Collenda: A Games Development Platform in reconfigurable environments using FPGA devices.

Gabriel Sá Barreto Alves
Technology Department
State University of Feira de Santana
Feira de Santana, Brazil
bielbarretoalves@gmail.com

Anfranserai Morais Dias
Technology Department
State University of Feira de Santana
Feira de Santana, Brazil
anfranserai@ecomp.uefs.br

João Carlos Nunes Bittencourt
Center of Exact and Technological Sciences
Federal University of Recôncavo da Bahia
Cruz das Almas, Brazil
joaocarlos@ufrb.edu.br

***Abstract*—In the last years, games has been used for teach software aspects, for example, programming logic, artificial intelligence, data structure, and others. This paper presents the validation of a GPU architecture based on sprites which aims to teach and integrate hardware and software elements through game programming in reconfigurable environments using FPGA devices. For this purpose, the project structure, execution process, instruction set, and some information about architecture of this video processor are described. In conclusion, some tests are described to prove the correct performance of modules that are essentials to the working of the project.**

***Index Terms*—Games, FPGAs, Hardware, Software**

I. Introduction

The use of *FPGA (Field-Programmable Gate Array)* as learning tool has increased significantly within university laboratories, allowing to create playful and interactive activities through the synthesis of digital circuits. For example, the creation of games using FPGA. The games are into the category of activities that provide great learning flexibility [2]. It helps assimilation of the techniques and approaches associated with integration between hardware and software elements [1] [3].

This project is the initial development of a teaching platform that uses games as the main motivation. Its purpose is to facilitate the comprehension related to hardware elements, as memory, registers, and others element of digital circuits, as well as, programming aspects through games developed with the platform.

This paper presents the validation of a GPU *(Graphics Processing Unit)* architecture based on sprites, that aims to teach and integrate hardware and software elements, through games programming in reconfigurable environments using FPGA devices.

In conclusion, will be shown tests for validating some modules present in the architecture. Such modules are essentials for the correct performance of the video processor.

II. Related Work

It's possible to find many projects related to games and learning about hardware and software. For example, games developed using microcontrollers or specific hardware, such as the FPGAs. The games through microcontrollers are implemented using a programming language, for example, the C language. The development is done through APIs that help create and control the game. A example, consists in devices based on Arduino, as *GameBuino* [5]. This approach is limited regarding the design of sprites and animations, requiring the creation of new hardware using a printed circuit board to help this activity.

The games developed in FPGA can be created through hardware description language (HDL) and its structure built using a synthesis tool, as Intel Quartus Prime. This approach is limited regarding the difficulty of creating new games, requiring each game to develop new hardware for animation and control. Example of projects with this approach can be found through classic games as Pong [3] and Tetris [4], implemented using hardware modules developed with an HDL language.

Comparing these approaches with the architecture proposed in this paper, the user can integrate the GPU developed with new hardware modules or use the instruction set through a programming language. For this reason, the user can use the high-level approach, as performed in microcontrollers, and the low-level approach, interconnecting hardware modules.

A similar project, but with a different approach other than Collenda, is the *MiSTer* [7]. It consists in a open source hardware implementation for consoles and old computers. This project re-implements old machines, such as Arcade, Atari, and others. *MiSTer* uses the *Terasic DE10-Nano* development board to controls most of the system's core, and a Linux kernel in which it allows the emulation of games using the hardware implemented within the FPGA.

Differently of these projects, the Collenda has the purpose of creating an environment in which the user will be able to develop games (as the examples previously mentioned) through hardware structuring, high level programming or a combination of both approaches.

III. GPU Architecture

This version of the GPU executes a restricted instruction set that allows initially to move and controls sprites on a VGA monitor with resolution of 640x480 pixels.

The GPU architecture can be integrate with others elements of hardware, and its instructions are received through of 2 data buses (dataA and dataB), according to Figure 1. This structure

is used because the first instruction developed needs more than 32 bits, and as Collenda was designed to be integrated with *Altera Nios II processor*, the same 32-bit structure was adopted. The instruction fields need to be sent at the same time for decoding, for this reason, it was necessary to insert these two input buses. The GPU also has an output bus that consist in the signals to the VGA monitor. Figure 1 shows the current version of the GPU architecture that has been implemented.

