echessboard

Alessia Gelmini, Benjamin Schmid Ties

August 2, 2025

1 Overview

We developed the project ”echessboard” for the course Advanced logic design by Prof. Roberto Passerone at the University of Trento. The goal of the project was to design a printed circuit board that functions as a digital chessboard. The current chess position is read out by the hardware on the PCB, and sent to an FPGA using the SPI bus. The data is decoded by dedicated logic on the FPGA. The FPGA design also contains a softcore RISC-V microprocessor running a C program that renders a visualization of the current chess position. Finally, the rendered image is displayed on a screen using the VGA interface.

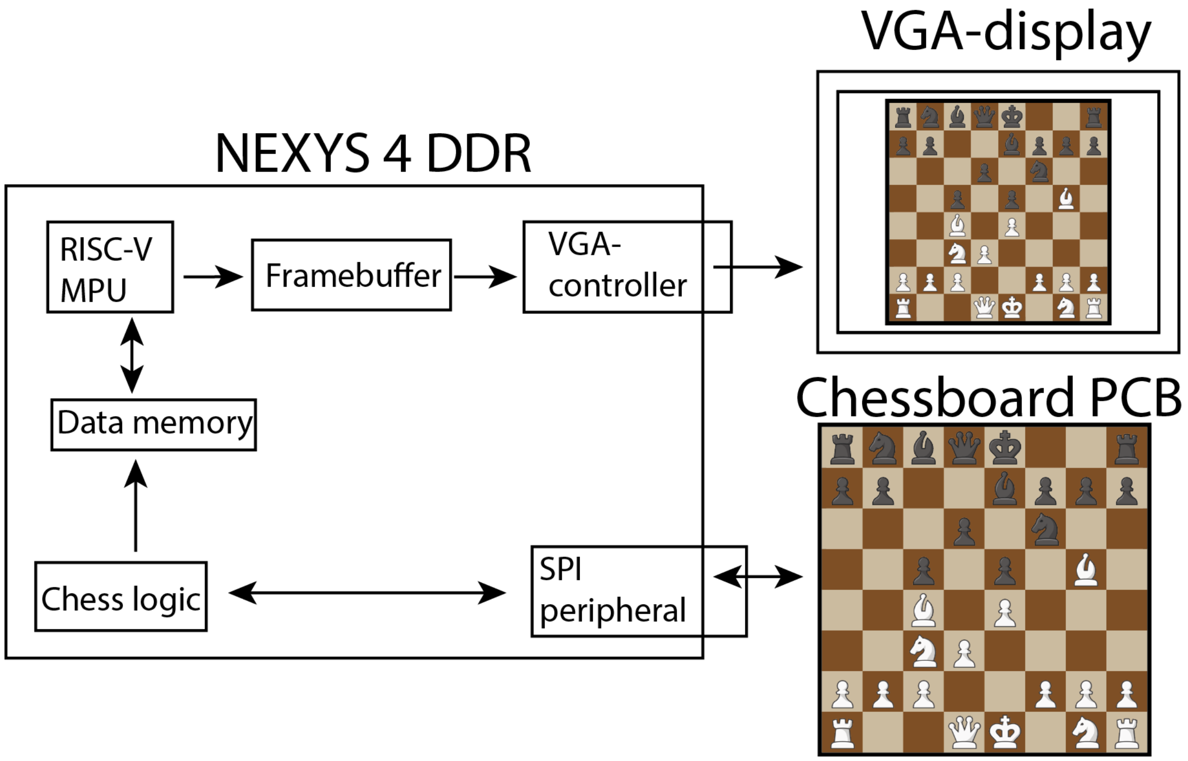


Figure 1: Block diagram of echessboard project

2 Chessboard PCB

The design for the digital chessboard PCB turned out to be relatively simple. Underneath every field of the chessboard, a hall sensor is placed. Chess pieces were 3d-printed with a cutout, in which a magnet was placed. The hall sensors read a logical 0 when a piece is present on a field and 1 when the field is empty. In total, there are 64 hall sensors. To read out the hall sensors, four 16-pin I2C based IO expanders were used. The hall sensors are connected column-wise to the IO expanders. The first

2 columns are connected to the first IO expander, the second 2 columns are connected to the second IO expander, and so on. The IO expanders have two separate ports, so every column is connected to a single port. The bits 0-7 of each port are connected to hall sensors on fields with row coordinate 1-8. The IO expanders are read out by an STM32 microcontroller. The data obtained is an array of eight bytes, one byte for every column. The data is sent via an SPI bus to the FPGA. The chessboard PCB also contains some buttons and LEDs for user interactions. The minimum number of controls was used while maintaining full functionality. The buttons are:

- Start/stop game
- Promotion (allows choosing piece type that a pawn promotes to)
- Confirm move

The LEDs are:

- Game ongoing
- Winner: white, black, draw
- Promotion: queen, knight, rook, bishop
- To move: white, black
- Illegal move

The PCB also contains a port for the ARM SWD interface to enable easy debugging and a USB-C port for power and communication with a computer.

