Block Breaker Game based on NIOS-II Soft Processor
ENGI-9865: Report of Course Design

Jiabao Guo
Student Number: 202096888
jiabaog@mun.ca
Memorial University of Newfoundland

***Abstract*—Targeting to the DE-10 Stadnard FPGA board, an attempt has been made to implement a Block Breaker video game based on the NIOS-II Soft Processor, using VGA port to output video data. This report also discussed in detail including the design of the architecture, the hardware and software implementation, state transition, and the performance and two profilling methods of the final work. In addition to the profiling methods, the authors also introduced the use of TCL scripts to realize the automatic QuestaSim test report generation, and a new simple testing using pure C and standard IO that is compatible with an embeded development environment.**

***Keywords*—**DE-10; FPGA; NIOS-II; Video Game; Block Breaker; VHDL; ASIC; Embedded Development

CONTENTS

I Introduction 1
- I-A Game Background . . . 2
- I-B The NIOS-II . . . 2
  - I-B1 Key Features . . . 2
  - I-B2 CPU Family . . . 2
- I-C Gameplay and Logics . . . 2
- I-D Reasons of Using NIOS-II . . . 2

II Architecture of Design 3
- II-A Overview . . . 3
- II-B Data Structures . . . 4
  - II-B1 Clocks . . . 4
  - II-B2 X and Y Axis of Ball . . . 4
  - II-B3 Blocks Data . . . 4
  - II-B4 Button Signals . . . 6
  - II-B5 Hex-es . . . 6
  - II-B6 Tray Position and Length . . 6
- II-C Hardware Side: Pseudo-random Number Generator . . . 6
- II-D Hardware Side: Data Merger . . . 6
- II-E Hardware Side: VGA Controller . . . 7
- II-F Software Side: Chrono, CoreGame, CoreIO and Logging . . . 7
  - II-F1 Chrono . . . 7
  - II-F2 CoreGame . . . 8
  - II-F3 CoreIO . . . 8
  - II-F4 Logging . . . 8
  - II-F5 Types . . . 8
- II-G State Transitions . . . 8
  - II-G1 ASM Chart . . . 8
  - II-G2 State Transition Diagram . . 8

III Profiling of Design 9
- III-A QuestaSim Simulation . . . 9
- III-B Automatic Test Report Generation based on TCL Script and QuestaSim . . 9
- III-C Discuss on Software Testing Framework 9
- III-D Introduction to "Fake Environment" . . 10
- III-E Test Result . . . 10

IV Appendix 10

References 10

I. INTRODUCTION

This project is mainly consist of the concept of the "Block Breaker" game and the application of NIOS-II Soft Processor that targeted to the DE-10 Standard Altera FPGA board. To describe our work in detail, we will discuss from 3 aspects, including:

- **Introduction.** In this section, we will first briefly introduce the background of this type of classic collision game, then discuss the use the NIOS-II Soft Processor to implement the game, and how our version of Block Breaker game works, and finally, from our own perspective, the reason of using the NIOS-II instead of pure VHDL.
- **Architecture of Design.** In this section, we will look into the hierachy of hardware components, explaining the design of each of them, then we will show the main game logic written in C that running on the NIOS-II Soft Processor. Also, in the end of this part we will present the game state transitions including an ASM chart and a state transition table.
- **Profiling of Design.** In this section, an attempt has been made to not only launch the QuestaSim (the successor of ModelSim) on the hardware side of the project, but also achieve automatic test report generation (a message box **showing an overall PASS/FAIL signal for each component**, followed with a list of testcases and the

PASS/FAIL status for each of them, **without having to check the waveform manually**) by one click. In addition, another attempt has been made to achieve the same thing on the software side, but with one more step that we called it **"Fake Environment"**.

A. Game Background

Block Breaker is a kind of very classic collision game. Your goal in this game is to launch the ball in the paddle (or call it tray) and break all the blocks suspended above.

B. The NIOS-II

This subsection is mainly based on the content of the official PDF document from Altera *Nios II Embedded Processor Backgrounder*, which can be found at the [1] of reference.

1) Key Features: The Nios II architecture is a RISC soft-core architecture with full 32-bit design:

- 32 general-purpose 32-bit registers,
- Full 32-bit instruction set, data path, and address space,
- Single-instruction 32x32 multiply and divide producing a 32-bit result.

2) CPU Family: Nios II classic is offered in 3 different configurations: Nios II/f (fast), Nios II/s (standard), and Nios II/e (economy). Nios II gen2 is offered in 2 different configurations: Nios II/f (fast), and Nios II/e (economy).

In this project, we are using the NIOS-II/e (economy) configuration as it does not require a license and is free for use.

C. Gameplay and Logics

In order to reduce the eye irritation caused by the bright sharp red color, the pictures in this section were darkened and may look blurry.

This subsection shows the gameplay of the project.

- **In fig. 2**: Before starting the game, the digital tubes display the "PRESS ANY KEY" periodly, telling the player to press any button to start the game.
- **In fig. 3**: The game starts and the VGA display shows the game screen as 1. In term of controlling the paddle, there are 4 buttons on the right bottom corner of the board. From left to right, they are: Move Left, Move Left, Move Right, Cheat, respectively. There are 2 move-lefts and 1 move-right. The extra cheat button, was another move-right, but changed temperarily for the purpose of presentation. Hitting the cheat button will force the current level to be win (pass), and the game will continue to the next level.
- **In fig. 4**: When clearing all the blocks in the current level, The "GOOD JOB" text flashes for 2 times to encourage the player to continue playing. Then, the game pauses and the "PRESS ANY KEY" in fig. 2 appears. After pressing any button, the game will continue to the next level.
- **In fig. 5**: If the ball fall below the paddle, then this is a fail. If you fail the level, the game displays the "LOSE" text and flashes for 2 times, indicating that you have lose. Then, the game pauses and the "BETTER NEXT TIME" in fig. 6 appears, telling the player that do not be discouraged.
- **In fig. 7**: When passing all levels, the text "GAME CLEAR" flashes for 2 times telling the player that the game is finished. Then, the game pauses and the "GAME OVER" in fig. 8 appears. Then, after pressing any button, the game will restart from the first level.

