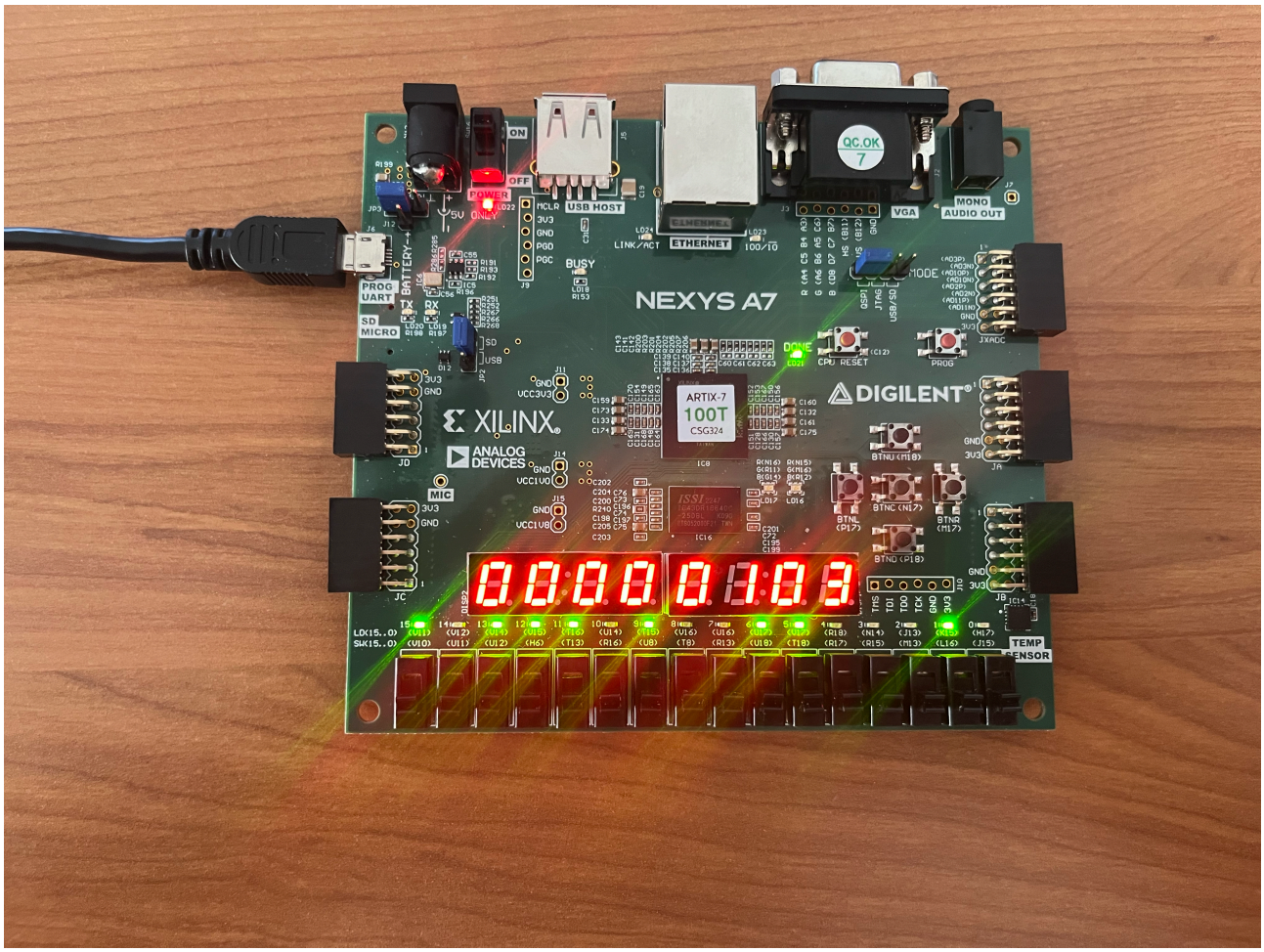
Logic Networks Project Report

MEMORY GAME REPORT



(FIGURE 1)

Giovanni Pagliani

18/12/2024

UNIVERSITÀ DEGLI STUDI DI TRENTO

Table of contents:

1. General Introduction

This project is a single-player memory game developed for the Nexys A7-100T FPGA board. It challenges the player's memory and reaction under increasing levels of difficulty, with interactive control via switches, buttons, and visual feedback through LEDs and 7-segment displays.