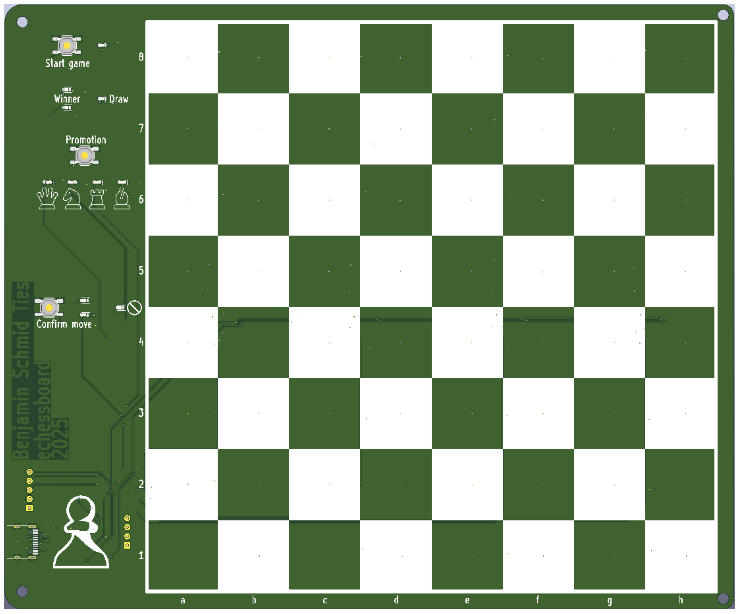


Figure 2: Front view of chessboard PCB (user interface)

3 SPI communication

Each move on the PCB is read by the STM32 microcontroller and sent via SPI to the FPGA, where it is validated and sent through a VGA module to the monitor.

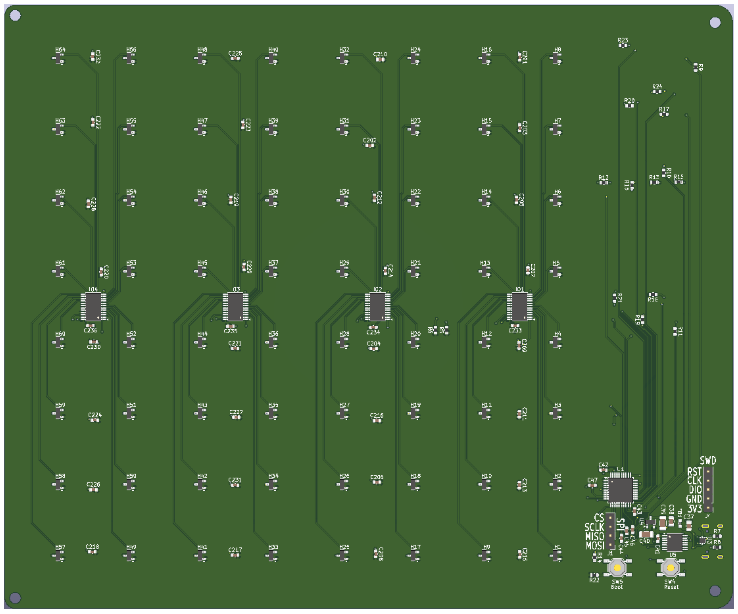


Figure 3: Back side of chessboard PCB

3.1 SPI introduction

SPI is a serial synchronous communication, which means that the clock is transmitted on a separate line from the data (SCKL channel). At each clock cycle, the master sends one bit to the slave through the MOSI (Master Out Slave In) channel, and the slave sends the data back to the master with the MISO (Master In Slave Out) line.

To start the communication, the master asserts low on the Slave Select (SS) line; then, based on the mode of operation, the data is transmitted. To close the communication, the master deasserts the chip select signal.

The modes of operation are based on the combination of clock polarity (CPOL) and clock phase(CPHA):

1. CPOL=0, CPHA=0: the clock starts low (logic state), and the signals are sampled on the rising edge of the clock;
2. CPOL=0, CPHA=1: the clock starts low (logic state), and the signals are sampled on the falling edge of the clock;
3. CPOL=1, CPHA=0: the clock starts high (logic state), and the signals are sampled on the falling edge of the clock; clock;
4. CPOL=1, CPHA=1: the clock starts high (logic state), and the signals are sampled on the rising edge of the clock;

3.2 SPI Master

In this project, the master is the STM32 microcontroller, while the slave is implemented on FPGA. The clock generated by the MCU has a frequency of 4MHz and is CPOL = 1. The data is transmitted at the falling edge, so the mode of operation is CPOL=1, CPHA=0.

The data sent by the master can be of 4 types:

1. Change in position: 64 bits that indicate the configuration of the board (1 if the piece is present, 0 otherwise), sent each time there is a change;
2. confirm move: the signal that the confirm button on the PCB has been pressed;
3. Promotion: The piece a pawn will be promoted to.
4. start game: the button to restart the game is pressed;

Type	Command	Encoding
Position	”00000000”	bit ’0’ = empty square, bit ’1’ = occupied square
Confirm move	”00000010”	bit ’1’ = move confirmed, bit ’0’ = no confirmation
Promotion	”00000001”	”xxx” based on the piece, refer to 2
Start game	”11111111”	restart the game

Table 1: Encoding of transmitted signals

Before sending the data, a byte indicating the type of message is sent (command).

Since the dimension of the sending data set for the STM32 microcontroller is one byte, the dimension of the command, of confirm move and substituting piece is also one byte.

In Table 1

The received data are used to switch on or off the LED on the PCB: if the move is not allowed, the error LED switches on; if the king of one player is eaten, the winner LED switches on.

3.3 SPI Slave

To implement SPI Slave on an FPGA, three processes have been used:

1. MOSI: for each bit received, a counter increments; when it reaches 8, it means that one byte (minimum size of transmitted data) is received; the data is then passed to the third process, the counter is reset, and the process is ready for another byte.

2. MISO: At each clock cycle, data is sent back to the master. The data is a byte, where 1 bit indicates the presence of an error, 2 bits encode the winner, and the others are zeros.

3. Communication with game control logic: once a byte is received, this process activates and based on the command (first byte sent), it increments a counter until the value of needed bytes for the encoded signal is reached: 8 byte for a complete board configuration and 1 byte for confirmed move and substituting piece. At this point, signals for the corresponding data are filled and can be used by the game control logic, so a valid signal is sent to this entity.

