

## Minesweeper Project

Alexis Le Glaunec (afl5), Agha Sukerim (ads13), Liam Waite (lcw6)

### 1-paragraph:

Minesweeper is a chip implementation of the [Minesweeper game](#), in which the goal is to clear the board without detonating mines. Initially, 1-3 mines are placed using a pseudo-random algorithm in a 3x3 grid. Then, the user can input a position by choosing a number between 1 and 9 on a [keypad switch](#). The chip will respond with a color on a LED dot matrix screen encoding the following information: red if it hits a mine (and the game ends), green if there is no mine among the 3-8 neighbors of the position and yellow if there is at least one mine in the position neighbors. If the position is yellow, a digital screen displays how many mines are adjacent. The initially unlit dot matrix screen will record and color in each position that is visited. The player wins if it clears all cells but the ones with mines. The chip will have the ability to remember the score, support game over and restart the game.

## Overview

In this project, we propose an implementation of the Minesweeper game, where a player tries to clear the 3x3 board without detonating mines. The user input `data_in` is a 4-bit signal to encode all of the board cells (ranging from 0 to 8 in a row-first arrangement).

## Registers

Our design requires 3 internal registers: three 4-bit registers of `mines` to store the mines' positions, with it being 0000 if a mine is not used. A 9-bit register called `cleared` is used to store the cells that have been cleared by the user (where “1” at position  $k$  means cell  $k$  is `cleared`), and a 4-bit register `score` is used to store the current score (i.e. number of cells cleared in the game). A 32-bit register `total_score` is used to store the cumulative score throughout all of the games. In total, we would need  $4 \times 3 + 9 + 4 + 32 = 57$  bits of memory and 6 registers.

## Finite State Machine and ALU

At every step, the user inputs a 4-bit value and we compare it against the mine position by performing a bitwise-and operation (`data_in & mine` for all mines). If  $(\text{data\_in} \& \text{mine}) \neq 0$ , then the game is over (as displayed on the screen) and the user can restart the game with a `rst` signal (in practice, a button). Otherwise, we record the cleared cell by updating the cleared register. Since `cleared` stores each position by bit while `data_in` stores the numbered position in binary, we will have to first convert `data_in` with a decoder (e.g. `data_in` = 0011 -> `data_inConverted` = 000000100). We can then record the cleared cell by updating the cleared register as follows: `cleared := cleared | data_in`. We also increment both the `score` and `total_score`. Then, the game goes on. If `cleared` is all-ones, the game is over, the user wins, and again the user can decide to start a new game using the reset button. Mine positions are automatically recorded in the `cleared` register for this to work. When the game is over, we reset all the registers except for `total_score`. In practice, implementing those operations requires the ALU to contain an increment unit, a 9-bit bitwise-AND, a 9-bit bitwise-OR, and a decoder.

## I/O

The design would require 5 input pins for the 4-bit user input signal and the `restart` signal. The outputs include a 3x3 LED matrix and an I<sup>2</sup>C display. The LED matrix would only need 6 outputs and multiplexing to control each row and column in order to light specific LEDs. For the display that outputs the number of nearby mines, it would only require 2 pins. In total, we expect to use 12 I/O pins.