Basic Gameplay

The game begins when the center button is pressed. Upon successful entry, all LEDs blink briefly to confirm the start. The player enters Level 1, indicated by 00 on the 7-segment display.
At each turn:

- All LEDs are turned off.
- After a short delay, a set number of random LEDs are turned on (6 in Level 1).
- These LEDs stay lit for a few clock cycles (1 second) and then turn off.
- This is the cue for the player to replicate the pattern using the switches located above the LEDs.
- Once ready, the player presses the center button again to enter the evaluation phase.

Scoring System

- +1 point for each correct match (LED was ON and switch is ON)
- -1 point for each incorrect guess (LED was OFF but switch is ON)
- 0 points if the LED was ON and the switch is OFF (no penalty in Levels 1–2)

After scoring:

- The player must manually turn off all switches, until this is done the lights will be lit up as a visual notice.
- Once all switches are OFF, a new pattern can be generated, or some Additional Controls may be checked.

To improve interaction, the remaining directional buttons on the board have been assigned new functionality during the post-evaluation phase (i.e., when all LEDs are turned off as a transition state):

- Right Button: proceed to the next turn
- Top Button: display the current pattern on the 7-segment display
- Bottom Button: display the number of turns taken
- Left Button (long press): restart the game from Level 1

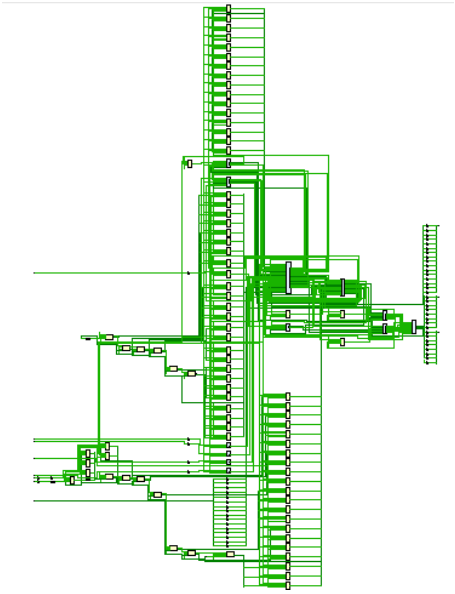
When the player reaches a score of 16, the game automatically proceeds to Level 2 (shown as 01). The flow is the same as Level 1, except that 8 LEDs are used instead of 6. The score resets to zero at the start of the level.
Upon reaching 16 points again, the player enters Level 3 (02). It also uses 8 LEDs and the same sequence of pattern display and evaluation. However, Level 3 introduces a stricter rule: the player is now penalized -1 if they forget to activate a switch corresponding to an ON LED.

Level 4 and 5 are structured to be slightly different, the behavior is the same of the second level (point wise), while introducing a countdown timer that forces the player to take faster decisions regarding the switch pattern he wants to send.

Once the player completes Level 5, the game enters a win state, where all LEDs display a wave-like animated pattern, celebrating the achievement.

2. Formal Description of the Project

Various components are needed for the game to function properly, each one of them interacts with the control unit, which coordinates everything.



