MASTERMIND ARCADE

SYMMETRICAL DESIGN

Both playing boards are loaded with the same program, so you just need to buy one design!

1 vs 1 game (Alice vs Bob)

Both players have to guess the SAME SECRET SEQUENCE (this increases the competitiveness!). The secret sequence is generated in Alice's board and sent to Bob's board, but none of them knows which is the secret sequence...

UNTIL **THE END**

Helena Calatrava, Albert Díez

ESDC - Project Report
Mastermind 2-Player Game

Contents

Please read this info related to the repo:
Link Repo: https://bitbucket.org/helenaCR/esdc_albert_helena/src/master/repo/
This repository contains:

- **Final Simulation:** simulation files, video explaining the simulation and README with an exhaustive description of the events that take part in the simulation.
- **Experimental Validation:** link to the final experimental validation video. https://youtu.be/HGz8_k-GD3A
- **Explanatory Content:** auxiliary information that has helped us during the project.
- **Docs:** old documents that have been delivered in Atenea.
- **Debugging Folder:** files used in Quartus to simulate parts of our system before performing the final simulation. These simulations are exhaustive.
- **VHDL Files:** VHDL files are only included in the repo. The whole project is also included (*.qpf* file and others).

1. Introduction

1.1. Document Description

This document is the Final Project Report of our ESDC Project: the Mastermind Game. The aim of this section is to introduce the project to the reader and also to give some details related to the contents we are delivering to the teachers: this document itself plus a cloud repo (detailed description in *subsection 1.3*).

In Section 2: Project Specifications, we are giving the updated SPECS of the project. Although there have been some deviations from the initial plan, the most relevant change from the initial project specs is the subsection related to What If Situations.

Section 3 shows the whole Block Diagram of the project. Also, we give a detailed explanation of each of the main blocks of our project. In this section we can see important changes from the initial Block Diagram, as we have been forced by our design to add new blocks in order to meet the initial specifications. Also, we think that our final approach makes the design easier to understand and also easier to follow for us when performing the final simulations, which are very complex.

Section 4 contains the algorithms of the main blocks. The state diagrams of the main blocks are included in Section 5. Both the algorithms and the State Diagrams have helped us when having to code all the VHDL files, also included in this Section 4.

For both the algorithms and the state diagrams, we have provided the reader with a link to the image (of either the algorithm or the state diagram) with a higher quality, as the quality of the images in this document was not as high as we wanted. The VHDL files are contained in the project repo, as we considered not nice from a format standpoint to include all the VHDL files in the document itself.

In Section 6, we are showing the contents of the README file that we have included in the repo. It was not necessary to include in this PDF but we think it is informative and descriptive for the reader, as it can be also seen as a *flux* diagram of our design in the form of a *log* file.

In Section 7 we have described the experimental validation that we have performed our teachers with, with a bit of explanation. Finally, in Section 8 we have written a small conclusion about the overall of the project, with both technical and academical details.

1.2. Introduction

Mastermind is a code-breaking game created in the 70s. We have chosen this game as our ESDC project because it was invented by Mordecai Meirowitz, an Israeli postmaster and telecommunications expert, and as new students of the Telecommunication Engineering Master it is an honour to implement this game.

In order to decide the rules of the game we have read the general instructions of the Mastermind game and we have been playing the mobile app Mastermind (by *Rottz Games*)

to get ideas related to difficulty levels and user interface items. The instructions given in the following lines are our specific design instructions.

1.3. Repository Documentation

As we have specified in the Contents page, we have created a repository with the following information:

- **Final Simulation:** simulation files, video explaining the simulation and README with an exhaustive description of the events that take part in the simulation.
- **Experimental Validation:** link to the final experimental validation video. https://youtu.be/HGz8_k-GD3A
- **Explanatory Content:** auxiliary information that has helped us during the project.
- **Docs:** old documents that have been delivered in Atenea.
- **Debugging Folder:** files used in Quartus to simulate parts of our system before performing the final simulation. These simulations are exhaustive.
- **VHDL Files:** VHDL files are only included in the repo.

The link to the repo is: https://bitbucket.org/helenaCR/esdc_albert_helena/src/master/repo/

The repository contains several README files that provide a description of the contents in each of the folders.

2. Project Specifications

2.1. Game Instructions

In this game, a secret sequence of four pegs is randomly proposed by the machine. The sequence contains colors that can be repeated. Also, the sequence can contain an empty peg. Two players are participating in our game and their objective is to guess the secret sequence in less attempts than their opponent, with a maximum of ten attempts.

Players have to choose between being Alice or Bob, and they can't be the same. The player that presses "A" will be Alice and the one that presses "B" will be Bob. Alice is in charge of choosing the game characteristics and Bob has to wait until the game starts.

The secret sequence will be generated in Alice's FPGA, which will send the sequence to Bob's FPGA. Although players have the secret sequence in their FPGAs, they never have access to it. Otherwise, the game would be corrupted.

No player will have advantage with respect to the other player because they will start at the same time. In each turn each player tries to guess the sequence simultaneously to the other player. **We decided that both players have to guess the same code in order to increase competitiveness.**

Alice will have to answer two questions. Depending on the answers, the generated secret sequence will be different. In **Table 2.2.1** we can see the sequence possibilities for each combination of answers. Color possibilities are shown in **Figure 2.2.1.**

Every time players try to guess the secret sequence, they get feedback from what we call the *small result pegs*. Players get a red peg for each correctly guessed peg, a white peg for each guess with the correct color but in the wrong position and an empty small peg for each wrong guess. This is explained in a clearer way in the diagram shown in **Figure 2.2.2**.

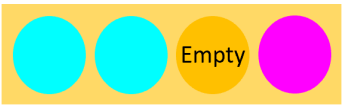
The order of the result pegs is randomized. Therefore, if we get a red peg in the first position, it does not mean that the first guessed peg is correct. It can be any of the other pegs. Also, players will see the result pegs of their opponent in order to keep track of the game evolution by knowing if they are doing it worse or better than their opponent.

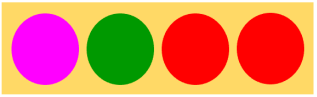
2.2. System Description

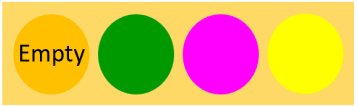
In **Figure 2.2.2,** we can see a general state diagram of the game. This is not meant to be the state diagram of any block of our design, it is just a guide that helps to understand the functioning of our system. However, in this state diagram, no frame types are named. In order to understand how frames will be used in our system, it is required to read the following contents of this subsection.

Figure 2.2.1. Color possibilities for VGA screen.

<table>
<tr><td colspan="2">4 PEG sequences
  </td><td colspan="2">Do you want to have one empty peg in the secret sequence?</td></tr>
<tr><td colspan="2"></td><td>Yes</td><td>No</td></tr>
<tr><td rowspan="2">Do you want to have one repeated color in the secret sequence?</td><td>Yes</td><td>Pegs can be Blue, Magenta, Red, Yellow, Green and Cyan. Also, there is one empty peg and one color is repeated once. Therefore, the amount of different colors in the secret sequence is 2.

Example sequence:
  </td><td>Pegs can be Blue, Magenta, Red, Yellow, Green and Cyan. One of the colors in the sequence is repeated once. Therefore, the amount of different colors in the secret sequence is 3.

Example sequence:
  </td></tr>
<tr><td>No</td><td>Pegs can be Blue, Magenta, Red, Yellow, Green and Cyan but also there is one empty peg. Therefore, the amount of different colors in the secret sequence is 3.

Example sequence:
  </td><td>Regular sequence. Pegs can be Blue, Magenta, Red, Yellow, Green and Cyan. Each color can appear only one time.

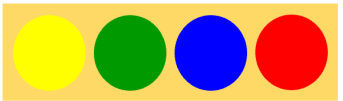
Example sequence:
  </td></tr>
</table>

Table 2.2.1. Description of the possible sequences depending on Alice's answers to questions.

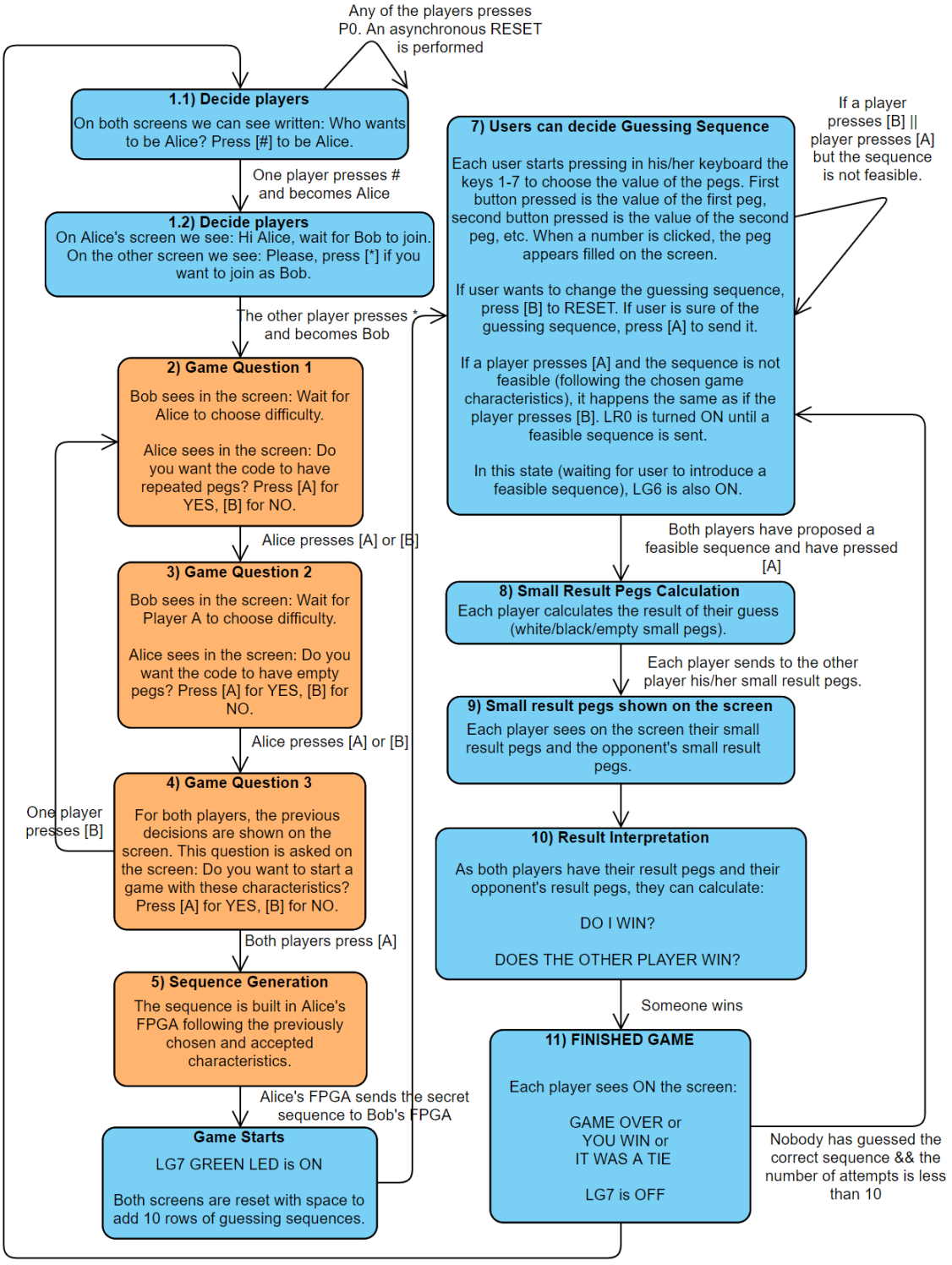


Figure 2.2.2. General State Diagram of our system. Orange states are states where Alice and Bob function in different ways.

As we have stated before, the diagram in **Figure 2.2.2.** is just a guide to understand the overall functioning of our system, but it is not meant to be an exact representation of the game flux with details. With this we mean that there are not FRAME TYPES written in this diagram, or input and output variables specified. However, we find it very descriptive and even necessary to check before going deeper in the project report.

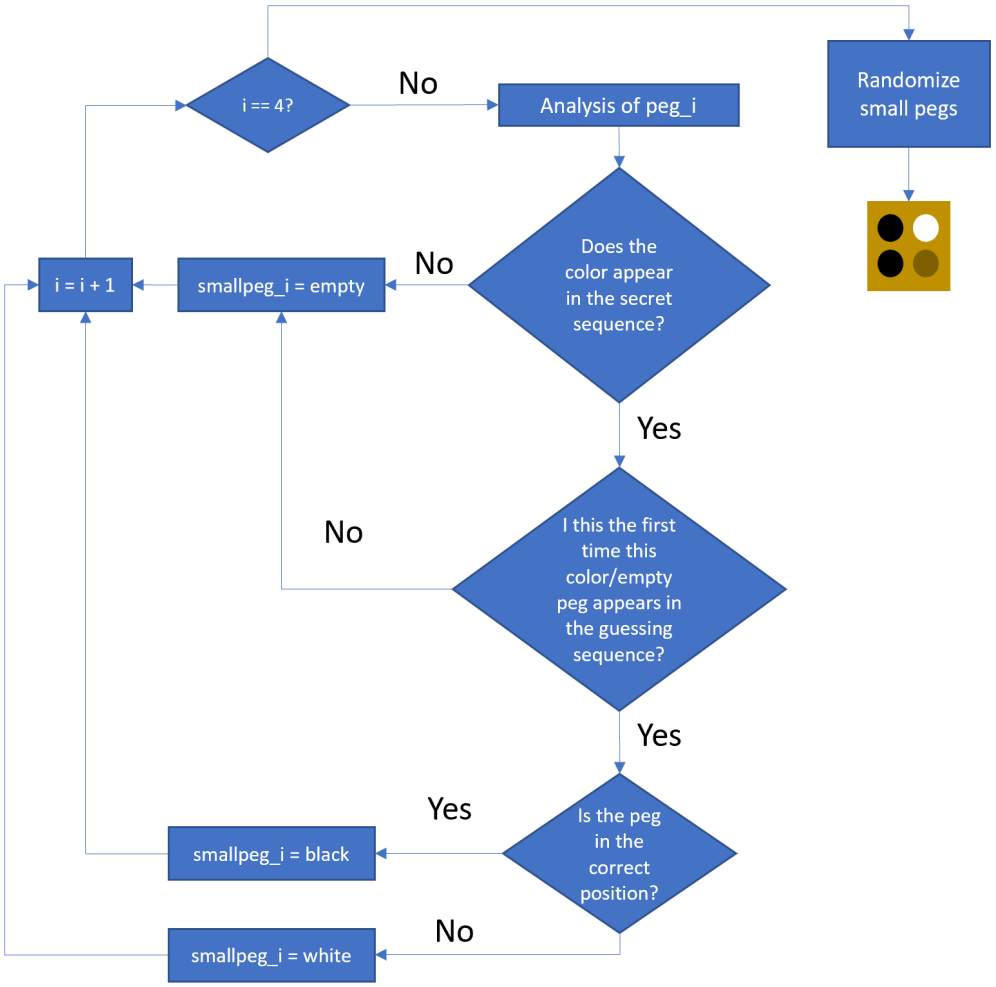


Figure 2.2.3. Diagram of the behaviour of the small result pegs.

In the following lines, we are giving an overall but detailed description of the system functioning.

- **[INITIAL STAGE]** The game starts with two boards connected. LED LG0 is turned on in both boards to inform that they still have to be identified. The player that presses [#] will become Alice and therefore her LG0 LED will turn off and her LG1 LED will turn on. The frame I AM ALICE will be sent to the other board. The player that instead presses [*] will become Bob and therefore her LG0 LED will turn off and her LG2 LED will turn on. The frame I AM BOB will be sent to the other board.
- When Alice receives the I AM BOB frame, she has to answer two questions related to the generation of the secret sequence. First, she will decide if the code has one repeated peg by pressing [A] to accept or [B] to cancel. Second, she will decide if the code has an empty peg by also pressing [A] to accept or [B] to cancel.
- The secret sequence is generated in Alice's FPGA. Alice sends this sequence to Bob in the frame SECRET SEQUENCE. However, it is a secret sequence and none of the players will ever have access to it.
- From this point, both FPGAs behave in the same way and the game starts. LG7 is turned on in both boards.

- **[GUESSING STAGE]** Each user can begin guessing the sequence by pressing the keys 1-7 of their keyboard as indicated in **Tab. 2.4.1**. When the system is waiting for the user to introduce a sequence, the LG6 LED is on. If a user is sure about their proposed sequence, they can fix it by pressing [*].
- When a user fixes the sequence, the LED LG6 is turned off and the board computes the result pegs and sends them to the other user with the frame type RESULT PEGS. Therefore, each user has their own result pegs and the result pegs of their opponent. At this moment, each user sees on the screen their result pegs and the result pegs of their opponent.
- With this information (own result pegs and opponent's result pegs) each user can calculate if the game is finished or has to continue. If any of the players has correctly guessed the sequence or this last turn was the tenth turn, LG1 or LG2 (depending if user is Alice or Bob) and LG7 are turned off and the squares corresponding to *Game Over / You Win / It's a Tie* is shown on each screen, depending on the result. Back to **[INITIAL STAGE]**
- If the game continues, then both users go back to entering the code (and LED LG6 is turned on again). Back to **[GUESSING STAGE].**

2.3. Frame Description

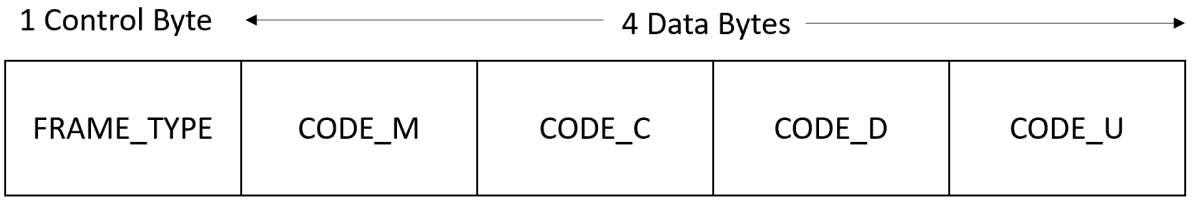


Figure 2.3.1. Frame structure (one box per byte).

