Chapter 12: Hardware Pong

Excerpt from the "Digital Logic Exploration Kit 1.0" Lab Manual for the XGameStation Micro Edition
www.xgamestation.com

12.1 - Introduction

In this final chapter I wanted to make something that would be a little bit cooler than the "usual" introduction to digital devices. So we are going to make an LED-based Pong game. Unfortunately, we can't pile up a ton of chips to make this game entirely in hardware. Instead, we will use the SX processor to handle a lot of the game's algorithms. While this chapter is relatively short, the comments found in the pong game's source code fill out the remaining details.

12.2 - Parts List

- 4 LEDs
- 1 K Ohm resistor
- 100 Ohm resistor
- Momentary switch
- Digital Logic Exploration Kit Breadboard and Wires

12.3 - Experiment

Our "creative" implementation of Pong is based around a single row of LEDs, which will act as our playfield. So how will a one-dimensional playfield work? A couple seconds after startup, the LEDs will light up individually giving the appearance of a ball traveling down the field and back. When the ball returns to the starting LED, the user has to press the switch in time to "hit" the ball and avoid losing a life. To make the game more interesting and challenging, the player loses a life by pressing the switch too early as well. As a final detail, the 4 LEDs will show the user the current level, in binary, before the gameplay begins.

When a life is lost the number of lives left is displayed, also in binary. After 8 or so successful hits the level is ended with the next level displayed on the LEDs. The higher levels increase the speed of the ball traveling back and forth. This will continue until it is impossible to continue for a non-Jedi player.

12.3.1 - The Source Code

The source code for the PONG demo is similar to some of the other experiments that we have run. It has the standard Delay macro and the LongDelay subroutine. The only difference with the **LongDelay** subroutine is that the caller of the function specifies the 3 counter values. This is so we can control how long the **LongDelay** is.

The source code is divided up into three different game states: **GAME_STATE_LEVEL**, **GAME_STATE_GAME**, and **GAME_STATE_LIVES**. The current game state is contained in the variable called **GameState**. **GAME_STATE_LEVEL** displays the current level on the LEDs for a designated time. The source code for this section is below

```
    mov   Temp1, Level                  ;Copy level to temp value
    rl    Temp1                         ;Shift left
    rl    Temp1                         ;Shift left
    rl    Temp1                         ;Shift left
    rl    Temp1                         ;Shift left
```

```
    and   Temp1, #LED_MASK                  ;Mask only the important upper bits
    mov   RB, Temp1;                        ;Display the level on the LEDs

    ;Now delay to show this display for approx 4 secs
    mov    Temp1, #$04                      ;Use temp value for loop
LevelDispDelay
    mov    W,#0                             ;Delay for aproximately 1 second
    mov    Counter1,W                       ;clear counters
    mov    Counter2,W
    mov    Counter3,W
    call   LongDelay                        ;call delay func
    decsz  Temp1                            ;decrement counter skip if 0
    jmp LevelDispDelay                      Jump back up


    ;We are finished now so continue onto the game
    mov    GameState, #GAME_STATE_GAME      ;next game state

    ;initialize the level counter
    mov    LevelCounter, #$06               ;6 successful hits per level

    ;Increase Pong speed
    sub     LevelSpeed, #$0F                ;increase level speed
```

This code starts by taking the current level that the player is on and shifts the value left 4 times since the LEDs are connected to the higher pin numbers on port RB. Then it delays for approximately a second. After this the game state is changed to **`GAME_STATE_GAME`**. The **`LevelCounter`** is reset to 6 hits per level and we increase the level speed by decreasing the variable **`LevelSpeed`** since this variable is used as a delay. The **`GAME_STATE_GAME`** section is the largest section and is broken up and shown below.

```
    mov     w, RB                           ;clear RB except the input
    and     w, #$01                         ;mask off the input
    or      w, #$10                         ;start light one
    mov     RB, w                           ;output it

    ;setup the loop
    mov    Temp2, #3

ShiftLEDsLeft

    ;now start our delay between light shifts
    mov    w, LevelSpeed
    mov    Counter1,W                       ;clear counters
    mov    Counter2,W
    mov    Counter3,W
    call   LongDelay                        ;call delay func

    ;shift the LEDs to the left
    mov    w, RB                            ;get the current light display
    and    w, #LED_MASK                     ;mask off just the LED value
    mov    Temp1, w                         ;move to temporary variable
    rl     Temp1                            ;rotate left the value
    mov    RB, Temp1                        ;finally display the value back onto the port

    decsz Temp2                             ;decrement our loop counter
     jmp ShiftLEDsLeft
```