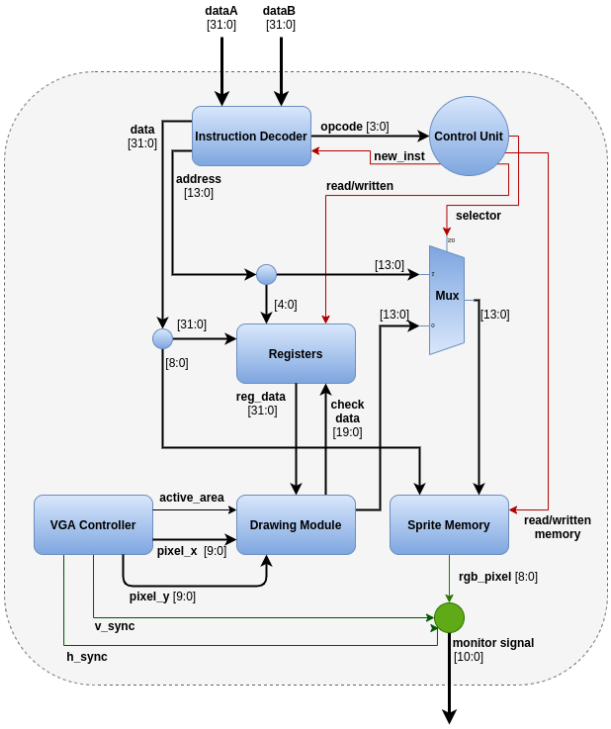

Fig. 1. Current Version of GPU's Architecture.

The Control Unit consists of a *State Machine* responsible for managing the reading, decode and execution process of instructions. The Registers are responsible for storing temporary information (coordinates, memory offset, and an activation bit) associated with sprites. These information are used to draw each sprite on the screen. The Drawing Module is responsible for managing the draw pixel process of the sprites and background color on the VGA monitor screen. The VGA Controller manages the monitor signals. This project has used a resolution of 640x480 pixels that is obtained with a clock frequency of 25Mhz. The Sprite Memory stores the bitmap for each sprite. In this version, the Sprite Memory consists in 16.384 words of 9-bits, 3 bits for each RGB component, and its memory positions are filled in synthesis time. Since the sprites size used are 20x20, each sprite will take 400 memory positions. It's possible to store 40 different sprites.

A. Instruction Set

The GPU has the following machine instructions in its current version:

Define Sprite (DS): This instruction setups the x and y sprite coordinates. In this instruction, the offset has 9-bits and will be used by the GPU to calculate the initial memory address of the bitmap that contains the colors for each sprite pixel. The instruction has 2 fields of 10-bits that inform the sprite's coordinates on the screen. Also, in this instruction, exists a bit confirming whether the sprite is activated or not, in other words, if it should be drawn at the moment. This bit is marked as *sp[29]* in Figure 2. These information are received through a 32-bit data bus, identified as dataB, in Figure 2. The dataA, carries the instructions operation code (opcode) and the registers addresses, as depicted in Figure 2. The information sent though the dataB input is stored into the Registers, according to the address carried in the instruction.

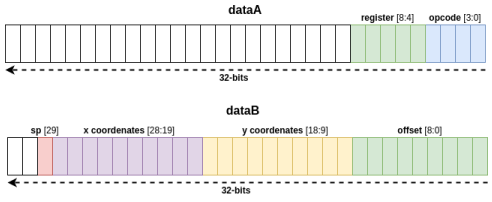

Fig. 2. Format of the Define Sprite Instruction.

Writing in the Sprite Memory (WSM): This instruction store or modify the content present in Sprite Memory, according to the address carried in the instruction. Its possible to change the background color of the screen using this instruction. The operation code (opcode) and the memory address are received through 32-bit input bus, identified as dataA (Figure 3). In the dataB, the R, G, and B signals are the RGB components for each pixel on the screen (Figure 3). The last memory address corresponds to the background colors of the VGA monitor.

Fig. 3. Format of the Memory Written Instruction.

B. Draw Pixel Process

The draw pixel process is the method used to draw sprites and background on the screen. This step are managed by the Drawing Module. It works together with other modules, such as the Registers for reading data for each sprite, the VGA Controller that informs the current x and y coordinates of the monitor, the active area signal [6], and the Sprite Memory.

The information on a VGA monitor is displayed horizontally, moving in the "forward" direction (left to right and top to bottom), for this reason, the sprites are drawn line by line. Every time the VGA Controller begins the addressable video area (active area) available to draw pixels, the Drawing Module starts its processing.

When a sprite is detected, the specific line of the sprite is drawn according to the current coordinates of the monitor

and the memory addresses that are calculated through the information stored in Registers. For each pixel, a new memory address is calculated. These addresses are used to access the bitmap of the sprites on the Sprite Memory. To use a sprite, it's necessary to select which one to use and store the sprite's data within the Registers. The Registers has 32 registers of 32-bits to store the information of each one. Figure 4 presents the execution flow of the Drawing Module.

