A Simple Pong in Assembly Language

Learning Goal: Create a complete program in assembly language.

Requirements: Nios2Sim Simulator, FPGA4U Board.

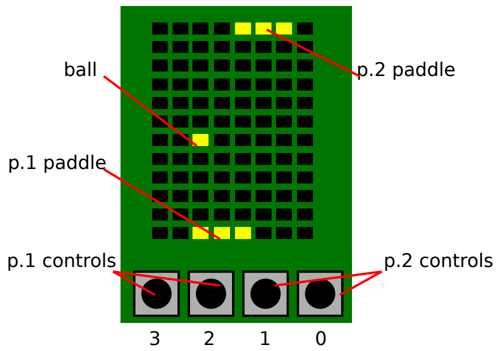
1 Introduction

During this lab, in assembly language, you will implement a simplified version of the well known game **Pong**. You will display the game on the LEDs of the **FPGA4U** and interact with the buttons.

Pong is a game for two players inspired by table tennis. It's made of a table, 2 paddles and a ball. Each player controls a paddle at one extremity of the table. With his paddle, one player can hit the ball back to his opponent. When a player misses the ball, his opponent wins one point, and the ball is reengaged by himself (i.e., by the one who lost the last point).

The 4 push buttons of the **FPGA4U** control the paddles. There are two buttons to control each paddle: one to move it up; the other to move it down.

The game is displayed on the LEDs of the **FPGA4U**, as illustrated in the following figure.



The current state of the game (i.e., the position and the velocity of the ball, the position of the paddles and the score) will be stored in the RAM. The following table shows an example of structure that you could use.

`0x1000`	Ball position on **x**
`0x1004`	Ball position on **y**
`0x1008`	Ball velocity on **x**
`0x100C`	Ball velocity on **y**
`0x1010`	Paddle 1 position
`0x1014`	Paddle 2 position
`0x1018`	Score of player 1
`0x101C`	Score of player 2

To improve the readability of your code you can associate symbols to values with the .equ statement. The .equ statement takes as arguments a symbol and a value. For example you may define the address of your structure elements and the address mapping of the peripherals as follow:

```
.equ    BALL,     0x1000 ; ball state
.equ    PADDLES,  0x1010 ; paddles pos
.equ    SCORES,   0x1018 ; scores

.equ    LEDS,     0x2000 ; LEDs address
.equ    BUTTONS,  0x2030 ; Buttons address
```

These symbols can be used to replace any numeric value of your code. You can use arithmetic expressions as well, like in this small example:

```
stw     zero, BALL (zero)    ; set ball x to 0
stw     zero, BALL+4 (zero)  ; set ball y to 0
stw     t0, LEDS+8 (zero)    ; set leds[2] to t0
ldw     t1, SCORE+4 (zero)   ; load the score of player 2 in t1
```

2 Drawing onto the LEDs

Your first exercise is to implement some procedure to display something on the LEDs. These procedures have to:

- Initialize the display with a `clear_leds` procedure.
- Turn on some pixels with a `set_pixel` procedure.

2.1 clear_leds

This procedure initializes to 0 the LEDs.

2.1.1 Arguments

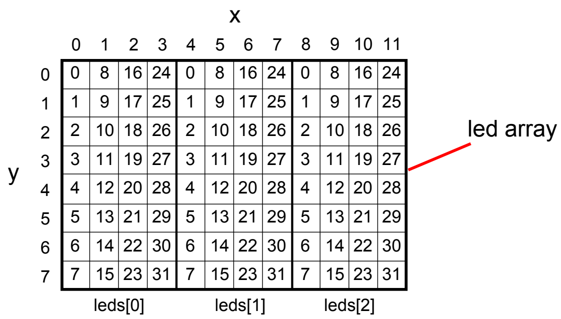
- None

2.1.2 Return Values

- None.

2.2 set_pixel

This procedure takes 2 coordinates as arguments and turns on the corresponding pixel on the display. The LED array has a size of 96 bits, or 3 words of 32 bits. The following figure translates the arguments **x** and **y** into a word and a bit position in this LED array.



This function should turn on one pixel within this LED array without modifying the value of the other bits.

2.2.1 Arguments

- `a0`: the **x** coordinate.
- `a1`: the **y** coordinate.

2.2.2 Return Values

- None.

2.3 Exercise

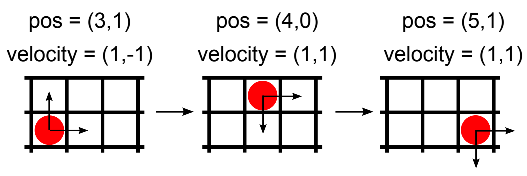
- Create a new `pong.asm` file.
- Implement the `clear_leds` and `set_pixel` procedures.
- Implement a `main` procedure that calls these procedures. It should:
  - First, call the `clear_leds` procedure to initialize the display.
  - Call the `set_pixel` procedure several times with different parameters to turn on some pixels.
- Simulate your program. Try to turn on different pixels.
- If you want to run this program on your FPGA4U board, follow the instructions of section 7.