The main obstacle to address is Clock Domain Crossing; in fact, the SPI slave works at the clock frequency of the FPGA, but the master sends the data with its clock frequency. The main problem when crossing a clock domain is metastability; in fact, the signal from the master is not synchronised with the clock of the slave, therefore it can enter a state of metastability, where its value is neither 1 nor 0. In particular, this happens for set-up time and hold time violations. To avoid this behaviour, the most used method is to introduce a double flip-flop between the input and the logic as shown in Figure 4. This is the solution adopted in this project.

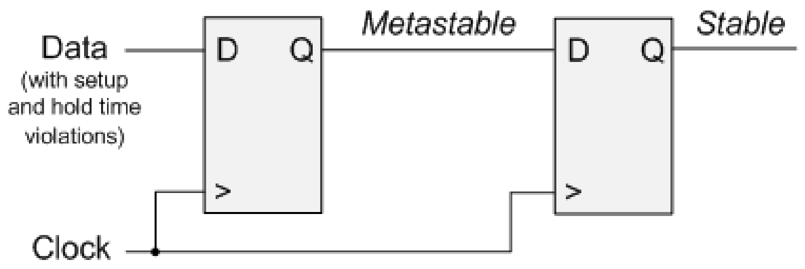


Figure 4: Double flip-flop for signal synchronisation

At the code level, the input signal from the master is sampled on the rising edge of the FPGA clock, first into a metastable register, and then into a second register, which stabilises the signal. To detect a rising edge on a signal when crossing from a slow to a fast clock domain, like in this case, the following code is used:

```
sclk_fedge <= not sclk_reg and sclk_meta;
```

This means that when the value on the first flip-flop is '1', but that on the second flip-flop is '0', then there has been a rising edge.

For the communication with the control logic, a final state machine with two states was used, the states and the condition for the transition are represented in 5

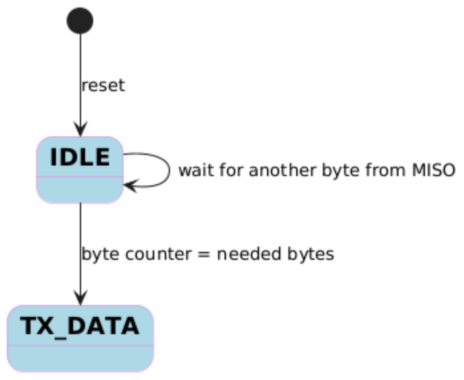


Figure 5: FSM for communication between SPI and game logic

4 Game Control logic

The data received by SPI is only binary data that indicates whether a square is occupied or not; in order to display the chessboard on the screen, the colour and the shape of the piece must be deduced.

To achieve this, a game control logic was implemented, which, based on the initial configuration of the chessboard, deduces the piece that has been moved and updates the configuration to be displayed on the screen. Another task of this entity is to verify whether the move is allowed or not and to send feedback to the microcontroller on the PCB, which controls the LED that signals an error. Along with the error signal, the game status is also transmitted: in progress, won by white, or won by black.

To check the validity of the move, six validators have been implemented—one for each type of piece. The overall structure of the FSM is shown in Figure 7.

The communication between the game control logic and the other modules is schematized in Figure

When a game starts, the start game button on the PCB is pressed, which resets the logic and initializes the state to *WAIT-SPI*; in this state, a valid signal is expected from the SPI module. As long as the signal doesn't arrive, the logic remains in this state.

When a valid signal arrives, the FSM enters the *INPUT-DETECTION* state: here, it checks if there is a change between the last saved configuration and the incoming position.

Elaborating the data only when the move is confirmed through the button is not enough; in fact, when a piece is captured, the configuration change would only allow detecting where the piece was lifted, but not where it was placed—since it could be placed on an already occupied square.

To detect this change, the start and end positions are updated each time the configuration of the board changes. A flag is used to record the number of pieces lifted and set on the board (for lifted pieces, the changed bit becomes '0'; for placed pieces, it becomes '1').

A maximum of two lifted pieces is allowed, as in the case of a capture; otherwise, the FSM enters the *ERROR* state.

Only when a move is detected does the state transition to *PIECE-DEDUCTION*, where—if the lifted piece is of the correct color—the corresponding validator is activated.

To communicate with these external entities, the handshake technique is used: a start signal activates the validator, and the FSM remains in this state until the done signal is received from the validator.

If the validator sends a valid response, the FSM moves to the *VALIDATOR-LOGIC* state, which updates the board configuration with the new changes. Moreover, it checks whether the move has captured the king—in which case the next state becomes *WIN*, and no further state changes occur until a new start game signal is received.

