Development of a Sprite-Based Architecture for Creating 2D Games in Reconfigurable Environments Using FPGA Devices

Gabriel Sá B. Alves[1], **Anfranserai M. Dias**[1], **Victor T. Sarinho**[2]

[1]State University of Feira de Santana - Technology Department

[2]State University of Feira de Santana
Lab. de Entretenimento Digital Aplicado – LEnDA

bielbarretoalves@gmail.com, anfranserai@uefs.br, vsarinho@uefs.br

Abstract. *The understanding of concepts worked on FPGAs combined with the creation of games in hardware platforms has helped in the understanding and assimilation of the necessary techniques and approaches to build digital game systems. This work presents the development and validation of a Sprite-Based Architecture that aims to develop 2D games in reconfigurable environments based on FPGA devices. To this end, a set of functions and other resources implemented in C language were develop, that aims to help the creation of games through the developed architecture.*

Keywords: *FPGA; 2D Games; Digital Systems.*

1. Introduction

For the development of electronic equipment, it is essential to understand the elements related to the creation of digital systems and circuits. Therefore, in courses such as Computer Engineering, Electrical and Electronics Engineering, universities use FPGA devices *(Field-Programmable Gate Array)* as a learning tool for the academic development of students on the elements of hardware, software and integration techniques.

FPGAs perform the physical simulation of sophisticated designs of circuits and digital systems and verify their operation on a physical device. In addition, there are several development boards on the market coupled with these chips, and structured with different peripherals, such as displays, buttons, among other input and output interfaces. These peripherals helps interaction with the system from the data entry to the graphic visualization of the obtained results.

Given these aspects, this work presents the development and validation of an architecture based on sprites that aim to develop two-dimensional games in reconfigurable environments using FPGA devices [Alves et al. 2021]. The intention here is to explore the possibility of provide a low cost portable video game for the creation of several 2D games using a single reconfigurable platform.

2. Related Work

Different academic projects are carried out in universities using the methodology of creating electronic games based on FPGA devices. Classic games such as *Snake* [Singla and Narula 2018], *Galaxian* [Xia et al. 2013], *Space Shoot* [Mishraa et al. 2017], *Pong* [Y.Q.Loh 2020], a functioning simuation of the sport cricket

[Egyir and Devendorf 2020] and others [Ali et al. 2021], are interesting examples of FPGA implementations. Such projects encourage students to develop skills in digital circuit modeling, system verification and testing. Most projects are modeled using HDL languages *(Hardware Description Language)* such as Verilog or VHDL *(VHSIC Hardware Description Language)* [Jiménez-Fernández et al. 2020] and synthesized through a CAD system, such as *Intel Quartus Prime*. However, this approach is limited in terms of the difficulty of creating new games, requiring the development of new control and animation hardware in each project.

Another approach is to use microcontrollers, where the development is carried out through a printed circuit board coupled with I/O peripherals and microcontrollers like Arduino. APIs (*Application Programming Interface*) are also available to facilitate game creation, game control and hardware access. The *GameBuino* [gam] and *Arduino Esplora* devices are some examples. However, this approach makes it difficult to use *sprites* and to make hardware improvements for the production of more complex games.

3. Proposed FPGA Architecture

Initially, was idealized the process of rendering and programming the games. The first step of the rendering process consists in the initialization of two memories responsible for storing the RGB color bits of the *sprites* and *background* of the screen. The second step corresponds to the print control of *sprites* and *background*. The graphic standard chosen was VGA (*Video Graphic Array*) with a resolution of 640x480 pixels. As a VGA monitor is scanned from left to right and from top to bottom, when the entire active area of the screen is scanned, the *sprites* can be updated, so that in the next *frame*, they are redrawn according to the new definitions.

For the game programming process, each game must be programmed using the C language through a general purpose processor. As the *sprites* information are stored at the hardware level, updates are made through instructions (commands) sent to the graphics module responsible for the game rendering process. The rendering process is independent, performing all control without the need of a software intervention. With this, the C language code will be responsible for defining the game logic and sending the commands to update the information pertaining to the objects to be rendered.