3 Computing the Next Position of the Ball and Displaying it

In this section, you must implement a `hit_test` procedure and a `move_ball` procedure. For the moment, ignore the paddles and simply make the ball bounce on the limits of the table.

The current state of the ball is represented by its position and its velocity vector. The position of the ball is determined by 2 coordinates, **x** and **y**, which point to 1 pixel of LED array. The velocity vector specifies in which direction the ball is moving. For this simple version of the **Pong**, the velocity vector will only take integer values between -1 and 1.

The `hit_test` procedure checks whether or not the ball hits a boundary of the table, in which case, it must modify the velocity vector to make the ball bounce on the border. This test can be done independently for each coordinate. For example, looking at the **y** axis, if the current position of the ball is 0, then the ball hits the upper bound of the table, and the **y** coordinate of the velocity vector must be inverted. The following figure illustrates this example.



The `move_ball` procedure computes the next position of the ball by adding the velocity vector to the current position vector of the ball.

3.1 hit_test

This procedure tests whether or not the ball hits the boundaries of the table, and modifies its velocity vector correspondingly.

3.1.1 Arguments

- None.

3.1.2 Return Values

- None.

3.2 move_ball

This procedure moves the ball depending on its velocity vector.

3.2.1 Arguments

- None.

3.2.2 Return Values

- None.

3.3 Exercise

- Implement the `hit_test` and `move_ball` procedures in your `pong.asm` file.
- Modify the `main` procedure. It should initialize the ball position and its velocity vector, and do these steps in an infinite loop:
  - Call procedure `clear_leds` to initialize the LEDs.
  - Call procedure `hit_test`.
  - Call procedure `move_ball`.
  - Call procedure `set_pixel` with the ball coordinates as arguments.
- Simulate your program to verify it.

4 Moving and Displaying the Paddles

In this section you will write a `move_paddles` and a `draw_paddles` procedure.

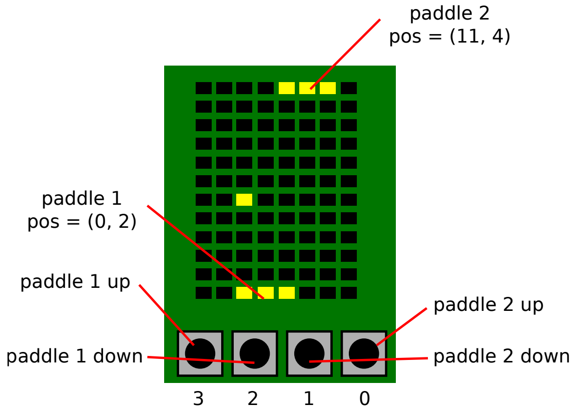
The `move_paddles` procedure reads the state of push buttons and moves the paddles correspondingly. The paddles can only be moved along the **y** axis. Their current position **y** is stored in memory. Their **x** coordinates are constants: 0 for the first paddle and 11 for the second.

The 4 push buttons of the FPGA4U are read through the **Buttons** module. The module has two registers, you can choose one of these two to implement your procedure:

Register	Name	31 . . . 4	3 . . . 0
0	`status`	*Reserved*	State of the Buttons
1	`edgecapture`	*Reserved*	Falling edge detection

Reading `status` will return the current state of the push buttons: if the bit n is 1, the button n is *currently* released, otherwise it is *currently* pressed. The `edgecapture` register detects when a button is pressed and activates its corresponding bit. The bit will stay at 1 until it is explicitly cleared to 0 by the program. In the simulator , you can observe the behavior of this module by clicking on the buttons and writing to its registers.

The `move_paddles` procedure must ensure that the paddles do not leave the table boundaries. Since the paddles are 3 pixels long, their **y** coordinate must stay between 0 and 5.



The `draw_paddles` procedure draws the paddles in the display. Using procedure `set_pixel`, it turns on 3 pixels for each paddle depending on their position. The coordinates refer to the top of the paddle, as shown in the previous figure.

The `draw_paddles` procedure calls another procedure, it thus has to save some registers on the stack. For that, it will have to use the **Stack Pointer** register (`sp`). Don't forget to initialize this `sp` register at the beginning of your main procedure. The following example saves some registers on the stack and restores them before returning to the caller.

```
; save ra and s0 registers:
addi    sp, sp, -8
stw     ra, 0(sp)
stw     s0, 4(sp)
; execute some code
...
; restore ra and s0 registers and return:
ldw     ra, 0(sp)
ldw     s0, 4(sp)
addi    sp, sp, 8
ret
```

4.1 move_paddles

This procedure moves the paddles depending on the push buttons state.

4.1.1 Arguments

- None.

