ARM Project – Pitfall!

Computer Systems (COS10004) – Assignment 2

Attempting to make Pitfall! using only bare metal assembly.

Name: Zac McDonald
Student ID: 102580465
Lab Session: 12:30pm Thursday

The Project

Prior to our assembly lectures I did some research to get exposure to assembly programming, particularly for early game consoles. Among that were some post-mortems from Atari 2600 developers, particularly the programmer behind Adventure (1980) and Pitfall! (1982). I thought they were really cool, so I wanted to try to recreate the original Pitfall! using ARM assembly for my assignment.

The Pitfall! post-mortem can be seen here:
https://www.youtube.com/watch?v=tfAnxaWiSeE

I wanted to recreate the entire game, however I had a number of problems regarding drawing to the screen and usage of arrays and structs. This along with time constraints lead to a lot of time needed for testing and researching how to use them correctly in FASMARM. After all of this, I only managed to partially implement the player controller and inputs.

The project is made up of a number of files, each grouping a set of functions and data stores relating to a particular aspect of the program. As I was unable to complete the project to my desired outcome, an amount of testing code is present without real gameplay having been completed. I do intend to complete the project outside of the assignment, as I've found it an entertaining challenge and very rewarding.

Assembly source files and their intended purpose:

Source File	Intended Purpose
main.asm	Intended to store the main game loop and Raspberry Pi startup code. Currently sets up for screen inputs and uses a core loop, with a constant repeat/tick rate, that handles updating player information, inputs and drawing the character to the screen.
constants.asm	Contains all application wide variables, namely the Pi model memory base address, screen settings and tick rate. Would be extended to include a number of game-specific variables.
gpio_functions.asm	Handles the setup and handling of all GPIO related functions. Sets up four LEDS connected to GPIOs 18, 17, 27 and 22, and provides functions for accessing them using indices 0 to 3 respectively. LEDs can be set to high or low, or to match a bit, using their index and the Pi base address. Also provides functions for working with two buttons connected to GPIO 23 and 24, indexed 0 and 1 respectively. Every update tick one can update buttons and then check the state of the individual buttons (using only their index). Button states include held, pressed (this update) and released (this update).

graphics_functions.asm	Provides functions for setting up the Raspberry Pi screen output using its mailbox system. Also provides a function for drawing pixels to the screen and another for the clearing of the screen (setting all pixels to black).
graphics.asm	Defines the sprite structure I used for the game (more info later), sprite rendering functions and all the encoded sprites from the original game. Note that although only the player sprites are currently used in the application, all of the object (non-background) sprites have been encoded and can be easily drawn to the screen.
player.asm	Handles all of the player logic. This includes the necessary data stores as well as functions for updating the player state, position and animation based on player input and in-game events. As of writing, only a few states are fully implemented, but detection and handling structure for all states is present.
animation.asm	Used to increment animation frames for entities based on the elapsed time. As only the player is implemented, only the player animation frame counter is incremented.

GPIO Functions

The gpio_functions.asm source file, as discussed, contains all the GPIO related code. It contains trivial functions to perform GPIO setup, pausing and setting LEDs to given states, all of which should be familiar from the tutorial tasks. It also contains an amount of logic for button input handling, built to allow for complex input behaviours where we can check for state changes rather than simple on and off.

This is achieved through using the UpdateButtons function that reads the on/off GPIO state of the buttons and stores them in an array (indexed by button number). Before doing so, it takes the previously stored on/off state and stores it in a separate array. This allows us to then compare the current and previous states; allowing us to detect the update on which a button changes state from up to down (pressed), down to up (released), or to detect no changes (either remaining up, or remaining held). This then allows us to construct complex behaviours such as detecting if both buttons are simultaneously pressed or detecting when exactly one button is released while the other is held.