Based on this operating principle, the proposed architecture was modeled (Fig. 1). Its structure consists of a general purpose processor, two FIFOS (*First In First Out*), a PLL (*Phase Locked Loop*) and a Graphics Processor. Nios II was chosen to act as a general purpose processor, which consists of a 32-bit *softcore* RISC processor with Harvard architecture developed by the company Altera. Its function is to execute the source code in C language of the games that will be programmed. The Graphics Processor is responsible for managing the game rendering process and executing a set of instructions that allow initially moving and controlling the *sprites*, as well as modifying the *layout* of the *background* of the screen. This set of instructions is received through the ***dataA*** and ***dataB*** buses, as seen in Fig. 1. The main outputs of the Graphics Processor consist of the horizontal (***h_sync***) and vertical (***v_sync***) sync signals from the VGA monitor, and the RGB color bits. As Nios and Graphics Processor have different *clock* frequencies, FIFOs are being used as intermediary devices for communication. The PLL is responsible for generating the *clock* frequencies necessary for the correct functioning of the architecture.

Nios stores all instructions that must be executed by the Graphics Processor in FIFOs. With the signal ***start*** at high logic level, the Written Pulse module will generate a single write pulse in sync with ***wrclk***, therefore, the instruction data present on buses ***data A*** and ***data B*** will be stored in FIFOs. Each FIFO initially has the capacity to store 16 words of 32-bits. When the ***wrfull*** signal is at a high logic level, it means that the FIFOs have reached their maximum capacity. In this way, the FIFOs' internal protection circuit is automatically activated to avoid possible *Overflows*. FIFOs facilitate the process of modeling new instructions, without the need to include new buses and/or intermediary devices between Nios and the Graphics Processor.

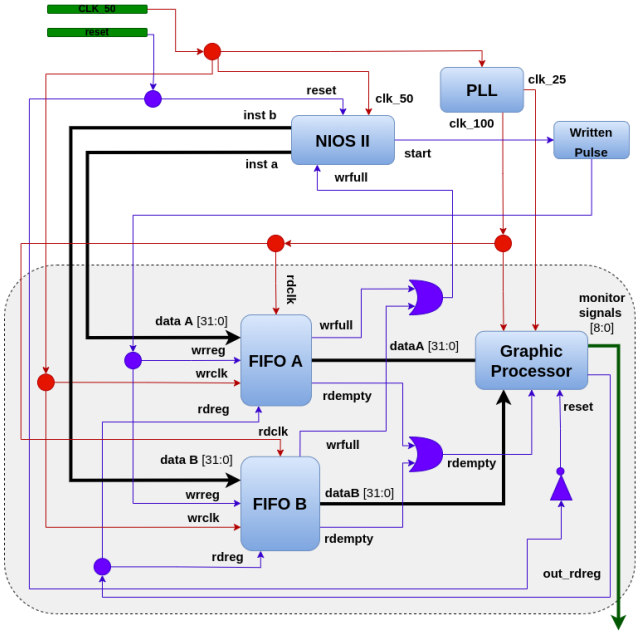

Figure 1. Representation of the proposed architecture.

4. Simplified API for the Production of 2D Games

After modeling and implementing the proposed architecture [Alves et al. 2021], some features in software such as functions, constants and composite types (*struct*) were built (Listing 1). Its objective is to abstract low-level aspects of the architecture and thus help in the programming of games using the C language. These resources are available in a header file (*graphic_processor.h*).

```
int set_sprite( int registrador, int x, int y, int offset, int
    activation_bit);

int set_background_block( int column, int line, int R, int G, int
     B);

int  set_background_color(int R, int G, int B);

void increase_coordinate(Sprite *sp, int mirror);

int  collision(Sprite *sp1, Sprite *sp2);
```

Listing 1. Auxiliary functions for the development of games and *sprites* control.