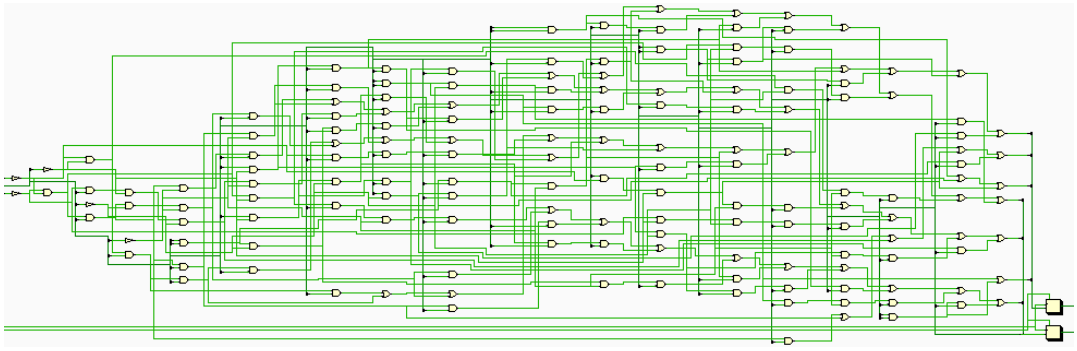
Debouncers

Used for each of our 5 button inputs. The purpose of a debouncer is to validate a button press, preventing errors and glitches. This is done by counting a minimum number of clock cycles for which the button must remain pressed. If this minimum duration is not reached, the button press is ignored, and not forwarded to the rest of the circuit. Each debouncer receives the button signal as “bouncy” (together with clock and reset signals) and outputs the validated stable pulse.

Binary to decimal converters

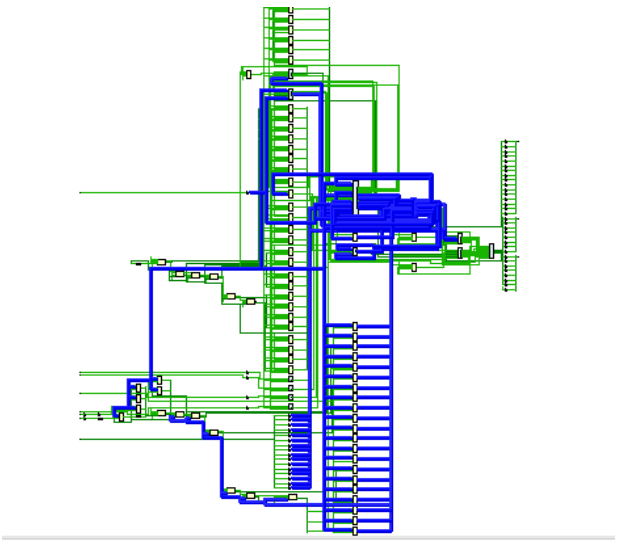
4 decoders are needed to go from 7-bit unsigned numbers to decimal digits. Inside this component, each combination of the seven segment is associated to a couple of 4-bit outputs, each one representing a decimal digits.

Snapshot of the internals of the converter:



The following components will be covered in more details later, but let's introduce:

Datapath

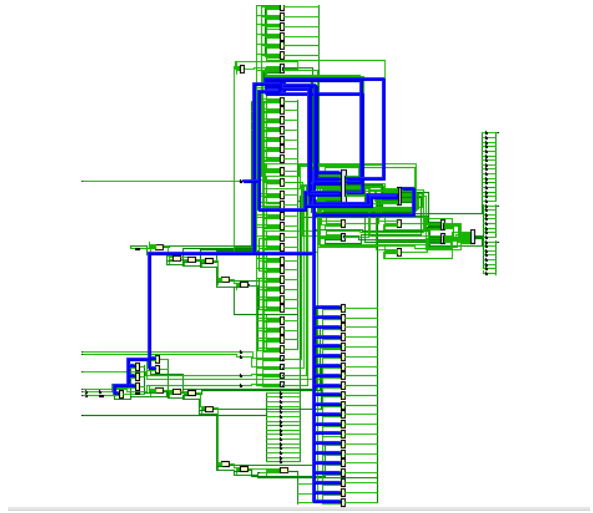


The datapath handles the core logic of the game's numerical state, including score calculation, turn counting, and level progression. It receives control signals from the control unit to determine when to evaluate the player's input and when to update internal registers. During the evaluation phase, the datapath compares the switch configuration provided by the player with the pattern stored internally, applying the game's scoring rules. The updated score is then made available to the display system. It also manages the turn counter and the timer, both of which are used by the control unit to decide the next state of the game.

