[3]==array[6]) && (array[6]==2)){
  cout << "WIN4" << endl;
  return 2;
 }
 if ((array[1]==array[4]) && (array[4]==array[7]) && (array[7]==2)){
  cout << "WIN5" << endl;
  return 2;
 }
 if ((array[2]==array[5]) && (array[5]==array[8]) && (array[8]==2)){
  cout << "WIN6" << endl;
  return 2;
 }
 if ((array[0]==array[4]) && (array[4]==array[8]) && (array[8]==2)){
  cout << "WIN7" << endl;
  return 2;
 }
 if ((array[2]==array[4]) && (array[4]==array[6]) && (array[6]==2)){
  cout << "WIN8" << endl;
  return 2;
 }

   return 0;
}
```