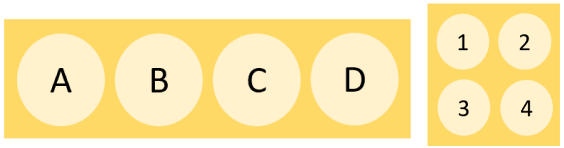


Figure 2.3.2. Sequence pegs (left) and small result pegs (right).

In the following list, we can see the different FRAME TYPES that are used in the whole design system:

- I AM ALICE $(A1)_{HEX}$
- I AM BOB $(A3)_{HEX}$
- SECRET SEQUENCE $(C1)_{HEX}$
- RESULT PEGS $(C3)_{HEX}$

The four bytes of data are only needed if the frame type is SECRET SEQUENCE or RESULT PEGS. In both cases, we want to know the value of four pegs. In the case of the frame type SECRET SEQUENCE, the values we want to know are the colors of the secret sequence. This frame is sent at the beginning of the game, only once. We use the four data

bytes as follows. In this case, only three bits for peg are required because colors will go from 1 to 7.

CODE_M: 0000 0$M_2M_1M_0$, where $M_2M_1M_0$ is the number associated with peg A
CODE_C: 0000 0$C_2C_1C_0$, where $C_2C_1C_0$ is the number associated with peg B
CODE_D: 0000 0$D_2D_1D_0$, where $D_2D_1D_0$ is the number associated with peg C
CODE_U: 0000 0$U_2U_1U_0$, where $U_2U_1U_0$ is the number associated with peg D

In the case of the frame type RESULT PEGS, the values we want to know are the colors of the result pegs of our opponent. This frame is sent at the end of each turn. In this case, only two bits for peg are required because colors will go from 0 to 2.

CODE_M: 0000 00M_1M_0, where M_1M_0 is the number associated with result peg 1
CODE_C: 0000 00C_1C_0, where C_1C_0 is the number associated with result peg 2
CODE_D: 0000 00D_1D_0, where D_1D_0 is the number associated with result peg 3
CODE_U: 0000 00U_1U_0, where U_1U_0 is the number associated with result peg 4

Repeated Peg (U_1)	Repeated Color (U_0)	Game characteristics
0	0	Regular sequence
0	1	Color is repeated once
1	0	One empty peg
1	1	One empty peg and a color is repeated once

Table 2.3.1. Bits in the frame type GAME CHARACTERISTICS.

2.4. User Interface

- Keyboard. We can see the use of each key in **Table 2.4.1.**
- ALTERA DE2 CYCLONE II:
  - Push button 0 (P0) to perform a global asynchronous reset.
  - RED LEDS
    - **LR0**: on if there has been an error when identifying both players (state 1.1 and 1.2 from **Figure 2.2.2.**).
  - GREEN LEDS
    - **LG7**: on if the game is in progress (state 6-11 from **Figure 2.2.2.**).
    - **LG6**: on if the user can enter a sequence (state 7 from **Figure 2.2.2.**).
    - **LG0**: on if the player can still choose character (from **Figure 2.2.2.**, state 1.1 in case of Alice but state 1.1 and 1.2 in case of Bob).
    - **LG1**: on if the user is Alice (until returning to state 1.1 from **Figure 2.2.2.**).
    - **LG2**: on if the user is Bob (until returning to state 1.1 from **Figure 2.2.2.**).

- Monitor screen connected by VGA to the FPGA. One monitor screen per FPGA (two int total, as there are two players in the game).
- RS-232 for the communication between the two FPGAs.

Keyboard Key	Use of Key
1	Red
2	Blue
3	Green
4	Yellow
5	Magenta
6	Cyan
0	Empty
[A]	Accept button
[B]	Cancel button
*	Key to become Bob
#	Key to become Alice

Table 2.4.1. Use of each game Keyboard Key.

2.5. Design Symbol

The design symbol of our general system is the following (see **Figure 2.5.1**).

Figure 2.5.1. System Design Symbol.*

* This design symbol is old. Keys that correspond to colours go from 0 to 6 and also the LR1 LED has not been used in the final version of our system.

2.6. Connections

The boards will communicate through a serial cable RS-232, using the three wires (Tx, Rx and GND) and plugged to the 9 PINS RS-232 board connector. In this project we will assume that there are no communication errors (to read more about this, see section *What If Situations*).

Regarding the system connections (please see **Figure 2.6.1**), it is also important to highlight the VGA connection between the two FPGAs and the two monitors available in the Laboratory. It is important to remember that baud rates must be the same for both FPGAs when establishing the connection.

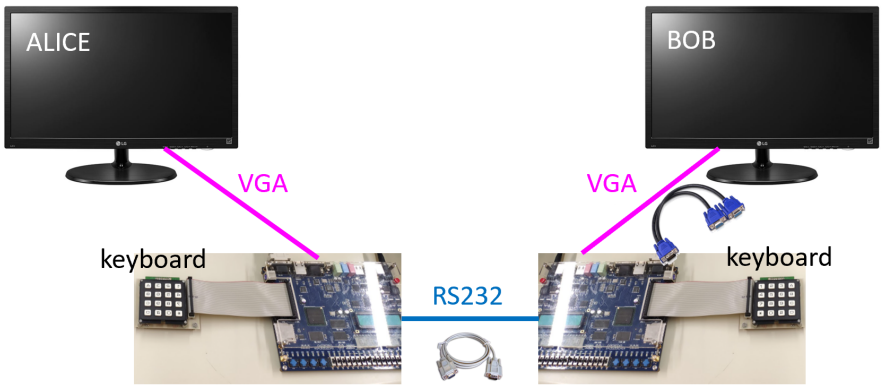


Figure 2.6.1. User Interface Scheme.

2.7. What If Situations

In the following lines, we are going to present some *What If Situations* that have been considered in the design of the system. We will also state some situations that have not been considered in the design of the system because, when doing simulations we have seen that situations are physically very improbable.

One player is identified as Alice, and the other player also wants to be identified as Alice. Therefore, this player presses the hash key.

We really wanted to make sure that if one player is identified as Alice, the other one can only be identified as Bob. It is very important that the roles of each board are clarified at the beginning of the game because the secret sequence is generated in Alice's board and also she is the one answering the questions related to the secret sequence. Therefore, if a user presses the hash key after the other user has already been identified as Alice, this user will enter in an **error state** (with the RED LED LR0 activated) and will have the opportunity to press either the hash or the ast key again. The same will happen until this user decides to press the ast key to be identified as Bob.

This is possible to prevent because when one user is identified as Alice, the other one receives the I AM ALICE frame and therefore knows that they can only be Bob. The same happens the other way round.

One player is identified as Bob, and the other player also wants to be identified as Bob. Therefore, this player presses the ast key.

The same happens as in the previous case. The user that has pressed the ast key after the other user has been identified as Bob will enter in an **error state** (with the RED LED LR0 activated) and will have the opportunity to press either the hash or the ast key again. The same will happen until this user decides to press the hash key to be identified as Alice.

The player proposes a not feasible guessing sequence. For example:

- **User presses [7] in case Alice has chosen not to have an empty peg in the secret sequence**
- **User repeats a color in case Alice has chosen not to have repeated colors**
- **User selects more than one empty peg (this is not a possible situation, see Table 2.2.1).**
- **User selects more than one repeated color (this is not a possible situation, see Table 2.2.1).**

We have finally agreed that, in these cases, the system is not going to complain or to report Alice or Bob about the error, as it is the duty of the users themselves to try to guess the secret sequence. If they want to include an empty peg when there is not an empty peg (and they already know it), it is their business to do so. The game is not designed to give hints to the players.

The two players press [#] to become Alice at the same time

We have checked in the simulations that in order to have a situation where the two players receive the I AM ALICE frame or the I AMB BOB frame, the two users should press the hash or the ast key with a time delay of the order of microseconds between each other. First of all, we have considered that in a normal game, this is not likely to happen. Also, if the objective of the users was to press the hash or the ast key at the very same time, it is true that the system could enter in a loop, but solving this is not the scope of our design, as we are aiming to build a prototype. Also, if this happens, the only thing that should be done is to restart both boards.

3. Block Diagram

3.1. Global System Description

In **Figure 3.1.1**, we can see the full system Block Diagram. However, as it is not possible to properly see all the signals in this image, we have done a more general Block Diagram by hand (see **Figure 3.1.2**) that makes it easier to understand how the whole system is connected. Also, when fully explaining the system, we will add some screenshots of particular parts of the Block Diagram from the *.bdf* file.

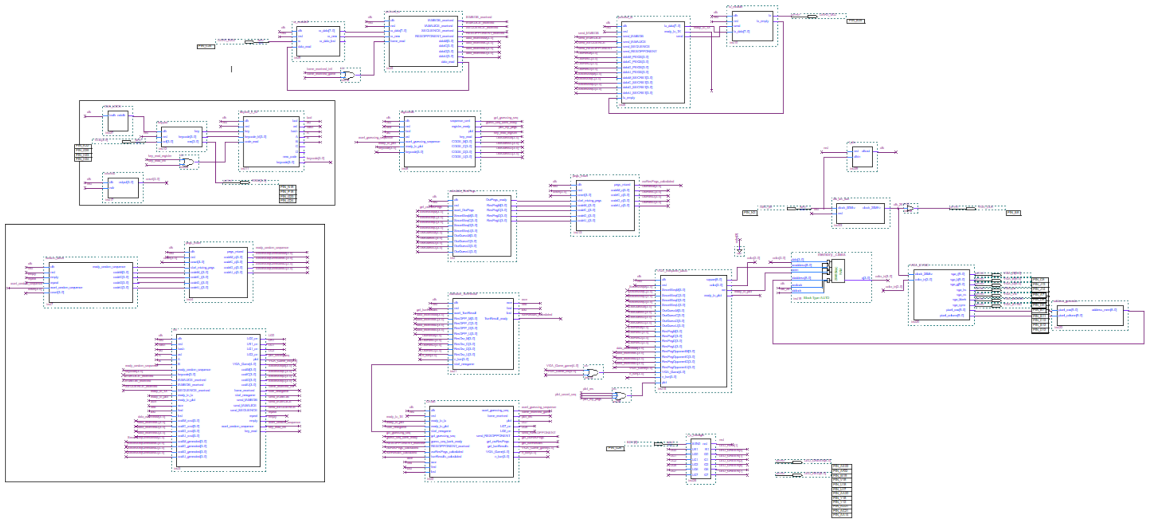


Figure 3.1.1. Full Block Diagram in Quartus Software (*block_diagram.bdf* file)

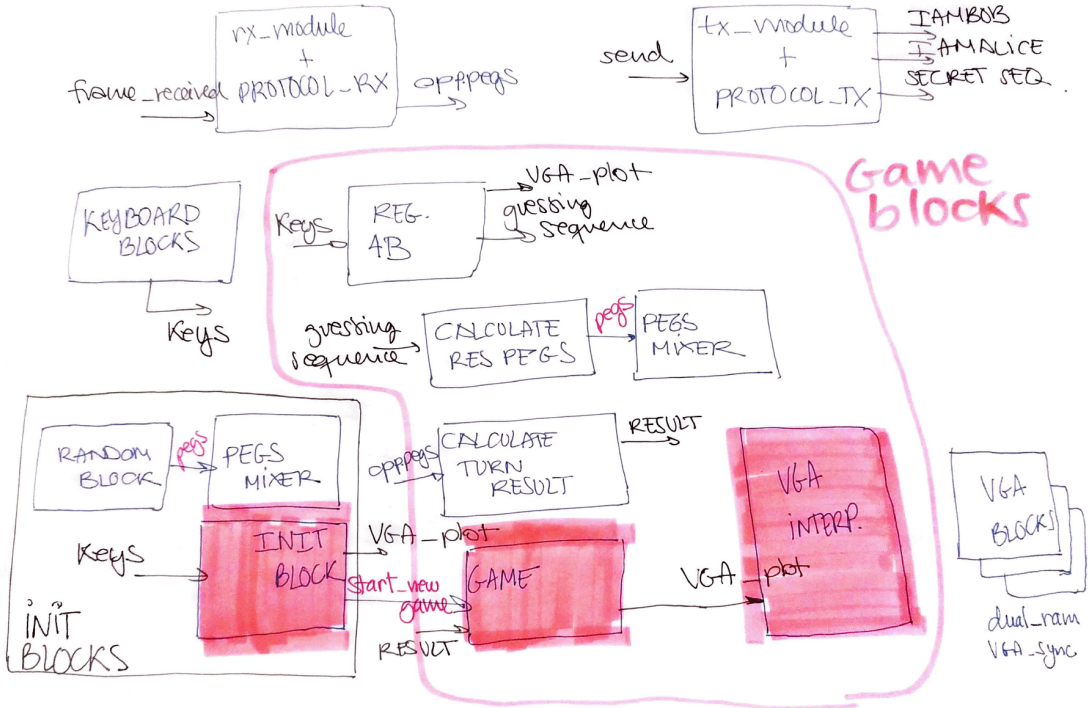


Figure 3.1.2. Summarized Block Diagram (done by hand)

Before starting, we would like to highlight that the only moment when the system differs between Alice and Bob is when they are in the Identifier Blocks. Basically, stages 2.a) and 2.b) are the parts when the system flux differs from both characters. However, as the system is symmetrical this is not any problem. In the following lines, we are going to explain in a lot of detail how the system works. We will describe the behaviour of our design through all the stages of the game.

Stage 1: System Init

The main blocks that take part in the System Init stage are the keypad (Keyboard UI), the Init Block, the block that creates a random sequence and the mixer block. Also, the VGA interpreter is used because there are some events that will have to be shown on the user screen.

A) Just to clarify → If a game has already happened before (AFTER STAGE 4 FINISHES):

If a game has already happened (the user has played Mastermind turns until the game has finished), independently of the game result (won/tied/lost), the user will have to press the # key in order to start again. When this key is pressed, the whole game will be reseted and everything will start again (LEDs will be turned off and the VGA screen will go background color entirely). It is important to highlight that LG6 and LG7 LEDs are turned off in the Game Block when the game has ended.

For example, LG1 and LG2 LEDS are deactivated here, because their function is to specify whether the user is Alice or Bob, but as we are starting a new game the user has still not a character associated with their FPGA.

B) If the system has just been initiated and we have not played yet:

When the system is initiated, the VGA Interpreter is notified of this need by the value of the 5 bits bus VGA_Game. The Main Block is waiting for the user to press either # or * in order to identify as a character. While the user has not been identified as one of the two characters, the LED LG0 will remain on, but at the moment the user has been identified as Alice or Bob, this LED will turn off. So far, this LED is turned on.

B.1) User wants to identify as Alice:

The user presses the # key because they want to be identified as Alice. We have to check if this is possible.

- **B.1.1) The opponent has already been identified as Alice:** We will know if this is the case because the input IAMALICE is high. We have to notify the user that the decision of being Alice is incorrect by turning on the LED LR1. This user, in this game, can only be identified as Bob. ***Start Stage 1 AGAIN***
- **B.1.2) The opponent has not been identified as Alice:** We will now if this is the case because the input IAMALICE is low. The LED LG0 will turn off (the character identification has finished) ***Move to 2.a): Character Identification as Alice***

B.2) User wants to identify as Bob:

The user presses the * key because they want to be identified as Bob. We have to check if this is possible.

- **B.2.1) The opponent has already been identified as Bob:** We will know if this is the case because the input IAMBOB is high. We have to notify the user that the decision of being Bob is incorrect by turning on the LED LR1. This user, in this game, can only be identified as Alice. ***Start Stage 1 AGAIN***
- **B.2.2) The opponent has not been identified as BOB:** We will know if this is the case because the input IAMBOB is low. ***Move to 2.b): Character Identification as Bob***

Stage 2.a): Character Identification as Alice

If the system flux is at this point, this means that the user has successfully been identified as Alice. The first thing that this block does is activate the output send_IAMALICE, in order to notify the opponent that this FPGA has been identified as Alice. Also, the LG1 LED is activated, indicating that this user is Alice. At this point, Alice will have to answer two questions related to the game difficulty. Now the VGA_Game 5 bits bus has to communicate to the VGA Interpreter that Alice has to answer the question: "**DO YOU WANT THE SECRET SEQUENCE TO HAVE REPEATED PEGS?**". Therefore, the square seen on the screen will change to *cyan*. To accept the proposal, the user has to press the A key on the keyboard and to decline it, the user has to press the B key on the keyboard (these are inputs of the Alice Identifier). After the user has answered this question, the VGA_Game bus will notify the VGA Interpreter that Alice has to answer the question: "**DO YOU WANT THE SECRET SEQUENCE TO HAVE EMPTY PEGS?**". Therefore, the square seen on the screen will change to yellow. To accept the proposal, the user has to press the A key on the keyboard and to decline it, the user has to press the B key on the keyboard. When the user has answered both questions, the Alice Identifier has enough information to generate the secret sequence. The sequence is generated randomly and mixed. After this, it is stored in the Init Block (that acts as a register for this matter). The Alice Identifier will activate the send_SEQUENCE output to ask the transmitter block to send the secret sequence to the opponent (who needs it, because they are Bob and they need the secret sequence to play). To do this, it is necessary to wait for the ready_to_tx input to be high, meaning that the transmitter bus is empty. Finally, to finish this stage, the start_newgame output is activated in order to start a new game (this pin is connected as an input of the Game Block). ***Move to 3: Mastermind Game***

As we have previously specified in the project SPECS document, both players have the secret sequence in their FPGA but they cannot access it because it is, as its name indicated, secret. For both cases (Alice and Bob), the secret sequence is stored in their identifier.

Stage 2.b): Character Identification as Bob

If the system flux is at this point, this means that the user has successfully been identified as Bob. The first thing that this block does is activate the output send_IAMBOB, in order to notify the opponent that this FPGA has been identified as Bob. Also, the LG2 LED is activated, indicating that this user is Bob.

The Bob Identifier block will wait until the input SEQUENCE_received is activated by the protocol rx block. At this point, the block will activate the frame_received output in order to turn the received flag low at the receiver block. The value of the received secret sequence will be stored in the output buses of the Init Block. Finally, to finish this stage, the start_newgame output is activated in order to start a new game (this pin is connected as an input of the Game Block). ***Move to 3: Mastermind Game***

Stage 3: Mastermind Game

If the user is at this point, this means that the start_newgame input of the Game (Mastermind) Block has been activated by the Init Block. At this point, the user has to notify the VGA Interpreter that a game has started and that therefore the whole screen has to be in background color. This is made by using the output VGA_Game of 5 bits. When the game starts, a series of steps will be repeated in a loop. This loop will end in case someone has guessed the sequence (either Bob or Alice) or in case the number of failed turns is 10 (maximum opportunities given by the game, based on the original classical Mastermind Game). In the following lines we can see the steps in the loop.