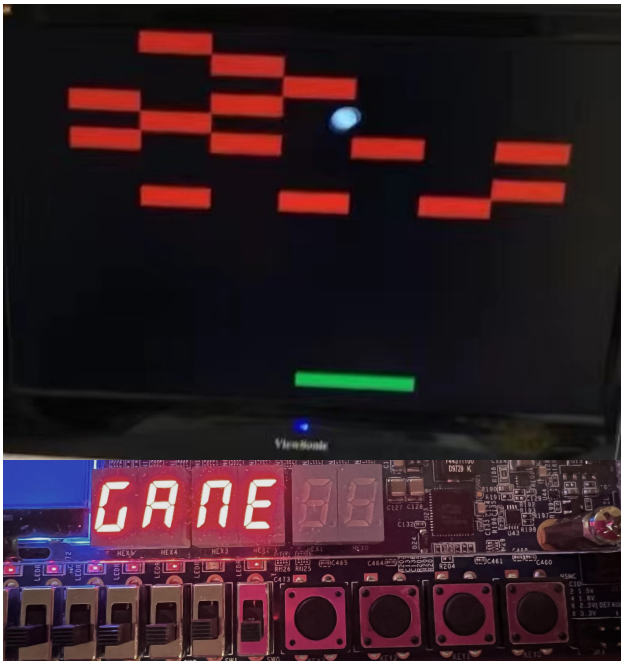

Fig. 1. Gameplay scene at level 1
There are 4 buttons on the right bottom corner of the board. From left to right, they are: Move Left, Move Left, Move Right, Cheat, respectively. There are 2 move-lefts and 1 move-right. The extra cheat button, was another move-right, but changed temperarily for the purpose of presentation. Hitting the cheat button will force the current level to be win (pass), and the game will continue to the next level.

D. Reasons of Using NIOS-II

As mentioned above, the game logic is written in C instead of a HDL. As you can see, in Part C (Gameplay and Logics), the game logic is simple but with an amount of small details. For example, flashing "Good Job" twice for 400ms each, is extremely easy in C while requires a certain amount of time to be implemented in HDL. To implement this in HDL, you need to first design a special frequency divider, or at least, calculate how many clock edges represent 400ms. Then, you have to design a new entity with EN (enable) port, CLK (clock) port, CLK_OUT (output clock) port etc., and/or at least, declare their corresponding signals in the architecture part of parent module. Finally, you need to wire the signals and ports, and the most importantly, you need to define a new state for it, and edit your state machine, adding the new state and its behaviour.

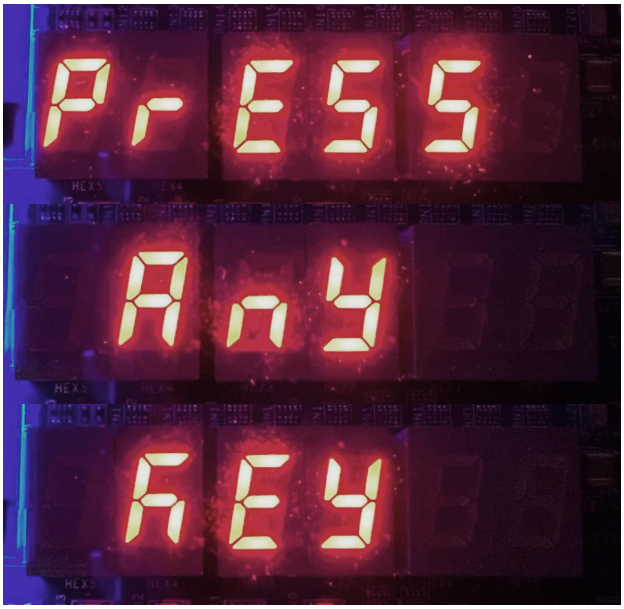

Fig. 2. Before Start Game: **"PRESS ANY KEY"**. The digital tubes display the "PRESS" and "ANY" and "KEY" periodly, telling the player to press any button to start/continue the game.

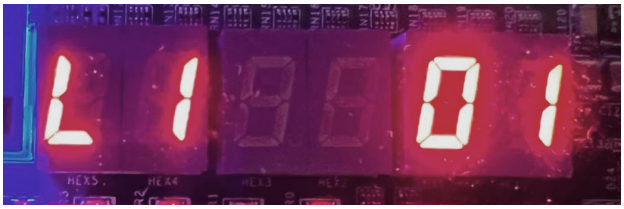

Fig. 3. In Game: **"L1 01"** means Level 1, Score 01. There are 3 levels in total. Score inherits between levels. The maximum score is 99.

From this fact, we clearly realized that in term of some specific logics, HDL can be extremely complicated. However, everything we need to do for the same logic in C is just as followed:

```
1    ci_set_string(0, "Good", 1);
2    msleep(400);
3    ci_set_string(0, "", 1);
4    msleep(150);
5    ci_set_string(0, "Job", 1);
6    msleep(400);
```

Where *ci_set_string* is a convenient function to convert characters to segment data of digital tubes and write them through PIO, written by ourselves.

This is why we need to use NIOS-II instead of VHDL, when working on the design of game logic.

II. Architecture of Design

In this section, we will look into the hierachy of hardware components, explaining the design of each of them, then we will show the main game logic written in C that running on the NIOS-II Soft Processor.

Fig. 4. Passing Level: **"GOOD JOB"**. The text flashes for 2 times to encourage the player to continue playing.

Fig. 5. Failing Level: **"LOSE"**. The text flashes for 2 times telling the player that he or she lose the game.

A. Overview

Fig. 9 is a coarse outline of our project. Since the entire game is based on NIOS, the C programming language will be the major way to illustrate the game logic instead of VHDL.

In the fig. 9,

- The ditigal tubes is used to display the game score,
- The buttons to control the paddle, the NIOS-II processor to run the game logic,
- Then, VHDL codes is used to convert the game scene into VGA signals,
- The LED Light Arrays are not used as planned.