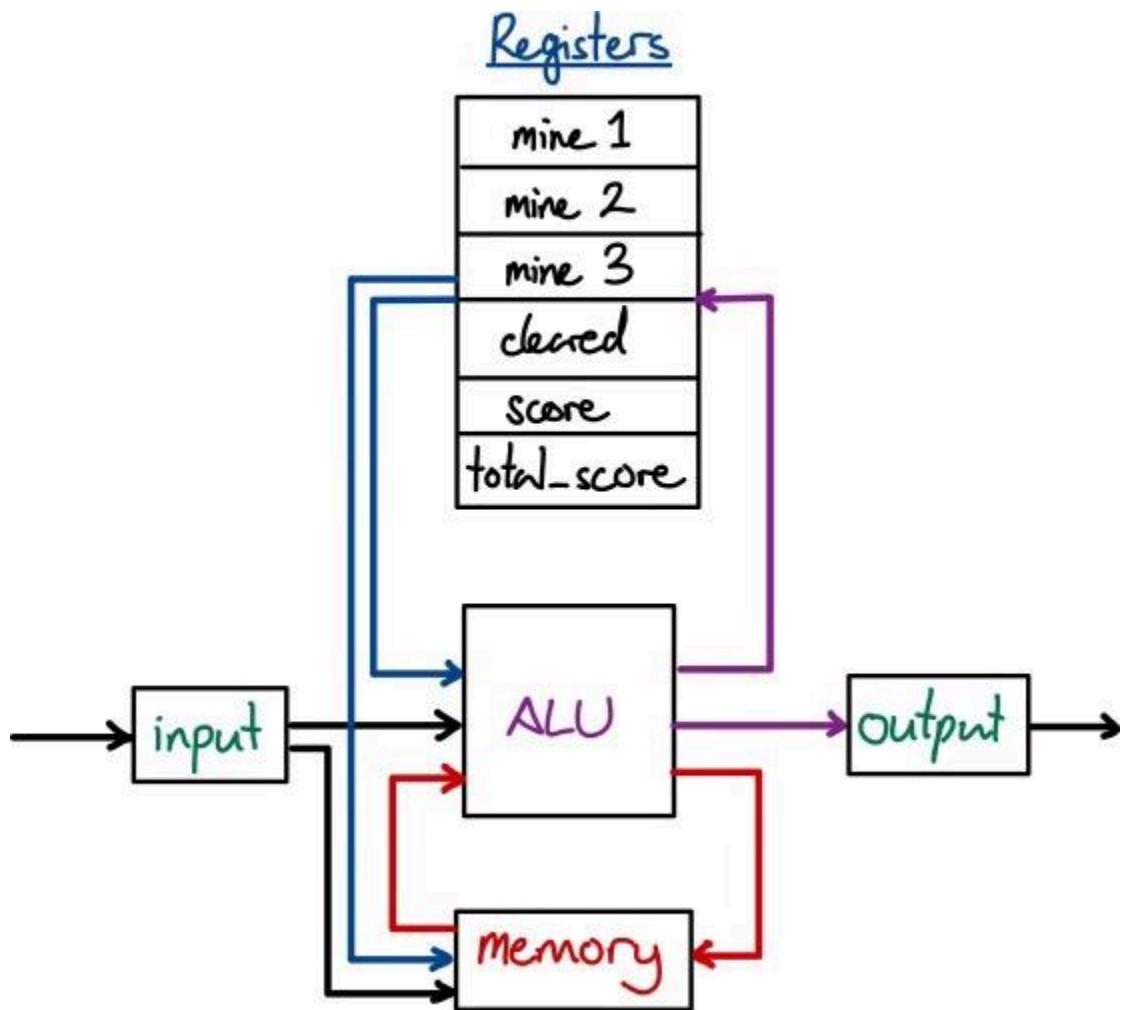
## Scalability

The game is by nature scalable; we can increase the board size which will result in larger registers and larger user input word size to fully utilize the chip area. There will be different levels consisting of a varying number of mines at the start of the game.

## Testing

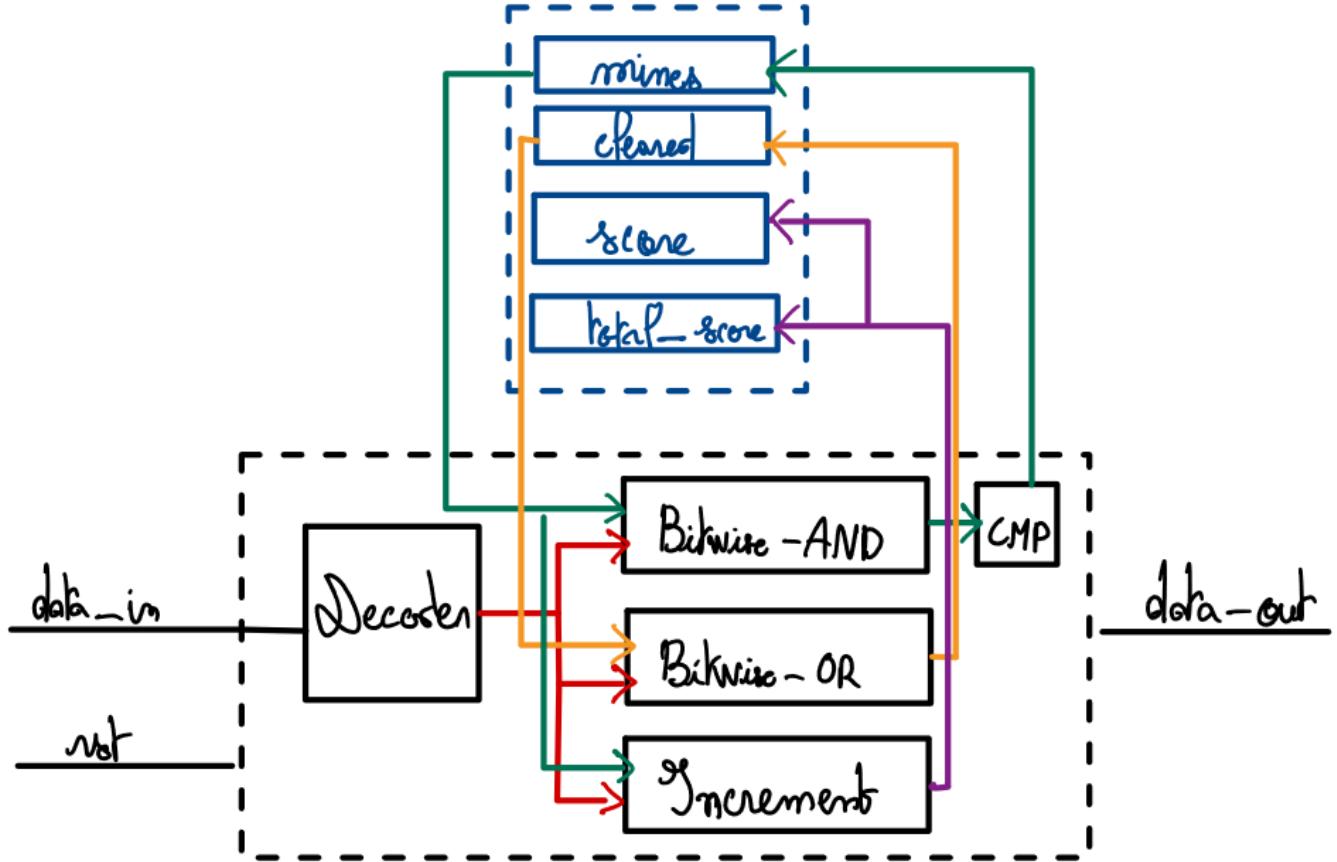
We intend to test our design at each step with testbenches simulating games with a series of user actions. In addition, we will incorporate SVA to catch any inconsistencies. We will unit test separately the components of the ALU.