This portion of the game loop starts by turning on the first LED. After it turns the light on it delays by **`LevelSpeed`**. Once the delay is finished the light is shifted to the left. This loop is repeated 3 times until the last light is turned on.

```
    ;Here we shift the LEDs back to the player
```

```
    ;setup the loop
    mov    Temp2, #3
    mov    Input, #0


ShiftLEDsRight

    ;now since we need to check for inputs on the rotate right we will write our own delay
here
    mov    w, LevelSpeed
    mov    Counter1,W                      ;clear counters
    mov    Counter2,W
    mov    Counter3,W

ShiftRightDelay
    decsz  Counter1                        ;decrement the counters and skip when zero
     jmp    ShiftRightDelay                ;jump

    ;we check here for input
    mov    Temp1,RB                        ;get the input
    and    Temp1, #$01                     ;mask off input
    cjne   Temp1, #$01,NoButton            ;skip if there is no input
    mov    Input, #$01                     ;set the input flag
    ;continue counting down
NoButton

    decsz  Counter2
     jmp    ShiftRightDelay
    decsz  Counter3
     jmp    ShiftRightDelay


    ;shift the LEDs to the right
    mov    w, RB                           ;get the current light display
    and    w, #LED_MASK                    ;mask off just the LED value
    mov    Temp1, w                        ;move to temporary variable
    rr     Temp1                           ;rotate right the value
    mov    RB, Temp1                       ;finally display the value back onto the port

    ;check for premature and correct input
    cjne   Input, #$01, NoInput            ;check for input flag
    ;now we need to see if this was premature
    cje    Temp2, #$01, NotPremature       ;check to see if this was premature
    ;they hit the button before the last light therefore they die
    mov    GameState, #GAME_STATE_LIVES    ;going to game state lives
    jmp    DoneStateGame                   ;end it right now

NoInput
    ;check to see if this is the last led
    cjne   Temp2, #$01, NoInputOk          ;check to see if this was premature
    ;they missed the button so go to gamestate lives
    mov    GameState, #GAME_STATE_LIVES    ;going to game state lives
    jmp    DoneStateGame                   ;end it right now


NoInputOk

    decsz Temp2                            ;decrement our loop counter
     jmp ShiftLEDsRight

NotPremature

    ;they hit it correctly so continue on
    decsz  LevelCounter                    ;decrement successful hit counter
```

```
     jmp    DoneStateLevel                  ;continue playing this level
```

Now we will proceed by shifting the pong ball back towards the player. The big difference here is that we need to check for player input on the switch. Since the **LongDelay** subroutine does not have the ability to check for input we will make our own delay and during that delay we will check for input. If the player does press the button the variable Input will be set. The LEDs are shifted to the right and then the Input variable is checked to see if the button was pressed before the pong ball is at the first LED by this line of code.

```
    cje    Temp2, #$01, NotPremature        ;check to see if this was premature
```

If they hit it correctly the **LevelCounter** is decremented and checked to see if it equals zero. If not we play the same level at the same speed. If the user didn't press the button the **NoInput** code section is run to determine if the PONG ball is at the last light. If either the player has missed the ball or if they pressed the button prematurely the **GAME_STATE_LIVES** game state is run.

```
    ;decrement the number of lives and display them
    decsz  Lives
     jmp    MoreLives

    ;no more lives so just flash the lights forever
NoMoreLives

    ;turn all lights off
    mov    RB, #00

    mov    w, #0
    mov    Counter1,W                    ;clear counters
    mov    Counter2,W
    mov    Counter3,W
    call   LongDelay                     ;call delay func

    ;turn all lights on
    mov    RB, #$f0

    mov    w, #0
    mov    Counter1,W                    ;clear counters
    mov    Counter2,W
    mov    Counter3,W
    call   LongDelay                     ;call delay func

    jmp    NoMoreLives
```

The above section of code is from the **GAME_STATE_LIVES** and it decrements the current player lives. It then checks to see if it is at zero and if it is the lights are turned off and on repeatedly over and over.

```
MoreLives

    ;there are more lives so just display them binary
    mov    Temp1, Lives                  ;Move lives to temp variable
    rl     Temp1
    rl     Temp1
    rl     Temp1
    rl     Temp1
    mov    RB, Temp1

    mov    Temp2, #3                     ;delay for 3 secs
LoopLivesDisp
    mov    w, #0
    mov    Counter1,W                    ;clear counters
    mov    Counter2,W
    mov    Counter3,W
```

```
call   LongDelay                        ;call delay func

decsz  Temp2                            ;decrement loop counter
jmp    LoopLivesDisp

;Game state back to game
mov    GameState, #GAME_STATE_GAME
```

This code is called when the player dies but still has lives left. It will display the number of current lives and then return back to the **GAME_STATE_GAME** code section.

