Gomoku Player Design

CE126 Advanced Logic Design, winter 2002

University of California, Santa Cruz

Max Baker
(max@warped.org)

Saar Drimer
(saardrimer@hotmail.com)

0. Introduction........ 3
0.0 The Problem........ 3
0.1 Strategy........ 3
1. Design........ 3
1.0 Overview........ 3
1.1 PC Components........ 4
1.2 Logic Design Components........ 5
2. Design Flow........ 7
2.0 Bidirectional BUS........ 7
2.1 Simplified Operation........ 7
3. Results........ 7
4. Conclusion........ 8

0. Introduction

0.1 The Problem

The assignment was to design a Gomoku player on a 16x16 board that played against a program run on a computer. Gomoku is a 5-in-a-row game played on a Go board using black and white game pieces. Our design must interface with the computer and decide on the next move according to the state of the playing board.

Our design must fit into 3 Xilinx XC4003E-PC84 FPGAs (including the computer-board interface FPGA) placed on the BORG II board. The BORG also has an 8K SRAM connected to one of the FPGAs.

0.2 Strategy

The computer holds the status of the board and in addition decides who wins. In order to make next move decisions, our design must also keep track of the 16x16 complete board.

Due to the constraints of resources we decided to evaluate 5x5 blocks to assess our next move. The block slides across the playing board covering all possible 5x5 blocks. These 5x5 blocks are then compared with a 5x5 pattern block and if there is a match a coordinate is sent to the computer as move.

We decided to use the onboard 8K SRAM as a ROM where we store the 5x5 pattern blocks and our desired move for every such pattern. A pattern file is loaded onto the SRAM and therefore sets the strategy of the player. This scheme enables us to have a highly versatile player because we can change its strategy on the fly without the need to change our logic design. In many ways this design is a learning machine that could add patterns as the game progresses and upon each loss. The pattern file could also be adapted to different opponent skills; for example, if playing against a machine or a human.

1. Design

1.0 Overview

The design is implemented using the *BORG II Board*, a custom prototyping board with five Xilinx 4000 series Field Programmable Gate Arrays (FPGA), a PC ISA bus interface, and an onboard 8KB SRAM. The design was achieved with a combination of schematic entry, hardware description languages (HDL), and truth-table style inputs. Specifically, Xilinx Foundation 3.1i is used for schematic entry and Verilog is used sparsely inside of Foundation. Finite state machines, as well as some combinational devices are created and minimized using the Mustang, Espresso and SIS tools.

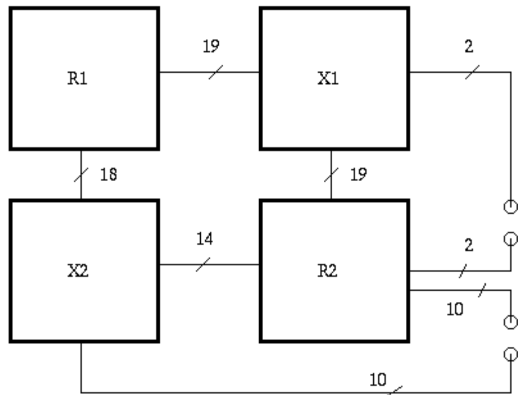


Figure 1. The Borg II interconnections

The design uses three of the four available Xilinx 4003e FPGAs. The first chip, R1, contains the interface to the PC and the latching of the white and black latest moves. The second chip, X1, contains the main state machine, the storage of the playing board, as well as most of the supporting combinational logic blocks. The third chip, R2, contains a state machine used to load the patterns from the onboard SRAM, a row by row comparator, and storage for the current pattern.

1.1 PC Components

The PC is running a Gomoku player written in C and always plays first (black). Once it has put a move on the board it sends the coordinate to the BORG board, de-asserts its *black_ready* signal waiting for the next white move.

The PC, in addition to programming the FPGA's, is used to generate the patterns and to store them into the SRAM. One program written in Perl, *makepat.pl*, takes an ASCII file containing patterns, and uses it to create a binary file that is the exact contents of the SRAM. A binary encoding of each of these is configured at the top of the program, as well as the output and input files; see appendix A for complete source code. Since the SRAM is 8 bits wide, only 5 bits are used from the whole byte to store the least and most significant bits in 2 consecutive bytes. After 10 bytes of pattern content a 6 bit coordinate is written giving a total of 11 bytes for each pattern. After each pattern is read from the input file, it is then rotated and mirrored the eight possible different ways. It is then checked for duplicates, and written to the output binary file. On one 35-pattern file, an average of 5.45 rotations was made from each seed pattern. The order of the pattern file determines the priority of each move and therefore, the overall strategy of the player.

