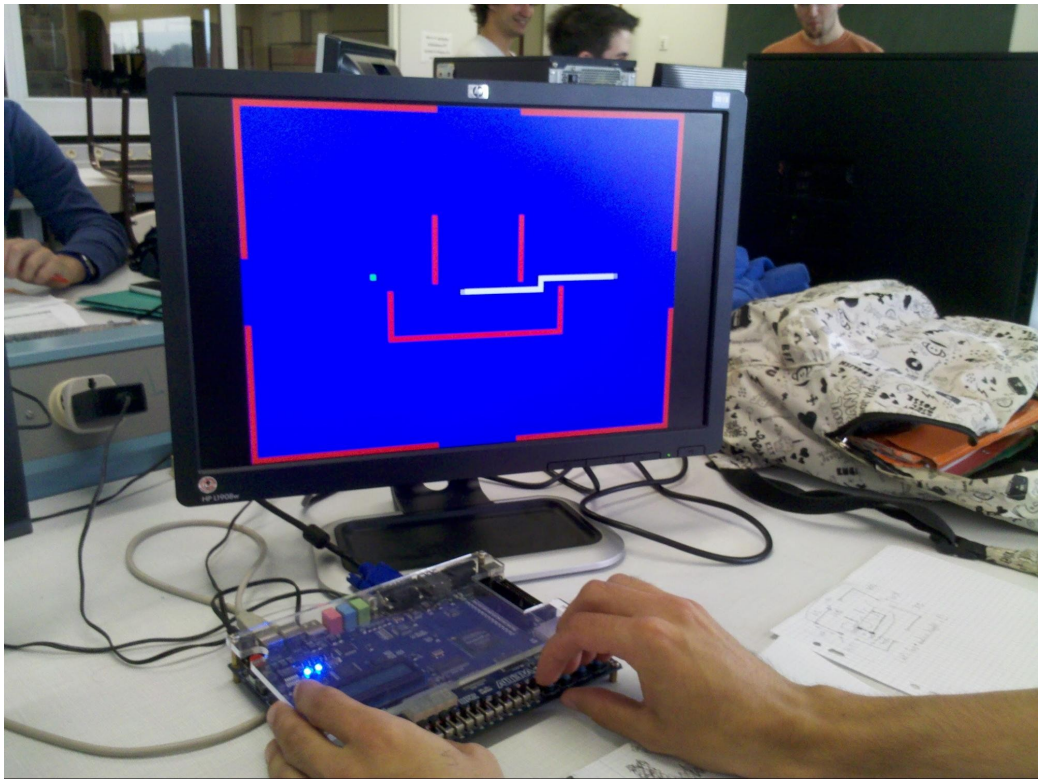
Corentin GUEZENOC
Miguel SÁNCHEZ DE LEÓN PEQUE
3A Majeure SERI

Rapport de TL - FPGA
Conception et implémentation d'un Snake sur carte FPGA Cyclone II



Abstract

Nous présentons dans ce rapport le travail que nous avons effectué dans le cadre du TL sur les FPGA : la conception et l'implémentation d'un jeu de Snake. Nous présentons en introduction la carte FPGA utilisée ainsi que le cahier des charges que nous nous sommes fixés, puis nous exposons l'implémentation matérielle du jeu : l'agencement de la mémoire, le fonctionnement du dispositif d'affichage à l'écran, l'utilisation du microprocesseur et l'utilisation des ressources de la carte Cyclone II. Ensuite, nous développons au sujet de l'implémentation logicielle : la manière dont nous avons géré le mouvement du snake, l'utilisation d'interruptions et l'interface utilisateur. Enfin, nous concluons en comparant le produit réalisé au cahier des charges, pour ensuite proposer des pistes d'améliorations futures.

I. Introduction

FPGA development board

For this project, we used DE2 development board by Altera. It features an Altera Cyclone II 2C35 FPGA, different I/O interfaces, memory, displays, switches, LEDs, clocks…

In this particular project, we are using the available push buttons, the 50 MHz clock and the VGA display port (apart, of course, of the Cyclone II FPGA, in which we will be embedding a Nios soft core).