(TURN STARTS)

A) Ask the user for a guessing sequence (VGA interaction)

The Game Block waits for the Guessing Sequence Register to be ready to use (waiting for the input guess_seq_bank_ready to be high). When the bank is ready, the Game Block output called want_guessing_seq is activated and asks the register to store a new guessing sequence (the want_guessing_seq register input is activated). When this happens, the register stores the value of the keyboard keys pressed by the user in the OurGuess MCDU 4 bits buses. As we have previously explained, each number pressed in the keyboard corresponds to a specific color.

We have to remember that the LED LG6 must be activated while the register is working. This means that when the user is introducing the guessing sequence, this LED will be turned on. Also, the LG7 LED will be turned on during the time between the activation of the start_newgame input and the activation of any of the finish game outputs (won/tied/lost).

The VGA Interpreter reads and updates on the screen the value of the OurGuess MCDU buses each time a number is pressed and the signal plot is activated. Therefore, everytime the user presses a key, we see its correspondent color on the screen. When the user is happy with the guessing sequence that they see on the screen, they have to press the * key. At this moment, the sequence_sent output of the register is activated, activating the got_guessing_seq input of the Game Block. Due to this, the Game Block knows that it has access to the guessing sequence entered by the user and that the result pegs can be calculated.

B) Compute the feedback for the entered guessing sequence (Generate the Result pegs)

The result pegs of the user are calculated by comparing the value of the secret sequence stored in the Init block and the value of the guessing sequence currently stored in the Guessing Sequence register. When the pegs are computed, this means that the FPGA has all the necessary information to show on the screen the entered guessing sequence and the result pegs. Therefore, if ready_to_plot is active, the VGA_Game 5 bits bus and the activation of the plot at the Game Block output inform the VGA Interpreter that it has to show on the screen the guessing sequence and the result pegs of the user.

It is interesting to highlight the fact that, at this point, the guessing sequence of this turn will be shown on the screen twice because we have in the guessing sequence row and also in the turn row, but it will disappear from the guessing sequence row at the moment we move to the next turn and press a new number. This is a way to keep track of the previously entered sequences in order to help the user to guess the secret sequence.

C) Send user result pegs and wait for the result pegs of the opponent

At this moment, the FPGA has to send to the opponent the result pegs so that the opponent has them. The Game Block makes this possible by activating the send_RESOPPONENT flag, which tells to the transmitter block that it is necessary to send the result pegs through the transmitter block (this can be done after making sure that the ready_to_tx input is high in a waiting state).

The Game Block enters in a new wait state that waits for the opponent result pegs to arrive at the receiver block. The Game Block is notified when this happens because the flag RESULTOPPONENT_received is activated by the receiver block. When this happens, the Game Block has to deactivate this receiver flag by activating the frame_received output.

D) Show on the screen the opponent result pegs

At this point, the result pegs of the opponent are located in the MCDU 4 bits buses of the receiver. The VGA Interpreter knows that it has to show on the screen the opponent result pegs due to the value of the VGA_Game output of the Game Block and the activation of the plot output of the Game Block, in case that ready_to_plot is active. The VGA Interpreter gets the value of the result pegs of the opponent and plots it. Then, the Game block will check if the game has finished or if it continues by using the block calculate_TurnResult, which checks if someone has won, loose or there is a tie. If the game finishes, the Init Block will know the result of the game with the won/lost/tied inputs. The LED LG7 is turned off because the game has finished.

(TURN FINISHES → Go back to A) in case nobody has guessed the secret sequence OR in case this turn was the 10th one)

Move to 4: Finished Game

Stage 4: Finished Game

If the system is at this point this means that the game has finished, which also means that one of the three finishing signals has been activated (won/lost/tied). By the activation of these signals, we are moving from working in the Game Block to working in the Init Block. The VGA_Game 5 bits bus informs the VGA Interpreter that the game result has to be shown on the screen. We see a square in the top right of the screen which indicates: "**YOU WIN**" or "**GAME OVER**" or "**IT'S A TIE**", and is represented in green, red, and blue, respectively. Also, on top of the screen we see the value of the correct sequence (the secret sequence), which is still stored in the Init block. Now we are at the moment when we want to start a game again, which corresponds to Stage 1.A).

3.2. Description of Individual Blocks

We consider it important to highlight that inputs such as *nrst* and *clk* will not be described in each block due to being reiterated and therefore not necessary information about the system.

Keyboard Block: keypad and keytest

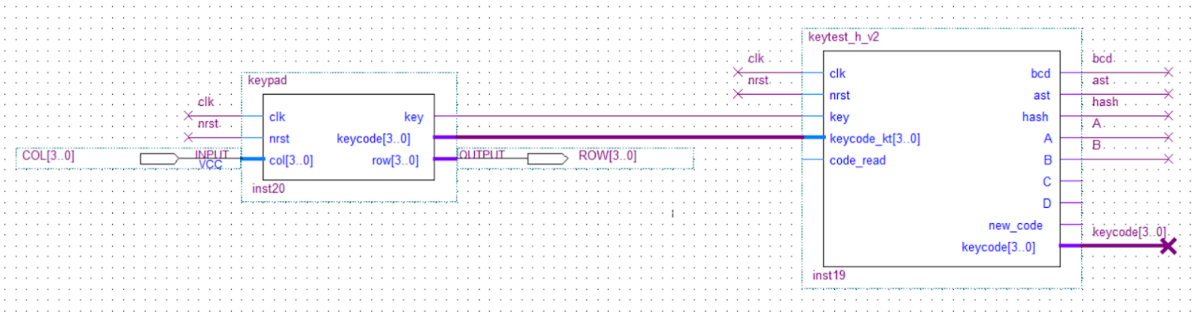


Figure 3.2.1 keypad and keytest blocks (*block_diagram.bdf* file from Quartus Software).

INPUTS:

- col[3..0] **(4 bits)**: signal to connect to the external keyboard.

OUTPUTS:

- row[3..0] **(4 bits)**: signal to connect to the external keyboard.
- bcd: flag signal that indicates when the user presses a number between 0 and 9 in the keyboard.
- ast: flag signal that indicates when the user presses * in the keyboard.
- hash: flag signal that indicates when the user presses # in the keyboard.
- A, B, C, D: flag signal that indicates when the user presses each letter in the keyboard.
- keycode[3..0] **(4 bits)**: code of the key pressed in hexadecimal. When the user presses any key from 0 to 9, keycode contains the corresponding hexadecimal code. * has de code E and # has de code F.

Register Bank

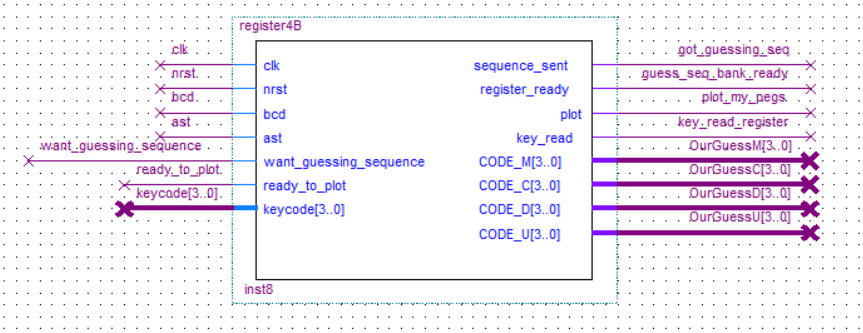


Figure 3.2.2. register bank blocks (*block_diagram.bdf* file from Quartus Software).

This register stores the guessing sequence entered by the user in each turn until the next turn starts or the game ends. In this design, there is only one register because there is only one sequence stored by the user.

INPUTS:

- bcd: flag that is activated when the user maintains a bcd key pressed. Everytime the user presses a bcd key, this flag will be activated and the bcd value (equivalent to one of the colors of the game) will be stored in the MCDU buses. Therefore, the color assigned to the bcd value will appear in the guessing sequence.
- ast: flag that is activated when the user maintains the [*] key pressed. The user must press it to send the guessing sequence that is currently stored in the MCDU buses. This means the user has to press the [*] key when they are sure about the introduced guessing sequence. When this input is activated, the register's output named *sequence_sent* is also activated to notify the Game Block that the sequence has been introduced.
- want_guessing_sequence: activated by the GAME block when the user can introduce the guessing sequence in the current turn. When this input is activated, the register starts 'functioning' and therefore the user is able to introduce the guessing sequence.
- ready_to_plot: this input comes from the VGA_interpreter block and is active when it is ready to plot a new object.
- keycode[3..0] **(4 bits bus)**: this 4 bits bus is an output of the Keyboard Block and contains the code of the key that is being pressed by the user. It is useful for our design because it contains the bcd value of the pressed key, and each bcd value corresponds to one color in the sequence.

OUTPUTS:

- sequence_sent: this output is activated when the user has introduced the guessing sequence and pressed [*]. It notifies the Game Block that the turn can continue because the user has already chosen their guessing sequence.
- register_ready: this output is activated when the register bank is ready to store a new sequence. This pin is read by the Game Block, who waits for it to be high and then asks the register for a new guessing sequence.
- key_read: this output is activated to inform the keyboard that the key has been read.

- plot: this output is connected to the VGA_interpreter to inform that it wants to plot the current guess each time the user presses a new color.
- CODE_M **(4 bits bus)**: 4 bits bus storing Peg A in the guessing sequence.
- CODE_C **(4 bits bus)**: 4 bits bus storing Peg B in the guessing sequence.
- CODE_D **(4 bits bus)**: 4 bits bus storing Peg C in the guessing sequence.
- CODE_U **(4 bits bus)**: 4 bits bus storing Peg D in the guessing sequence.

Transmitter Block

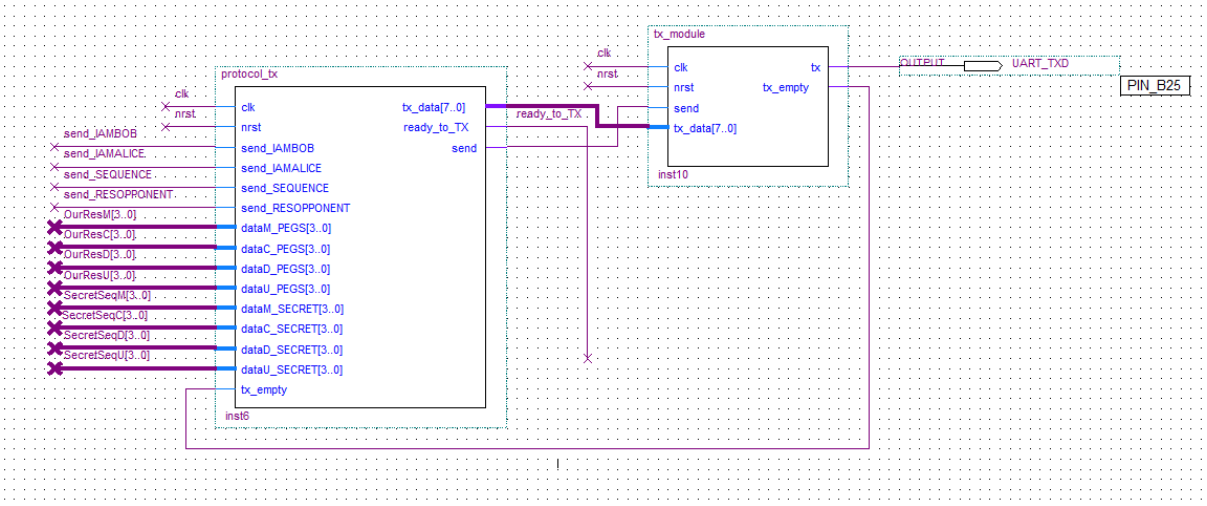


Figure 3.2.3. protocol_tx and tx_module blocks (*block_diagram.bdf* file from Quartus Software).

a. protocol_tx

INPUTS

- send_IAMBOB: Active high. Signal that, when active, it means the frame “IAMBOB” has to be sent.
- send_IAMALICE: Active high. Signal that, when active, it means the frame “IAMALICE” has to be sent.
- send_SEQUENCE: Active high. Signal that, when active, it means the frame “SEQUENCE” has to be sent.
- send_RESOPPONENT: signal that, when active, it means the frame “RESOPPONENT” has to be sent.
- dataM[3..0] **(4 bits)**: Bus of 4 bits where the Peg A or Peg 1 is transmitted, in case the frame to transmit is “SEQUENCE” or “RESOPPONENT”.
- dataC[3..0] **(4 bits)**: Bus of 4 bits where the Peg B or Peg 2 is transmitted, in case the frame to transmit is “SEQUENCE” or “RESOPPONENT”.
- dataD[3..0] **(4 bits)**: Bus of 4 bits where the Peg C or Peg 3 is transmitted, in case the frame to transmit is “SEQUENCE” or “RESOPPONENT”.
- dataU[3..0] **(4 bits)**: Bus of 4 bits where the Peg D or Peg 4 is transmitted, in case the frame to transmit is “SEQUENCE” or “RESOPPONENT”.

- tx_empty: When low, it indicates that the UART is currently transmitting data and consequently busy. If high, it indicates that the UART is not transmitting data (i.e. the UART is idle).

OUTPUTS

- tx_data[7..0] **(8 bits)**: Data byte to be transmitted.
- ready_to_TX: Active high. When active it means that the transmitter is not transmitting any frame and is ready to be used.
- send: Synchronous signal, active high. It should be activated one clock period to start a new transmission (awakening signal). This signal can only be set to one when the output TX_EMPTY is high, i.e., when the UART is not currently transmitting any data.

b. tx_module

INPUTS:

- send: Synchronous signal, active high. It should be activated one clock period to start a new transmission (awakening signal). This signal can only be set to one when the output TX_EMPTY is high, i.e., when the UART is not currently transmitting any data.
- tx_data[7..0] **(8 bits)**: Data byte to be transmitted.

OUTPUTS:

- tx: Serial output. The serial transmission protocol is: START BIT (low level) + PAYLOAD (8 data bits, LSB first) + STOP BIT (high Level). While idle (not transmitting), TX is set to high level. The speed of the serial transmission can be configured by setting internal parameters. As a matter of fact, this configuration sets how many clock cycles a transmitted bit is held at the output (see waveform below). The figure shows that each bit transmitted is held for N clock cycles. This implies that for transmitting a byte, it is required (10*Clock period) seconds (10 = 8 payload bits + start bit + stop bit).
- tx_empty: When low, it indicates that the UART is currently transmitting data and consequently busy. If high, it indicates that the UART is not transmitting data (i.e. the UART is idle).

Receiver Block

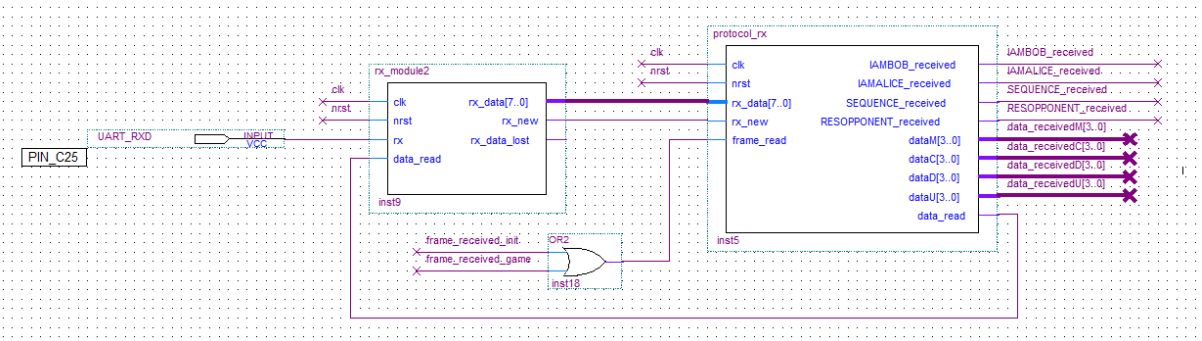


Figure 3.2.4. protocol_rx and rx_module blocks (*block_diagram.bdf* file from Quartus Software).

a. protocol_rx

INPUTS:

- rx_data[7..0] **(8 bits)**: Payload of the last byte received. To be read by another block.
- rx_new: Flag signal (active high). It is active when a new byte is received. This flag signal deactivates when the input DATA_READ is activated for one clock period.
- frame_read: ACK signal that acknowledges that the block “GAME” has received the frame “RESOPPONENT” and that the block “Init” has received the frame “IAMALICE”, “IAMBOB” or “SEQUENCE”.

OUTPUTS:

- IAMBOB_received: Active high. When active means that the frame “IAMBOB” has been received. Signal connected to “main_block” and “Alice_identifier”.
- IAMALICE_received: Active high. When active means that the frame “IAMALICE” has been received. Signal connected to “main_block”.
- SEQUENCE_received: Active high. When active means that the frame “SEQUENCE” has been received. Signal connected to “Bob_identifier”.
- RESOPPONENT_received: Active high. When active means that the frame “RESOPPONENT” has been received.
- dataM[3..0] **(4 bits)**: Bus of 4 bits where the Peg A or Peg 1 is transmitted, in case the frame to transmit is “SEQUENCE” or “RESOPPONENT”.
- dataC[3..0] **(4 bits)**: Bus of 4 bits where the Peg B or Peg 2 is transmitted, in case the frame to transmit is “SEQUENCE” or “RESOPPONENT”.
- dataD[3..0] **(4 bits)**: Bus of 4 bits where the Peg C or Peg 3 is transmitted, in case the frame to transmit is “SEQUENCE” or “RESOPPONENT”.
- dataU[3..0] **(4 bits)**: Bus of 4 bits where the Peg D or Peg 4 is transmitted, in case the frame to transmit is “SEQUENCE” or “RESOPPONENT”.
- data_read: This input should be activated by the block that reads the information received by this UART, to acknowledge that the last received byte has been read. Synchronous signal, active high. This signal should be active one clock period.

b. rx_module2

INPUTS:

- rx: Serial input. The serial transmission protocol is: START BIT (low level) + PAYLOAD (8 data bits, LSB first) + STOP BIT (High Level). As the transmitter sets the serial line to high level when the transmitter is idle, the receiver activates when detects that the input serial line goes to zero. The speed of the serial transmission can be configured by setting internal parameters. As a matter of fact, this configuration sets how many clock cycles a transmitted bit is supposed to be at the input. The configuration parameter must be the same at both the receiver and transmitter. As in the transmitter, if a bit is supposed to be held at the input for N clock cycles, once the UART is activated, it takes (N*10 clock period) seconds to finish the reception of a byte.
- data_read: This input should be activated by the block that reads the information received by this UART, to acknowledge that the last received byte has been read. Synchronous signal, active high. This signal should be active one clock period.