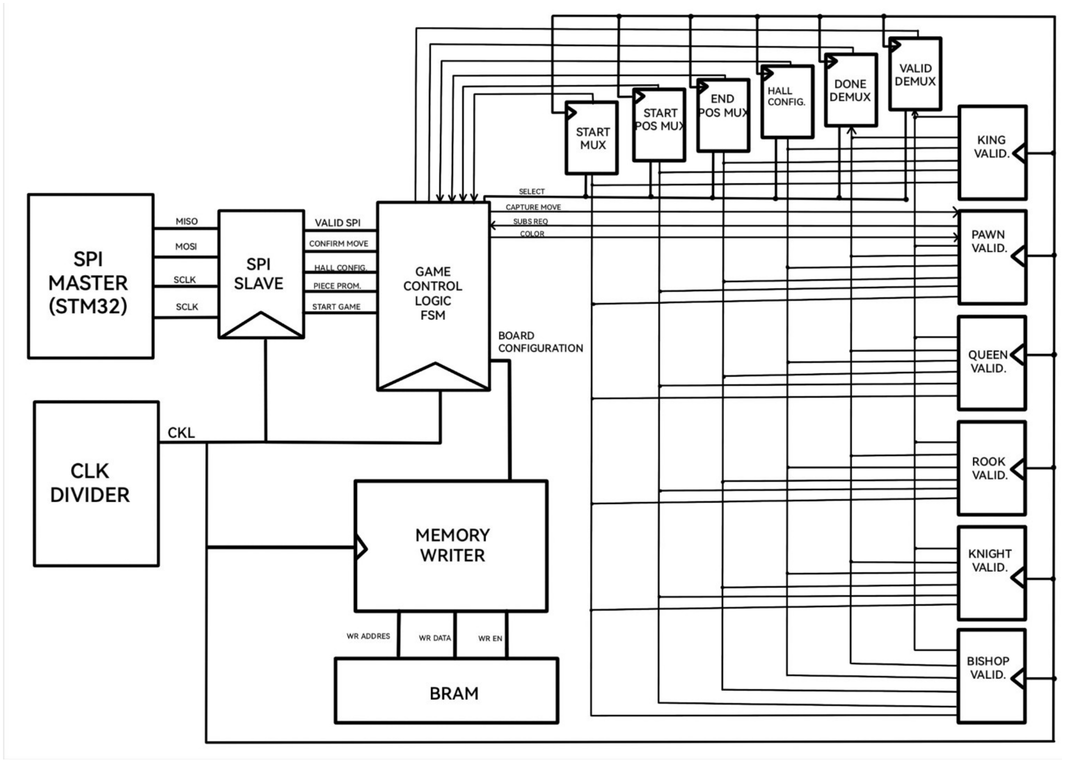


Figure 6: FSM for communication between SPI and game logic

SHAPE	empty	pawn	rook	bishop	knight	queen	king
ENCODING	”000”	”001”	”010”	”011”	”100”	”101”	”110”

Table 2: Encoding of piece shape’s

If the move does not end the game, the state returns to *WAIT-SPI*, where it waits for another configuration change.

Lastly, if the FSM enters the *ERROR* state, the move is discarded, and a new valid move is expected.

6

4.1 Validators

Six validators have been implemented to verify the validity of each piece; to select the right validator, the shape of the piece that started the move is used as a selection signal for six multiplexers/demux 2. These are used to connect the right validator to the *start, valid* and *done* signals for the handshake and the start position, end position, and hall configuration signals used for the validation.

4.2 Memory Writer

Another entity is used to save each new configuration into a BRAM so that it is accessible by the chess position renderer. The width of the BRAM is 32 bits, and each shape is encoded with 8 bits: one for the colour, four zero bits, and 3 bits for the position. Then, one row of the chessboard is saved in 2 memory addresses. The memory is updated at each change of the clock cycle, but a better implementation would be to enable this entity with a signal coming from the logic control, so that the memory is updated only when a real change happens.