The game logic was written in C, and where we used VHDL was only on controlling the VGA port, because only hardware description language (the HDL) have the direct access to the high frequency clock and achieve the synchronized programming.

Fig. 10 shows the top level of the project. It is a BDF (Block Diagram File) in Quartus II, and it is a very good way to visualize the design of the project. The BDF is essentially a clearer way to represent a VHDL/Verilog HDL/Other HDLs that is specifically used for top level logics that wiring every components together. From this figure, we can extract following information:

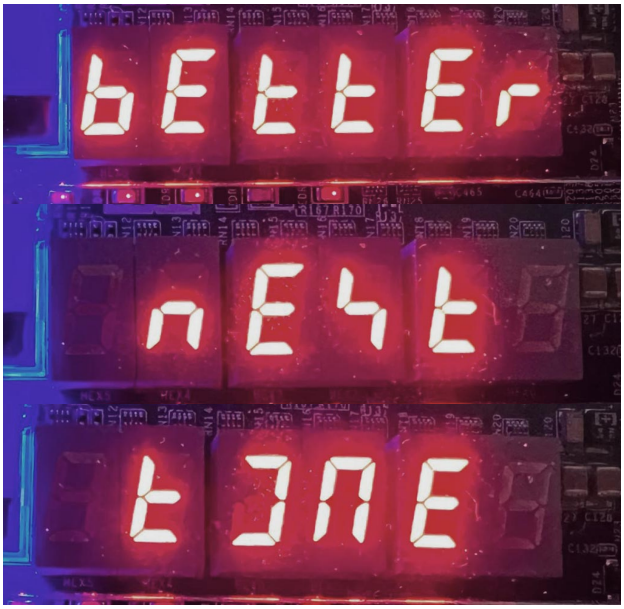

Fig. 6. Failing The Game: **"BETTER NEXT TIME"**. The text flashes for 2 times telling the player that do not be discouraged. "BETTER NEXT TIME" only appears after "LOSE".

Fig. 7. Passing all levels: **"GAME CLEAR"**. The text flashes for 2 times that express the congratulations to the player for passing all levels.

- **The input/outputs of the NIOS-II processor.** This includes all data structures that used in the game logic, and will be explained later;
- **The input/outputs of the VGA port.** This includes all signals that used in the VGA port, and will be explained later;
- **The input/outputs of the Phase-locked loop.** This includes all signals that used in the PLL, an external IP core. The PLL will also be explained later.

B. Data Structures

1) Clocks: There are 2 clocks in the design:

- **The 50MHz clock.** This clock is used to drive the game logic, and it is connected to the NIOS-II processor.
- **The 25MHz clock.** This clock is used to drive the VGA display to work as 640x480@60Hz, and it is connected to the VGA port.

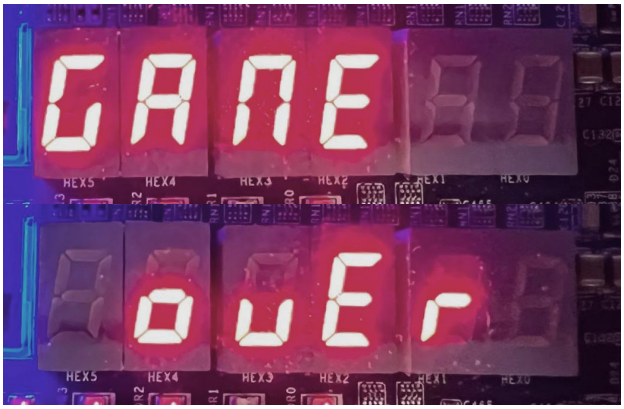

Fig. 8. Passing all levels: **"GAME OVER"**. The text flashes for 2 times indicating that the player has passed all levels and is returning to the first level now. "GAME OVER" only appears after "GAME CLEAR".

2) X and Y Axis of Ball: Both $ball\_x$ and $ball\_y$ are 16-bit unsigned integer.

There is a special 2D coordinate system defined as following:

- **The origin (Zero Point).** The origin point (0,0) is the **LEFT TOP CORNER** of the screen.
- **X axis.** Also known as the **Horizontal axis**, goes from LEFT to RIGHT, range from 0 to 640.
- **X axis.** Also known as the **Vertical axis**, goes from TOP to BOTTOM, range from 0 to 480.

This coordinate system is same as the windows in Microsoft Windows Operating System.

3) Blocks Data: The blocks data is an array of 8-bit unsigned integer with length 9, that is, $blocks\_data$[71 downto 0] or alt_u8 $blocks\_data$[9].

For example,

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	0	0	0	0	0	0	0
1	0	1	0	1	0	1	0	0
2	0	0	1	0	1	0	0	0
3	0	0	0	1	0	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

represents that there are 3 blocks in the second row, 2 blocks in the third row and 1 block in the forth row, no blocks in other places.

For another example, regarding the blocks distribution in the fig. 1, the corresponding blocks data will be:

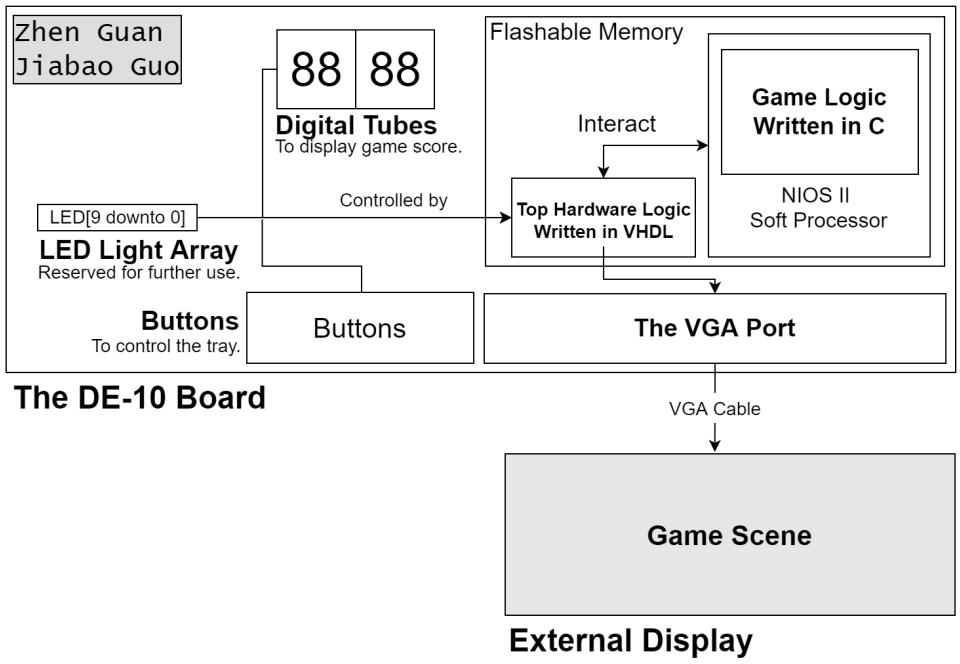

Fig. 9. Architecture of project.

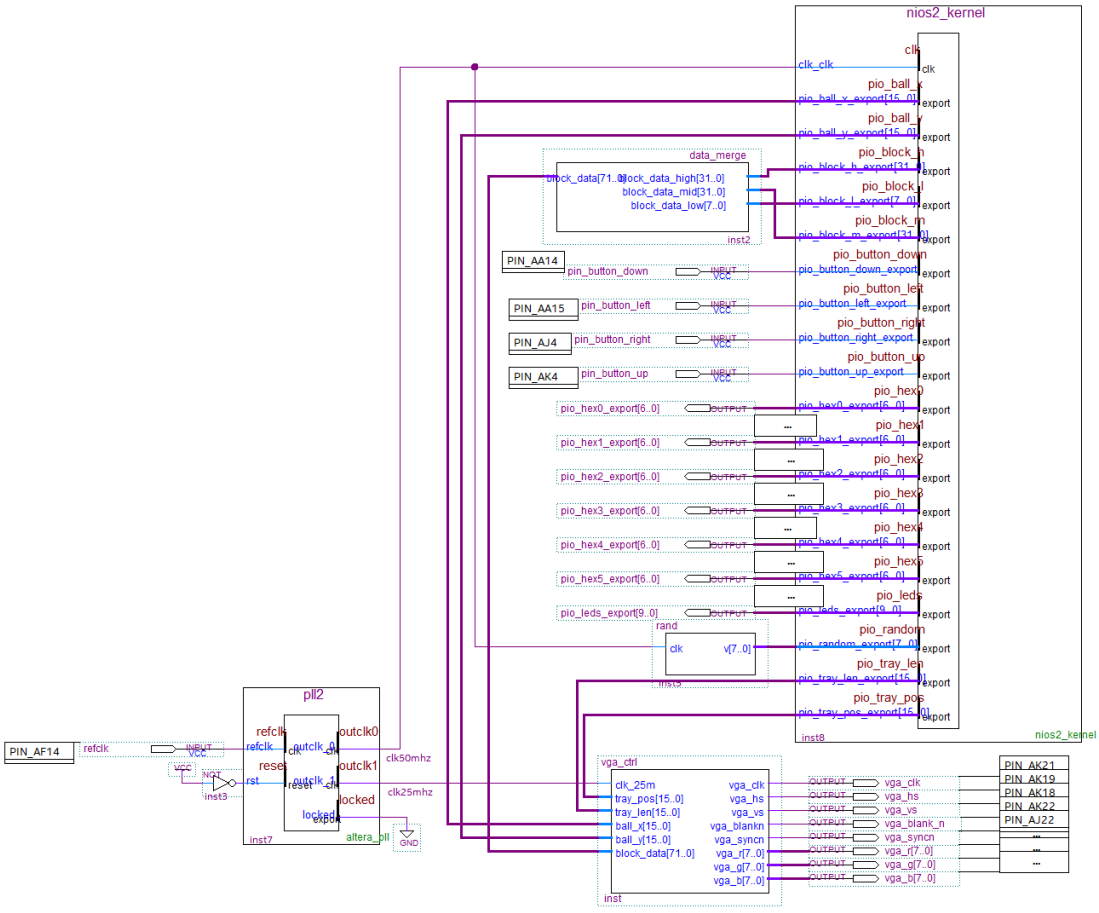

Fig. 10. Top-level design of project.

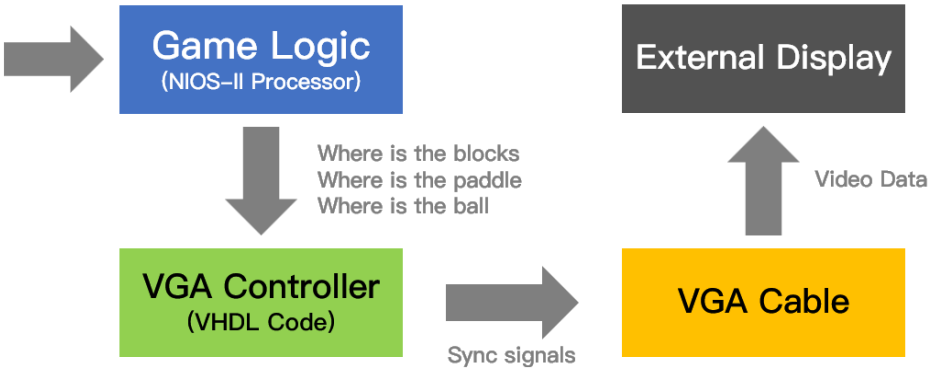

Fig. 11. Architecture of project (Clearer Version).