4.1.2 Return Values

- None.

4.2 draw_paddles

This procedure draws the paddles on the display.

4.2.1 Arguments

- None.

4.2.2 Return Values

- None.

4.3 Exercise

- Implement the `move_paddles` and `draw_paddles` procedures in your `pong.asm` file.
- Modify the `main` procedure:
  - Initialize the `sp` register.
    - To which value should you initialize it?
  - Initialize the position of the paddles.
  - Call the `move_paddles` and `draw_paddles` procedures in a loop.
- Simulate your program to verify it.

5 Testing if the Ball Hits a Paddle

In this section, you will modify the `hit_test` procedure to take the paddles into account. The `hit_test` procedure will now return a value in `v0` telling whether a player missed the ball.

The procedure has to verify if the ball hits one of the paddles, and correct the velocity vector if it is the case. For this simple version of the **Pong**, you can make the assumption that if the ball is directly on the side of the paddle and the ball is moving toward the paddle, it hits the paddle; otherwise it misses it.

If a paddle misses the ball, the procedure has to return in `v0` the winner's ID (i.e., 1 for player 1, and 2 for player 2); otherwise it returns 0.

5.1 hit_test (final version)

This procedure tests whether or not the ball hits the boundaries of the table or a paddle, and modifies its velocity vector correspondingly. If there's a winner, `v0` return the winner's ID.

5.1.1 Arguments

- None.

5.1.2 Return Values

- `v0`: The winner's ID, if there's any; otherwise 0.

5.2 Exercise

- Modify the `hit_test` as described.
- Create a `display_game` procedure that will display the game on the LED making use of some of the previous procedures that you did implement.
- Modify the `main` procedure to read the value returned by the `hit_test` procedure. It must stop the game when a player misses the ball.
- Simulate your program to verify it.

6 Updating the Score and Displaying it on the LEDs

In this section, you will implement a `display_score` procedure to display the score on the LEDs. The following `font_data` section contains a font definition for hexadecimal characters. Each `.word` statement defines the font of the character in comments.

```
font_data:
    .word 0x7E427E00 ; 0
    .word 0x407E4400 ; 1
    .word 0x4E4A7A00 ; 2
    .word 0x7E4A4200 ; 3
    .word 0x7E080E00 ; 4
    .word 0x7A4A4E00 ; 5
    .word 0x7A4A7E00 ; 6
    .word 0x7E020600 ; 7
    .word 0x7E4A7E00 ; 8
    .word 0x7E4A4E00 ; 9
    .word 0x7E127E00 ; A
    .word 0x344A7E00 ; B
    .word 0x42423C00 ; C
    .word 0x3C427E00 ; D
    .word 0x424A7E00 ; E
    .word 0x020A7E00 ; F
```

To draw a character on the LEDs, you must load the corresponding word from this section, and store it into the LEDs module. For example, if the score is 3-8, you must:

- Load the word 3 and store it into leds[0].
- Load the word 8 and store it into leds[2].
- (Optional) Draw a separator character in leds[1].

6.1 display_score

This procedure draws the current score on the display.

6.1.1 Arguments

- None.

6.1.2 Return Values

- None.

6.2 Exercise

- Copy the `font_data` section to the end of your code.
- Implement the `display_score` procedure.
- Modify the `main` procedure to implement the final behavior of the game. You are free to add any other procedure to implement it. The main procedure should:
  - Initialize the game state.
  - Start a round.
  - Wait for a winner.
  - Update the score in the **RAM**.
  - Display the score on the LEDs.
  - Reinitialize the position of the ball and the paddles.
  - Start a new round and so on, until a player reaches some score (e.g., 15 pts).
- Simulate your program to verify it.
- Follow the instructions of section 7 to try it on your FPGA4U.

7 Running your Program on the FPGA4U DE0-nano

This section describes the necessary steps to run the program on the **FPGA4U DE0-nano** board.

- You need a working **Nios II** CPU.
- Normally, your CPU Quartus project should have the TCL script for pin assignments already run. If it is not the case, then run the TCL script by going through to Processing > TCL.
- In your `pong.asm` program, add a `wait` procedure to slow down its execution speed.
  - For example, this procedure could initialize and decrement a large counter and return when it reaches 0.
  - In your `main`, add a call to your `wait` procedure. You can call it after having displayed the game on the LEDs, for example.
  - Do not forget to comment that call when going back to simulation.
- In the **Nios2Sim** simulator, export your program in a `.hex` file (File > Export to Hex File...). Save it to the `ROM.hex` file of your CPU Quartus project.
- Compile your Quartus project.
- Program the **FPGA**.
- Every time you modify your program, do not forget to regenerate the `.hex` file, and compile the Quartus project again before programming the **FPGA**.

8 Submission

You don't need to submit anything for this lab. However, you have to show your complete implementation to the assistants during one of the labs.