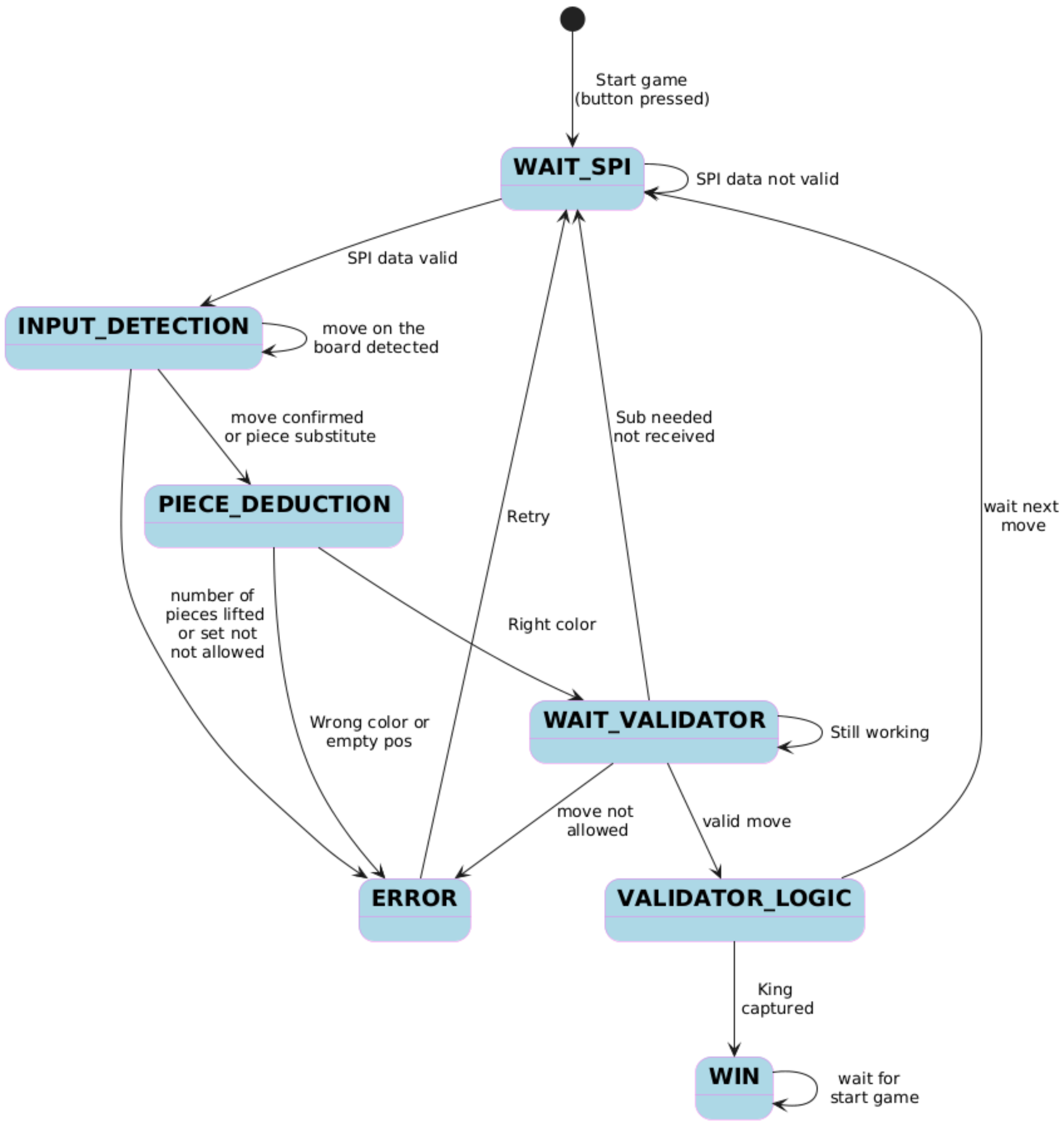


Figure 7: Block diagram of the connections between different modules of the game

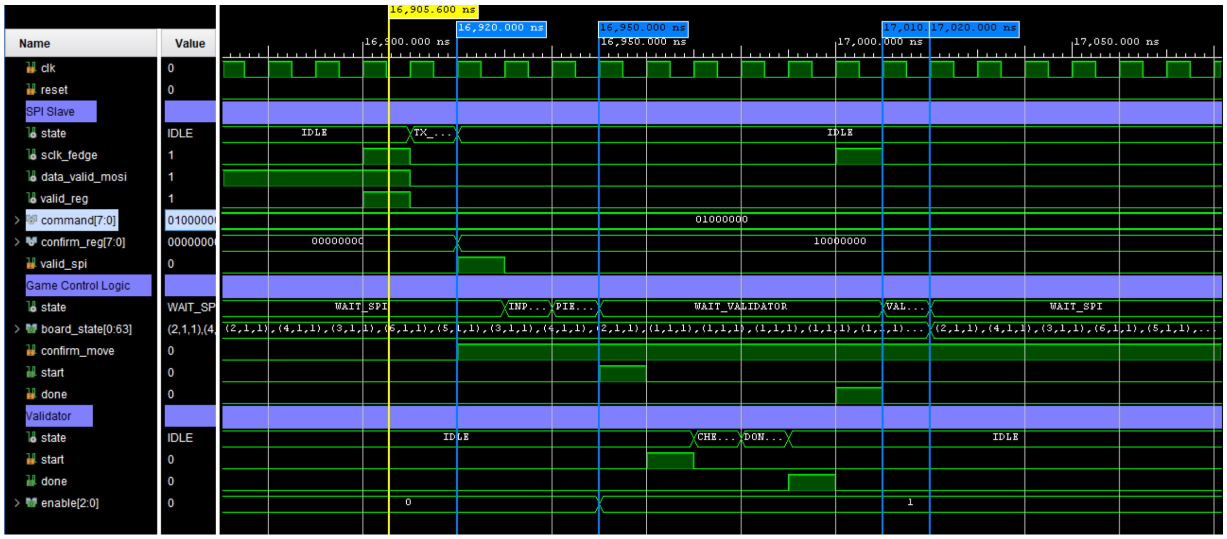


Figure 8: Timing diagram showing some changes in signals when a confirmation move arrives

5 Timing Diagrams

To verify the correct functionality of the entities, a behavioral simulation in Vivado is used. The following is a summary of the results:

In Figure 8, the behavior and communication between the SPI Slave, the Game Control Logic, and the Validator are depicted. The valid register is set by the MOSI process when one byte arrives, which synchronizes the signal entering the final state machine with the external clock. In the IDLE STATE, if the byte counter is zero, the command is stored, and based on this command, the number of additional bytes required to complete the transmission is determined. For the "confirm move" operation, only one more byte is needed. Once this byte is received, a valid spi signal is set, and the Game Control Logic reads the data. In the INPUT DETECTION state, if the confirm move is detected, the state transitions to PIECE DEDUCTION, activating the corresponding validator for the moved piece. The Game Control Logic then waits for the done signal from the validator.

In Figure 9, the scenario is illustrated where a change is detected, but the confirm move has not been pressed. Using a copy of the hall sensor configuration, the difference with the new sensor values is detected, and the index of the changed piece is stored in a register. This index will later be used for move validation when the confirm move signal is received.

In Figure 10 shows the complete interaction between the entities. Each blue cursor represents one full move: the first is a normal move, followed by two capture moves, and another normal move.

Finally, Figure 11 illustrates how the validators are activated through the multiplexers. In Figure **??**

6 RISC-V Chess position renderer

The C program running on the RISC-V microprocessor takes as input the decoded chess position and outputs an image into a framebuffer memory. Images of each chess piece are placed in the data memory. The goal was to optimize for speed as much as possible, because the RISC-V microprocessor was not pipelined or otherwise optimized for speed. By making the observation that only 4 distinct colors are strictly necessary to render a complete chessboard, each pixel can be represented by 2 bits. These 2 bits can then be converted into 12 bit RGB colors by the VGA image generator. By implementing only 2 bits per pixel, 16 pixels can be rendered at once using logical operations on a 32-bit word. The pieces were converted from PNG images to a binary array. Each field of the chess board has a size of 48x48 pixels. This size is convenient because 48=16*3, which means a row of the image can be represented using 3 32-bit words. The array for a piece image is then 3x48 32-bit integers (3 integers for every row and 48 rows). The original chess piece images were resized to 48x48 pixels. Every pixel of the transparent background was converted to 00 and each pixel contained within the piece