OUTPUTS:

- rx_data[7..0] **(8 bits)**: Payload of the last byte received. To be read by another block.
- rx_new: Flag signal (active high). It is active when a new byte is received. This flag signal deactivates when the input DATA_READ is activated for one clock period.

Init Block

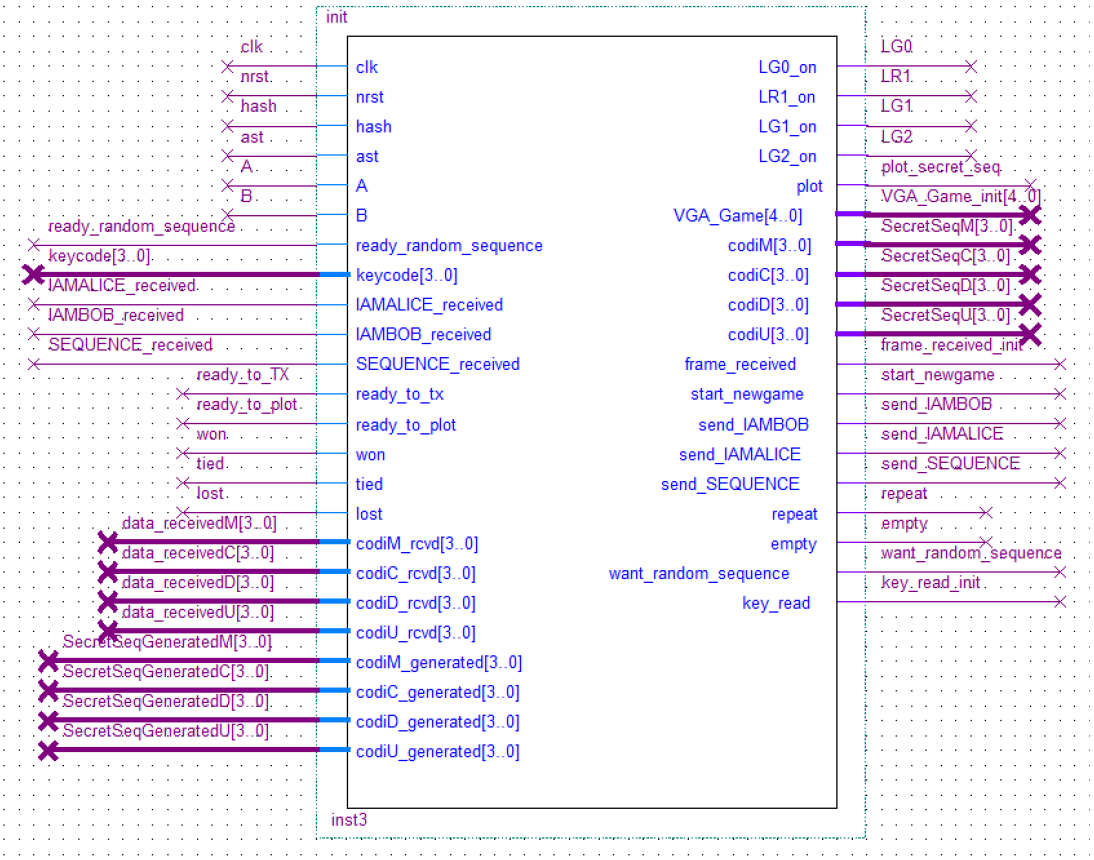


Figure 3.2.5. Alice_identifier block (*block_diagram.bdf* file from Quartus Software).

The aim of this block is to make possible that the user identifies themselves as the character Alice. It is connected to the Main Block and to the Mastermind/Game Block.

INPUTS:

- hash: activated when the user presses the hash key.
- ast: activated when the user presses the ast key.
- ready_random_sequence: activated when the random sequence has been generated and mixed.
- keycode[3..0]: just in case we want to check the value of the key in an exhaustive manner.
- IAMALICE_received: activated when the other player is Alice.
- IAMBOB_received: activated when the other player is Bob.
- SEQUENCE_received: activated when we are Bob and we have received the sequence from Alice's board.

- ready_to_plot: activated when the VGA interpreter is telling us that the VGA is free to plot whatever we want.
- won/tied/lost: activated when the game is finished.
- codiX_rcvd[3..0]: these inputs contain the value of the received secret sequence (if we are Bob).
- codiX_generated[3..0]: these inputs contain the value of the generated secret sequence (if we are Alice).
- ready_to_tx: this input is activated when the transmitter block is ready to send data to the opponent (the tx bus is empty because it is not working right now).
- A: this input is activated when the key [A] is being pressed by the user (this will be used when Alice is answering the questions shown on screen, in order to accept the difficulty options).
- B: this input is activated when the key [B] is being pressed by the user (this will be used when Alice is answering the questions shown on screen, in order to decline the difficulty options).

OUTPUTS:

- LG0_on: activated when the user can still choose who is their character.
- LR1_on: activated when the user wants to be Alice and the other character is already Alice (or the other way round). See the section *What If Situations*.
- LG1_on: Activated when the user is Alice.
- LG2_on: Activated when the user is Bob.
- plot: Activated when we want to plot something from this block.
- VGA_Game[4..0] **(5 bits)**: Contains information related to what we want to plot from this block.
- codiM[3..0], codiC[3..0], codiD[3..0], codiU[3..0] **(4 bits)**: it contains the secret sequence (no matter if we are Alice or Bob, this block functions as a register as it is storing the secret sequence in these outputs).
- frame_received: activated when we have received a frame and we have also processed it.
- start_newgame: activated when we want to start a new game (this is done when the character identification has been finished).
- send_IAMBOB: this is activated when the user has press * and we want to inform the other user about this.
- send_IAMALICE: this is activated when the user has press # and we want to inform the other user about this.
- send_SEQUENCE: activated when we are Alice and we want to send to the other user the secret sequence that has been generated in our board.
- repeat: activated when the answer to the first question is ‘A’ and therefore there is one repeated peg.
- empty: activated when the answer to the second question is ‘B’ and therefore there is one empty peg.
- want_random_sequence: activated when Alice has been identified and has answered the two questions, so we want the secret sequence generated.
- key_read_init: activated when we have read a key from the keyboard.

Game Block

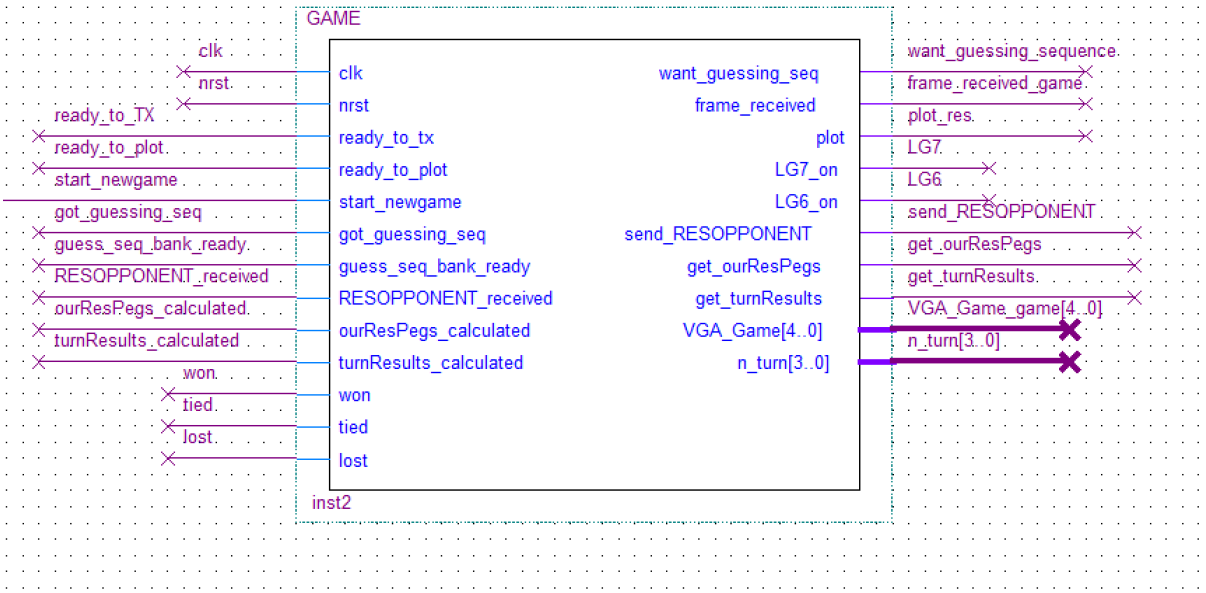


Figure 3.2.6 mastermind_block (*block_diagram.bdf* file from Quartus Software).

This block is one of the main system blocks, as it directs the functioning of the game itself (this is why it is called the Game/Mastermind Block). It is connected to both character identifiers (Alice and Bob), to the guessing sequence register, to the receiver and transmitter blocks and to the VGA interpreter.

INPUTS:

- start_newgame: this input is activated by one of the character identifiers (the one corresponding to the character of the FPGA). It indicated the moment when a new game can start.
- got_guessing_seq: it is activated by the register bank and tells the Game Block that the user has already introduced the guessing sequence of the current run.
- guess_seq_bank_ready: it is activated by the register bank when it is ready to store a new guessing sequence. The Game Block asks the register bank for a new guessing sequence when this input is high (meaning that the bank is free for us).
- RESULTOPPONENT_received: this flag is high when the frame with the result pegs from the opponent has arrived to the receiver block. When this flag is activated, the output frame_received has to be also activated to indicate to the receiver block that the RESULTOPPONENT_received flag can be deactivated.
- ready_to_tx: this input is activated when the transmitter block is ready to send data to the opponent (the tx bus is empty because it is not working right now).
- ready_to_plot: activated when the VGA interpreter is telling us that the VGA is free to plot whatever we want.
- ourResPegs_calculated: activated when the block calculate_ResPegs has finished calculating our result pegs.
- turnResults_calculated: activated when the block calculate_turnRes has finished calculating our turn results.
- won: activated when the game has finished and we have won.

- tied: activated when the game has finished and we have tied.
- lost: activated when the game has finished and we have lost.

OUTPUTS:

- LG7_on: this output is connected to the I/O manager indicating when the LG7 LED must be on. The LG7 must be on when the game is in progress (during all the game turns, from the moment when start_newgame is activated to the moment any of the finished game signals - won, tied, lost- is activated).
- LG6_on: this output is connected to the I/O manager indicating when the LG6 LED must be on. The LG6 must be on when the user can introduce a new guessing sequence, which means when the register bank is being used.
- frame_received: this output is activated in order to turn the flag RESULTOPPONENT_received off in the receiver block, notifying the receiver that the Game Block has already been notified with the fact that the FPGA has received the result pegs from the opponent.
- plot: Activated when we want to plot something from this block.
- VGA_Game[4..0] **(5 bits)**: Contains information related to what we want to plot from this block.
- want_guessing_seq: activated when we want to ask the register to store the following guessing sequence introduced by the user.
- n_turn: the value of this output corresponds to the number of the current turn.
- want_ourRespegs: activated when we want to ask the block calculate_resPegs for the result of our guess (our res pegs).
- want_turnResult: activated when we want to ask the block calculate_turnResult for the result of the current turn.
- send_RESOPPONENT: activated when we want to send our result pegs to the opponent user.

calculate_ResPegs

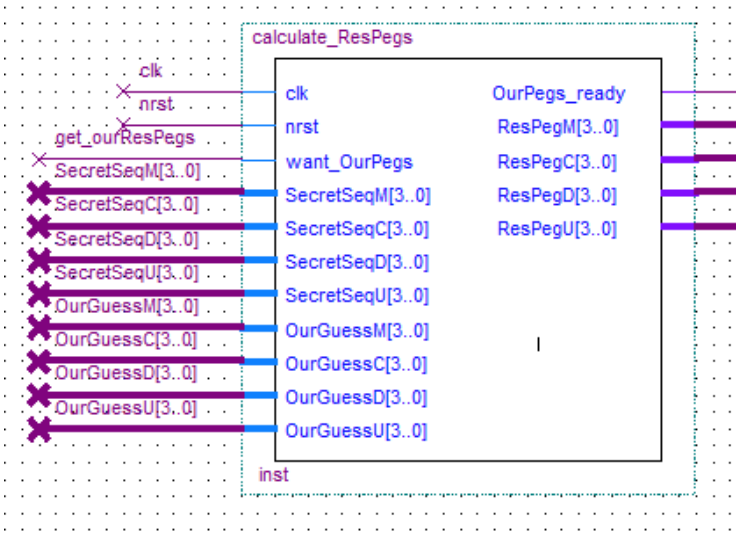


Figure 3.2.7 calculate_ResPegs (*block_diagram.bdf* file from Quartus Software).

This block calculates the result pegs of the user guess code.

INPUTS:

- want_OurPegs: signal activated by the block “GAME” when it wants to calculate the result pegs of the user guess.
- SecretSeqM[3..0], SecretSeqC[3..0], SecretSeqD[3..0], SecretSeqU[3..0] **(4 bits)**: values of the secret sequence from “Init”.
- OurGuessM[3..0], OurGuessC[3..0], OurGuessD[3..0], OurGuessU[3..0] **(4 bits)**: values of the user guess from the “Register”.

OUTPUTS:

- OurPegs_ready: this signal is activated when the result pegs have been calculated and the “pegs_mixer” block can begin with the randomization of its positions.
- ResPegM[3..0], ResPegC[3..0], ResPegD[3..0], ResPegU[3..0] **(4 bits)**: values of the result pegs of the user.

calculate_TurnResult

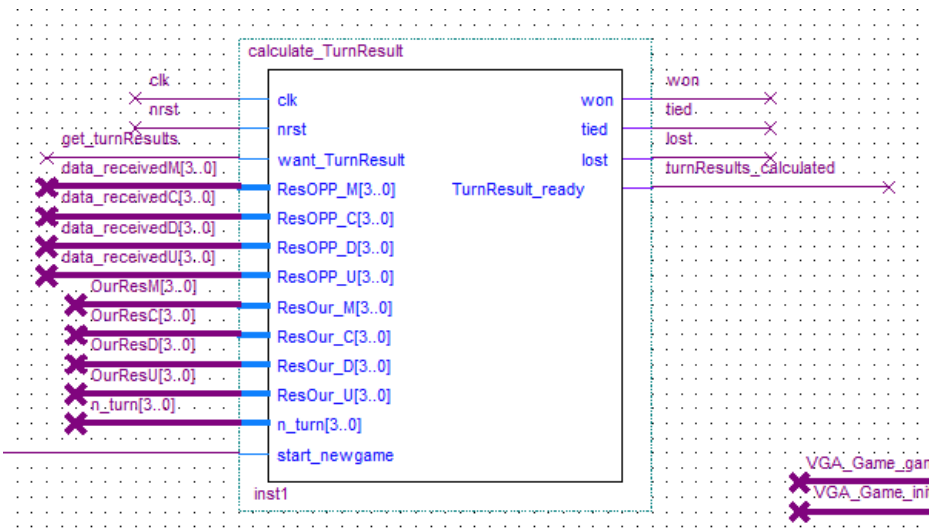


Figure 3.2.8 calculate_TurnResult (*block_diagram.bdf* file from Quartus Software).

This block checks if the user has won/loose or if there is a tie.

INPUTS:

- want_TurnResult: this signal is activated from the block “GAME” and, when activated, this block starts with the measurement.
- ResOPP_M[3..0], ResOPP_C[3..0], ResOPP_D[3..0], ResOPP_U[3..0] **(4 bits)**: values of the result pegs of the opponent.
- ResOur_M[3..0], ResOur_C[3..0], ResOur_D[3..0], ResOur_U[3..0] **(4 bits)**: values of the result pegs of the user.
- n_turn[3..0] **(4 bits)**: This signal indicates in which turn of the game is the user playing, from 1 to 10. If it is not in the game, it’s value is 0.

- start_newgame: this signal is activated when there is a new game, so this block can be reinitialized.

OUTPUTS:

- won: if the user has the correct result pegs and the opponent does not.
- tied: if both the user and the opponent has the correct result pegs.
- lost: if the opponent has the correct result pegs and the user does not.
- TurnResult_ready: when this block has finished measuring the results of this turn, it activates this signal.

random_block

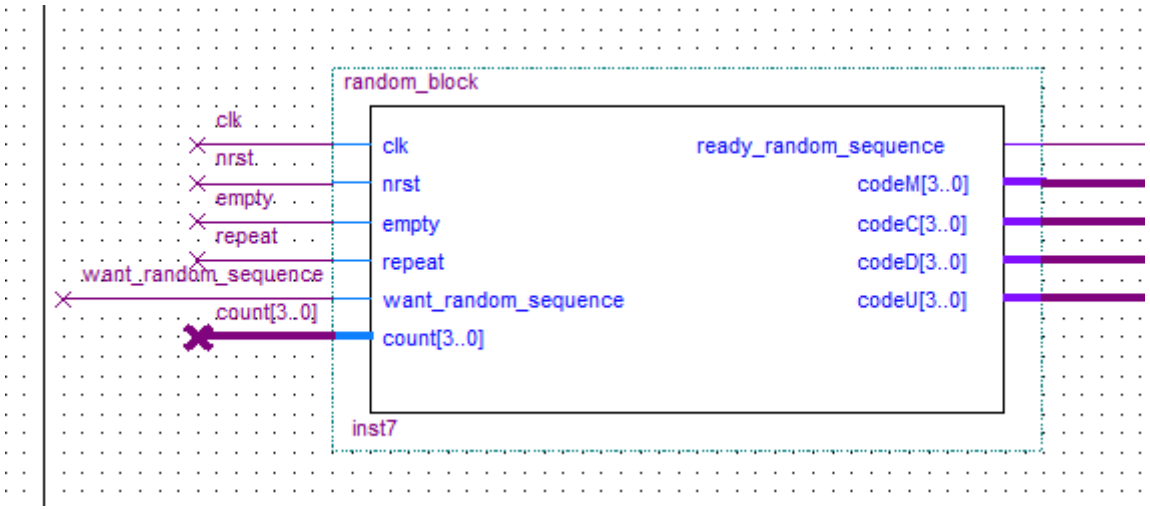


Figure 3.2.9 random_block (*block_diagram.bdf* file from Quartus Software).