Control Unit

The control unit is the brain of the game. It manages the game's finite state machine (FSM), orchestrating transitions between various phases: initialization, pattern generation, waiting for player input, evaluation, transition between turns, and level advancement. It listens to debounced button signals and player actions, enabling and disabling other components based on the current game state. It generates control signals such as pattern_enable, score_enable, and turns_enable, selects what to show on the display, and sets the LED behavior (including the final wave animation at the win state). It is also responsible for enforcing game rules across levels and reacting to special button actions like restart or checking level/turn count.

Pattern Generator



The pattern generator is responsible for producing the random sequence of active LEDs at the beginning of each turn. It uses the Fisher–Yates shuffle algorithm to select a given number of unique positions in a 16-bit vector and set those bits to 1. The number of active LEDs depends on the current level (6 in Level 1, 8 in Levels 2 and 3). To improve randomness we use the number of clock cycles between pattern display and the center button press as a source of entropy to vary the seed.

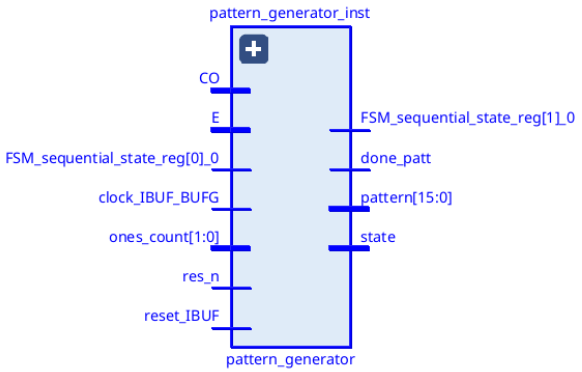
7-Segment Display Driver

The 7-segment display driver controls the two onboard displays of the Nexys A7 by managing the output of four BCD digits (two per display). It receives the converted BCD signals from the binary_to_bcd module, which represent values such as the current score, level, turn count, or timer depending on the current game phase. It converts each 4-bit BCD digit into the corresponding 7-segment pattern, driving the correct segments to display numbers from 0 to 9. The control unit selects which value (level, score, turns, timer) should be shown by enabling or disabling display modes accordingly.

3. Pattern Generator

The pattern generator creates a 16-bit vector (pattern) with a specific number of 1s in it. The number of 1s is selected through the input signal ones_count.

Instead of placing the 1s in fixed positions, the module shuffles their positions randomly, so each time the result looks different, even with the same input.



How It Creates a Random Pattern

The logic is based on a well-known method called the **Fisher–Yates shuffle**, also known as the **Knuth shuffle**. This is a way to take an ordered list and mix it up randomly and fairly.

Here's how it works in our case:

- First, we create a pattern with exactly the requested number of 1s at the beginning => (e.g., 1111110000000000 for six 1s).
- Then, we shuffle the bits in the pattern so that the 1s end up in random positions.

This way, the pattern always has the right number of 1s, but their positions are different every time.
To decide which bits to swap during the shuffle, we use a Linear Feedback Shift Register (LFSR). An LFSR is a small circuit that creates a long sequence of pseudo-random bits using a mathematical formula based on the polynomial:

$$x^{16} + x^{14} + 1$$

In VHDL, that looks like this:

```
lfsr <= lfsr(14 downto 0) & (lfsr(15) XOR lfsr(13));
```

Making It Actually Random

LFSRs are only useful if they start with a different value each time. If you always reset the LFSR to "111...1", it will always generate the same shuffle sequence.

To fix this, we added a free-running counter that increases constantly with each clock cycle. Every time a new pattern is requested (when enable = '1'), the current value of this counter is copied into the LFSR as the seed.
This way, even if the player presses the button at a slightly different time, the seed changes, and so does the random pattern.