To further streamline this process, I've made four additional button functions, each taking a single parameter: the button number. GetButton simply returns 0 or 1 depending on if a button is currently down (1) or up (0). GetButtonDown similarly returns 0 or 1; if the button was pressed last frame it will return 1, otherwise it will return 0. GetButtonUp works in the same manner, but instead checks for if the button was released last frame. Both work by simply comparing the current and previous states of the given button. For example GetButtonDown checks if previously the button was up (0) and is now currently down (1). The fourth function, GetButtonState, calls each of the other button functions and returns a single value to describe a buttons change of state in the last update. It returns 0 if it remained up, 1 if it remained held, 2 if it was pressed and 3 if was released. In doing player inputs I mainly make use of the GetButtonState function.

Update Buttons (assembly source):

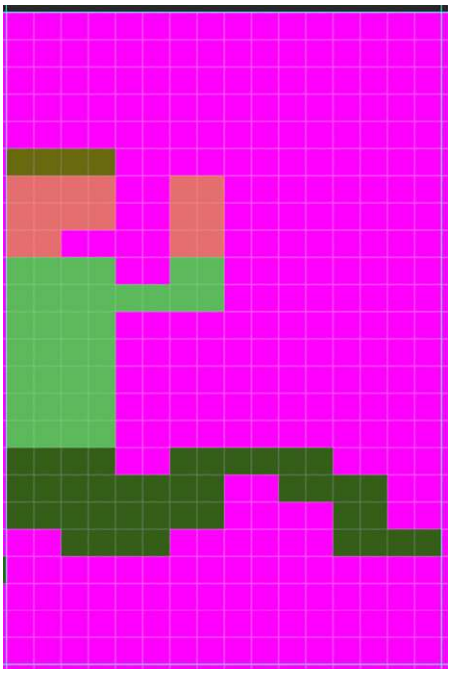
```
UpdateButtons:
; Updates the state arrays for buttons
; params r0 = BASE ADDRESS
; states become 0 if up and 1 if down
PUSH { lr, r4 }
    orr r0, GPIO_OFFSET
    ldr r0, [r0, #52]           ; Get the GPIO states

    adr r1, GPIO_BUTTON_CURR_STATE  ; Get the address of the current state array
    adr r2, GPIO_BUTTON_PREV_STATE  ; Get the address of the previous state array
    ldrb r3, [r1, #0]               ; Set previous state for button 0
    strb r3, [r2, #0]
    ldrb r3, [r1, #1]               ; Set previous state for button 1
    strb r3, [r2, #1]

    adr r2, GPIO_BUTTON_PINS    ; Get the address of the button pins array
    ; r0 = GPIO states
    ; r1 = Current State array
    ; r2 = GPIO pins

    ; Update current state for button 0
    ldrb r3, [r2, #0]           ; Get the GPIO pin number of button 0
    mov r4, #1
    lsl r4, r3                  ; Set the n-th bit to high, rest low (n = pin)
    tst r0, r4                  ; Check the n-th bit in the GPIO state
    mov r4, #0                  ; If the GPIO is low, we will set 0
    movne r4, #1                ; If the GPIO is high, we will set 1
    strb r4, [r1, #0]           ; Store state (0 or 1) in array

    ; Update current state for button 1
    ldrb r3, [r2, #1]           ; Get the GPIO pin number of button 1
    mov r4, #1
    lsl r4, r3                  ; Set the n-th bit to high, rest low (n = pin)
    tst r0, r4                  ; Check the n-th bit in the GPIO state
    mov r4, #0                  ; If the GPIO is low, we will set 0
    movne r4, #1                ; If the GPIO is high, we will set 1
    strb r4, [r1, #1]           ; Store state (0 or 1) in array
POP { pc, r4 }


; Stores the pin numbers of the buttons, indices 0-1
align 4
GPIO_BUTTON_PINS:
db 23, 24

; Stores the current state of the buttons, indices 0-1
; Use to detect if a button is pressed, held or released
align 4
GPIO_BUTTON_CURR_STATE:
db 0, 0

; Stores the previous state of the buttons, indices 0-1
; Use to detect if a button is pressed, held or released
align 4
GPIO_BUTTON_PREV_STATE:
db 0, 0
```