Specifications

The gameplay basics can be described as such : "The player controls a long, thin creature, resembling a snake, which roams around on a bordered plane, picking up food (or some other item), trying to avoid hitting its own tail or the "walls" that surround the playing area. Each time the snake eats a piece of food, its tail grows longer, making the game increasingly difficult. The user controls the direction of the snake's head (up, down, left, or right), and the snake's body follows. The player cannot stop the snake from moving while the game is in progress, and cannot make the snake go in reverse" (source : Wikipedia).

Inspired by this description, our specifications for the Snake game will be the following :

- The player controls a long and thin creature which roams around on a rectangular plane in order to pick up food
- The player controls the direction of the head (up, down, left or right)
- The snake cannot go in reverse
- The player cannot stop the snake from moving

- If the snake hits its tail or a wall, the game is over
- When the snake eats food, the tail grows longer and the snake’s speed increases, making the game increasingly difficult
- The game will propose several levels with different wall configurations
- The game will propose several map sizes
- The game will display scores when it is over

II. Hardware implementation

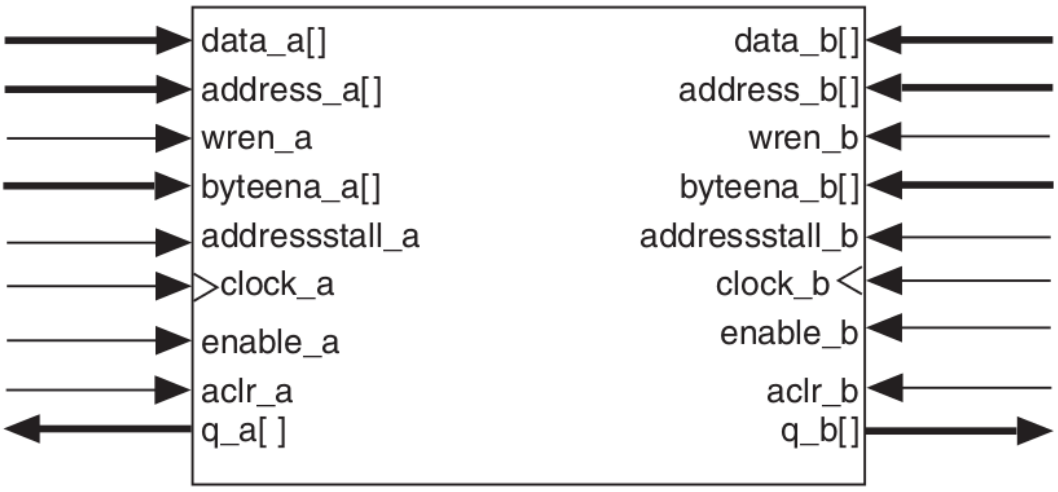
Memory

One of the most useful features of the FPGAs is the on-chip RAM memory. Both Xilinx and Altera, the two main FPGA manufacturers, create devices with this on-chip distributed memory. This memory structures address the on-chip memory needs of FPGA designs: RAM, FIFO buffers, ROM...

Cyclone II devices feature M4K memory blocks which provide:

- Over 4096 memory bits per block (4608 including parity bits)
- TDP operation.
- Input masking during writes (byte enables).
- Up to 250 MHz operation.

TDP configuration allows simultaneous read/write operations.



One port is used by the processor for reading and writing operations, the other is used by the display logic, only for reading.

As we already mentioned this memory blocks can operate at up to 250 MHz, so we'll analyze if that is enough for our purpose:

- uP interactions: the uP needs to make only a few modifications (i.e. update two data values: tail and head or up to three with the food) in each frame. The rest of the matrix (walls, snake body and empty areas) will remain unmodified each frame. The idea is to interrupt the processor each millisecond to execute the main operations. A millisecond period is equivalent to a 1 KHz operation which is, by far, a lower frequency that the block memory operation.
- Displaying interactions: to display all the data stored through the VGA port we need to, in the worst scenario, read data at a frequency of ~25 MHz. That is again, by far, a lower frequency than the block memory operation.