States of the Pattern Generator

The module works as a simple state machine, going through the following steps:

idle	Waits for enable = '1' to start
load	Builds the base pattern with the right number of 1s
shuffle	Mixes up the bit positions using the LFSR
done_state	Sets the done signal to '1' when the pattern is ready

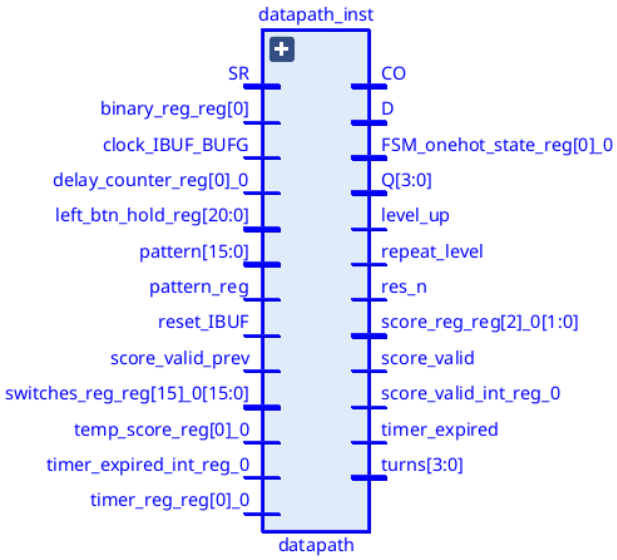
Obviously You can trigger a new pattern only after enable goes back to ‘0’.

Conclusion on Pattern Generator

The pattern generator reliably produces 16-bit vectors containing exactly the number of 1s specified by the ones_count input. To ensure randomness, it uses a variation of the Fisher–Yates shuffle algorithm, which rearranges the bits without altering the count of active bits. The randomness of the shuffle is driven by a 16-bit Linear Feedback Shift Register (LFSR), seeded at runtime with a free-running counter. This seeding method ensures that the patterns vary from one generation to the next, even if the system is reset, since the counter introduces time-based variability

4. The Datapath

This datapath module implements the core data logic for a memory game. It performs per-bit comparisons between the current pattern and the switch inputs, updates the game score and turn counter, and manages a countdown timer used in higher levels (level 4 and 5).



Block 1: Per-bit Pattern Comparison Logic

```
if pattern_reg(bit_index) = '1' and switches_reg(bit_index) = '1' then
      temp_score <= temp_score + 1;
elsif pattern_reg(bit_index) = '0' and switches_reg(bit_index) = '1' then
      temp_score <= temp_score - 1
elsif pattern_reg(bit_index) = '1' and switches_reg(bit_index) = '0' and level = "010" then
      temp_score <= temp_score - 1;
end if;

if bit_index = 15 then
      state <= score_calculated;
else
      bit_index <= bit_index + 1;
end if;
```

This logic compares each bit of the stored pattern and the user's switches. The score is adjusted accordingly: a correct match increases the score, while a mismatch may decrease it. In level 3 (level = "010"), missing an active bit incurs an extra penalty.

Block 2: Countdown Timer Implementation

This block handles the countdown from 10 to 0, activated when timer_enable is high, and resetted when is low. Every second (based on DELAY_CYCLES), timer_reg decrements. When it reaches 1, the module sets timer_expired_int, which signals the control unit to evaluate the player's input.

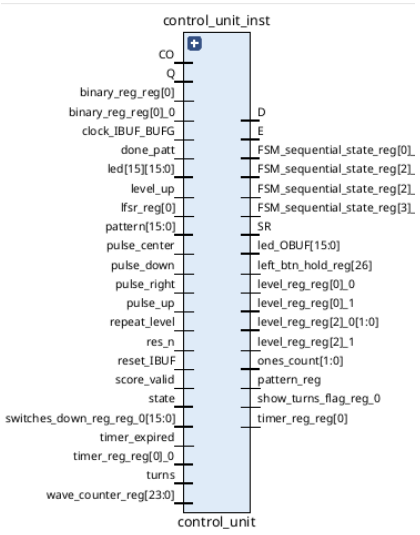
```
if timer_enable = '1' then
   if timer_reg = 0 then
      timer_expired_int <= '0';
   elsif delay_counter = DELAY_CYCLES then
      delay_counter <= 0;
      timer_reg <= timer_reg - 1;

      if timer_reg = 1 then
         timer_expired_int <= '1';
      else
         timer_expired_int <= '0';
      end if;

    else
          delay_counter <= delay_counter + 1;
          timer_expired_int <= '0';
    end if;

else
    if level = "011" then
      timer_reg <= 10;
    else
      timer_reg <= 5;
    end if;
      delay_counter <= 0;
      timer_expired_int <= '0';
 end if;
```

Conclusion

The datapath effectively isolates the computational logic of the game. It maintains game state consistency and supports both regular and timed gameplay modes, providing critical outputs like score_valid, level_up, and timer_expired.