Sprite Handling

For the sprites in the game, I elected to use a simple colour palette to make encoding of sprites simpler. I chose a 16 colour palette based on the Pico-8 palette (see: https://www.lexaloffle.com/pico-8.php) translated into 16-bit colour. These colours can then be referenced using a 4-bit value (range 0-15). To simply even further I embraced limitations present on the Gameboy system where each sprite has 4 colours, with one being a chroma key. This allows us to represent each pixel using 2-bits (range 0-3). In each sprite structure I define a (functionally) 2-byte value that defines the colour palette, where the lower 3 nibbles represent the chosen colours. In the pixel information, these colours can be referenced using pixel values of 1 to 3, using 0 as transparent. Each sprite has dimensions 16px by 24px. An example sprite is shown encoding is shown below:

Example Sprite:

```
 align 4
 Sprite_Char_Swing_0:
 dw $0EB3            ; Colour choices 14, 11 and 3
 dw $00000000
 dw $00000000
 dw $00000000
 dw $00000000
 dw $00000000
 dw $FF000000
 dw $55050000
 dw $55050000
 dw $50050000
 dw $AA0A0000
 dw $AAAA0000
 dw $AA000000
 dw $AA000000
 dw $AA000000
 dw $AA000000
 dw $AA000000
 dw $FF0FFF00
 dw $FFFF0FF0
 dw $FFFF00F0
 dw $0FF000FF
 dw $00000000
 dw $00000000
 dw $00000000
 dw $00000000
```



In the DrawSprite function, we obviously need to make sense of all this binary information. To do so we need to follow some set steps (relate to below source):

1) Extract the current colour palette values into an array
2) Repeat (for each pixel in each row):
    a. Create a 2-bit mask and shift it to the appropriate position
    b. AND the mask with the row information to extract single pixel information
    c. Shift the result to occupy the lowest 2-bits so we can read its value
    d. Check that the colour is not zero, if it is: skip to the next pixel
    e. Calculate the pixels coordinates, using origin (x, y) values and iterators
    f. Call the DrawPixel function providing the calculated (x, y) and read colour

Creating the logic for this function was rather difficult as a lot of bitwise functions are needed and we need to be aware of how to translate data from its encoded form to a usable and meaningful form.