The VGA Controller has two counters. The horizontal to count pixels in each line and another (vertical counter) to count lines in a frame. First, it's verified whether these counters are in the active area according to the signal sent by VGA Controller. When the monitor is in the active area, the current x and y coordinates provided also by this controller are sent for comparison between the coordinates of all sprites that have been stored in the Registers.

This comparison allows you to check whether the current coordinate belongs to a sprite. The result defines whether the sprite must be drawn. Pixels that are outside to a sprite belongs to background.

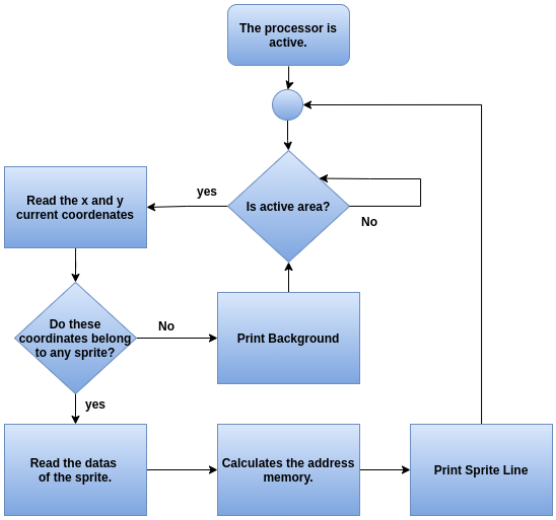

Fig. 4. Execution flow of the Drawing Module.

IV. Results

This section aims to include the tests that were carried out to validate the operation of modules present in GPU's architecture, as the Control Unit, Instruction Decoder and the Drawing Module. The correct operation ensures that GPU works as expected.

A. Instruction Decoder:

The following test aims to validate the Instruction Decoder through the insertion of data in the input buses (***dataA*** and ***dataB***) and check the answer produced in the output signals after the processing.

For the test, it was used two instructions in the simulation. The first simulates the receipt of an instruction to write in the Sprite Memory, and the second, an instruction to set up information of a sprite. After processing, intends to observe the operation code, address, and content to be store. In Figure 5, the signal lines and buses are counts from top to bottom.

According to Figure 5, it's possible to note that in the first rising edge of the signal ***clk_en*** (line 1) the insertion of data in ***dataA*** (line 3) and ***dataB*** (line 4) buses is done. However, before this first edge, there wasn't valid data on the buses and the ***new_instruction*** signal (line 2) was at a high logic level, so that, no instruction is decoded. This ensures a default value on all output buses and nothing will be executed by the GPU.

In the second rising edge of the ***clk_en*** (at 30ns), the first instruction were processed and the output is specified as expected. Line 5, present the opcode value, the line 6 present the address that was received, and the line 7 present the content to be store in the Sprite Memory.

See that, in this new pulse, the ***new_instruction*** signal is placed again in high logic level and default value is generated in the next rising edge.

At 50ns, in the third rising edge, the second instruction is inserted. The ***new_instruction*** signal is disabled allowing for new instructions to be decoded, and the outputs are exposed on the fourth rising edge at 70ns generating the correct output as expected.

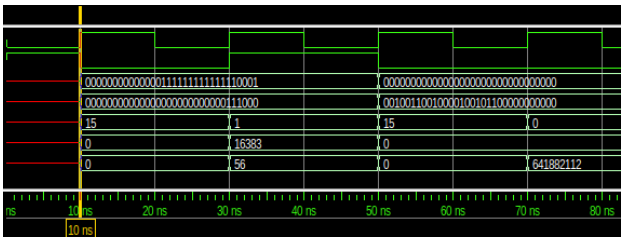

Fig. 5. Instruction Decoder Simulation

B. Control Unit:

This test aims to ensure the state transitions are working as expected so that everything is defined at the correct time. The Control Unit works with a frequency of $f = 100Mhz$ ($T = 10ns$). In Figure 6, the signal lines and buses are counts from top to bottom.

For the Control Unit validation was specified an inputs sequence that corresponds to its input buses, as shown in Figure 6:

- Line 3 - **opcode**: Identification of the instruction to be executed.
- Line 4 - **printtingScreen**: It informs if the screen is being drawing or not.
- Line 5 - **doneInst**: Execution status of an instruction.
- Line 6 - **fifo_empty**: Status of the instruction queue.

The tests were based on the state transition between the writing instructions in the Sprite Memory and the Registers. The output buses were not considered to minimize the complexity of this analysis.

In Figure 6, from 25ns to 105ns, the execution process of the instructions is carried out. When the ***doneInst*** signal is

placed at the high logic level, the unit returns to the initial state. It represents that the instruction finished its execution, thus, it restarting the reading process and executing a new instruction.