The memory structure will be the result of combining a multiplexed number of memory blocks. The number of multiplexed stages will increase with the number of memory blocks added to the memory structure, but even taking into account the synthesizer inefficiency and the resulted path delays, the memory structure frequency operation will not be decreased below 25 MHz, which is the minimum MFO needed.

Therefore, the decision taken was to create a memory structure that simulates a big on-chip memory of the total desired size:

Memory size = 3 bits * max_matrix_width * max_matrix_height

Where:

- 3 bits are used for each spot and represent the possible state of it (8 possible states): empty, wall, food, head, and the four possible link relations of a body piece. Food and head states could be out of the states enumeration, but as long as we can not reduce the number of states to 4, we would be needing 3 bits to represent the state and, therefore, we can include this two states in the memory structure, simplifying again the design.
- max_matrix_width: arbitrary value that sets the maximum matrix width, which has been set to 160, for playability and resource-saving reasons. With this value, we can divide the VGA frame into 160 squares of 4 pixels width, which is the smallest size that we consider playable.
- max_matrix_height: has been set to 120 for the same reasons.

The 8 possible states for each memory address are enumerated below:

- 000: empty
- 001: snake
- 010: walls
- 011: food
- 1xx: chain relation

    - 100: chained upward address
    - 101: chained rightward address
    - 110: chained downward address
    - 111: chained leftward address

Although the memory reads operations could have been reduced, setting a bigger data port width, that would have also required a more complex logic interface to translate all the data read to the corresponding display information. Therefore and due to the non-limiting frequency operation requirements, a data port width of 3 bits have been chosen.

The displaying memory interface port is reading data at the VGA frequency, so this interface has been simplified a lot: we check which pixel we are displaying and we read the corresponding value for that pixel. The uP memory interface port is reading and writing at 50 MHz, which is not really necessary as the uP read/write operations are not needed to be fast and, furthermore, the uP can not change the output bits (and therefore the data, read enable and address ports) at that speed. Anyway this frequency has been selected to test the memory structure performance, which we already supposed could operate at higher frequencies than 25 MHz (and 50 MHz as we have already proved).

The resulting memory structure has then:

- 160 * 120 = 19200 addresses.
- 3 bits data ports.
- Read/write operation for the uP interface.
- Read operation for the display interface.
- 50 MHz and 25 MHz clock frequency for the uP and display ports, respectively.

The memory blocks have also been configured to be tolerant to read and write simultaneous operations using a WAR strategy and, therefore, showing the old data to the read request in case of collision.

Displaying

The game is displayed on a 480*640 pixels screen using VGA standard. To implement the displaying module, we use four modules : *cptadpixel*, *synchro*, *reader* and *rvb*.

- The two first ones are used to generate a one-bit signal (video) that indicates to the *rvb* module when to display images on the screen, and two signals *adr_pixel* and *adr_ligne* which are counters that go through the screen matrix at the frequency VGA_CLK.
- The *reader* module’s function is to convert the *adr_pixel* and *adr_ligne* signals into a memory address signal *adrmem* that will correspond to the way we use the DPRAM, according to the value of the input *level* (size of the gaming map).
- The *rvb* module takes as an input what the DPRAM reads at the address *adrmem* and convert it into colors on the screen :

    - 000 = empty : blue
    - 001 = snake head : white
    - 010 = walls : red
    - 011 = food : green
    - 1xx = snake body : white

Also:

- We have implemented a link - named level - from the processor to the displaying module, which commands the size of the playing map used, i.e. the size of an elementary square on the screen.

Microprocessor

The design has a Nios II embedded processor (soft core), provided by the manufacturer (Altera) as an IP. Nios II is a 32-bit embedded-processor architecture designed specifically for the Altera family of FPGAs.

Adding an embedded soft core to the design is resource consuming and, as there is no sleep or standby mode for the processor, it is a good idea to utilize it as much as possible. Also, the uP flexibility is a plus for non-critical speed tasks. That's why most of the game logic/rules have been software implemented.