The **set_sprite** function is used to position a *sprite* on the screen. The field **Opcode** is defined internally, before sending the command to the Processor. Its return is 1 if the operation was sent successfully, and 0 otherwise. The **set_background_block** function is used to model the *background* by filling the 8x8 pixel blocks. Its parameters consist of the components **R**, **G**, **B** and the row and column value that represents the block to be filled. Internally, the row and column parameters are used to calculate the corresponding address in *Background* Memory. The **set_background_color** function is used to set the *background* base color, whose parameters consist of its RGB components. In an architecture, as the register used for the base color is fixed, the **Register** field is defined internally, as well as the **Opcode**. Its return is 1 if the operation was sent successfully, and 0 otherwise. The **increase_coordinate** function is responsible for updating the x and y coordinates of a moving *sprite* according to its movement angle and displacement value. Its parameters consist of passing by reference a variable of type *Sprite* and an integer value that informs whether the *sprite* coordinates should be mirrored when leaving the active area of the VGA monitor. After performing the update, it is necessary to use the **set_sprite** function to draw the *sprite* in the new coordinates. In the Listing 2, is presented two structs used to store *sprites* information.

```
typedef struct{
        int coord_x, coord_y;
        int direction, offset, data_register;
        int step_x, step_y;
        int active,  collision;
} Sprite;
typedef struct{
        int coord_x, coord_y, offset;
        int data_register, active;
} Sprite_Fixed;
```

Listing 2. Constants and structures for a game production.

The **collision** function is used to check if a collision has occurred between two *sprites*. It implements the Rectangle Overlay technique, using the height and width of a *sprite* to define the boundary areas. It is checked whether the print area of one *sprite* is inside the print area of the other. Its parameters consist of passing by reference two variables of type *Sprite* with their attributes **coord_x** and **coord_y** properly filled in. Its return is 1 if the collision is detected, and 0 otherwise.

5. Obtained Results

Regarding the game loop of the proposed architecture, Figure 2 illustrates a general representation of the execution flow for the developed games. The first action performed is the initialization of the *sprites* that will be used during the game. This process consists of declaring and initializing variables of composite types that will store the *sprites* information. For this, the *struct Sprite* presented previously was used. Then, through the *set_sprite*, *set_background_color* and *set_background_block* functions, the initial *sprites*, the background base color and the scenario construction (in the case of *Space Invaders*) are introduced in the screen.

Starting the main loop, there is the reading of the control buttons. This reading is done through the function ***IORD(base, 0)*** available in the file *altera_avalon_pio_regs.h*.

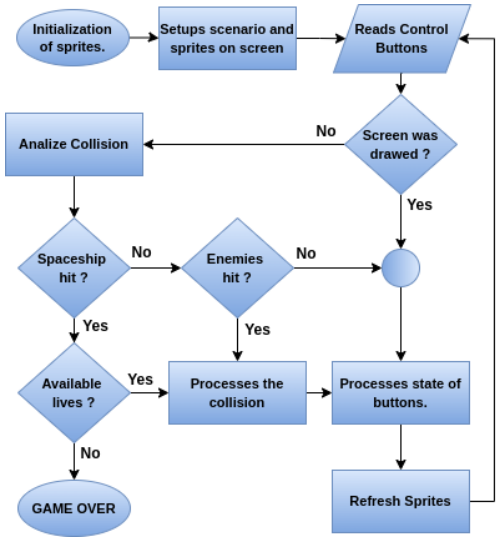

Figure 2. Game execution flow.

The *base* parameter consists of the memory address of the registers that store the state of each button. These addresses are found in the *system.h* file. Both libraries were developed by the company Altera to facilitate access to peripherals connected to the Nios II processor system. The access to these libraries is done after creating the project in the Eclipse IDE integrated with the Quartus Prime platform. Once the buttons are read, it is checked whether the time to scan a screen has already been reached.

The memory address is also used in the IORD function to check whether the instructions sent to Graphic Processor were read. If a screen has been drawn, the instructions was executed and the game is in progress, then the *sprites* are updated through the *increase_coordinate* function and the update commands transmitted to the Graphics Processor through the *set_sprite* function.