### Intermediary Block Diagram (see below for complete)

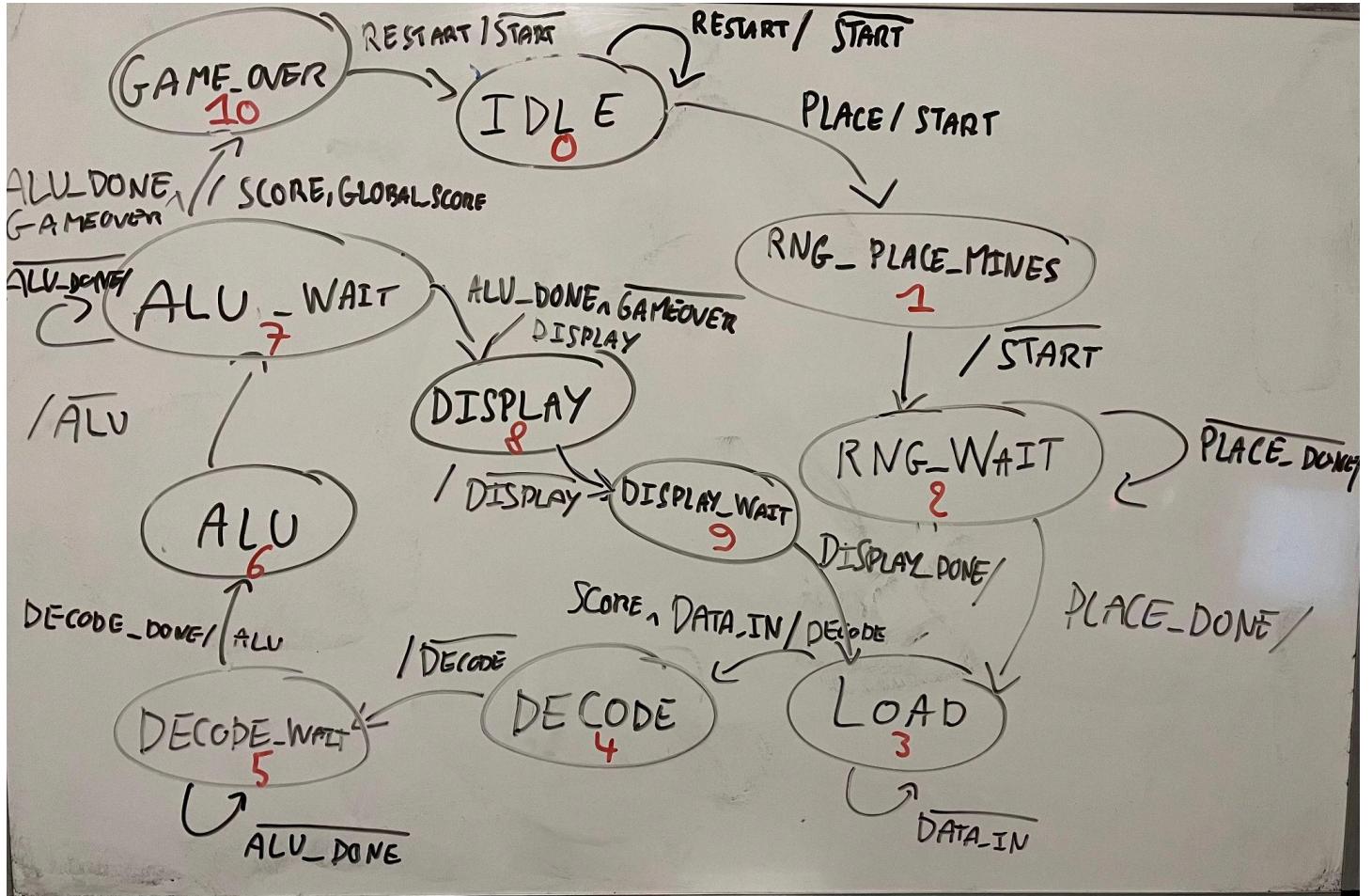


## Schematics of the ALU

This diagram shows more details about the inside of the ALU, and how it would interconnect with the registers.



## Bubble Diagram



In this diagram, we call four distinct data paths:

1. RNG: random number generator to place the mines (number depending on the level chosen)
2. DECODE: decode the user input into a bitmask
3. ALU: perform the arithmetic operations (bitwise-and, bitwise-or, increment)
4. DISPLAY: display the cleared cells and indicate how many mines are nearby the last cleared position