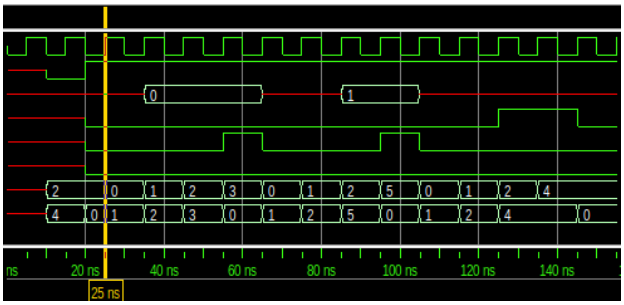

Fig. 6. Control Unit Simulation

From of 105ns the reading process of a new instruction is initiated. The ***printtingScreen*** shows that new processing for the screen has started. The Control Unit is redirected and keeps waiting until the screen is drawn. Consequently, halting the execution of new instructions in this period, since, the Drawing Module uses the data from the Sprite Memory and the Registers. Each instruction is executed with 4 clock pulses. Within this time interval, a new instruction is read, decoded and executed.

C. Drawing Module:

This test aims to verify if the memory addresses are been calculated correctly or not, by the Drawing Module. The test was based on the data insertion in the input buses to simulate the execution process of the module. Also, in Figure 7, the signal lines and buses are counts from top to bottom.

It's possible to analyze, in Figure 7, after the Drawing Module detects the x and y coordinate values (line 6 and 7) and the ***sprite_on*** signal (line 9) is set to a high logic level, it starts the calculating process for the memory addresses in order to access the color bits of each sprite's pixel detected. Note that, the area for drawing pixels on the monitor is flagged by the ***active_area*** input signal (line 5). Consequently, this process only starts when it's at the high logic level.

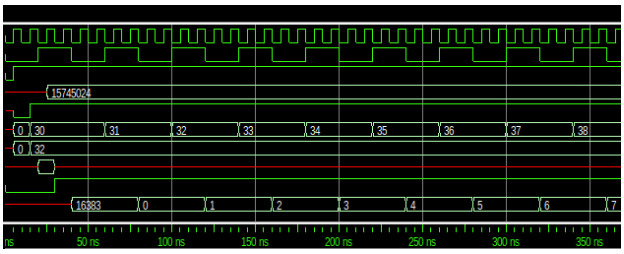

Fig. 7. Drawing Module Simulation.

The addresses are generated at each falling edge of *clk_pixel* (line 2) with a period of $T = 40ns$ and frequency $f = 25Mhz$, in accordance with the operation of the VGA Controller. At each rising edge of the *clk_pixel*, a new coordinate is generated, and in the falling edge, if a *sprite* has been detected, the corresponding memory address is calculated. As the *sprites* have a default size of 20x20 pixels, in each sprite line, 20 new addresses are calculated for every sprite detected. The addresses are generated according to the memory offset for each sprite.

V. Conclusion

This paper presented the architecture, instruction set, and validation of a GPU based on sprites that allow move and controls elements on a VGA monitor with resolution of 640×480 pixels. This project has been proven to be a viable solution to games development and the building a teaching platform using games as the main motivation.

This project is still under development, with some other steps to be improved and validated, such as its integration with *NIOS II* processor. This will allow a developer to use Collenda to program a game using the C language. We also intend to develop improvements in the sprites storage, in order to provide better flexibility to the programmer, and the development of input interfaces that allow player interaction with games.

References

[1] Sanchez-Elez, M., and S. Roman. "Learning Hardware Design by Implementing Student's Video-Game on a FPGA." Frontiers in Education: Computer Science and Computer Engineering (2015): 24-30.

[2] Squire, Kurt. "Video games and learning." Teaching and participatory culture in the digital age (2011).

[3] R. Szabó and A. Gontean, "Pong game on FPGA with CRT or LCD display and push button controls," 2014 Federated Conference on Computer Science and Information Systems, 2014, pp. 729-734, doi: 10.15439/2014F181.

[4] K. Liu, Y. Yang and Y. Zhu, "Tetris game design based on the FPGA," 2012 2nd International Conference on Consumer Electronics, Communications and Networks (CECNet), 2012, pp. 2925-2928, doi: 10.1109/CECNet.2012.6201435.

[5] GAMEBUINO. GameBuino. Available in: "https://gamebuino.com/". Accessed in: March 26th 2021

[6] Santos, Italo S., Joao Carlos N. Bittencourt, and Anfranserai M. Dias. "Desenvolvimento de um Jogo de Corrida em FPGA."

[7] MiSTer. Available in: "https://github.com/MiSTer-devel/Main_MiSTer/wiki". Accessed in: May 25th 2021