# ECE 350 Final Project - Rotating Memory Game

Andre Lim

Spring 2025

My project's goal was to build the popular memory card game, where the player must match pairs of face-down cards arranged in a $4 \times 4$ array. In addition, I aimed to augment the design with a few additional gameplay features, namely:

1. Accelerometer-based movement to steer the player's block avatar
2. Bullets firing through lanes in which the player's block can be steered through

**Project Components**

1. **VGA Display**: All visuals were hard-coded in Verilog. VGA display receives outputs via the FPGA's built-in VGA port
2. **CPU Core**: The CPU core as designed throughout the course. It processes the $x$- and $y$-coordinates of the player's block avatar input for collision detection with cards, then updates variables stored in specific memory locations to control screen display. We run the CPU with a 50 MHz clock.
3. **Nexys A7 FPGA**: Built-in ADXL362 3-axis MEMS accelerometer is used to gather acceleration data, which controls the movement of the player's block avatar. The on-board LEDs and 7-segment display on the FPGA is used to display the $y$-coordinate of the player's block avatar, as well as the time taken until the completion of the game. Input is added in the form of a start button (BTNR), which is used to transition from a start screen to the actual game.

## Gameflow FSM

In state 0, no cards are flipped.

In state 1, 1 card is flipped. If the player remains in or re-enters the range of the first flipped card, we remain in state 1. We transition to either state 2 or state 3 when the plater enters the range of a second card.

In state 2, the first and second flipped cards do not share the same color. We turn both cards face-down and transition back to state 0. The number of lives is decremented by 1. While the player has not exited the range of the second card upon first entry, we prevent further penalty on the player. If number of lives is equal to 0, the game ends.

In state 3, the first and second flipped cards share the same color. We make both cards disappear and transition back to state 0. The score is incremented by 1. If the score is 8 i.e. all cards have been matched, the game ends and the time taken to complete the game is shown on the 7-segment display of the FPGA.

## Main Design

The visual display was primarily controlled in Verilog, namely by the *VGA-Controller* module, which takes in a series of inputs from the RAM memory module, namely $b1\_SOLVED$, ..., $b16\_SOLVED$ and $b1\_ON$, ..., $b16\_ON$, that keep track of each card's status. These inputs determine if a card has been flipped or matched, thus controlling the card's appearance on the screen. While the player's block avatar is not in the range of any of the 16 cards, we continuously loop through the 16 checks for the 16 cards.

If a card is matched, it is hidden from view. If an unmatched card is ON, its true color is revealed, and if it is OFF, it is shown as white. We were constantly reading each variable in sync with the VGA clock, so the display would always be updating.

The variables are stored in specific memory locations that are updated by the assembly instructions passed to the processor. The processor is also used to detect when the player's block avatar is in the range of any one of the 16 cards so that the display can be updated accordingly. The $x$- and $y$-coordinates of the player's block avatar are passed from the *VGAController* module into registers \$*gp* and \$*sp*, which are then accessible by assembly instructions read by the processor. Within the RAM memory, the $x$- and $y$-coordinates of the top-left vertex of each card are stored in specific locations, thus facilitating the card boundary comparison checks in our assembly code, following which $b1\_SOLVED$, ..., $b16\_SOLVED$ and $b1\_ON$, ..., $b16\_ON$ are updated with the *sw* instruction. To keep track of the state of the game, a register \$*t9* is updated accordingly.

For each of the cards from 1 to 16, we compare the coordinates of the boundary edges of each card with the $x$- and $y$-coordinates of the block read from registers \$*gp* and \$*sp*. If any of the cards has been matched, the check for that card is bypassed. If the check for a card is determined to be successful, the assembly instructions follow a branch to update the state of the game accordingly. Furthermore, the colors of each card are stored in specific locations in RAM memory, and are assigned to the cards in a pseudo-random order, determined by a 4-bit linear feedback shift register (LFSR).

Lastly, we add bullets that pass through the gaps between the cards, which the player must avoid while traversing the grid. This is controlled in the *VGAController* module in Verilog. Each bullet consists of a small rectangle which is translated from the top of the VGA display to the bottom. Upon reaching the bottom, the $y$-coordinate is set to 0, and the bullet is translated in the downward $y$-direction again.

### Accelerometer Data

The Nexys A7 FPGA has a built-in ADXL362 accelerometer, which we use to simulate the effect of gravity within the game. The state machine used to obtain input from the accelerometer via an SPI communication protocol is as follows:

1. Wait for 6 ms to allow sensor to power up and go into standby mode
2. Send a write instruction (0x0A)
3. Send the register address to write to (0x2D)

4. Send the data to be written (0x02), which sets the mode of the accelerometer to measurement mode
5. Wait for 40 ms to allow sensor to calculate valid data after being put into measurement mode
6. Send a read instruction (0x0B)
7. Send the register address to read from (0x0E). This obtains the LSB of $x$-axis data
8. Read $x$-axis LSB
9. Read $x$-axis MSB
10. Read $y$-axis LSB
11. Read $y$-axis MSB
12. Read $z$-axis LSB
13. Read $z$-axis MSB
14. Wait for 10 ms to allow calculation of another set of valid data, loop back to Step 6 and repeat.