This block generates a random sequence of pegs for the secret sequence, taking into account if the options empty and repeat are active. A mixing of the pegs is necessary to obtain a complete random sequence.

INPUTS:

- empty: if the option “empty peg” was selected, this signal is high.
- repeat: if the option “repeated peg” was selected, this signal is high.
- want_random_sequence: when the block “init” wants to generate a secret sequence, it activates this signal.
- count[3..0] **(4 bits)**: counter used for the randomization.

OUTPUTS:

- ready_random_sequence: when the sequence has been generated, this signal is activated to let the “pegs_mixer” block that it can start mixing the pegs.
- codeM[3..0], codeC[3..0], codeD[3..0], codeU[3..0] **(4 bits)**: values of the secret sequence that are sent to the “pegs_mixer” block.

pegs_mixer

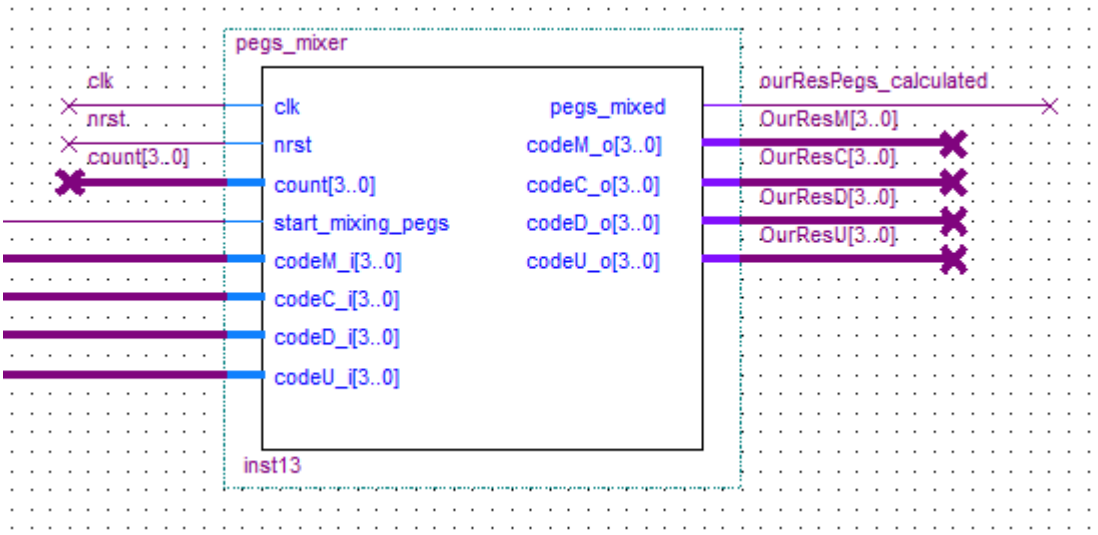


Figure 3.2.10 pegs_mixer (*block_diagram.bdf* file from Quartus Software).

This block mixes the input pegs to obtain a randomized sequence. It is used in two places, to mix the secret sequence generated by the “random_block”, and to mix the result pegs of the user in the block “calculate_ResPegs”.

INPUTS:

- count[3..0] **(4 bits)**: counter used for the randomization.
- start_mixing_pegs: when the “random_block” or the “calculate_ResPegs” block have finished, they activate this signal to start with the mixing of the pegs.
- codeM_i[3..0], codeM_i[3..0], codeM_i[3..0], codeM_i[3..0] **(4 bits)**: values of the sequence not mixed.

OUTPUTS:

- pegs_mixed: when this block has finished with the mixing, it activates this signal to let the “init” block or the “GAME” block that the sequence is mixed and ready.
- codeM_o[3..0], codeM_o[3..0], codeM_o[3..0], codeM_o[3..0] **(4 bits)**: mixed values of the sequence.

Video Graphics Array (VGA)

a. VGA_interpreter

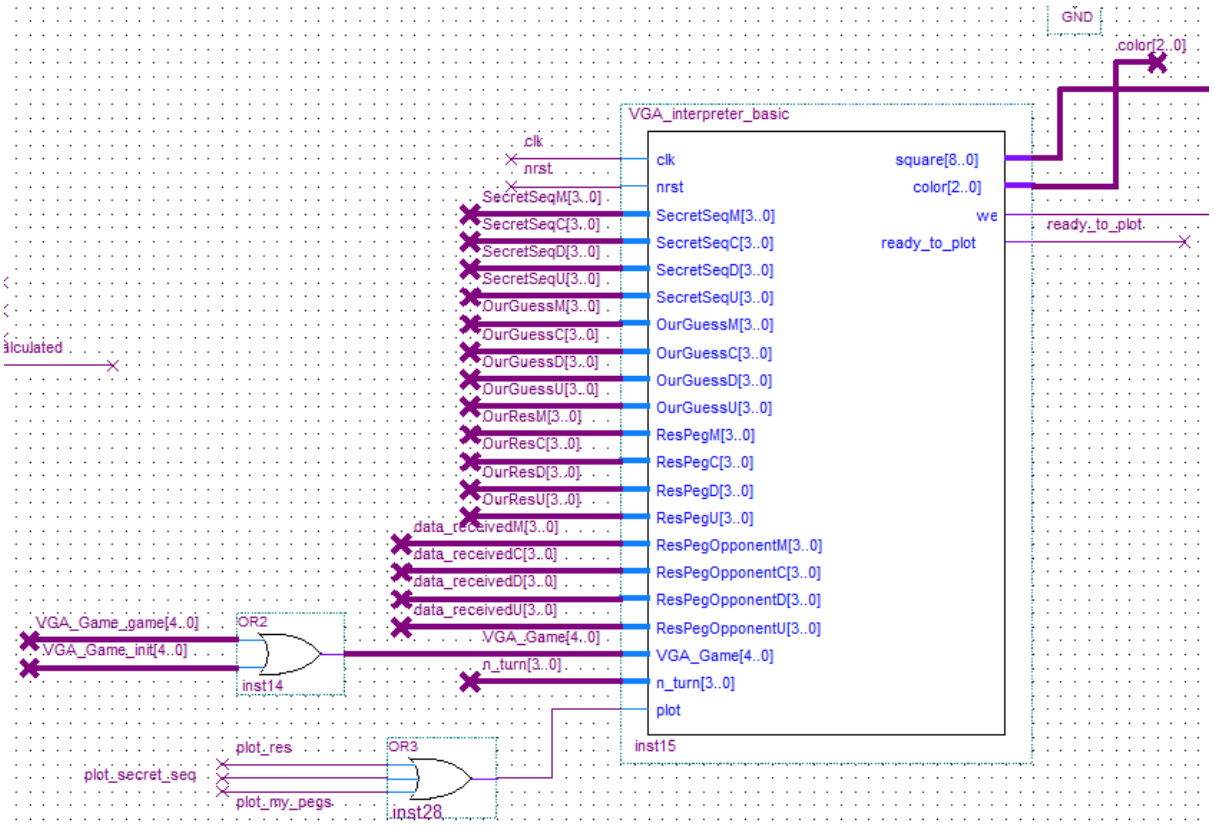


Figure 3.2.11 VGA_interpreter_basic (*block_diagram.bdf* file from Quartus Software).

This block takes the secret sequence, guess code, result pegs and opponent result pegs and consecutively sends for each square the corresponding color. This block is aware of the current turn so it knows in which position a given sequence and the results should be plotted. It also displays the initial screen and the questions, represented in the top left square, and the game over screen, represented in the top right square.

INPUTS:

- SecretSeqM[3..0], SecretSeqC[3..0], SecretSeqD[3..0], SecretSeqU[3..0] **(4 bits)**: values of the colors of the 4 pegs of the user code, i.e., the correct code generated by Alice. These signals come from Alice_identifier if the user is Alice or from Bob_identifier if the user is Bob.
- OurGuessM[3..0], OurGuessC[3..0], OurGuessD[3..0], OurGuessU[3..0] **(4 bits)**: values of the colors of the 4 pegs of the current guess code. These signals are the 4 last colors that the user selected. The signals come from the Guessing Sequence Register and are updated each time the user presses a new color when the user has to introduce the guess code.
- ResPegM[3..0], ResPegC[3..0], ResPegD[3..0], ResPegU[3..0] **(4 bits)**: These are the result pegs of the user that come from the Game block.

- ResPegOpponentM[3..0], ResPegOpponentC[3..0], ResPegOpponentD[3..0], ResPegOpponentU[3..0] **(4 bits)**: These are the results pegs of the opponent that is received in the Receiver Block.
- VGA_Game[4..0] **(5 bits)**: This signal indicates in which phase the game currently is. When the plot signal is activated, this value is read in order to know what has to be plotted. The phases of the game are, from 1 to 10: STARTING SEQUENCE, QUESTION 1, QUESTION 2, BLACK, PLOT GUESSING + OUR RESULT, PLOT OPPONENT RESULT, PLOT CURRENT GUESS, YOU WIN, IT'S A TIE, YOU LOOSE.
- n_turn[3..0] **(4 bits)**: This signal indicates in which turn of the game is the user playing, from 1 to 10. If it is not in the game, it's value is 0.
- plot: Active high. This signal is active when the block GAME, Init or Register sends the signal plot in order to plot any type of object.

OUTPUTS:

- square[8..0] **(9 bits)**: This value indicates the address of the square to be plotted. There are a total of 15x20=300 squares on the screen. This signal is connected to the "dual_ram".
- color[2..0] **(3 bits)**: This value indicates the color that corresponds to the signal "square[14..0]". Connected to "dual_ram".
- we: Active high. If active means that the two signals above are written in "dual_ram". Connected to "dual_ram".
- ready_to_plot: this signal indicates that this block is ready to plot something new. Connected to "GAME", "Init" and "Register".

b. dual_ram

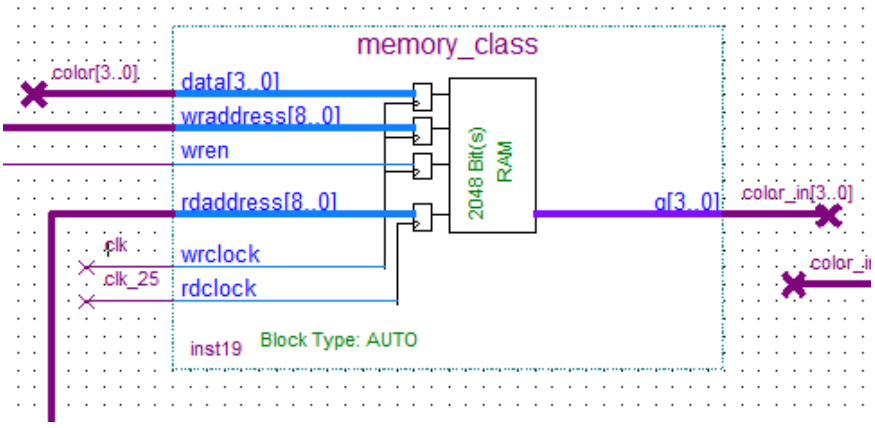


Figure 3.2.12 dual_RAM (*block_diagram.bdf* file from Quartus Software).

This block is used to write and read the different colors for each square in a RAM. The block "VGA_interpreter" stores data in this RAM and the block "VGA_SYNC" reads from this RAM to know the color at each pixel of the screen. This block has two clocks, the clock of the writing and the clock of the reading.

INPUTS

- wraddress[8..0] **(9 bits)**: address of the square from "VGA_interpreter".

- data[2..0] **(3 bits)**: color of the square from “VGA_interpreter”.
- wren: write enable from the “VGA_interpreter”. If active saves the color of the square from “VGA_interpreter” to the RAM at the specified address.
- rdaddress[8..0] **(9 bits)**: Signal from “address_generator” in which is indicated the square address so the “VGA_SYNC” can obtain the correspondent color.

OUTPUTS

- q[3..0] **(4 bits)**: Signal connected to “VGA_SYNC”. The signal is the correspondent color from the address that the “VGA_SYNC” asks. Only the last 3 bits are read.

c. VGA_SYNC

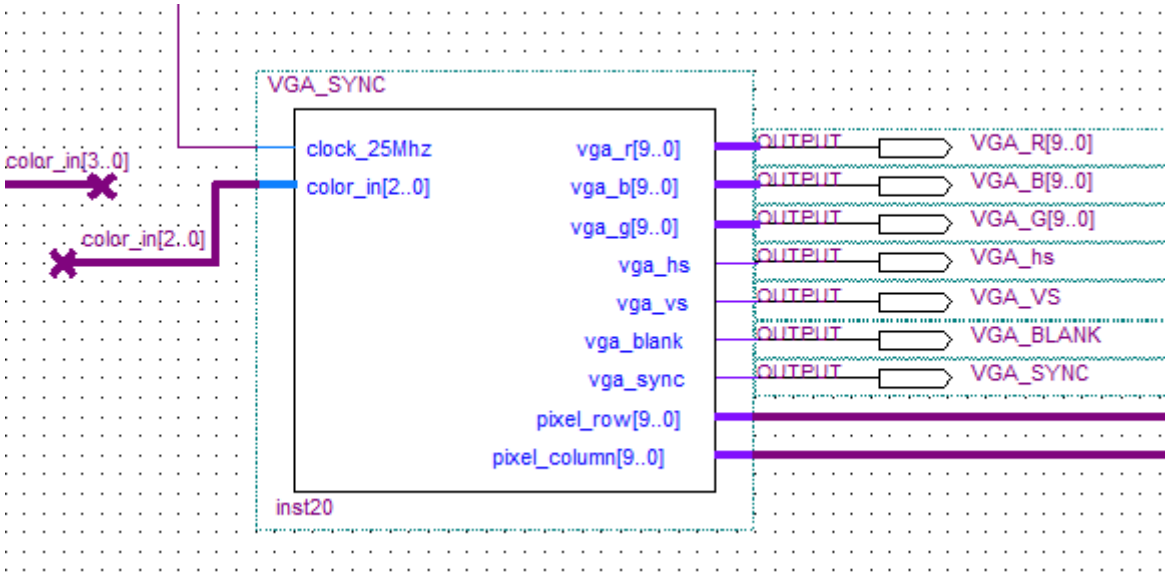


Figure 3.2.13 VGA_SYNC (*block_diagram.bdf* file from Quartus Software).

This block is the one connected directly to the screen. It displays synchronously the different colors for each pixel in the screen. It informs the block “address_generator” of the next pixel to display, and then the value is received from the block “dual_ram”.

INPUTS

- color_in[2..0] **(3 bits)**: color that will be shown in the screen in the pixel that was indicated in “pixel_row” and “pixel_column”. Connected to “dual_ram”.

OUTPUTS

- vga_r[9..0], vga_b[9..0], vga_g[9..0] **(10 bits)**: colors displayed in the screen in the correspondent pixel which depends of the time.
- vga_hs: horizontal synchronizer.
- vga_vs: vertical synchronizer.
- vga_blank: when active, all the screen is blank.
- vga_sync:
- pixel_row[9..0] **(10 bits)**: row value of the next pixel to show in the screen.
- pixel_column[9..0] **(10 bits)**: column value of the next pixel to show in the screen.

d. address_generator

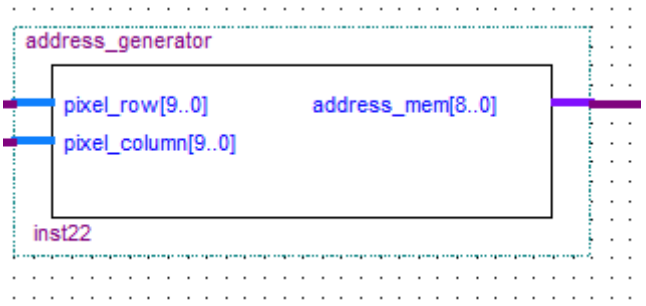


Figure 3.2.14 address_generator (*block_diagram.bdf* file from Quartus Software).

This block takes the value of the row and column of a pixel and gives the corresponding address of the square. It just takes the more significant bits of the row (4 bits) and of the column (5 bits).

INPUTS

- pixel_row[9..0] **(10 bits)**: row value of the next pixel to show in the screen.
- pixel_column[9..0] **(10 bits)**: column value of the next pixel to show in the screen.

OUTPUTS

- address_mem[8..0] **(9 bits)**: address that contains the information of the pixel in a given row and column.

User Interface

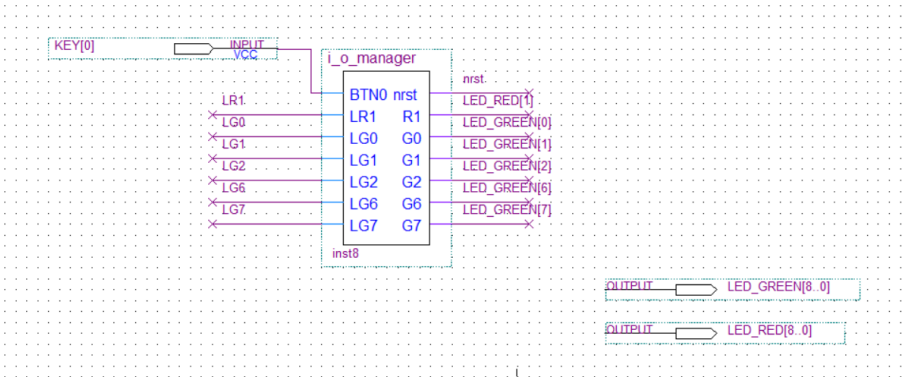


Figure 3.2.15 i_o_manager (*block_diagram.bdf* file from Quartus Software).

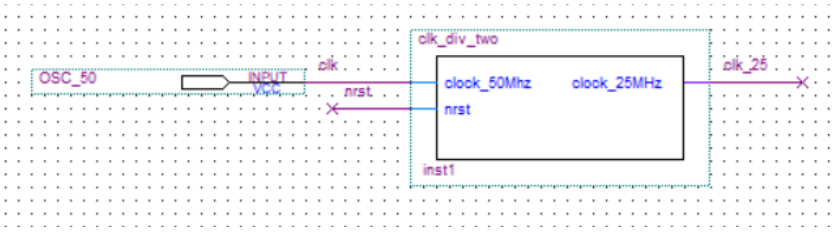


Figure 3.2.16 clk_div_two block (*block_diagram.bdf* file from Quartus Software).

a. i_o_manager

This block is used to connect the signals of the LEDs and the nrst to the board.