Draw Sprite (assembly source):

```
DrawSprite:
; Draws a sprite given its address and x/y coordinates (top left)
; params r0 = sprite address (i.e. adr r0, Sprite_Char_Idle_0)
; params r1 = x coordinate
; params r2 = y coordinate
; params r3 = face direction, 0 = normal, 1 = flipped
PUSH { lr, r4-r9 }
    mov r9, r3                          ; Save the direction for later

    ; Load the sprites colours into our CURRENT_SPRITE_COLOURS array
    ldrh r3, [r0, #0]
    adr r4, CURRENT_SPRITE_COLOURS      ; Get the current colours palette
    adr r5, PALETTE                     ; Get the palette array address

    mov r6, $0F00                       ; Get the first colour index only
    and r6, r6, r3
    lsr r6, #7                          ; Shift the index so it represents the number properly, move 1
less so it is double (2 bytes per colour)
    ldrh r6, [r5, r6]                   ; Load the colour from the palette
    strh r6, [r4, #0]                   ; Store the colour in the CURRENT_SPRITE_COLOURS array

    mov r6, $00F0                       ; Get the second colour index only
    and r6, r6, r3
```

```
    lsr r6, #3                          ; Shift the index so it represents the number properly, move 1
less so it is double (2 bytes per colour)
    ldrh r6, [r5, r6]                   ; Load the colour from the palette
    strh r6, [r4, #2]                   ; Store the colour in the CURRENT_SPRITE_COLOURS array

    mov r6, $000F                       ; Get the third colour index only
    and r6, r6, r3
    lsl r6, #1                          ; Double the index (2 bytes per colour)
    ldrh r6, [r5, r6]                   ; Load the colour from the palette
    strh r6, [r4, #4]                   ; Store the colour in the CURRENT_SPRITE_COLOURS array

    ; Draw each pixel with the correct colour
    ; r0 = sprite address
    ; r1 = x coordinate
    ; r2 = y coordinate
    ; r3 = current sprite row
    ; r4 = CURRENT_SPRITE_COLOURS array
    ; r5 = counter (0-24 for rows, 0-16 for each pixel)

    mov r5, #0              ; Initialise the row index
    DrawSprite_DrawRow:
    mov r6, r5              ; Get the current row index
    lsl r6, #2              ; Multiply by 4 (each row is 4 bytes long)
    add r6, #4              ; Skip the 2 bytes defining colours
    ldr r3, [r0, r6]        ; Read in the associated row
    PUSH { r5 }
        mov r5, #0                      ; Initialise the pixel index
        DrawSprite_DrawPixel:
            mov r6, $C0000000               ; Setup the mask to extract single pixel information (2
bits)
            mov r7, r5                      ; Get the current index
            lsl r7, #1                      ; Multiply the index by 2 (bits per pixel) so it equals
the number of bits we need to shift the mask
            lsr r6, r7                      ; Shift the mask

            and r6, r6, r3                  ; Read the pixel information, r6 now contains the offset
pixel info
            mov r8, #30                     ; We need to shift by 30 - (index * 2) to get the number
properly
            sub r8, r7                      ; 30 - (index * 2)
            lsr r6, r8                      ; Shift it, r6 now contains the pixel info (not offset)

            cmp r6, #0                      ; If the colour is zero - nothing to draw
            beq DrawSprite_DrawPixel_End
            sub r6, #1                      ; Otherwise subtract 1 to get the proper colour index
            lsl r6, #1                      ; x2 because 2 bytes per colour

            POP { r7 }                      ; Recall the y coordinate into r7
            PUSH { r7 }

            PUSH { r0-r3 }
            ldrh r0, [r4, r6]               ; Load the correct colour into r0

            cmp r9, #0                      ; Check if we need to flip the image
            addeq r1, r5                    ; Get the x coordinate (not flipped)
            addne r1, #16                   ; Get the x coordinate (flipped)
            subne r1, r5

            add r2, r7                      ; Get the y coordinate
            bl DrawPixel
            POP { r0-r3 }
        DrawSprite_DrawPixel_End:
        add r5, #1
        cmp r5, #16
        bne DrawSprite_DrawPixel
    POP { r5 }
    add r5, #1
    cmp r5, #24
    bne DrawSprite_DrawRow
POP { pc, r4-r9 }

; Stores the possible colours we can use indexed 0-15. They are based on the PICO-8 palette
align 4
PALETTE:
dh $0000, $290A, $792A, $042A, $AA86, $5AA9, $C618, $FF9D, $F809, $FD00, $FF64, $0726, $2D7F, $83B3,
$FBB5, $FE75

; Stores the current sprites colours
```

```
align 4
CURRENT_SPRITE_COLOURS:
dh $0000, $0000, $0000
```

The Player

The player function, as may assume, contains all the logic related to the player character controller. It contains two main functions: UpdatePlayer and PlayerGetFrame. A function PlayerDraw handles the drawing of the function, simply making a call to DrawSprite using the sprite address received from PlayerGetFrame. The player has five states: idle, running, jumping, swinging and climbing. As of submission, I have only implemented idle and running, *with an **incomplete** attempt at implementing jumping.*

UpdatePlayer contains all input checking and the transitions between states as well as each states behaviour. PlayerGetFrame uses the current state value and a frame counter (incremented over time) to determine the correct frame of each states animation to show. Again, note only limited states have been implemented, *however the checks for each state are present* in both functions. The idle state will be active if no buttons are currently down. Run will be active if exactly one button is held. If we are in the Run state, we can transition to the Jump state by pressing the other button. Interacting, which would initiate the climbing action, is triggered by pressing both buttons simultaneously.