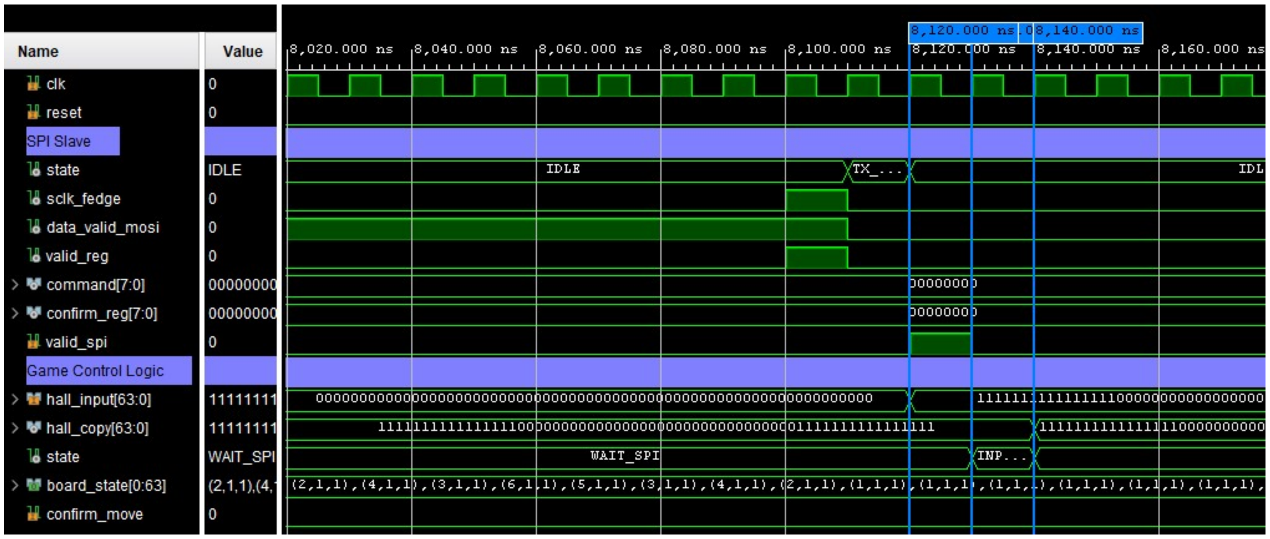


Figure 9: Timing diagrams showing some changes in signals when a new position arrives

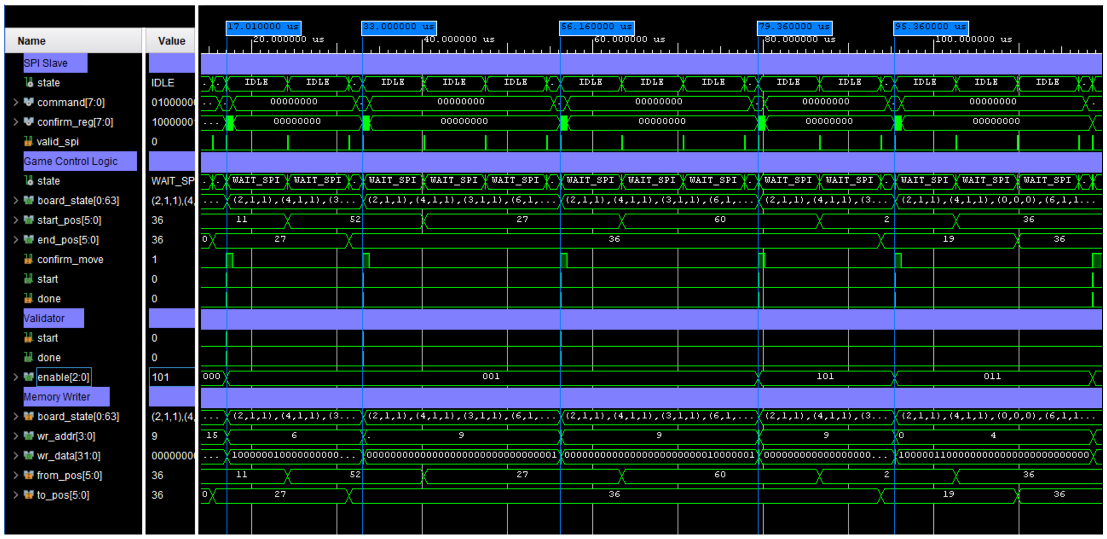


Figure 10: Timing diagrams showing the change in states and signals for each entity, when some moves are simulated through testbench