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	1	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0
2	0	0	0	1	0	0	0	0
3	1	0	1	0	0	0	0	0
4	0	1	0	0	0	0	0	0
5	1	0	1	0	1	0	1	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	1	0
8	0	1	0	1	0	1	0	0

Representing in flat, then we have

$blocks\_data$[71 downto 0]=

"010000000010000000010000101000000100
000010101010000000000000001001010100"

4) Button Signals: $button\_left$, $button\_up$, $button\_down$, $button\_right$ are four 1-bit std_logics (button signals), corresponding to fig. 1, from left to right respectively, **having a inversed logic that 0 represents "Pressed" and 1 represents "Not Pressed".** Their names are from the previous design (Guess My Number) and they are not standing for 4 directions now.

The inverted logic is because keeping a signal high is easier than keeping low. This avoids a lot of accidentally-triggers.

5) Hex-es: hex[0] hex[5] are 6 digital tubes. Each of them is a 7-bit unsigned integer, or a 7-bit std_logic_vector.

A digital tube is represented in "GFEDCBA" format, with every bit inverted that 1 means OFF and 0 means ON. For example, "0000000" represents "8", "0110000" represents "3", "0001000" represents "A".

6) Tray Position and Length: Both $tray\_len$ and $tray\_pos$ are 16-bit unsigned integers. The tray, or call it paddle, is vertically fixed on the y =SCREEN_HEIGHT $-$ 52, and its X coordinate of left side is represented by $tray\_pos$.

The $tray\_len$ stores the length of the paddle, in pixels. The longer the paddle is, the easier to catch the ball, the easier the game is.

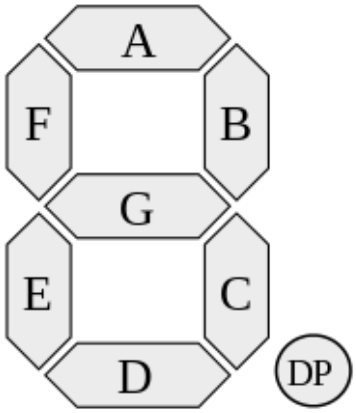

Fig. 12. 7-Segment Digital Tube. [2].

C. Hardware Side: Pseudo-random Number Generator

With a high frequency clock it is easy to generate random numbers. The random number generator in our system is actually an 8-bit counter. Along with the clock, the counter counts from 0 to 127 and then restart from 0. The user hit the "Start Game" button at unpredictable time, therefore the value in the counter is also unpredictable, but in a known range.

This module is not used in this project due to the change of the project subject, but is reserved for further use because it is simple and effective. It was useful in previous subject "Guess My Number Game based on NIOS-II Soft Processor".

D. Hardware Side: Data Merger

There is a limitation on the NIOS-II processor that we can not transmit more than 32-bit data in a single PIO port. However, although we compressed our blocks data to the extreme, it still occupies 72 bits. To transmit the blocks data we had to split it

into a 32+32+8 form, that are high 32-bit, middle 32-bit and low 8-bit. After that, we did bitwise concatenation to restore the blocks data.

For example, if we have

blocks_data[71 downto 0]=

"01000000001000000001000010100000010000001010101000000000000000001001010100"

then

*blocks_data_high*32[71 downto 40]=

"010000000010000000010000101000000"

*blocks_data_mid*32[39 downto 8]=

"01000000101010100000000000000010"

*blocks_data_low*8[7 downto 0]=

"01010100"

Finally, to restore the original data, we do bitwise concatenation:

```
    blocks_data <=
        blocks_data_h
        & blocks_data_m
        & blocks_data_l;
```

There are also bitwise reverse problem: we found that the data in the PIO output is not in the same order as input. So we had to reverse the data before passing in the PIO port:

```
alt_u32* gp_block_data_high = reverse_bit(
    (alt_u32*)(data + 0) );
alt_u32* gp_block_data_mid = reverse_bit(
    (alt_u32*)(data + 4) );
alt_u8* gp_block_data_low = reverse_bit(
    (alt_u8*)(data + 8) );
```

Having the defination of *reverse_bit* that

```
alt_u32 reverse_bit(alt_u32 a) {
    a = ((a >> 1) & 0x55555555)
        | ((a & 0x55555555) << 1);
    a = ((a >> 2) & 0x33333333)
        | ((a & 0x33333333) << 2);
    a = ((a >> 4) & 0x0F0F0F0F)
        | ((a & 0x0F0F0F0F) << 4);
    a = ((a >> 8) & 0x00FF00FF)
        | ((a & 0x00FF00FF) << 8);
    a = ( a >> 16) | (a << 16);
    return a; }
```

The reverse_bit function is generated by Github Copilot.

E. Hardware Side: VGA Controller

It requires a group of specific values to make the monitor run on a certain resolution but we could not find one from the user manual. Finally we consulted the internet (https://electronics.stackexchange.com/questions/532192/intel-de1-soc-vga-controller-in-vhdl) for the correct code to generate Hsync and Vsync signals.

F. Software Side: Chrono, CoreGame, CoreIO and Logging

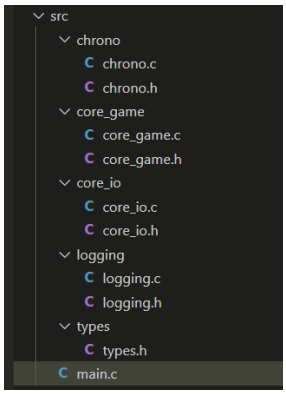

Fig. 13. Source Tree of Sofrware Side

The design of software side (game logic side) is divided into 5 parts.

1) Chrono: The chrono module seals *alt_types.h* and provide a *msleep* function to sleep in milliseconds.

In addition, this module provide a convenient macro that

```
#define time_warp(
    variable,
    kiloperiod,
    proc) \
if ((++(variable)) >= 1000*(kiloperiod))\
    { variable=0; proc; }
```

This is an implement of frequency divider. The use of *time_warp* is easy:

```
// Requires a static variable.
static int i = 0;

// Some kind of loops
for (;;) {
    time_warp(i, 25, {
        if (is_button_left_pressed()) {
            // No fluctuations here!
            // Do something.
        }
    });
}
```

This is a 25000 divider that slows down the clock by approximately 25000 times (Of course there are no real "clk"s, but this do helps with the fluctuation of the button).