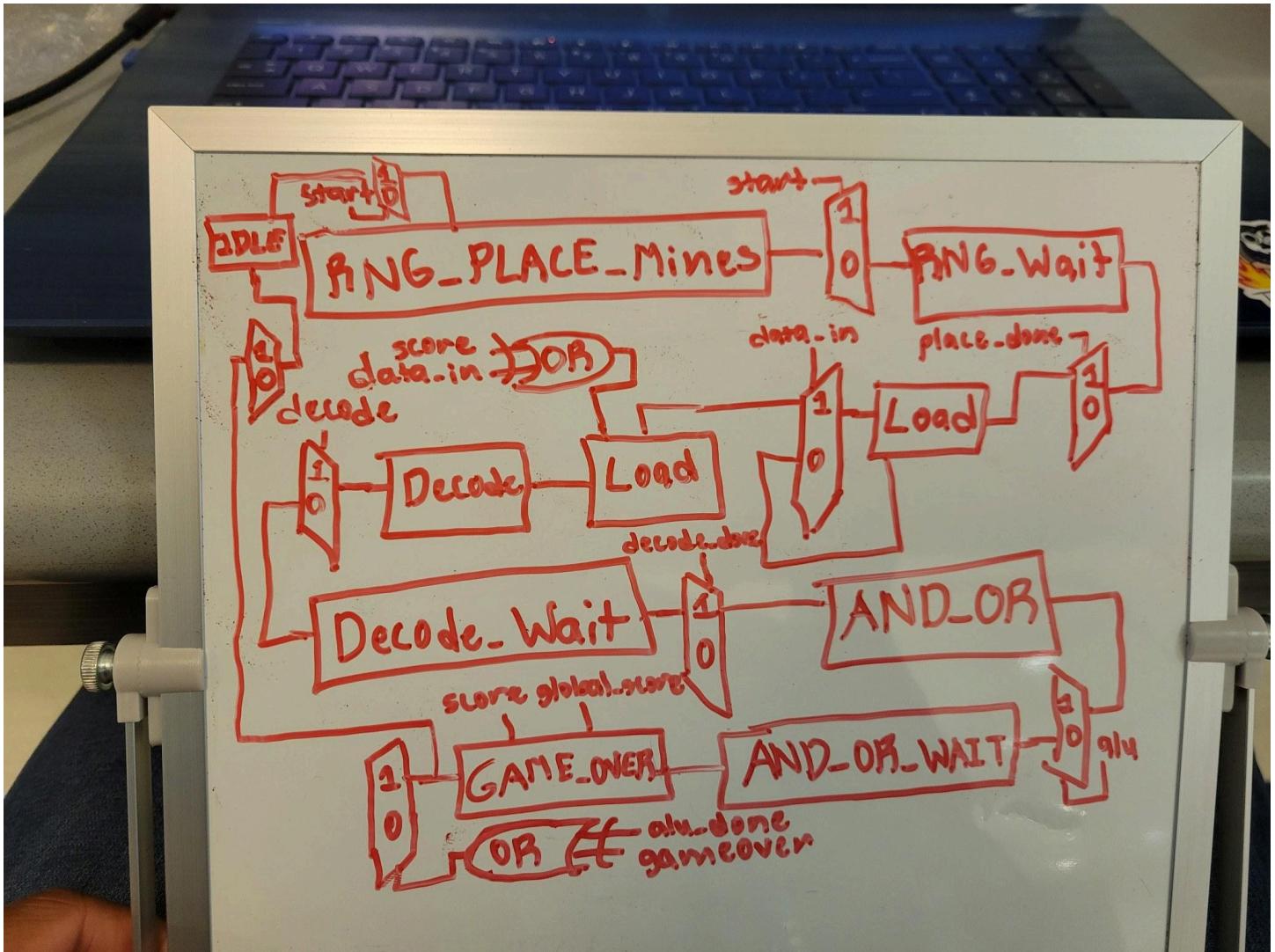
For the ALU, we would use a second FSM consisting of 3 stages:

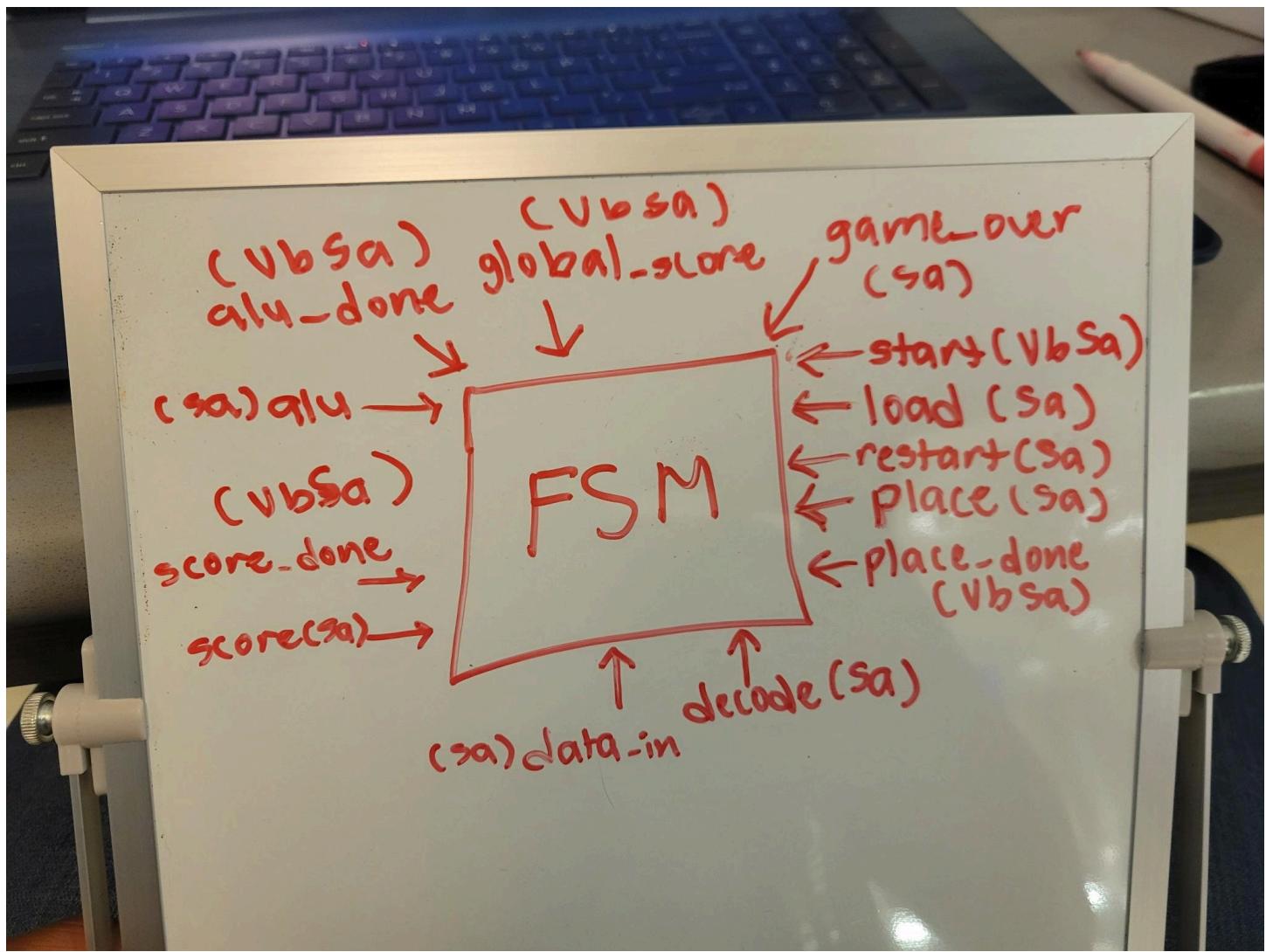
1. Bitwise-and to compute `data_in & mine` (to know if the user detonated a mine)
2. Bitwise-or to keep track of the cleared cells
3. Increment to increment the score (based on number of cleared cells)

Each step consists of 1 state. So, in total (main\_FSM + ALU\_FSM), we end up with 13 states.

## Block Diagrams

We have designed the block diagrams for the ALU and the FSM:





## FSM Pseudo-Code

1. module Minesweeper(
  - 1.1. input wire [3:0] data\_in, //4-bit input for user cell selection
  - 1.2. input wire [3:0] reset, // RESET signal
  - 1.3. output wire [5:0] led\_matrix, // Output for 3x3 LED matrix
  - 1.4. output wire [1:0] nearby,
  - 1.5. output wire [4:0] score ); // Output for I2C display
2. Internal Registers
  - 2.1. reg [3:0] mine1, mine2, mine3
  - 2.2. reg [8:0] cleared
  - 2.3. reg [3:0] score
  - 2.4. reg [31:0] total\_score
3. Internal Constants
  - 3.1. parameter IDLE = 0
  - 3.2. parameter RNG\_PLACE\_MINES = 1
  - 3.3. parameter RNG\_WAIT = 2

- 3.4. parameter LOAD = 3
  - 3.5. parameter DECODE = 4
  - 3.6. parameter DECODE\_WAIT = 5
  - 3.7. parameter AND\_OR = 6
  - 3.8. parameter AND\_OR\_WAIT = 7
  - 3.9. parameter GAME\_OVER = 8
  - 3.10. parameter WIN = 9
4. Internal Variables
- 4.1. reg [2:0] state
  - 4.2. reg [2:0] next\_state
5. Decoder
- 5.1. wire [8:0] data\_inConverted
  - 5.2. Decoder4to9 decoder(data\_in,data\_inConverted)
6. ALU Operations
- 6.1. wire mine\_detected //decides if game is over
  - 6.2. assign mine\_detected = *compare data\_in and all mine1, mine2, mine3, assigning HIGH if any AND is HIGH*
7. Random Mine Placement Generator
- 7.1. Simple algorithm to generate three 4-bit positions for mine1, mine2, mine3
  - 7.2. Check that mine1 != mine2 != mine3, or do again
  - 7.3. output place\_done
8. FSM Logic
- 8.1. *High 'reset' input all return to IDLE*
  - 8.2. IDLE
    - 8.2.1. If (~reset): next\_state = RNG\_PLACE\_MINES
  - 8.3. RNG\_PLACE\_MINES
    - 8.3.1. Start Random Mine Placement Generator
    - 8.3.2. next\_state = RNG\_WAIT
  - 8.4. RNG\_WAIT
    - 8.4.1. If (~place\_done): next\_state = LOAD
  - 8.5. LOAD
    - 8.5.1. If (mine\_detected): next\_state = GAME\_OVER
    - 8.5.2. If (~mine\_detected): next\_state = DECODE
  - 8.6. DECODE
    - 8.6.1. To decode data\_in to data\_inConverted
    - 8.6.2. If (~mine\_detected): next\_state = DECODE\_WAIT
  - 8.7. DECODE\_WAIT
    - 8.7.1. If data\_inConverted is new, next\_state = AND\_OR
  - 8.8. AND\_OR
    - 8.8.1. Update 'cleared' with 'data\_inConverted' (cleared := cleared | data\_in)
    - 8.8.2. If (cleared == 9b1), next\_state = WIN
    - 8.8.3. If not, next\_state = LOAD
  - 8.9. GAME\_OVER