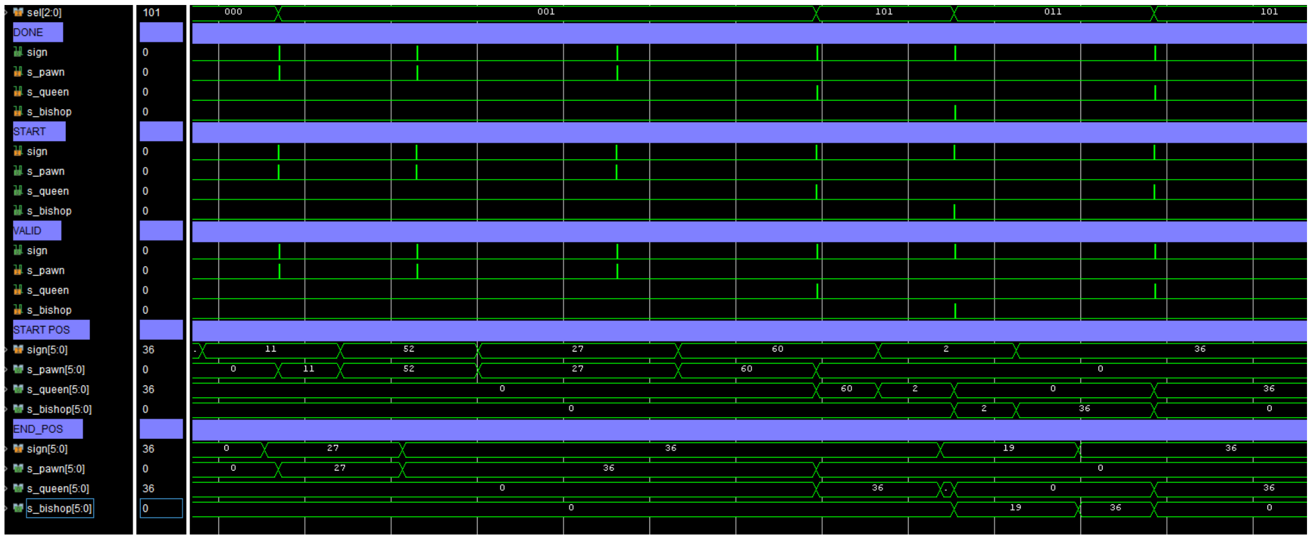
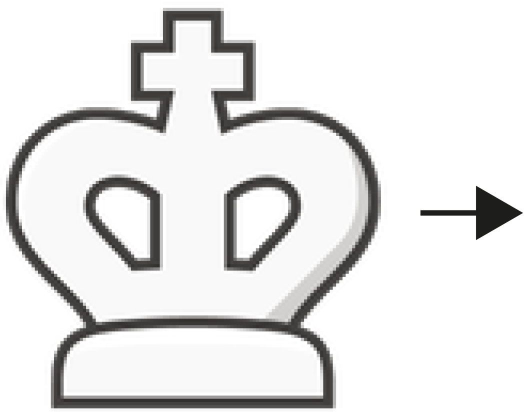


Figure 11: Timing diagrams showing the start, done and valid sequence for the activated validator

was converted to 11. Using logical operations, the image is rendered 16 pixels at a time. Using the riscv cross compilation toolchain, the program was compiled for the RV32I instruction set and a linker script was used to configure instruction and data memory. The .text section was extracted and used to initialize RISC-V instruction memory and the .rodata and .srodata sections were used to initialize data memory. The .rodata contains the constant chess piece arrays for rendering, and 64 bytes for the chess position, which is decoded by dedicated logic.

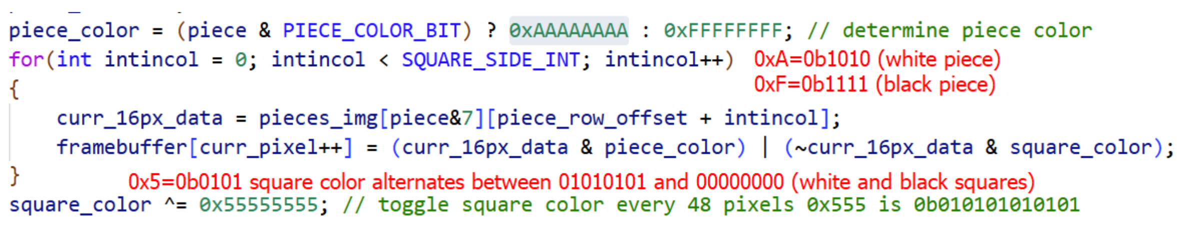


```
00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000001111111111110000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000001111111111110000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000001111111111110000000000 00000000000000000000000000000000
00000000000000000000000000000000 00001111111111111111111111110000 00000000000000000000000000000000
00000000000000000000000000000000 00001111111111111111111111110000 00000000000000000000000000000000
00000000000000000000000000000000 00001111111111111111111111110000 00000000000000000000000000000000
00000000000000000000000000000000 00001111111111111111111111110000 00000000000000000000000000000000
00000000000000000000000000000000 00001111111111111111111111110000 00000000000000000000000000000000
00000000000000000000000000000000 00001111111111111111111111110000 00000000000000000000000000000000
00000000000000000000000000000000 00000000001111111111110000000000 00000000000000000000000000000000
00000000000000000000111111111111 11111111111111111111111111111111 11111111111100000000000000000000
00000000000000000011111111111111 11111111111111111111111111111111 11111111111111000000000000000000
00000000000000001111111111111111 11111111111111111111111111111111 11111111111111110000000000000000
00000000000000111111111111111111 11111111111111111111111111111111 11111111111111111100000000000000
00000000000011111111111111111111 11111111111111111111111111111111 11111111111111111111000000000000
00000000000011111111111111111111 11111111111111111111111111111111 11111111111111111111000000000000
00000000000011111111111111111111 11111111111111111111111111111111 11111111111111111111110000000000
00000000000011111111111111111111 00001111111111111111111111111100 11111111111111111111110000000000
00000000001111111111111111110000 00000000111111111111111111000000 00001111111111111111110000000000
00000000001111111111111111110000 00000000111111111111111100000000 00001111111111111111110000000000
00000000000011111111111111111100 00000000111111111111111100000000 00111111111111111111110000000000
00000000000011111111111111111111 00000000111111111111111100000000 00111111111111111111000000000000
00000000000011111111111111111111 00000000111111111111111100000000 11111111111111111111000000000000
00000000000000111111111111111111 11000000111111111111111100000011 11111111111111111100000000000000
00000000000000111111111111111111 11110000111111111111111100001111 11111111111111111100000000000000
00000000000000001111111111111111 11111111111111111111111111111111 11111111111111110000000000000000
00000000000000000011111111111111 11111111111111111111111111111111 11111111111111000000000000000000
00000000000000000000111111111111 11111111111111111111111111111111 11111111111100000000000000000000
00000000000000000000001111111111 11111111111111111111111111111111 11111111110000000000000000000000
00000000000000000000000011111111 11111111111111111111111111111111 11111111000000000000000000000000
00000000000000000000000000111111 11111111111111111111111111111111 11111100000000000000000000000000
00000000000000000000000000111111 11111111111111111111111111111111 11111100000000000000000000000000
00000000000000000000001111111111 11111111111111111111111111111111 11111111110000000000000000000000
00000000000000000000001111111111 11111111111111111111111111111111 11111111110000000000000000000000
00000000000000000000111111111111 11111111111111111111111111111111 11111111111100000000000000000000
00000000000000000000111111111111 11111111111111111111111111111111 11111111111100000000000000000000
00000000000000000000111111111111 11111111111111111111111111111111 11111111111100000000000000000000
00000000000000000000111111111111 11111111111111111111111111111111 11111111111100000000000000000000
00000000000000000000111111111111 11111111111111111111111111111111 11111111111100000000000000000000
00000000000000000000111111111111 11111111111111111111111111111111 11111111111100000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000
```