12.3.2 - Building and Running the Device

Now take a look at the schematic in Figure 12.1. Go ahead and build this circuit and hook it up to the XGS. Load the source code into the XGS Micro Studio and program the XGS ME. All of the code is commented and I suggest spending some time reading through it to thoroughly understand how all of this is working. Once you're ready turn the switch to run mode and see how far you can get in levels. One trick I've learned is to press and hold the switch when the ball gets to the second to last LED, but after a while (level 10 or so) it gets pretty hard. When the player has lost all of their lives, all the LEDs will blink on and off, indicating game over. Check out figures 12.2 and 12.3 to see a completed implementation.

Figure 12.1 - Hardware pong schematic.

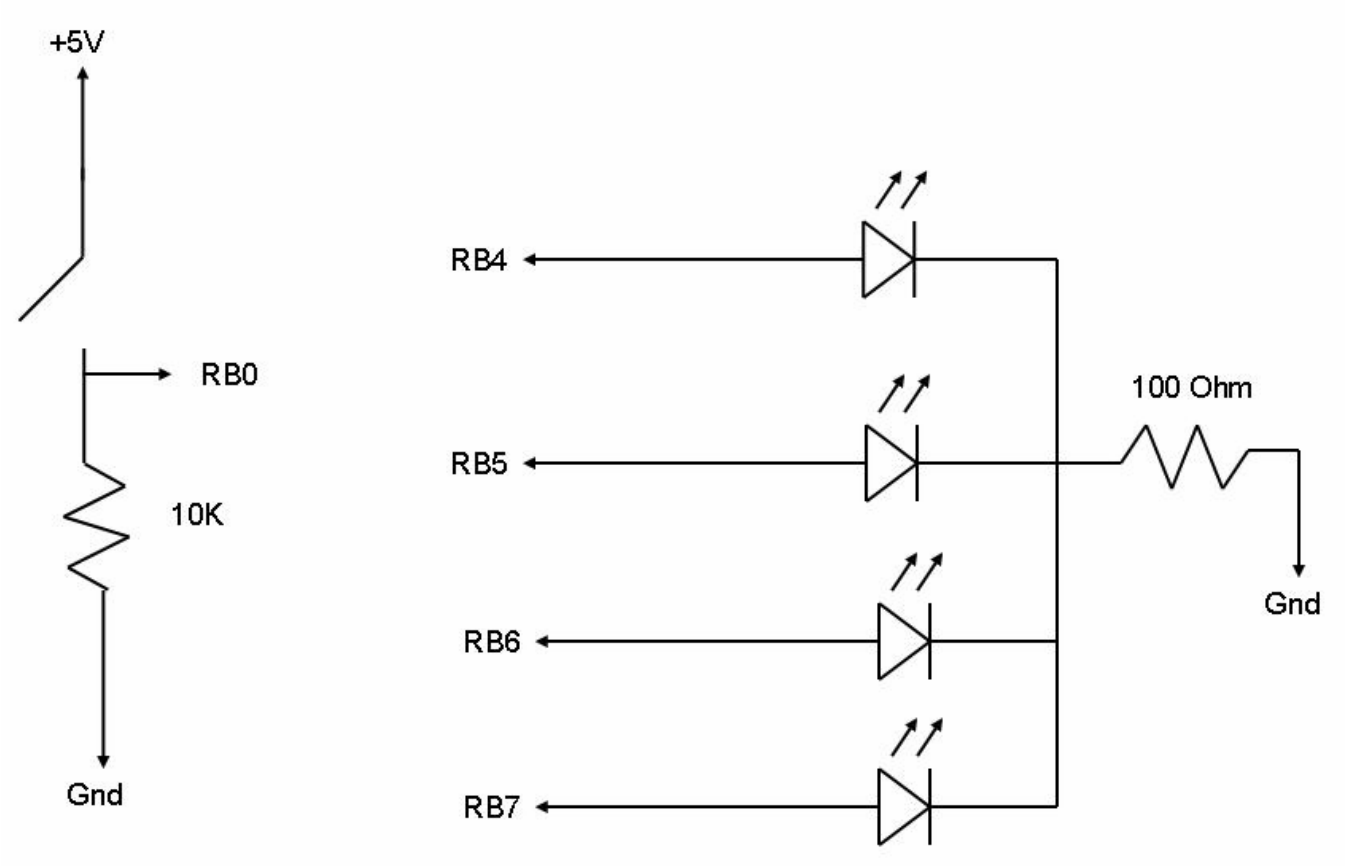


Figure 12.2 – Pong!