Here is a list of some of the peripherals that have been used (excluding some which are part of the base Nios system, like the SDRAM controller, etc.):

- Clock source (clk_0): connected to the DE2 50 MHz clock.
- Interval Timer (timer_1): used to trigger system ticks interruptions each millisecond.
- PIOs:
    - input_mem_data: read memory data.
    - output_mem_adr: memory address.
    - output_mem_wren: memory write enable bit.
    - output_mem_data: data to be written in the memory.
    - input8bits: push buttons (attached interruptions).
    - output_level: tells the hardware display reader the size of the memory matrix.

Resources utilization

Summary for normal effort and balanced optimization technique synthesis:

- Total logic elements: 5047/33216 (**15%**)
    - Total combinational functions: 4514/33216 (**14%**)
    - Dedicated logic registers: 3135/33216 (**9%**)

- Total registers: 3203
- Total pins: 76/475 (**16 %**)
- Total memory bits: 187776/486840 (**39%**)
- Embedded Multiplier 9-bit elements: 4/70 (**6%**)
- Total PLLs: 1/4 (**25%**)

Summary for extra effort and area optimization technique synthesis:

- Total logic elements: 4961/33216 (**15%**)
    - Total combinational functions: 4389/33216 (**13%**)
    - Dedicated logic registers: 3135/33216 (**9%**)
- Total registers: 3203
- Total pins: 76/475 (**16 %**)
- Total memory bits: 187776/486840 (**39%**)
- Embedded Multiplier 9-bit elements: 4/70 (**6%**)
- Total PLLs: 1/4 (**25%**)

As we can see, little area optimization has been achieved, reducing between 0-1% the resource utilization compared with the balanced optimization technique synthesis.

Summary for fast effort and area optimization technique synthesis:

- Total logic elements: 4959/33216 (**15%**)
    - Total combinational functions: 4386/33216 (**13%**)
    - Dedicated logic registers: 3135/33216 (**9%**)
- Total registers: 3203
- Total pins: 76/475 (**16 %**)
- Total memory bits: 187776/486840 (**39%**)
- Embedded Multiplier 9-bit elements: 4/70 (**6%**)
- Total PLLs: 1/4 (**25%**)

Almost no difference compared to extra effort synthesis report. That is mainly because the Nios processor is occupying most of the logic area:

- Total combinational functions: 3844/4386 (**88%**)
- Dedicated logic registers: 2994/3135 (**96%**)
- Total memory bits: 64896/187776 (**35%**)

Only area optimization technique has been tested, as speed is not a constraint for this project.

III. Software implementation

The game is basically a state machine with the following states: *welcome*, *select_size*, *select_walls*, *ready*, *play* and *score*. Those states will be described later on the “Interruptions” section.

Snake’s movement

The snake has a discrete movement. Indeed, we choose a frequence (the game speed), at which we update the snake’s head and tail. To update the head, the program updates the memory square corresponding to the present head, replacing the *head* state by a *up*, *down*, *left* or *right* state according to the player’s choice of direction. The position of the future head is also calculated, and the correspondent memory square is updated to the *head* state. The tail’s update is performed simply by erasing the present tail, which means the program writes the *empty* state in the corresponding memory square. Thanks to this strategy, the whole snake is kept in memory and, more importantly, the direction of each piece of it, making the program able to give to it this “sneaky” movement. Indeed, one of the major problems in this project is to implement it so that the tail passes by every place the head has visited before, in the same order.

As noted in the specifications, the goal of the game is for the snake to eat as much food (randomly disposed on the game map) as possible. To make the game increasingly difficult, we make it grow and go faster each time it eats a food item. The implementation of the growth is simple : when the snakes eats food, its tail is not updated at its next headway. Thus, the snake’s size increases by one elementary square. As for the speed increase, each time the snake eats food, its speed (global variable) is multiplied by a constant factor.

Interruptions

The main function does nothing but initialize some parameters and configure the interruptions, and then stucks at an infinite loop doing nothing (sadly, there is no sleep mode for the Nios processor).

The first interruption is attached to the DE2 buttons events. The function that handles the interruption will behave in a different way depending on the game state:

1. ***welcome***: For the welcome screen, it will just basically change to the next state.
2. ***select_size***: For the size selection screen, it will print the snake for the selected size (depending on the button pressed) and wait for confirmation until the user presses again that button. After confirmation, some variables will be initialized so set the matrix size, and the game state will be changed.
3. ***select_walls***: For the walls selection screen, the behavior will be similar. Different walls will be drawn until the user confirms the selection clicking again the same button. After confirmation, the first food will be drawn at a random position and the game state will be changed.