Pattern file encoding:

```
. : don't care
b : blank spot
B : blank spot -> next move here
X : Black piece
O : White Piece
```

A pattern example from patter file:

```
B....
.X...
.OX..
...X.
....b
```

SRAM encoding:

```
00 : blank spot
01 : black piece
10 : white piece
11 : don't care
```

A converted pattern:

```
11110000
11110000
11111000
11101000
11101000
11011000
11111000
10111000
01111000
01111000
10010000
```

You might have noticed that the converted pattern is inverted. This is because the design takes reads in the LSB first (more on this in the conclusion section).

The second program, *borgmem*, written in C, reads and writes to the onboard SRAM. A binary input file is read byte-wise and used to fill the onboard SRAM. See the source code for various command-line parameters.

1.2 Logic Design Components

Main FSM

This block was written as mustang input file and minimized using the Espresso and SIS tools. This FSM controls the overall progress of the design, from the first *black_ready* signal from the computer to the *white_ready* signal telling the computer the next white move is ready. There were several iterations of this FSM and it was originally written as a Mealy machine. The debugging process made us change the FSM to a Moore machine in order to be sure it is working properly. The chosen encoding for the state machine is *one-hot* in order to save on LUT which were scarce and use the more abundant flip-flops. Both the Mealy and Moore implementations are included in this report.

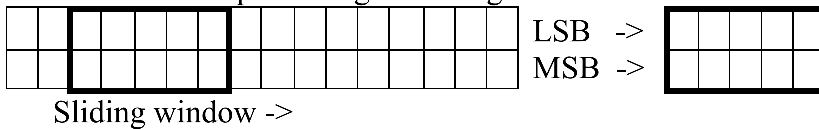
Game Board Storage

The current status of the game board is stored internally using the FPGA's look-up tables (LUT) used as RAMs. As the board is 16x16 large, 8 bits are needed to address a specific square (4 bits for X and 4 for Y). Since each square is one of three possible states: blank, white or black, two bits are needed to encode the state of each square. Each LUT has 16 bits, and is addressed one bit at a time using the

library part RAM16X1S. Using the RAM16X2S macro, each 16x2 RAM holds a column of squares. Connecting the address of the lines from each RAM together means that one row (16 squares) of the board can be addressed at a time. Two 3-to-8, active-high decoders are used to provide a mechanism to write to a single square of the board. The output of the decoders are connected to the write-enable of the RAMs, so that when data inputted is written, one full row is addressed, but only one square is enabled.

Game Board Window
The output of the game board block (*board*) is a 16x2bit row. From this row a *window* of five board spots (10-bits) is compared against a row of a pattern. The *game board window* uses ten 5-bit multiplexers to take the 32-bit input of the current row to make the 10-bit output of the five-square window.

One 16x2 row representing a whole game board row with a single row output:



SRAM Controller
A finite state machine created in Verilog controls the onboard SRAM. This FSM uses a manual state assignment in order to make each output a state bit to insure glitch free signals for the SRAM. A start signal from the main state machine tells the controller to fill the pattern store with the next 11 bytes of memory from the SRAM. When the pattern store is full a done signal is raised for the main FSM. While waiting for a start signal the SRAM control pads are put into tri-state so that the PC can take control of the SRAM. The PC can then change the patterns that the entire design is using, and thereby dynamically change and refine the strategy used without the need to reload.

Pattern Store
The pattern store holds the current pattern from the SRAM used by the comparator to search for the next white move. The Pattern store is implemented using 50 flip-flops (5x5x2), as well as some binary counters used in storing and retrieving pattern rows. An additional 6 flip-flops are used to store the location, or *spot*, of the next move relative to the 5x5 pattern. The translator maps this location to the placement of the window on the game board (8 bit output).

Comparator
The comparator is fed the current line of the current board window and is compared to the current line of the pattern. In the case of a don't pattern care spot, the comparator "ignores" the comparison of that spot.

Translator
In each of the patterns there is a 6 bit coordinate (3 bits for X and 3 bits for Y), called *spot*, indicating where to put the next white piece in the 5x5 window. In order to translate this coordinate to a valid 8 bit coordinate, the translator is fed the current board address and adds to it the 3 bit X and Y spot coordinate yielding a valid board spot.