When the magnitude of the input accelerometer data is equal to 0, the block's translational movement is set to 0 as well. Since the orientation of the accelerometer within the FPGA is not known, we manually observe and assign the directions in which rotation of the accelerometer about a particular axis changes the $x$- or $y$-axis data. These are then later used to determine the directions in which the player's block avatar will be translated, by trial and error. The ultimate goal is to have the player's avatar block translate in the downward direction, as viewed by the player holding the console (VGA display), simulating the effect of gravity acting on the plater's block avatar.

## Processor Changes

A *fill $rd, $rs, immed* instruction is added to the processor. The wire *is_fill* detects when the instruction is a *fill* instruction and when the first *fill* instruction reaches the Execute/Control logic (i.e. on the rising clock edge when *is_fill* is true and *fill_active* is false), *fill_active* is set to true. While fill_active is true, every pipeline stage is frozen (by setting *latch_enable* and *stall_enable* to false) to devote one full cycle to each memory write, of which there are 16 in total. *address_dmem* steps through [*fill_base*, *fill_base*+1, ...]. *wren* is held high unconditionally and *data* is driven from our 4-bit RNG module, so each memory location gets a pseudo-random value from 0 to 15. The *fill* instruction has opcode 01001. The randomly allocated card IDs are then used to assign random colors to each of the cards before we enter the actual gameloop.

## Challenges

The documentation for the built-in ADXL362 accelerometer was relatively complex and I was unsure of the exact orientation of the accelerometer within the Nexys A7 FPGA, thus accelerometer data along the $x$-, $y$- and $z$-axes had to be manually calibrated to suit the ideal motion of the player's avatar block via trial and error.

## Assembly code

### Register and Memory Layout

**$t9** : Current game state

**$t7** : ID of the previously flipped card

**$t6** ID of the currently flipped card

**$s7** Score (number of matched pairs)

**$k0** Number of lives used (mismatches)

**Memory 20–35** store each card's color value

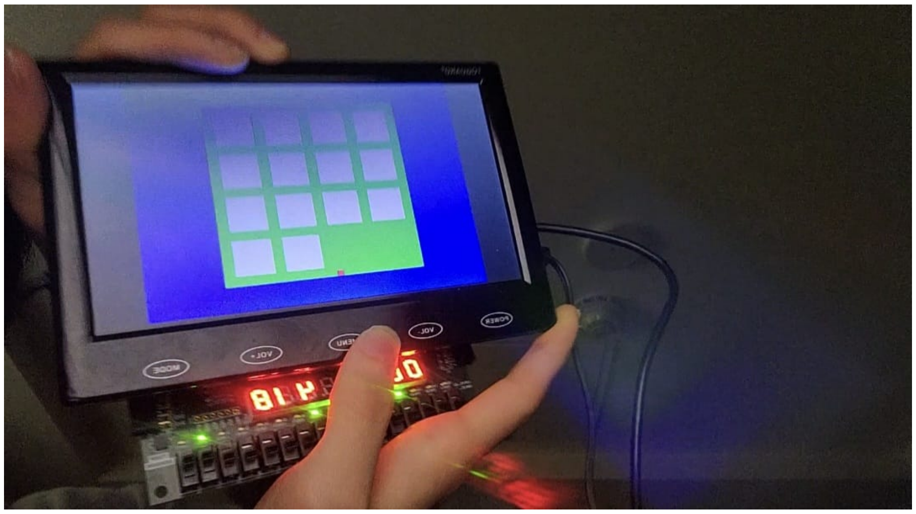
**Memory 40–55** store each card's $x$-coordinate (top-left vertex)

**Memory 60–75** store each card's $y$-coordinate (top-left vertex)

Essentially, we set up registers (game state, score, lives) and load two memory arrays: one for card colors and one for card coordinates.

**Card Detection**: In a continuous loop, we read register-mapped $x$- and $y$-coordinates of the player's avatar block and compare them to each card's memory mapped top-left vertex $x$- and $y$-coordinates, allowing us to determine which card, if any, the player is on. The gameflow FSM is as described above. The game halts when all cards are matched, or when all lives are lost.

**Pictures**



Two cards have been successfully matched and have disappeared from the grid. The player controls the red avatar block by rotating the VGA display with the FPGA