According to the game loop (Figure 2), if the scan time is not reached, collision processing between *sprites* is performed. If the scan time has been reached, the states of each control button are processed, and the *sprites* are updated according to the new information. For the *Asteroids* game, the player will only have one life, so if the ship is destroyed the game is over. For *Space Invaders* game (Figure 3), the player will have 3 lives. If the ship is hit and it has extra lives, the game is returned to its normal execution after a period of 4 seconds from the game level in which the player is.

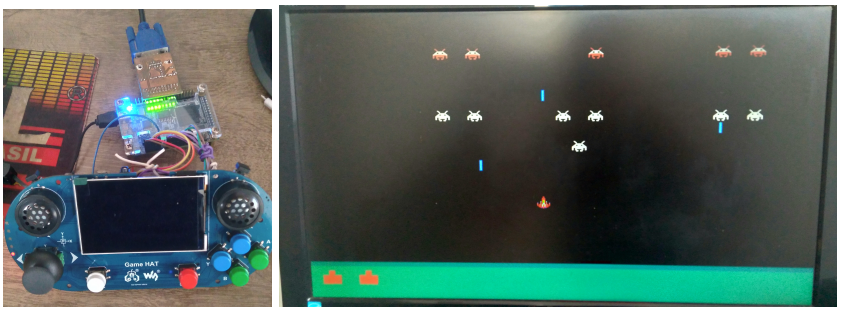

Figure 3. Initial prototype and the developed Space Invaders game.

Regarding the initial prototype, the entire architecture was compiled and synthesized on the Quartus Prime Lite 20.1 platform. Signals for movement control, ship firing, game pause and game resume operations are obtained through external hardware with buttons and a *joystick* (Figure 3). Execution tests were performed by the DEO-Nano development board coupled with the FPGA *Altera Cyclone IV EP4CE22F17C6N* chip. As identified problem, performing an animation in the limit of 31 *sprites* on the screen causes a loss in the update speed due to the limit of sending 13 instructions per *frame*. It is intended to increase this value, in order to allow the control of more *sprites* in the same *frame*, thus achieving a better performance in the execution of the implemented games.

6. Conclusions and Future Work

This work presented the validation of an architecture based on sprites that aim to create two-dimensional games in reconfigurable environments. To this end, two games was create based on functions and others resources implemented in C language that aims to help the creation of games through the developed architecture.

As future work, new functionalities for the Graphic Processor will be developed, such as the processing of geometric shapes and the change from the VGA graphic standard to HDMI (*High Definition Multimedia Interface*), in a way that allows the display of games directly on the external hardware display seen in Figure 3. A sound module will also be developed, in order to guarantee to the user a better immersion experience during all phases of the developed games. Finally, it is also necessary to apply the use of the architecture by other game developers, in order to assess the production environment as a whole, the required learning curve, possible learning difficulties, and the architecture's ability to develop different types of game genres.

References

Gamebuino. `https://gamebuino.com/`. Accessed in: 2022-01-18.

Ali, S. I. et al. (2021). A strategical approach for implementing digital games on fpga. *Turkish Journal of Computer and Mathematics Education (TURCOMAT)*, 12(10):3539–3549.

Alves, G. S. B., Dias, A. M., and Bittencourt, J. C. N. (2021). Collenda: A games development platform in reconfigurable environments using fpga devices.

Egyir, R. J. and Devendorf, R. P. (2020). Design of a cricket game using fpgas.

Jiménez-Fernández, C. J., Oliva, C. B., Fernández, P. P., Soto, A. G., Ordóñez, F. E. P., and Barrero, M. V. (2020). Learning vhdl through teamwork fpga game design. In *2020 XIV Technologies Applied to Electronics Teaching Conference (TAEE)*, pages 1–5.

Mishraa, A., Kumarb, A., and Pariharc, R. (2017). Design and fpga implementation of space shoot game. *International Journal of Control Theory and Applications*, 10(30).

Singla, N. and Narula, M. S. (2018). Fpga implementation of snake game using verilog hdl.

Xia, S.-M., Xu, X.-L., Qin, L., and Liu, C.-H. (2013). Galaxian game on altera de2-115 fpga architecture.

Y.Q.Loh (2020). Designing a game on fpga using verilog.