2) CoreGame: This module handles the real main game logic, including ball movement, ball collision detection, state transitions, level(stage) designs, flashed text logics, etc. For detailed information please refer to the source code (core_game.c).

3) CoreIO: This module handles all interactions with the PIO port, for example, digital tubes write-back, ball positions write-back, charater-to-segment-data converter, etc. This module is also designed to be the only one that interacting with the PIO. For detailed information please refer to the source code (core_io.c).

4) Logging: This module is only for logging, providing log_info and log_error. In executables running on NIOS-II, the standard output stream $stdout$ is read by the UART debug port and displayed through nios2-terminal. For detailed information please refer to the source code (logging.c).

5) Types: This is a header file providing TRUE, FALSE and UNREFERENCED_PARAMETER support.

```
1  typedef alt_u8 BOOL;
2  #define TRUE (1)
3  #define FALSE (0)
4  #define UNREFERENCED(x) ((void)(x))
```

G. State Transitions

Design Thoughts: In order to reduce the design complexity and make state transition more explicit, we intend to design out ASM chart in "One-Hot Encoding" form and it would be implemented as a prototype of "Moore State Machine".

List of Notations: We assume that

$$\begin{aligned} \text{State } IDLE &= S_0 \\ \text{State } IN\_GAME &= S_1 \\ \text{State } WIN &= S_2 \\ \text{State } LOSE &= S_3 \\ \text{State } OVER &= S_4 \end{aligned}$$

and also abbreviations:

$$\begin{aligned} reset &= r \\ cheat &= c \\ \text{“Press Any Key”} &= PAK \\ \text{“Good Job”} &= GJ \\ \text{“Game Clear”} &= GC \\ \text{“Better Next Time”} &= BNT \end{aligned}$$

where $cheat$ represents the Cheat Button mentioned in the Section I.

State Assginments:

State	Q_1	Q_2	Q_3	Q_4	Q_5
S_0	0	0	0	0	1
S_1	0	0	0	1	0
S_2	0	0	1	0	0
S_3	0	1	0	0	0
S_4	1	0	0	0	0

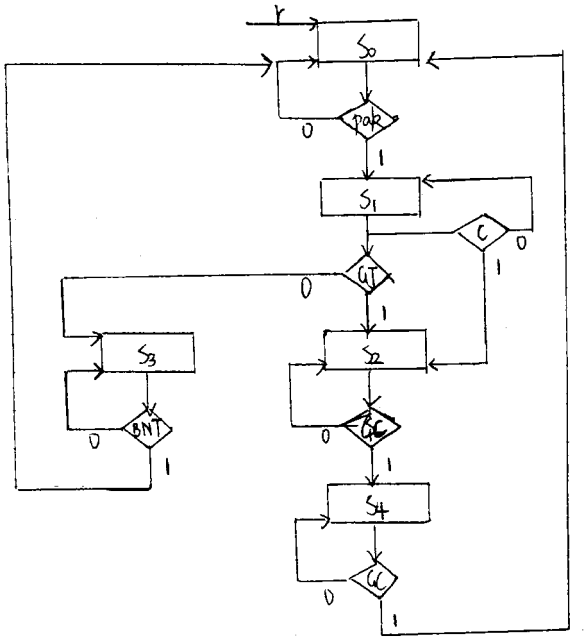

Fig. 14. The ASM chart of the design.

1) ASM Chart: **Fig. 14 is the ASM chart of the design.** This figure may be laid on other pages because of some typesetting policies of LaTeX.

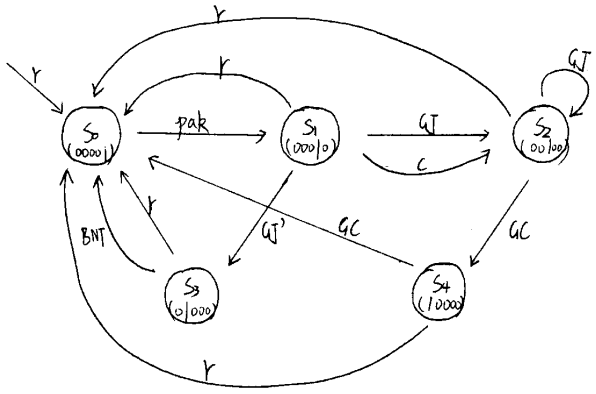

Fig. 15. The state transition diagram of the design.

2) State Transition Diagram: **Fig. 15 is the state transition diagram of the design.** This figure may be laid on other pages because of some typesetting policies.

III. Profiling of Design

A. QuestaSim Simulation

QuestaSim is a newer version of ModelSim with 64-bit machine support. we are using QuestaSim for the testbench on hardware side.

The test project is located at the /sim path of project root.

As you can see in fig. 16, the simulation is targeted to 3 components:

- VGA Controller,
- Data Merger,
- Random Number Generator.

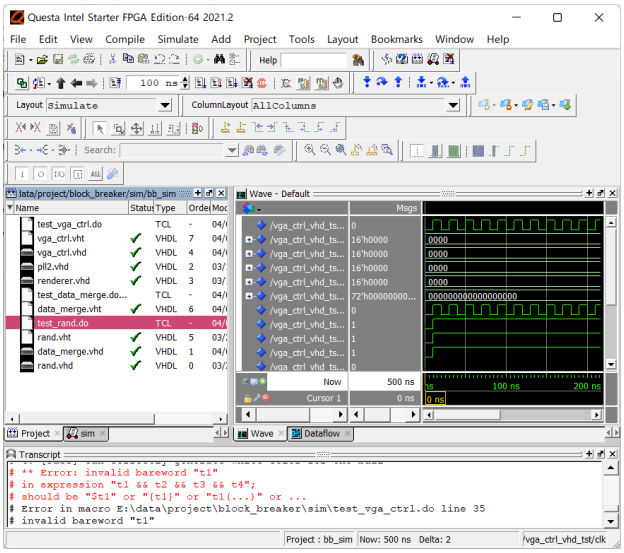

Fig. 16. QuestaSim. Can be launched via command "vsim"

B. Automatic Test Report Generation based on TCL Script and QuestaSim

We designed 3 TCL scripts for the 3 test subjects (VGA Controller, Data Merger and Random Number Generator).