INPUTS

- BTN0: nrst is active when the button P0 is pressed
- LR1: Red led, activated from the main block.
- LG0: Green led, activated from the “main” block. It indicates that the user can choose to be Alice or Bob.
- LG1: Green led, activated from “Alice identifier” block. It indicates that the user is Alice.
- LG2: Green led, activated from “Bob identifier” block. It indicates that the user is Bob.
- LG6: Green led, activated from the “game” block. It indicates that the user can enter a sequence.
- LG7: Green led, activated from the “game” block. It indicates that the game is in progress.

OUTPUTS

- nrst: Active low, asynchronous.
- R1.
- G0, G1, G2, G6, G7.

b. clk_div_two

INPUTS

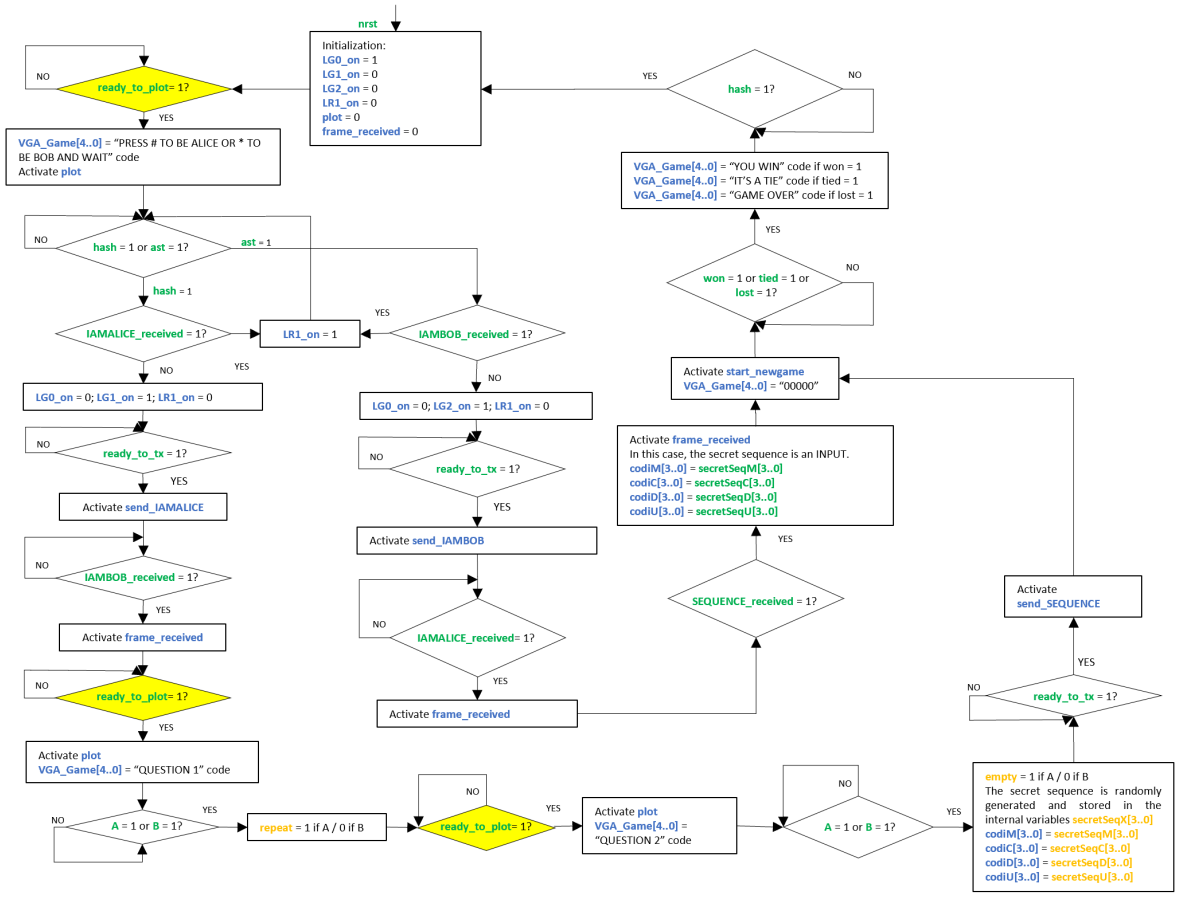
- clock_50Mhz: clock with a frequency of 50 MHz.

OUTPUTS

- clock_25MHz: clock with a frequency of 25 MHz. It is used for the blocks “VGA_SYNC” and “dual_ram”.

4. Algorithm of the Blocks

Init Block



Init Block Algorithm: LINK to see with higher quality (please ask the students if the hyperlink is broken)

This Block (Init Block) is a mix of the Main, Alice Identifier and Bob Identifier Blocks. We have finally decided to put these blocks together in order to simplify the interconnection of input/output variables in our schematic (*.bdf* file).

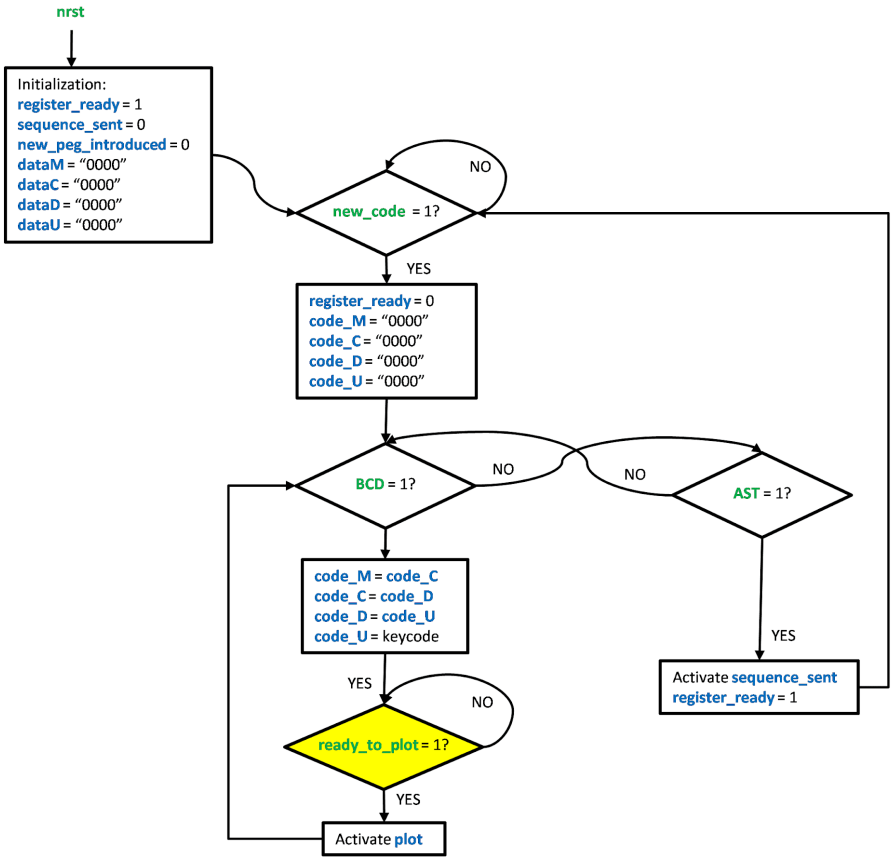
Internal Variables:

repeat: True if Alice has decided that the secret sequence has repeated colors.
empty: True if Alice has decided that the secret sequence has empty pegs.

SecretSeqX[3..0] stores the secret sequence.
secretSeqM[3..0]; secretSeqC[3..0]: secretSeqD[3..0]: secretSeqU[3..0]

For each output variable, and with the objective of making decisions/calculations, we will create an internal equivalent variable:
output_variable_IN

Guessing Register



Guessing Register Block Algorithm: LINK to see with higher quality (please ask the students if the hyperlink is broken)

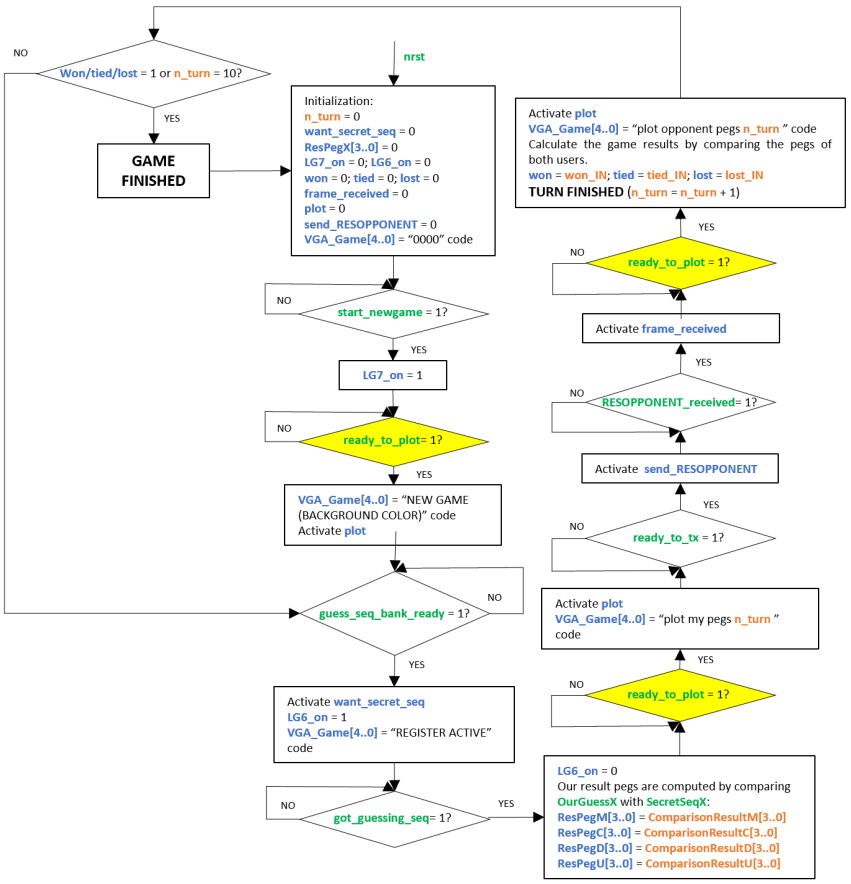
This register waits until the VGA Interpreter says (by activating the ready_to_plot register input) that the VGA is free and can plot something new. The register does this everytime the user has introduced a new peg and therefore there is the need to plot this new peg on the screen. Therefore, the procedure is the following one:

1) User introduced a new peg.
2) The register waits until it is possible to plot on the screen
3) The register output ***plot*** is activated and the peg is plotted.
4) We return to 1) unless user presses the * key.

Internal Variables:

For each output variable, and with the objective of making decisions/calculations, we will create an internal equivalent variable:
output_variable_IN

Mastermind (Game) Block



Mastermind (Game) Block Algorithm: LINK to see with higher quality (please ask the students if the hyperlink is broken). We could say that this is the most important block of the system because the main processes that happen when the game is ON happen in this block. The interconnection between this and the other blocks is the key to understanding our design.

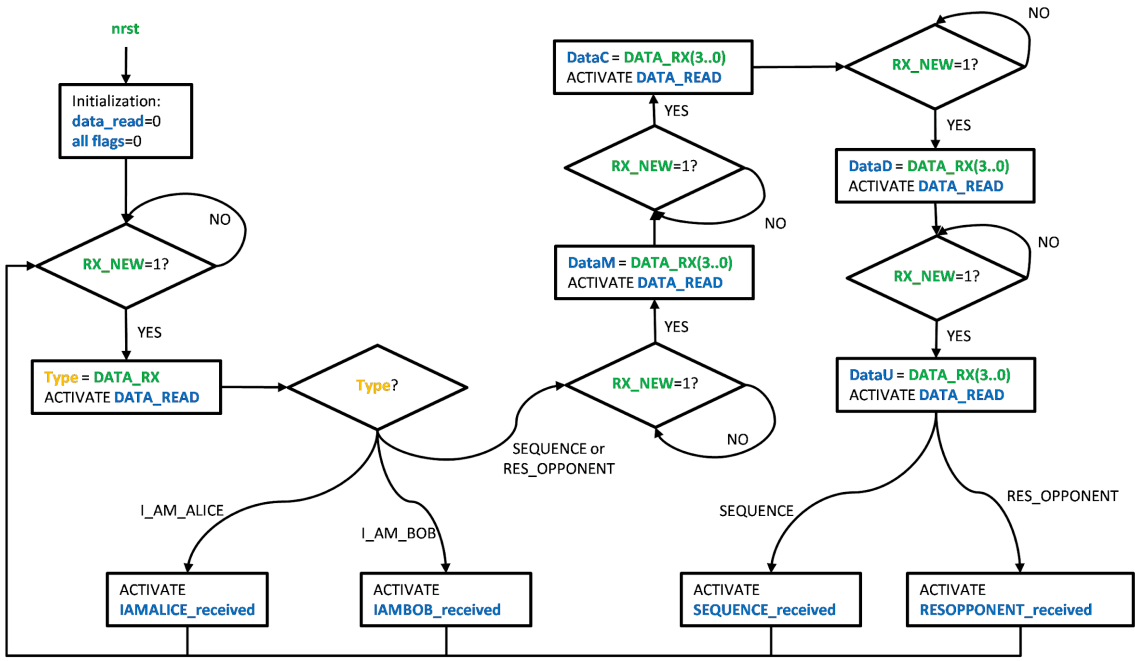
Internal Variables:

n_turn: this variable stores the value of the current turn (it has a maximum of 10). The maximum is 10 because in the Mastermind Classical Game definition, the challenge is to guess the secret sequence in 10 turns.

ComparisonResultX[3..0] stores the result of the comparison between the secret sequence and the user guessing sequence.
ComparisonResultM[3..0]; ComparisonResultC[3..0]; ComparisonResultD[3..0] ComparisonResultU[3..0]

For each output variable, and with the objective of making decisions/calculations, we will create an internal equivalent variable: ***output_variable_IN***

Receiver Block



Receiver Block Algorithm: LINK to see with higher quality (please ask the students if the hyperlink is broken)

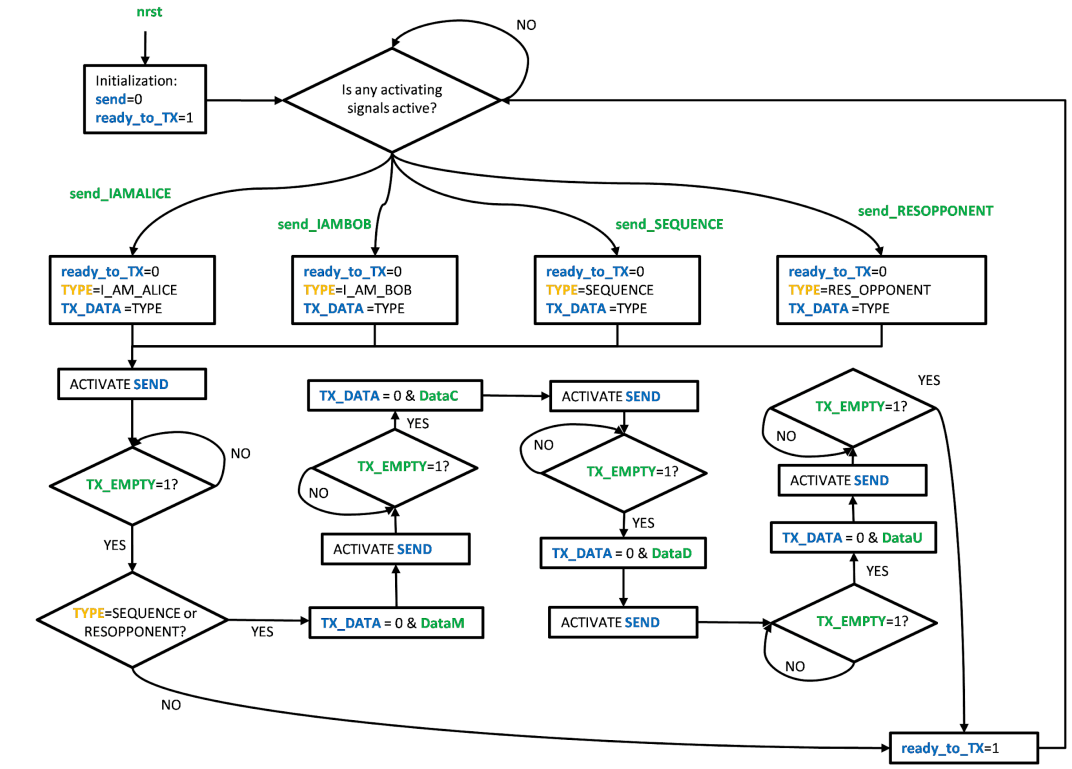
The algorithm of our system Receiver Block is quite similar to the one in the Design 1 from the Laboratory. The possibilities that we have (taking into account the FRAMES that we have in our system) are the following ones:

- IAMALICE_received
- IAMBOB_received
- SEQUENCE_received
- RESOPPONENT_received

Internal Variables:

Type: As in the Laboratory Design 1 design, this variable stores the FRAME TYPE that is being received by our system.