5. The Control Unit

Your control_unit manages the full flow of the memory game using a finite state machine. It responds to user inputs (buttons and switches), drives the pattern generation process, controls LED and timer logic, and manages transitions between levels — including the addition of timed levels (level 4 and 5) and a manual reset through button hold.



A. Menu Interaction (Up/Down/Right)

```
-- in clocked process:
 if state = menu_actions then
    if up_btn = '1' then
      show_pattern_flag <= '1';
      show_turns_flag   <= '0';
    elsif down_btn = '1' then
      show_turns_flag <= '1';
      show_pattern_flag <= '0';
    elsif right_btn = '1' then
      show_pattern_flag <= '0';
      show_turns_flag   <= '0';
    end if;
  end if;

      (...)
--in fsm:
when menu_actions =>
  led_mode <= "00";

  if show_pattern_flag = '1' then
     led_mode <= "01";
  end if;
  if show_turns_flag = '1' then
     turns_enable <= '1';
  end if;
  if right_btn = '1' then
     next_state <= delay_after_clear
  end if;
```

In the menu_actions state, users can toggle between displaying the pattern again or the number of turns using the up/down buttons. Right button cancels the selection.

In levels 4 and 5, the timer is automatically started in wait_for_btn_eval. If the player doesn't press the center button in time, timer_expired triggers an automatic evaluation of their input.

State Table

State	Description	Key Input/Output
idle	Waits for the player to start the game.	**In**: center_btn → transitions to blink_all
blink_all	Briefly turns on all LEDs at the beginning of the	**Out**: led_mode = "10", uses delay_counter
generate_pattern	Triggers the pattern generator.	**Out**: enable_patt = '1'
wait_pattern_done	Waits for the pattern generator to finish.	**In**: done_patt = '1'
show_pattern_once	Displays the generated pattern once.	**Out**: led_mode = "01", uses delay_counter

wait_for_btn_eval	Waits for center button or timer to expire before evaluating switches.	**In**: center_btn, timer_expired **Out**: start_compare, timer_enable
wait_score	Waits for the datapath to validate the score.	**In**: score_valid (registered)
wait_switches_down	Waits for all switches to be lowered before	**In**: all_switches_down **Out**: led_mode = "10"
menu_actions	Menu mode: allows re-showing pattern or turns.	**In**: up_btn, down_btn, right_btn **Out**: led_mode, turns_enable
delay_after_clear	Short delay after menu before checking score.	Uses delay_counter
check_score	Decides whether to level up, retry, or end game.	**In**: level_up, repeat_level (registered)
next_level	Moves to next level, restarts sequence.	Transitions to blink_all
end_state	Final state when game is completed successfully.	**Out**: led_mode = "11" with wave animation

Conclusion on Control Unit

This control unit is tailored precisely to the game's behavior, incorporating level logic, timers, display modes, and user-driven resets. It smoothly coordinates with the datapath and other modules, handling gameplay complexity while keeping FSM logic readable and modular. The integration of a manual reset and timed level progression makes it flexible and player-friendly.

6. Results

The memory game was successfully implemented and simulated using VHDL on the Nexys A7 FPGA board. The core functionalities, including pattern generation, input comparison, score tracking, level progression, and LED feedback, behaved as expected. The system correctly responded to user inputs, generated randomized LED patterns, and updated the score and turns across all levels. The transition to timed levels (Levels 4 and 5) functioned properly, with the timer starting at the correct moment and triggering automatic evaluation when expired. The manual reset feature via long press on the left button was integrated and verified. Finally, reaching the end state correctly activated the wave-like LED animation. Overall, the game meets all specified design goals.

7. Credits

Code images have been generated using Carbon.