- 8.9.1. If (reset): next\_state = IDLE  
 8.10. WIN  
 8.10.1. If (reset): next\_state = IDLE
9. Decoder4to9  
 9.1. Function/module to convert 4-bit data\_in position to 9-bit position

```
// FSM Logic
always @(posedge rst or posedge data_in) begin
  if (rst) begin
    // Reset logic, except total_score
    state <= IDLE;
    cleared <= 0;
    score <= 0;
    // Mines should be randomly assigned in a practical implementation
  end else begin
    case(state)
      IDLE: next_state <= CHECK_MINE;
      CHECK_MINE: begin
        if (mine_detected) next_state <= GAME_OVER;
        else next_state <= UPDATE_SCORE;
      end
      UPDATE_SCORE: begin
        if (cleared == 9'b111111111) next_state <= WIN;
        else next_state <= IDLE;
      end
      GAME_OVER: begin
        // Reset all except total_score
        next_state <= IDLE;
      end
      WIN: begin
        // Increment total_score and reset for a new game
        total_score <= total_score + score;
        next_state <= IDLE;
      end
    endcase
  end
end

// Sequential Logic for state transition
always @(posedge clk) begin
  state <= next_state;
  case(state)
    CHECK_MINE: begin
      // Game over logic
    end
    UPDATE_SCORE: begin
      cleared <= cleared | data_inConverted;
    end
  endcase
end
```

```
score <= score + 1;
total_score <= total_score + 1;
end
GAME_OVER: begin
    // Game over logic
end
WIN: begin
    // Win logic
end
endcase
end

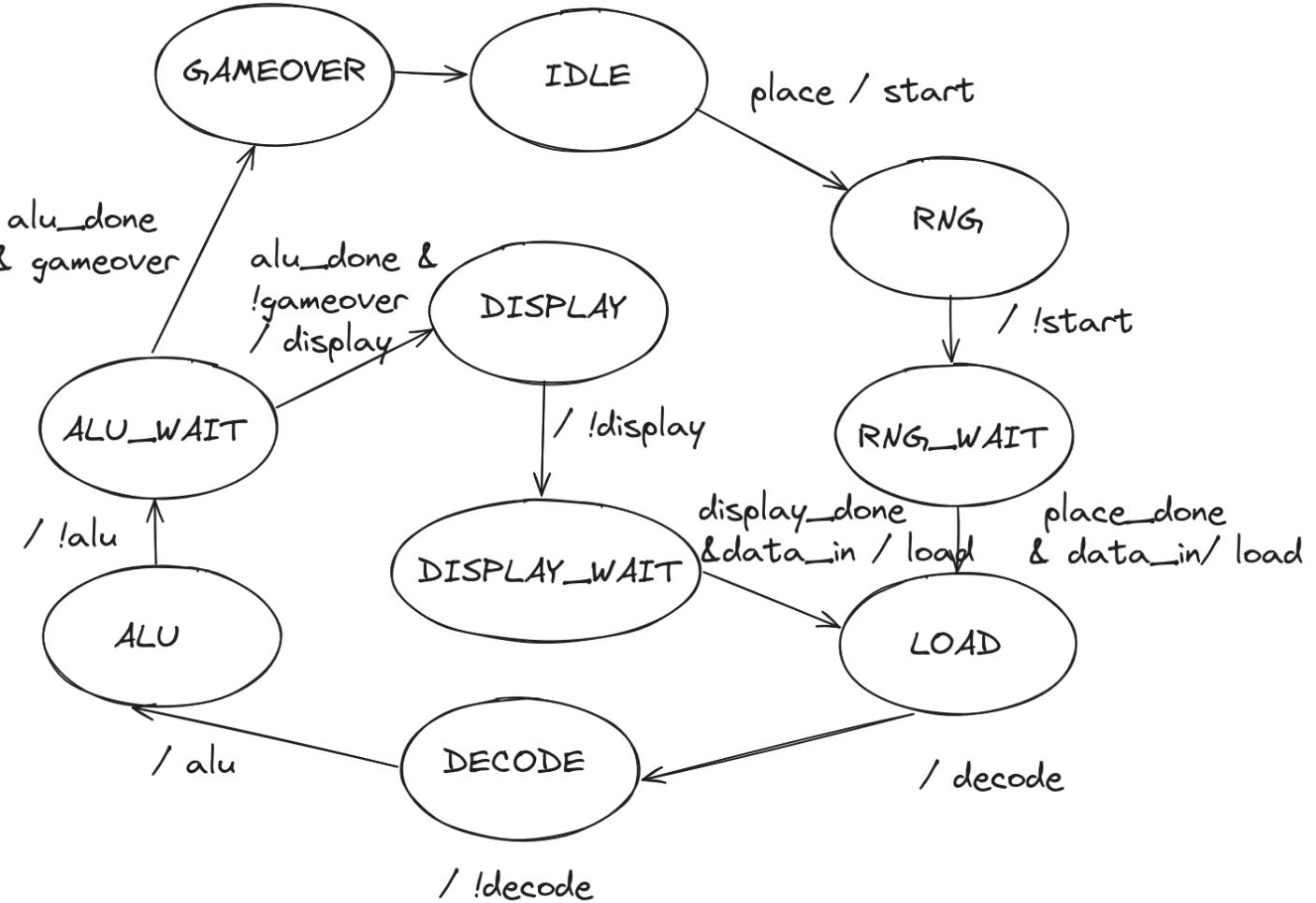
// Output Logic (simplified for pseudocode)
// Actual implementation would control the LED matrix and I2C display

endmodule

module Decoder4to9(input [3:0] in, output reg [8:0] out);
    always @ (in) begin
        out = 1 << in;
    end
endmodule
```

## Controller FSM

Bubble diagram:



Please note that in order to keep the bubble diagram readable, we do not represent self-loops. Those can be deduced to be the default edge taken if no other is available.