Transmitter Block



Transmitter Block Algorithm: [LINK to see with higher quality] (please ask the students if the hyperlink is broken)

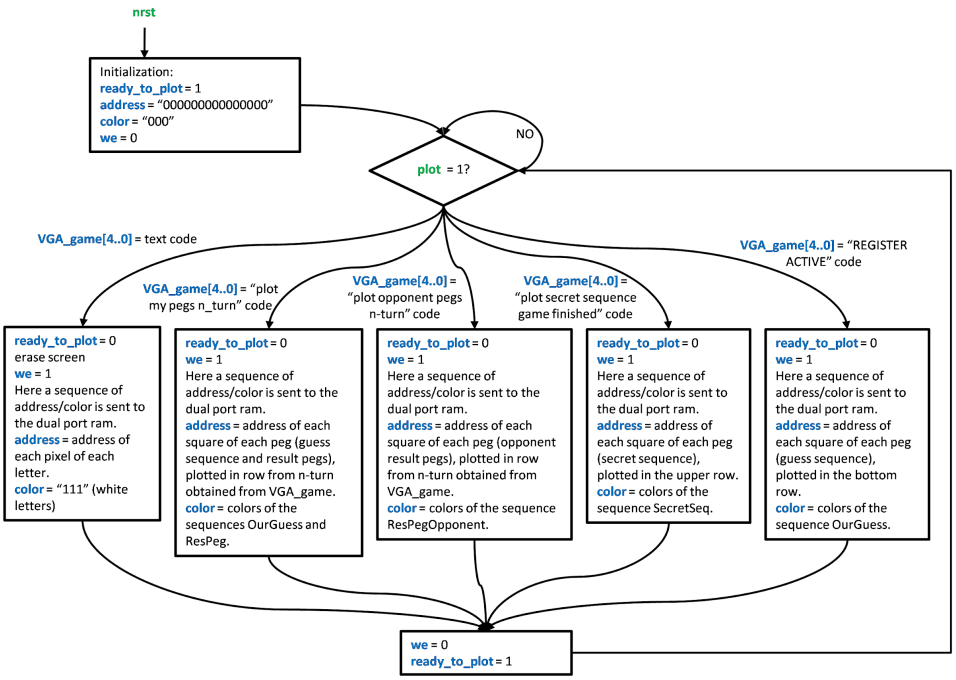
The algorithm of our system Transmitter Block is quite similar to the one in the Design 1 from the Laboratory. The possibilities that we have (taking into account the FRAMES that we have in our system) are the following ones:

- SEND_IAMALICE
- SEND_IAMBOB
- SEND_SEQUENCE
- SEND_RESOPPONENT

Internal Variables:

Type: As in the Laboratory Design 1 design, this variable stores the FRAME TYPE that is being received by our system.

Video Graphics Array (VGA) Block



VGA interpreter Block Algorithm: LINK to see with higher quality (please ask the students if the hyperlink is broken)

*This algorithm corresponds to an older version. The newer version is basically the same but with more options, and is described better in the state diagram. The blocs that should be present for the different values that the signal "VGA_Game[4..0]" can have are the following:

- plot starting question
- plot question 1
- plot question 2
- erase all screen
- plot guessed code + our result pegs
- plot opponent results
- plot current guess
- plot YOU WIN
- plot IT'S A TIE
- plot YOU LOOSE

Also, the addresses are of 9 bits.

Basically, the behaviour of this algorithm is the following. It enters in a waiting state until the activating signal ***plot*** is high. This activating signal can come from the following blocks:

- Init Block
- Game (Mastermind) Block

- Guessing Register

When the activating signal is activated, the output ready_to_plot turns to 0 in order to inform the blocks that the VGA Interpreter is busy (all the blocks will enter in a wait state until this signal is high if what they want to do is plotting). What the VGA Interpreter does is read the value of the signal VGA_Game and, depending on the value of this signal, plotting one thing or another.

Internal variables:

row: this variable depends on the value of n_turn and defines at which row of the screen the squares of the guess code and result pegs have to be plotted. It is concatenated with col to form the address sent to the RAM.
row_aux: this variable is used when erasing all the squares of the screen. It is concatenated with col
col: this variable is used to help when forming the address, and it corresponds to the column at which the pegs are plotted.
color_SecretSeq[11..0]: This vector contains the colors of the secret sequence concatenated in order M, C, D, U.
color_OurGuess[11..0]: This vector contains the colors of our guess concatenated in order M, C, D, U.
color_ResPeg[11..0]: This vector contains the colors of the result pegs concatenated in order M, C, D, U.
color_ResPegOpponent[11..0]: This vector contains the colors of the result pegs of the opponent concatenated in order M, C, D, U.

Blocks calculate_ResPegs, calculate_TurnResult, random_block and pegs_mixer

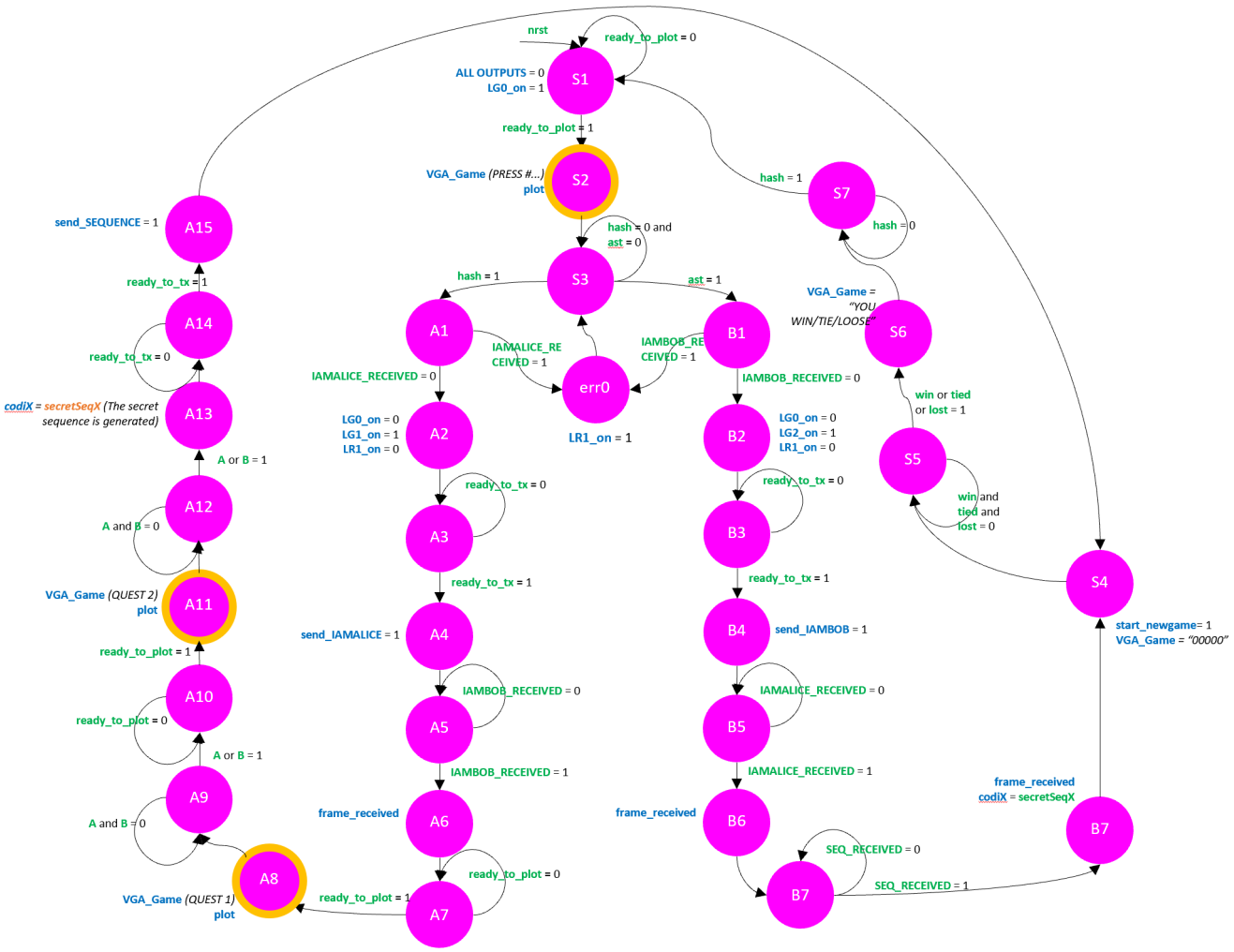
---

All these blocks have been created as auxiliary blocks during the last stages of the project, as the initial idea was to perform all these calculations inside other blocks such as the Game and Init Blocks. When we were in advanced stages of the design, we saw that it was easier for us to split tasks made by the leading blocks in several tasks that could be performed by smaller blocks. For example, the calculation of the turn result or the pegs mixing could be done in blocks created to specifically do this task and not in the Game block or in the init block.

As these blocks have been created, we could say, “on the fly”, we consider it appropriate to attach the material that we have needed to compute them in VHDL: the state diagrams. However, we have not needed to draw the algorithm of these blocks. This is mostly because the key point of the blocks is the operations and there is not any complexity in the activation of signals such as *frame_received.* Therefore, in this section, no algorithm has been attached. In order to understand the behavior of this block it is necessary to see the state diagrams in the next section.

5. State Diagram of the Blocks

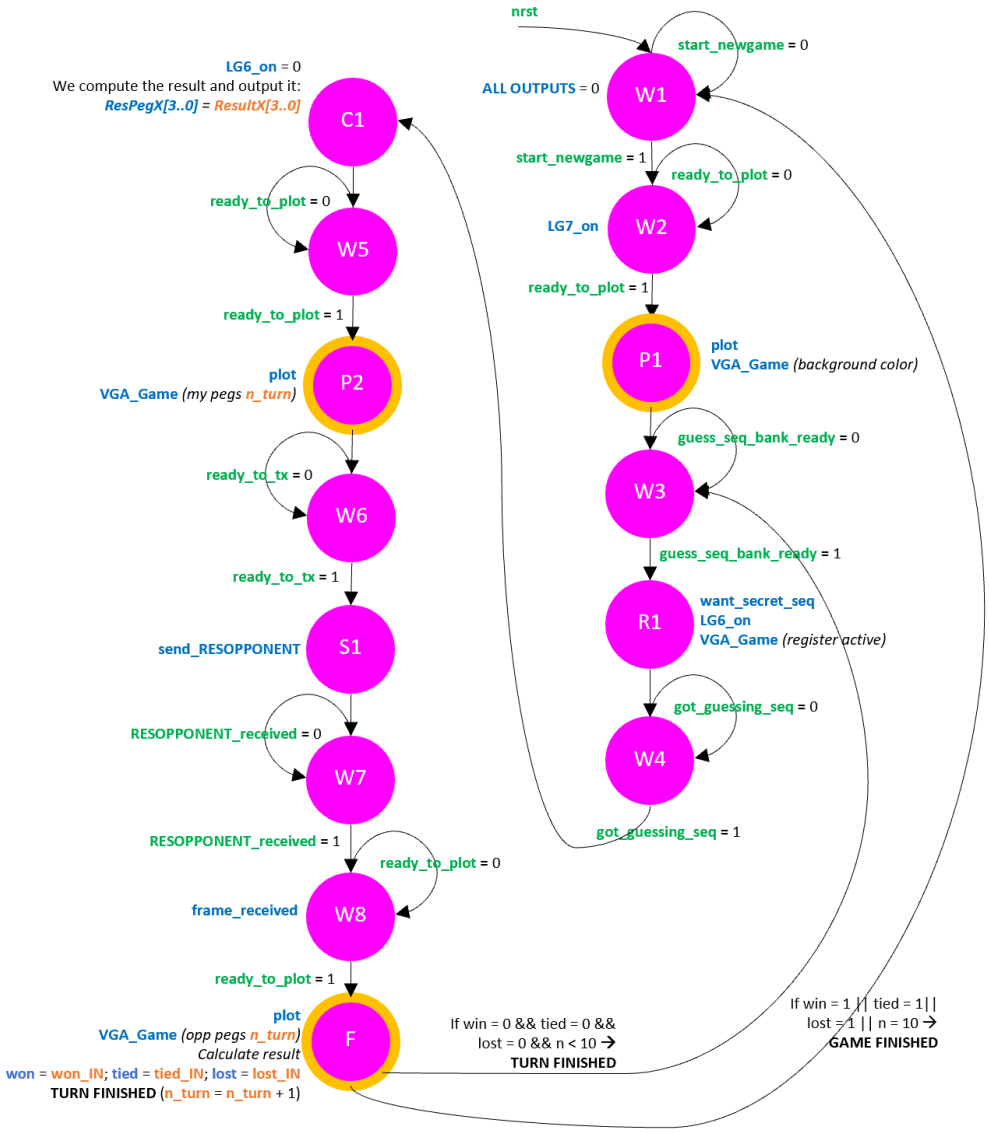
Init Block



Init Block State Diagram: [LINK to see with higher quality] (please ask the students if the hyperlink is broken)

States highlighted with an orange line are the ones where the VGA plots on screen what is indicated in the VGA_Game signal. As **our design is symmetric**, we have defined the Init Block State Diagram like this (we have two possible paths to follow depending on whether we are Alice or Bob). States in the Alice path are named with a letter A and an ID of the state. States in the Bob path are named with a letter B and an ID of the state. General states (common states for both characters) are named with a letter S (of 'State') and an ID of the state.

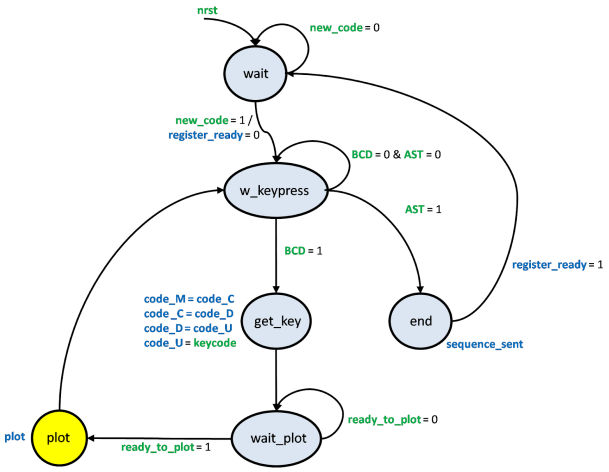
Mastermind (Game) Block



Mastermind (Game) Block State Diagram: [LINK to see with higher quality] (please ask the students if the hyperlink is broken)

Again, states highlighted with an orange line are the ones where the VGA plots on screen what is indicated in the VGA_Game signal. Waiting states are named with letter W and a state ID. States that interact with the register are named with letter R and a state ID. States where something is plotted on screen (VGA plotting) are named with letter P and a state ID. The final state is named with letter F. Also, the name of the state where the secret sequence is compared to the guessing sequence is ‘C1’.

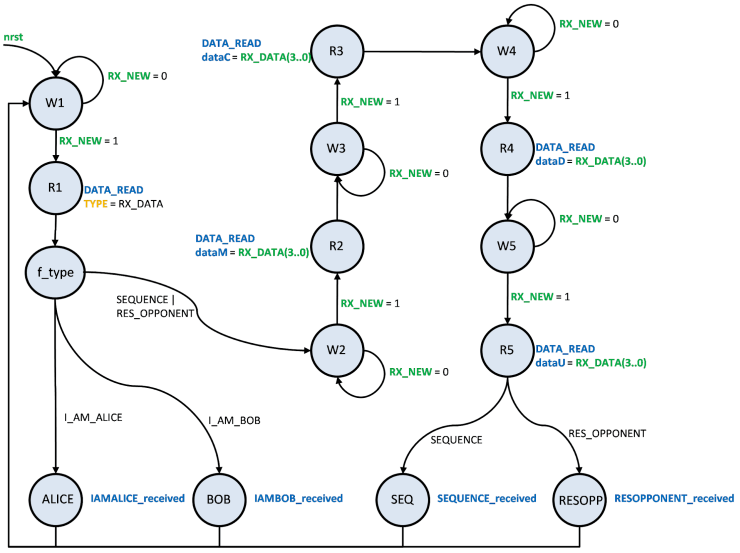
Guessing Register



Guessing Register State Diagram: LINK to see with higher quality (please ask the students if the hyperlink is broken)

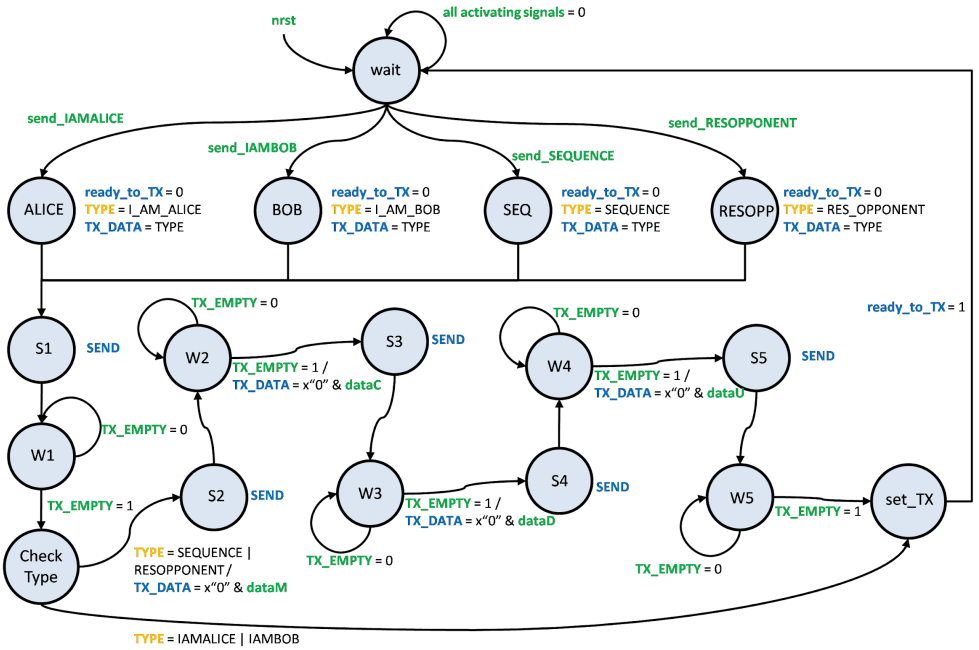
The state “plot”, filled with yellow, is the one where the VGA plots on screen what is indicated in the VGA_Game signal, which in this case will be the current guess.