2. Design Flow

The following is a simplified version of the decision making process. Many control signals are omitted because they are irrelevant for this discussion. A close examination of the main FSM and the schematics will reveal their function.

2.0 Bidirectional BUS

There was a need to transfer the 10 bits of the board window row, called *brow*, from X1 to R2 and in turn transfer the 6 bit *spot* from R2 to X1. In addition to the control signals, there were not enough hard wired connections between the two FPGAs (there are 19 connections between R2 and X1, as seen in Figure 1). To solve this we combined the two signals into a bus called *brownspot* which is controlled by the *busc* signal coming from the main state machine. The only time *spot* is on the bus is when a matching pattern is found and *spot* is needed for the translator to produce valid 8 bit coordinate for the white move.

2.1 Simplified Operation

The following simplified overall procedure is started when *black_ready* is deasserted:

```
store black move on the local board
reset all counters (except address counter)
load next pattern
Until (window_counter = 5) do {
        until !(line match) do {
                Compare line of pattern and board_window
                Window_counter + 1 }
        if window_counter = 5 (MATCH!)
                put spot on the bus
        else {
                load next pattern
                reset window_counter }
        }
assert white_ready
go back to initial wait state
```

There will never be a case where is there is no pattern match because the last pattern in the pattern file will always account for the first move.

3. Results

Our results were promising -- we managed to complete games up to 184 moves (92 each player) against the computer; see Figure 2. It is difficult to construct a pattern file to exactly match the computer's strategy and therefore our player loses. On the other hand, while playing against a human, our player often wins. Our losses are not discouraging because of the way our strategy works: whenever there is a loss, a pattern could be added at the right priority and eliminate that particular loss. A comprehensive learning machine could learn and add patterns to the pattern file and win very often, maybe always. Unlike other designs, our player is highly modular and capable; due to the lack of time we did not invest more in producing better pattern files.

```
   A B C D E F G H J K L M N O P Q            #  black  white
16 . . * . . . O . . . . . O . . * 16         77   M2     N2
15 . O O . . . * . O . O * * * . O 15         78   M3     M4
14 . . * O * O * * * O * O O O O * 14         79   N3     K1
13 * O * * * O * O * O . O * * * * 13         80   P3     O3
12 . . * O O * * O * O O . * O * * 12         81   O6     L1
11 . O O . * O O O O * * * * O . O 11         82   P5     P2
10 . . * O * * O * * O * O O O * . 10         83   Q6     O4
 9 O . * . O O O * O O O O * . O . 9          84   Q4     N7
 8 . * O O * * * * O * * * O * * * 8          85   Q5     Q3
 7 O O * * * O O * * O * * O O O O 7          86   Q8     Q7
 6 . . * O O * * O O * O O * * . * 6          87   C14    P1
 5 . O O . . * O O O * O * . O * * 5          88   C16    C15
 4 . . * . O * * O * * O O . O * * 4          89   F2     B15
 3 . . . . . * . O . O . * * O * O 3          90   F3     G2
 2 . . . . . * O * O * * * O * O . 2          91   F4     F1
 1 . . . . . O . . . O O . . . O . 1          92   F6     F1
   A B C D E F G H J K L M N O P Q
      Search Depth 2  Seed 1016327465
    BLACK/bono      vs.     WHITE/Borg         BORG response 80 (5,0)
Bye bye
Ha ha, I won
```

Figure 2. A loss after 184 moves.

According to the implementation reports, X1 can run at a frequency up to 34MHz and R2 up to 62MHz.

4. Conclusion

This design took more than 150 hours in a period of 2 weeks to complete. It took intensive work, concentration and creativity to produce this fine design. We both learned a lot about logic design, PC interface, logic minimization, schematics capture tools and working as a team.

Among other minor error we discovered early on in the process of debugging we encountered an error that took a few days to correct. When we visualized the pattern loading into the SRAM we thought it corresponded to the way the machine read it. This is a classic “endian” problem where the SRAM loading was reversed to what our logic expected. After laborious efforts and simulation, we caught on to this detail, fixed the SRAM loading program and the player worked without any change to the logic design itself.

We would like to thank Martine Schlag for her help with this project.

Attached Materials:
FSM, PLA, EQN files (floppy)
Source codes of PC components (floppy)
Pattern files (floppy)
NCB and MCS files (floppy)
Design schematics printouts
Implementation reports for X1 and R2
FSM drawings