## Data Path Files

We consider two modules for the data path: the main *dp.v* file to handle the main functionalities of the data path and the random number generator *rng.v* that supplements the main *dp.v* file.

The main datapath (*dp.v*) focuses on all of the game logic relating to user inputs and updating the board. The second clock phase is reserved for done flags for the FSM. In *dp.v*, we implemented the following functionalities:

- Decode the input data into a 1-dimensional indexing of the cells on the 5x5 board
  - Into same format as how *mines* are stored
- Calculate the number of neighboring mines (*n\_nearby*) at a given user's chosen cell
  - Case statement
- Update the board at each turn
  - Record each chosen position on *temp\_cleared*
- Determine if the user has exploded the mine
  - *gameover* register
- Determine if the user has won the game by clearing all cells except for the mines
  - Compare *mines* bit array with decoded data input

To randomly generate mine placements, a separate datapath is used (*rng.v*), outputting the mine positions, *mines*, that are subsequently provided to *dp.v*. It uses the linear congruential generator (LCG) formula to randomly set a given number of bits in the 25-bit *mines* array. The values for the multiplier and increment will be user-provided.

Linear Congruential Formula:  $X_{n+1} = (aX_n + c) \bmod m$

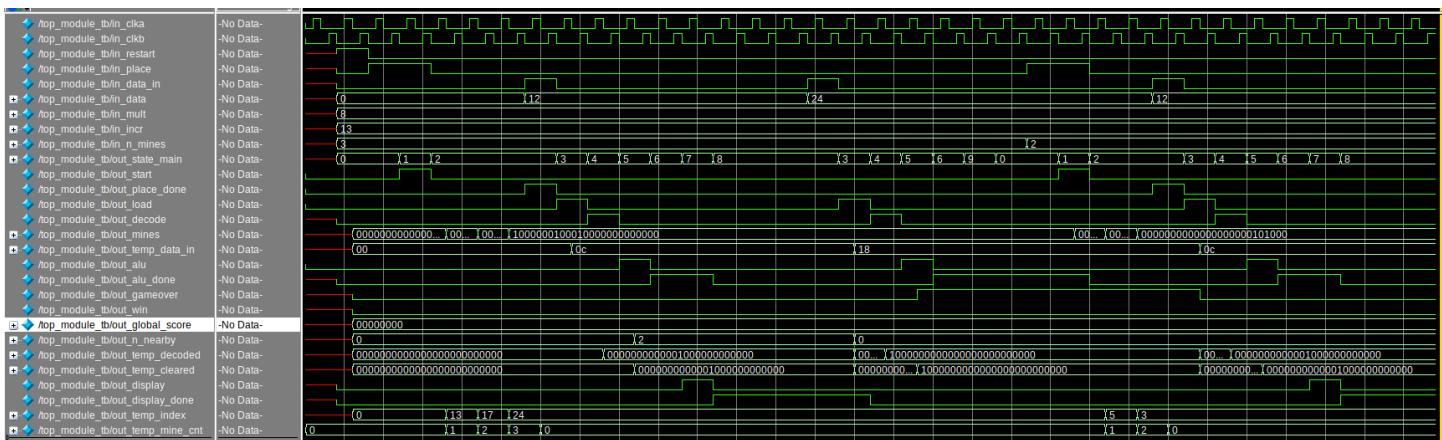
## Testbench

Our testbench consists for now in two rounds of the minesweeper game:

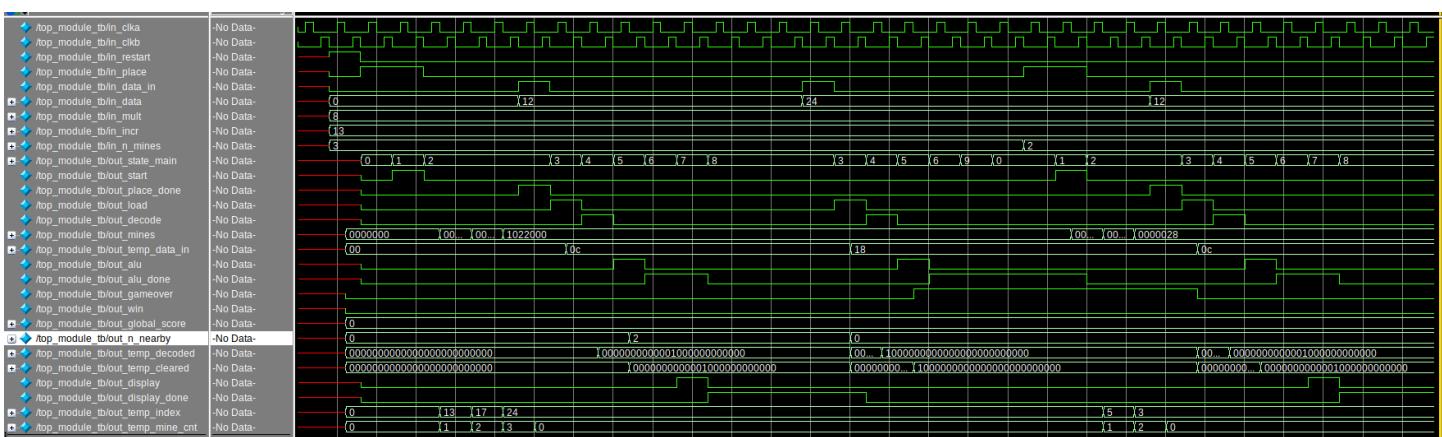
1. At cycle 2, we restart the FSM with a level difficulty of 3 mines
2. Between cycles 3 and 7, we generate the mine using RNG (handshake with *rng.v*)
3. From cycle 8 to cycle 16, we run one iteration of the game, loading the user input (clearing cell 12) up to the display of the board. As there is no mine on cell 12, we continue the game
4. From cycle 17 to cycle 24, we run the second iteration of the game, loading the user input (clearing cell 24). As there is a mine on cell 24, we go to gameover state.
5. We start a new game for a level difficulty of 2 mines
6. From cycle 28 to 36, we clear cell 12.

This testbench is a proof-of-concept for our design. We will extend the testbench from 6. until we win (by clearing all the 23 non-mine cells) to show an example of victory. Right now, the score corresponds to the number of victories since the last restart.

Questa:

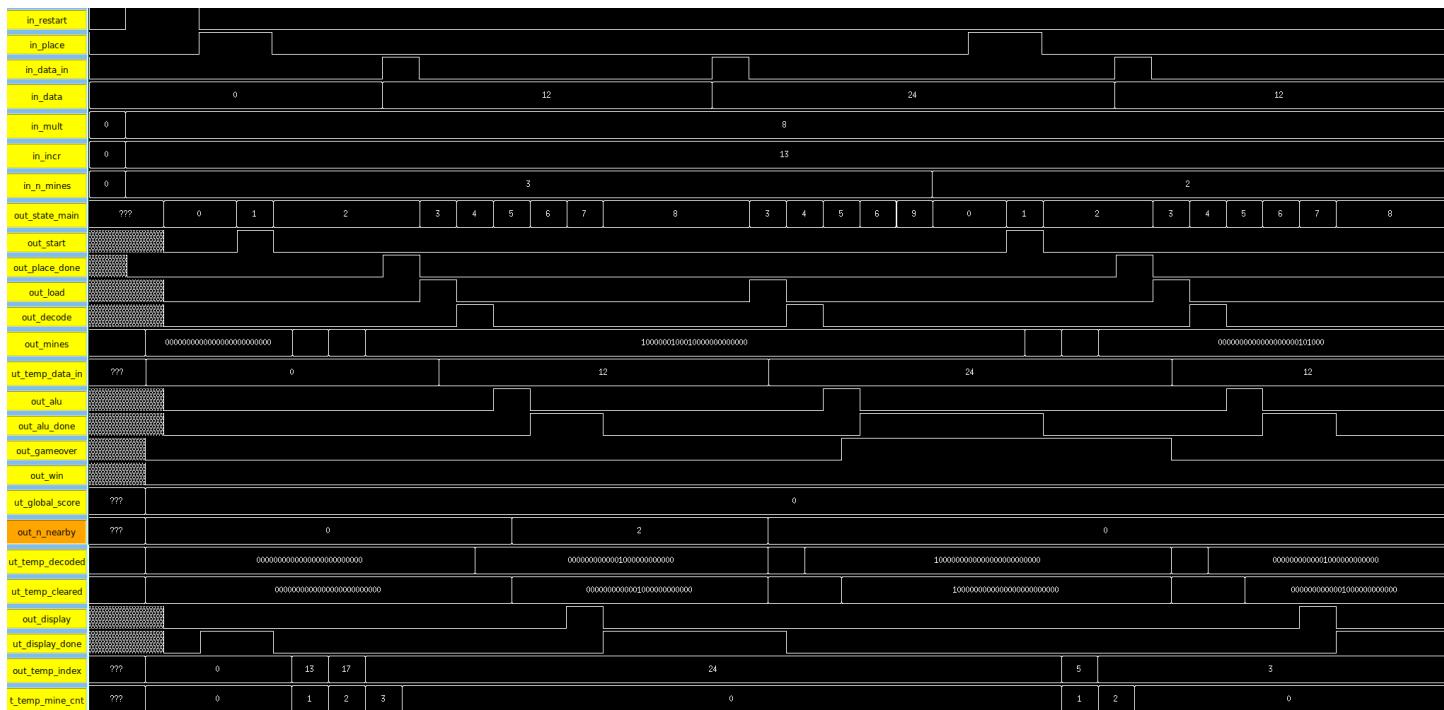


## Questa post DC:



We observe no difference in the testbench behavior between before and after synthesis. The n\_nearby works as expected and gameover is triggered only when the user chooses a cell with a mine.

Irsim:



In Irsim, we need to add one more cycle at restart to give enough time to reset the flipflops. Irsim result is consistent with the Questa results.

Magic:

Our chip design on Magic is about 800x800 lambda, which should make it fit with standard padframe.