As of fig. 17, after executing each of the scripts, the test report is generated automatically, a message box will be popped up, showing an overall PASS/FAIL signal for each component, followed with a list of testcases and the PASS/FAIL status for each of them, without having to check the waveform manually) by simply clicking the menu item "Execute" in the GUI.

For detailed TCL script code please refer to the source code (/sim/test_rand.do, /sim/test_vga_ctrl.do and /sim/test_data_merge.do).

C. Discuss on Software Testing Framework

Existing C/C++ testing frameworks are not suitable for our testing needs. The list of frameworks we had evaluated are:

- **CTest**. Not suitable bacause this is an executable-level test based on CMake. The target platform (NIOS-II CPU Architecture) does not support this.

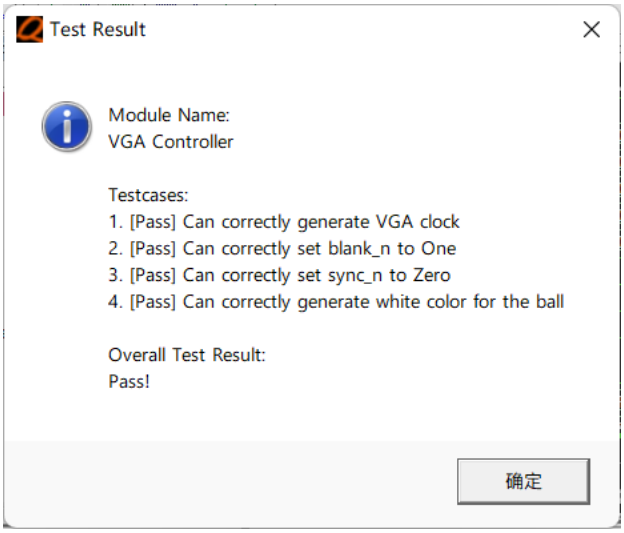

Fig. 17. Test result generated by TCL Script. The message box is shown by calling WinAPI "MessageBoxA/W" which is localized according to the host machine's language. The Chinese text in the button represents the meanning of "Confirm" or "OK".

- **CUnit**. Not suitable bacause implementing a CUnit requires adding external libraries and this is not a good selection for embedded development. Although we can use macros to generate a kind of "Release Build" that compiles without testing framworks, this is still causing too much code change.
- **QuestaSim on NIOS-II**. This is an official method to run QuestaSim on projects using NIOS-II. However, this requires a long list of steps to initiate and currently there are neither enough documentation on this, nor enough related discussions on any forums. We can not use this high risk method.

We need a more flexible testing framework that:

- Has nearly no change to existing code, to ensure maximum compatibility;
- Using no third-party libraries, so that it can work well with embedded development;
- Or can be runned directly on the development environment, without having to run on the FPGA board, such as QuestaSim.

Taking the above into consideration, we have written a very simple C testing framework for this project only, temperarily called "Easy Test", with the following advantages:

- Has no code change to the existing project;
- Using only pure C and standard IO (stdio.h);
- Based on CMake that allows module-level testing and easy to manage the compiler parameters;
- Running the test directly on the developemnt environment, without having to turn on the FPGA board.

Regarding the source code of Easy Test please also refer to

the files that submitted along with this report (/test/framework/easytest.c).

D. Introduction to "Fake Environment"

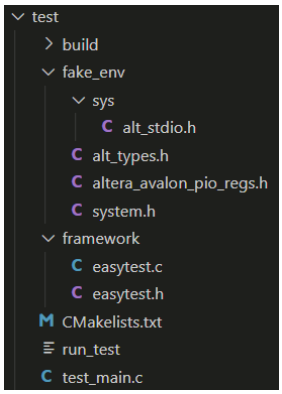

Fig. 18. Easy Test and "Fake Environment"

We also created **a fake NIOS-II runtime library** to work with the testing, temperarily called "Fake Environment". This is creating a partial NIOS-II runtime on the PC but blocking all hardware-related operations.

For example, *altera_avalon_pio_regs.h*, which is an important NIOS-II runtime library, providing IO operations for the PIOs, while in the fake environment, all IO operations will do nothing but still return a legal value. This has no actual impact to the testing, because according to our design of the software side, unable to access the hardware will not influence the state transition.

E. Test Result

Fig. 19 shows the test result of the software side. The test result is generated by **Easy Test**.

Fig. 20 shows the test result of the VGA Controller.

Fig. 21 shows the test result of the Rand Generator.

Fig. 22 shows the test result of the Data Merger.

All the tests were finished in very short time, and passed all testcases. This is identical to the actual run on the FPGA board. The gameplay is seamless and had a very good user experience and very challenging difficulty.

We are trying to continue to improve the testing framework, making it compatible with any future works. We are also trying to improve the user experience, by providing a more friendly interface and a more intuitive testing process.

```
[100%] Built target bb_test
[INFO] Block Breaker Game Test
[INFO] ======================
[INFO] Zhen Guan, Jiabao Guo
[INFO] Memorial University

 TESTING  Core Game Logics
      PASS - Ball X position middle
      PASS - Ball Y position bottom
      PASS - Init tray length should be 128
      PASS - Init tray X position middle
      PASS - Init ball X direction left
      PASS - Init ball Y direction up
      PASS - Init score should be 0
      PASS - Init level should be 1
      PASS - Init game state should be IDLE
      PASS - Game state should be IN_GAME after key press
[INFO] You Win!
      PASS - Game state should be WIN after clear blocks
 PASSED  (11/11 Passed).

 TESTING  IO Logics & Util Functions
      PASS - Function seg_from_char test (1/3)
      PASS - Function seg_from_char test (2/3)
      PASS - Function seg_from_char test (3/3)
      PASS - Reverse bit test (1/3)
      PASS - Reverse bit test (2/3)
      PASS - Reverse bit test (3/3)
 PASSED  (6/6 Passed).

 TESTING  Failure Example
      PASS - This testcase is always pass
      FAIL - This testcase is always fail
 FAILED  (1/2 Passed, 1 Failed).
➜ test
```

Fig. 19. Software Testing using Easy Test. **The last test is failed because its a test to show what a failed test looks like.**

IV. Appendix

References

[1] Altera. *Nios II Embedded Processor Backgrounder* (PDF), https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/pr/nios2_backgrounder.pdf.