4. ***ready***: In this game state the user will see the snake, the walls and the first food drawn in the screen. Once he/she presses a button, the game will start. Game state will be changed to “play”.
5. ***play***: In this game state we will only store the last pressed button. The system tick interruptions will handle, for each frame, the update of the snake. Therefore, the game state will be changed in this case from the system tick interruptions handler, in case a collision occurred during the game.
6. ***score***: This is the final state, in which the game stops and the score is displayed. A new event captured by the buttons interruptions will change the state back to the initial “welcome” state.

The second interruption is configured to be executed automatically each 1ms. Basically, it is a counter that, depending on the level, would count up to a number of milliseconds. When the count reaches the target:

- Head position is updated.
- Score is decremented a little bit (this way you get a penalization if you spend too much time moving around without eating food).
- If no food was eaten, then the tail will be updated too.
- If food was eaten, then the tail will not be updated and the score will be incremented.
- If there was a collision (with a wall or the snake’s body), then the game is over and the score is displayed.
- Snake speed is incremented (that means the frame period will be decremented by a factor).
- Counter of milliseconds is set to 0 for the next frame.

User interface

The user interacts with the system through the four available push buttons in the DE2 board.

The user interface is divided into different screens depending on the game state:

- ***Welcome screen***: it is just a white screen, waiting for the user to push a button in order to change the game state.
- ***Size selection screen***: it is a blue screen. The snake (in white) will be drawn upon user interaction (different sizes will be displayed for different buttons). Once the user has found the desired matrix size, he/she can confirm pressing again the same button.
- ***Walls selection screen***: in this case different walls will be displayed (in red) upon user interaction. Again, once the user has found the desired matrix size, confirmation is expected by pressing again the same button. Once the user confirms, the first food (in green) will be displayed and the game will wait again for the user to press any button before starting the game.

- ***Play screen***: in this screen the background will be blue, the snake will be displayed in white, the walls in red and the food in green. It is the same as in the walls selection screen. The only difference is that in this case the snake will be moving!
- ***Score screen***: upon collision, the game will be finished and a red screen will be displayed. When the user presses a new button, the welcome screen will be displayed again.

IV. Conclusion

We have reached every goal we had set in the specifications, including up to four different wall configurations and up to four different maps, making the player able to experiment sixteen different game environments. It also offers increasing difficulty, making the experience more entertaining.

We have achieved a system that utilices **less than 15%** of the FPGA resources, except for the:

- Total memory bits: 187776/486840 (**39%**), which was necessary to store the 19200 elements of the memory matrix in order to create the highest playable matrix size (160x120). Moreover we have to take into account that the Nios softcore is making use of 35% of the memory we reserved.
- Total PLLs: 1/4 (**25%**).

In order to reduce the amount of memory utilized, we should make use of an external memory, which was not allowed in this project.

V. Future work

Although the most important parts of the game have been already implemented there is some work that could be done in some areas:

- The **user interface** could be more esthetic: welcome screen could display a snake image instead of just the white background; score and hints could be displayed directly on the screen instead of using the “printf” function to display that information on the terminal.
- **Keyboard integration** would also be a very interesting feature, as the push buttons available in the DE2 board do not work always as expected and as they are not arranged in a way that makes the user feel comfortable while controlling the snake movement.
- **Game sound** could be added, and should be configurable (some users would find the game sound rather annoying).
- **Power consumption optimizations**. As we know, for example, that we could reduce the operation frequency of the memory ports which are connected to the microprocessor.

APPENDIX. List of acronyms and abbreviations

DPRAM: Dual-Port Random Access Memory
BRAM: Block Random Access Memory
FPGA: Field Programmable Gate Array
IP: Intellectual Property
MFO: Memory Frequency Operation
RAM: Random Access Memory
TDP: True Dual Port
uP: Microprocessor
WAR: Write After Read