Figure 12: Conversion of piece image to 48x48 pixels(3x48 32-bit words)

7 VGA controller and image generator

The VGA controller uses a clock signal to generate vertical and horizontal sync signals, which are transmitted to a compatible screen. It also outputs x and y coordinates used by the image generator. For a resolution of 640x480, with a refresh rate of 60Hz, a 25MHz clock signal is necessary. Initially,



```
piece_color = (piece & PIECE_COLOR_BIT) ? 0xAAAAAAAA : 0xFFFFFFFF; // determine piece color
for(int intincol = 0; intincol < SQUARE_SIDE_INT; intincol++)  0xA=0b1010 (white piece)
{                                                               0xF=0b1111 (black piece)
    curr_16px_data = pieces_img[piece&7][piece_row_offset + intincol];
    framebuffer[curr_pixel++] = (curr_16px_data & piece_color) | (~curr_16px_data & square_color);
}         0x5=0b0101 square color alternates between 01010101 and 00000000 (white and black squares)
square_color ^= 0x55555555; // toggle square color every 48 pixels 0x555 is 0b010101010101
```

Figure 13: Logic operation for rendering 16 pixels at a time

the horizontal sync signal goes high. At every clock cycle, one pixel is sent, using 3 4-bit DACs for red, green, and blue intensities. After 640 pixels have been transmitted, there is a number of clock cycles where no pixels are transmitted, composed of front porch, sync pulse, and back porch. The horizontal sync signal stays high during the front porch, goes low during the sync pulse, and goes back high during the back porch. This indicates to the screen that the next row will be transmitted. After 480 rows have been transmitted, there is a number of rows during which no pixels are transmitted, but the horizontal sync signal is still pulsed for every row. The vertical sync signal stays high during the vertical front porch, goes low during the vertical sync pulse, and goes back high during the vertical back porch. The image generator uses the x and y coordinates generated by the VGA controller to address the framebuffer memory and load data. After every 16 clock cycles (32-bit word, 2 bits per pixel -¿ one word contains 16 pixels), new data has to be loaded from the frame buffer. The 4 least significant bits of the x coordinate are used to index the current pixel within the current word. One pixel (2 bits) is extracted from the word and used to retrieve a 12-bit RGB color from a look-up table. These 12 bits are then output to the corresponding VGA DAC pins and transmitted through the VGA cable to the screen.

8 RISC-V microprocessor

The RISC-V microprocessor used to render the current chess position implements the full RV32I base integer instruction set. The data memory has a base address of 0, and the framebuffer has a base address of 0x80000000. The RISC-V microprocessor runs at a clock speed of 6.25 MHz clock, derived by dividing the external clock of 100MHz by 16.

9 Future Developements

A possible future development for the game logic is implementing checks and checkmates on the king. This can be achieved by introducing an additional state in the finite state machine (FSM), following the state that validates the move. In this new state, the validators should be decoupled from the main game logic and attached to a separate FSM that simulates all possible moves of the opponent's pieces. If any of these simulated moves target the king's position and are deemed valid by the validators, a check is detected and signaled to the player.

Detecting checkmate is more complex. For every threat to the king, there may be a counter-move that blocks or captures the attacking piece. Therefore, another simulation is required to evaluate all possible defensive moves. If none of them successfully remove the threat, the game can conclude it's a checkmate.

Another planned improvement is optimizing the validators themselves. Currently, a full FSM is used, but in reality, only a process that checks the initial move and updates the move state and validity is necessary. Replacing the FSM with a simpler procedure could significantly improve performance by reducing the time the game logic waits for validation, thereby shortening the critical path.

Finally, memory writes can be optimised by updating only the blocks that change, and only when a change occurs. The memory consists of 16 bytes, and each move affects at most two positions on the board: the starting square and the destination square. Therefore, at most two memory blocks need to be updated per move. A dedicated finite state machine, as illustrated in Figure 14, can be implemented to handle these selective memory updates efficiently.

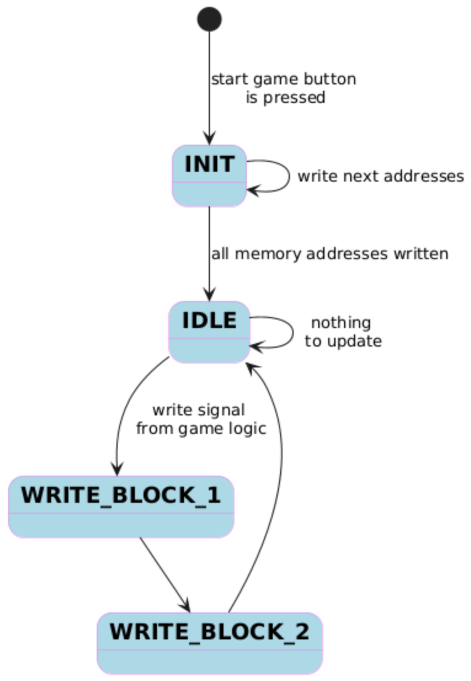


Figure 14: FSM for the write to memory entity for future developements