[2] Tan, Yin, H., Xu, J., Jiang, F., Chen, T., & Li, W. (2021). *Design and Manufacture of Programmable Electrochromic Digital Tube*. In Advances in Graphic Communication, Printing and Packaging Technology and Materials (pp. 366-372). Springer Singapore. https://doi.org/10.1007/978-981-16-0503-1_54

[3] E. Monmasson and M. N. Cirstea, *FPGA Design Methodology for Industrial Control Systems—A Review*, in IEEE Transactions on Industrial Electronics, vol. 54, no. 4, pp. 1824-1842, Aug. 2007, doi: 10.1109/TIE.2007.898281.

[4] M. N. Cirstea, A. Dinu, J. Khor, and M. McCormick, *Neural and Fuzzy Logic Control of Drives and Power Systems*. Oxford, U.K.: Elsevier, 2002.

[5] D. L. Perry, *VHDL*. New York: McGraw-Hill, 2004.

[6] C. Cecati, *Microprocessors for power electronics and electrical drives applications*, IEEE Ind. Electron. Soc. Newsl., vol. 46, no. 3, pp. 5-9, Sep. 1999.

[7] S. Brown, *FPGA architectural research: A survey*, IEEE Des. Test. Comput., vol. 13, no. 4, pp. 9-15, Winter 1996.

[8] Internet sites and on line journals dedicated to FPGAs, such as: "FPGA-FAQ". [Online]. Available: http://www.fpga-faq.com/, "FPGA and Structured ASIC Journal," http://www.fpgajournal.com/

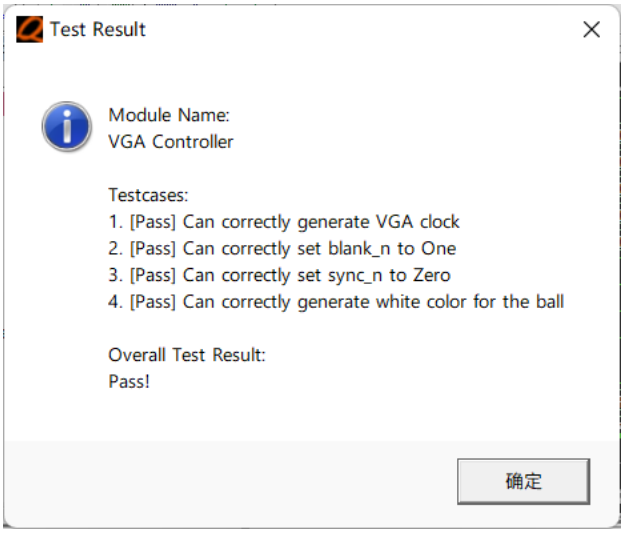

Fig. 20. Test Report of VGA Controller

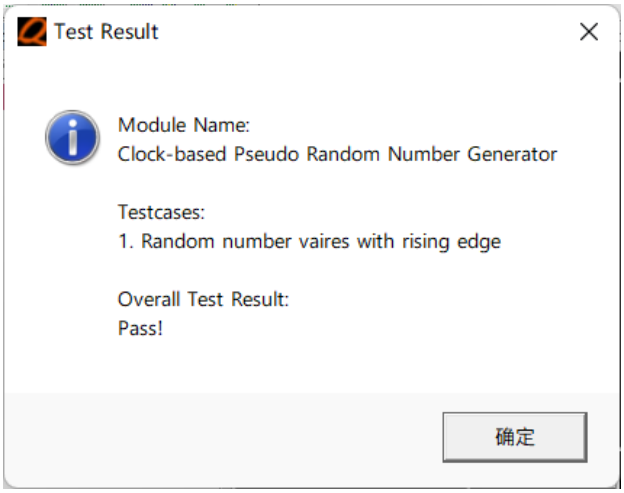

Fig. 21. Test Report of Rand Generator

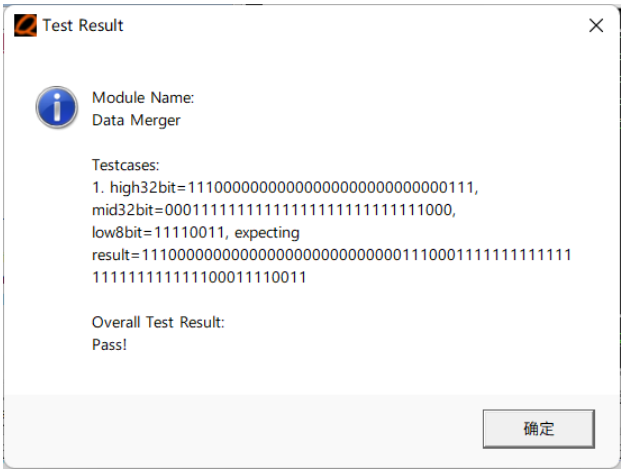

Fig. 22. Test Report of Data Merger

[9] Mishra, A. et al. (2018) *VGA Application in Text Display Using FPGA*, in Intelligent Communication, Control and Devices. [Online]. Singapore: Springer Singapore. pp. 1575-1585.

[10] Gazi, O. & Arli, A. C. (2021) *State Machines using VHDL FPGA Implementation of Serial Communication and Display Protocols*. 1st ed. 2021. [Online]. Cham: Springer International Publishing.

[11] Cohen, B. (1999) *VHDL Coding Styles and Methodologies*. 2nd ed. 1999. [Online]. New York, NY: Springer US.

This report is at an end and thank you very much to review our work.