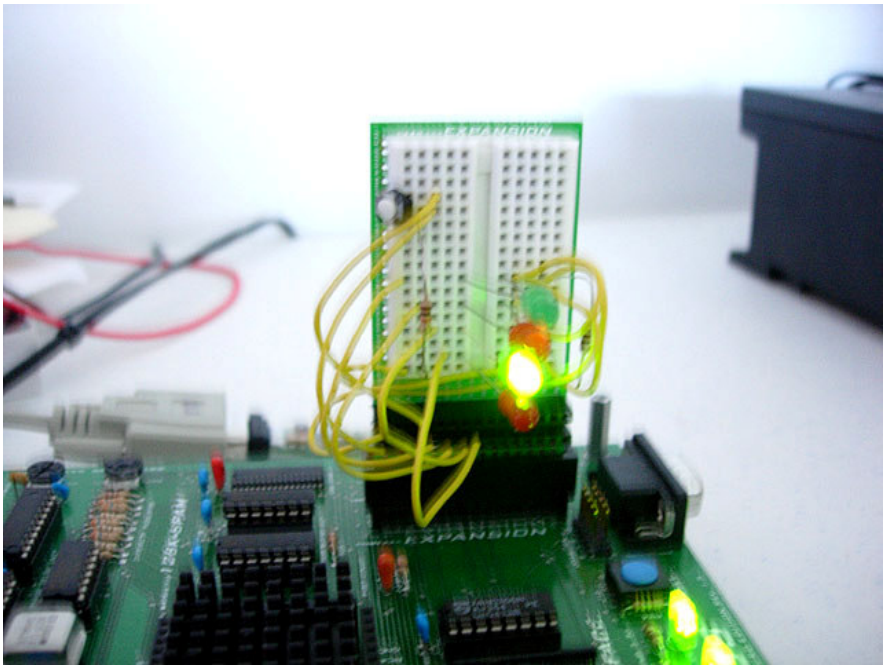
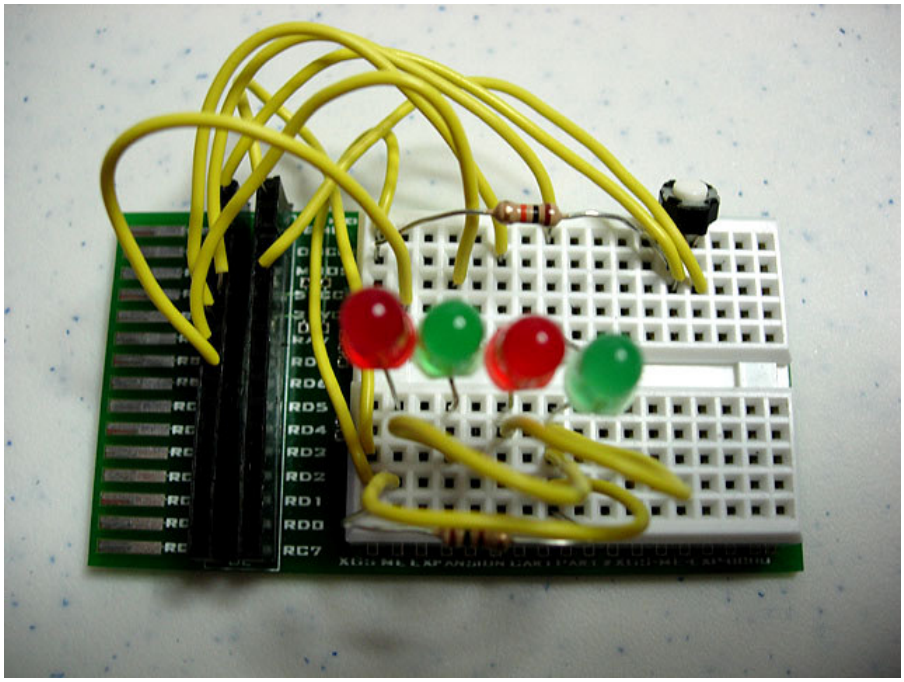


Figure 12.3 – More Pong.



12.4 - Summary

That about wraps up the XGameStation Micro Edition Digital Logic Exploration Kit! We finished up with a nice playable game which could be easily changed to add some more features, like sound through the 8 Ohm PC speaker. You can use these building blocks that you have learned throughout this manual to make larger experiments and add some of your own expansion boards to the XGS. If you come up with anything particularly cool, don't hesitate to share with the community.