Receiver Block



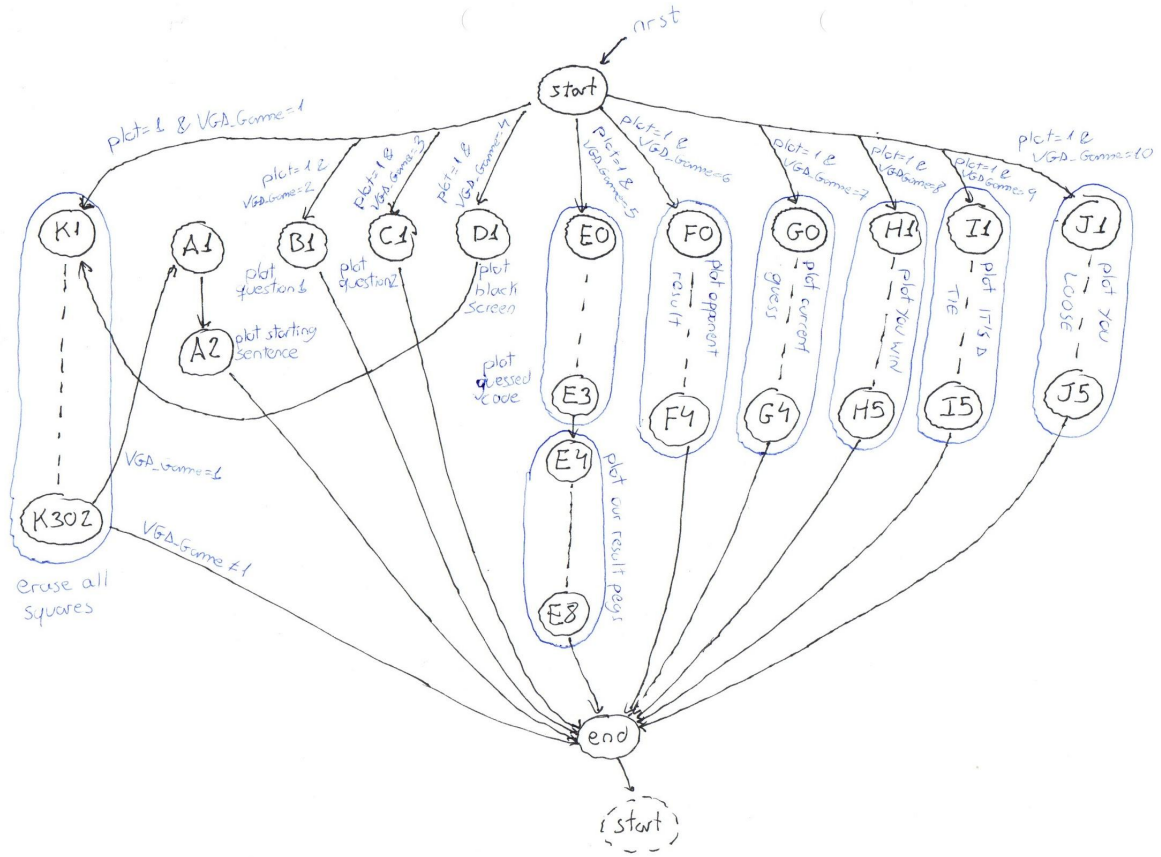
Receiver Block State Diagram: LINK to see with higher quality (please ask the students if the hyperlink is broken)

Transmitter Block



Transmitter Block State Diagram: [LINK to see with higher quality] (please ask the students if the hyperlink is broken)

Video Graphics Array (VGA) Block

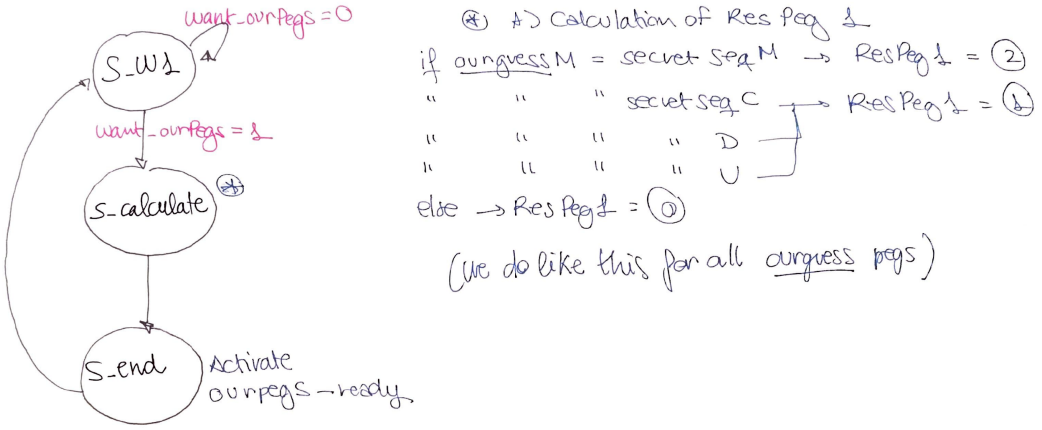


VGA interpreter block state diagram: LINK to see with higher quality (please ask the students if the hyperlink is broken)

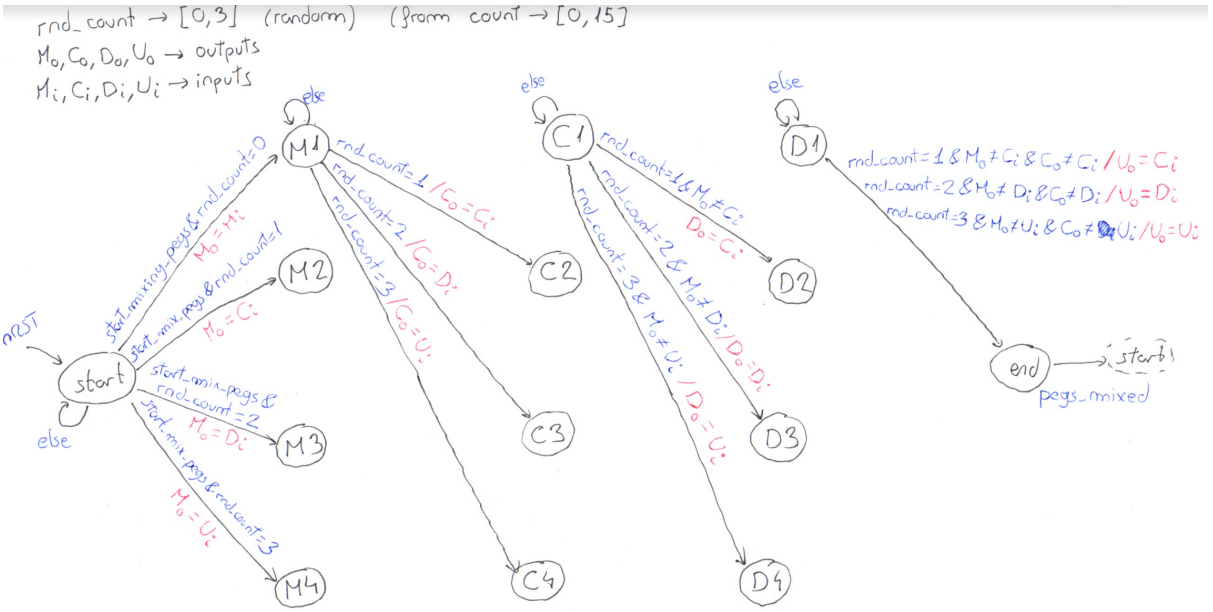
Blocks calculate_ResPegs, calculate_TurnResult, random_block and pegs_mixer

Being in advanced stages of the design, we saw that it was easier for us to split tasks made by the leading blocks in several tasks that could be performed by smaller blocks, and this is the reason why we created these auxiliary blocks. As these blocks have been created, we could say, “on the fly”, we consider it appropriate to attach the material that we have truly used to compute the VHDL files. Therefore, we are attaching the corresponding state diagrams.

calculate_ResPegs

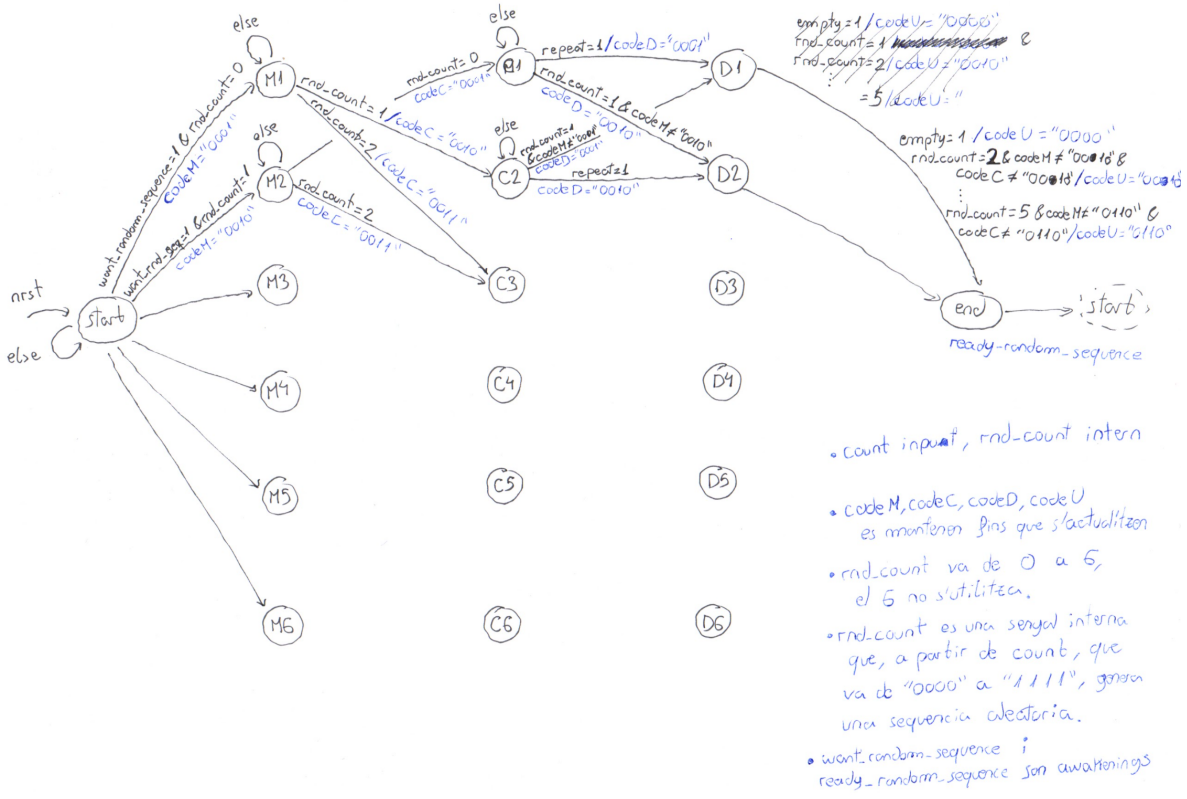


pegs_mixer



calculate_TurnResult

random_block



6. Final Simulation

6.1. Readme

The final simulation of the system is included in the cloud repo, as explained in the subsection 1.3. Repository Documentation. Here we include the content of the README inside the folder *0_folder_simulation* from the cloud repo.

```
Final Simulation (simulation_repo.vwf)
      - clock 20 ns

- [approx time instant] : event
- [60 ns] : the plot signal from both init blocks is activated to inform the VGA interpreter that screen must plot the starting sequence (VGA_Game = "00001"). As ready_to_plot is 1, both users move to state S2 from init block.
- [120 ns] : user 1 presses the hash key and therefore she is identified as Alice. Her LED LG1 is on because this indicates that she is Alice. LG0 turns off as this user is already identified.
- [210 ns] : Alice starts sending the frame IAMALICE
- [1.24 us] : the unidentified user receives the frame IAMALICE
- [1.46 us] : the unidentified user tries to be Alice by pressing hashtag but the init block enters in error state (LED R1 activated), as this user can only be Bob.
- [1.67 us] : the unidentified user rectifies and presses the ast key. Now he is identified as Bob. We see how the LED LG2 is activated. LG0 turns off as this user is already identified.
- [1.75 us] : Bob sends IAMBOB and starts waiting to receive the secret sequence
- [2.8 us] : Alice receives the frame IAMBOB. Bob checks the type of the sent frame, and as it is not a sequence, he can stop using the transmitter.
- [2.95 us] : for Alice, the first question is shown on screen and Init waits for Alice to answer the first question.
- [4.2 us] : Alice answers the first question with an A (there will be repeated pegs in the secret sequence)
- [4.4 us] : Alice answers the second question with a B (there will be no empty pegs in the secret sequence)
- [4.53 us] : the random blocks generates the secret sequence
- [4.61 us] : the mixer mixes the secret sequence
- [4.81 us] : Alice starts sending the secret sequence to Bob
- [4.83 us] : Alice can already start playing!! (but no worries, all of these events are so fast that the users will have the feeling of starting at the same time). The game starts for Alice and she can start introducing a guessing sequence in the register.
- [5.85 us] : Bob starts receiving the secret sequence (the transmitter of Alice has to send several frames)
- [10.2 us] : Bob has received the whole secret sequence. Now we can check in the outputs (Secret1C and Secret2X) how the secret sequences are the same. Both users can start playing! When the game starts, the LED LG7 is on, and also the LED LG6 is on when the user can enter a guessing code.
THE SECRET SEQUENCE IS 1335
```

- [10.27 us] : the signal n_turn from both players has been updated and now has the value 1 (we are in the first turn). VGA_Game is now "00111", indicating that the introduced pegs will be shown on screen.
**********FIRST TURN
Each player starts introducing their guessing sequence:
PEG M TURN 1:
- [10.8 us] : Alice introduces the M colored peg (with a value of 4).
- [10.82 us] : Bob introduces the M colored peg (with a value of 3).
- [13.1 us] : the VGA notifies that it has plottes the M pegs (in this case it happens at the same time with Bob and Alice because we are making that they play at the same time more or less).
PEG C TURN 1:
- [13.2 us] : Alice introduces the C colored peg (with a value of 3).
- [13.22 us] : Bob introduces the C colored peg (with a value of 1).
PEG D TURN 1:
- [17.44 us] : Alice introduces the D colored peg (with a value of 3).
- [17.46 us] : Bob introduces the D colored peg (with a value of 2).
PEG U TURN 1:
- [22.8 us] : Alice introduces the U colored peg (with a value of 5).
- [22.82 us] : Bob introduces the U colored peg (with a value of 2).
We have tried to press keys that should not be pressed right now and it continues working properly.
- [26.56 us] : Alice and Bob press the ast key and the register delivers the guessing sequence to the game block.
We have tried to continue changing the sequence after pressing asterisc, and it works properly (the sequence cannot change after pressing ast key).
THE GUESSING SEQUENCE INTRODUCED BY ALICE IS 4335
THE GUESSING SEQUENCE INTRODUCED BY BOB IS 3122
- [26.75 us]: The result pegs of Alice have been calculated (they contain 3 2's, because she has guessed the right color and place of 3 pegs, and one 0, because 4 is not contained in the secret sequence)
- [26.75 us]: The result pegs of Bob have been calculated (they contain 2 1's, because he has guessed the right color but not the right place of two pegs)
- [26.85 us]: Result pegs have been mixed (of both users at the same time because we are simulating that they play at the same time)
OURPEGS IN ALICE USER (our_pegsX1):
2202 (correct numbers and mixed!!)
OURPEGS IN BOB USER (our_pegsX2):
1010 (correct numbers and mixed!!)
We can see the pegs of each user before sending them (and therefore, in their correspondent boards) in the outputs our_pegsX1 and our_pegsX2.
- [26.93 us]: The protocol tx of Alice starts sending her result pegs to Bob.
- [26.93 us]: The protocol tx of Bob starts sending her result pegs to Alice.
- [32.3 us] : The protocol rx of Alice has received all the result pegs of Bob (we see how our_pegsX2 is the same as opponent_pegsX1)
- [32.3 us] : The protocol rx of Bob has received all the result pegs of Alice (we see how our_pegsX1 is the same as opponent_pegsX2)
- [32.47 us] : The turn result has been calculated for Alice.
- [32.47 us] : The turn result has been calculated for Bob, a new turn starts because nobody has won and n_turn < 10. N_turn now will be 2 (SECOND TURN).
**********SECOND TURN

- [50.42 us] : Alice and Bob start introducing the new guessing sequence (it does not have to be like this, but for the easiness of the simulation we do it like this)
- [90.42 us] : Alice and Bob press the ast key to send their guessing sequence.
THE GUESSING SEQUENCE INTRODUCED BY ALICE IS 1335 (it is the correct one!! Alice will win or at least tie!)
THE GUESSING SEQUENCE INTRODUCED BY BOB IS 4122 (it is not the correct one, therefore Alice will win and Bob will loose)
We can check the guessing sequences by checking the outputs our_guessX1 for Alice and our_guessX2 for Bob.
- [90.51 us] : the result pegs of both Alice and Bob have been calculated.
- [90.59 us] : the result pegs of both Alice and Bob have been mixed.
- [96.01 us] : Alice and Bob have received the result pegs of their respective oponents.
OURPEGS IN ALICE USER (our_pegsX1):
2202 (correct numbers and mixed!!) --> this will be the sequence of result pegs received by Bob
OURPEGS IN BOB USER (our_pegsX2):
1010 (correct numbers and mixed!!) --> this will be the sequence of result pegs received by Alice
- [96.01 us] : both users have received the result pegs of their opponent
- [96.21 us] : the result of the turn has been calculated. The VGA_Game value corresponds to the code that tells the VGA interpreter to plot the sentence YOU WIN. In the case of Bob, the VGA_Game value corresponds to the code that tells the interpreter to plot the sentence YOU LOST.
We see how the signal won1 is high and the signal lost2 is also high.
**********ALICE HAS GUESSED THE SEQUENCE, THE GAME ENDS
- [96.23 us] : the game block is again in the first waiting state (W1)
In order to start again another game, users have to press the hash key.

7. Experimental Validation

As it is explained in the subsection 1.3. Repository Documentation, in the folder *1_experimental_validation* of the repo, there is the link to the final video of the experimental validation. In the video, we show the correct functioning of our system presenting the different cases of configurations and results: tied game, lost game and won game.

The different configuration cases are:

- AA: repeated pegs, one empty peg.
- AB: repeated pegs, no empty peg.
- BA: no repeated pegs, one empty peg.
- BB: no repeated pegs, no empty peg.

Link to the video: https://youtu.be/HGz8_k-GD3A

8. Conclusion

The main objective of our project was to build a symmetrical digital system that could be uploaded to two FPGA boards so that two players could play the Mastermind Game by interacting with monitors (via VGA). Our idea was that both players played against the same secret sequence in order to increase competitiveness. Also, this sequence would be personalized depending on the answers of one of the users (Alice) to the questions presented in the game. We also wanted to provide random feedback (in relation to the result pegs) in order to make the game more difficult. There are some *What If Situations* that we were taking into account in the design of our system.

We have accomplished all these objectives and this is why we state that our result is successful. We have found very interesting that, although we had spent a lot of time designing the Block Diagram, Algorithms and State Diagrams of the blocks at the beginning of the project, in advanced stages of the project we have found the need to split big blocks in smaller ones in order to have subdivided tasks. This has made our system easier to understand and implement.

We would really like to have more time to implement more features and to experiment more with the VGA, as we have really enjoyed the process of this project. It has been an enriching experience since the beginning and we have learned a lot. We hope we have more opportunities to work with FPGAs in the future.