An extract of the UpdatePlayer function, alongside the struct used for storing the player information can be seen below:

```
PlayerUpdate:
; Updates the player information based on events this frame.
PUSH { lr, r4-r6 }
    ; Read in the current button input states
    mov r0, #0
    bl GetButtonState
    mov r4, r0
    mov r0, #1
    bl GetButtonState
    mov r5, r0

    mov r0, Player_Info
    ldrb r6, [r0, #4]                                   ; Make record of the state last frame -
use it to check for change and reset frame counter

    ; If both buttons just pressed and we are grounded (interact)
    PlayerUpdate_Check_Interact:
    ldrb r1, [r0, #2]                                   ; Load grounded value
    cmp r1, #1                                          ; Check that player is grounded
    cmpeq r4, #2                                        ; Check right button held
    cmpeq r5, #2                                        ; Check left button held
    bne PlayerUpdate_Check_Move_Right
    ; Call interact if equal flag is set

    b PlayerUpdate_End_State_Checks

    ; If one button held, run in that direction and set the facing direction
    ; Check movement right
    PlayerUpdate_Check_Move_Right:
    cmp r4, #1
    cmpeq r5, #0
    bne PlayerUpdate_Check_Move_Left
    ; Do move right
    mov r1, #0                                          ; Set facing direction to 0
    strb r1, [r0, #3]
    ldrb r1, [r0, #0]                                   ; Load the x coordinate
    ldrb r2, [r0, #5]                                   ; Load the speed
    add r1, r2
    strb r1, [r0, #0]                                   ; x = x + speed
    ldrb r1, [r0, #2]
    cmp r1, #1                                          ; Check if grounded
    moveq r1, #1                                        ; Set state to running if not jumping
```

```
    strbeq r1, [r0, #4]
    b PlayerUpdate_Check_Jump

    ; Check movement left
    PlayerUpdate_Check_Move_Left:
    cmp r5, #1
    cmpeq r4, #0
    bne PlayerUpdate_Check_Jump
    ; Do move left
    mov r1, #1                                          ; Set facing direction to 0
    strb r1, [r0, #3]
    ldrb r1, [r0, #0]                                   ; Load the x coordinate
    ldrb r2, [r0, #5]                                   ; Load the speed
    sub r1, r2
    strb r1, [r0, #0]                                   ; x = x - speed
    ldrb r1, [r0, #2]
    cmp r1, #1                                          ; Check if grounded
    moveq r1, #1                                        ; Set state to running if not jumping
    strbeq r1, [r0, #4]
    b PlayerUpdate_Check_Jump
```

...

```
; Stores the player information
align 4
Player_Info:
db 24                   ; 0: Player x coordinate
db 60                   ; 1: Player y coordinate
db 1                    ; 2: Grounded: 1 = grounded or 0 = not grounded
db 0                    ; 3: Face direction 0 = right, 1 = left
db 0                    ; 4: State: 0 = idle, 1 = run, 2 = jump, 3 = swing, 4 = climb
db 3                    ; 5: Speed (pixels per update)
db 0                    ; 6: Animator frame counter
db 0                    ; 7: Current Sprite
```

Assumptions and Problems

Obviously, the initial scope for this project was a huge undertaking, but I was (and still am) confident I can complete it given more time. Development of it was rather straightforward after I had spent a time experimenting with and learning how to use and construct both arrays and structs in ARM assembly.

The two largest problems I experienced greatly impacted the project, having me spend days trying to solve them, mostly in vain.

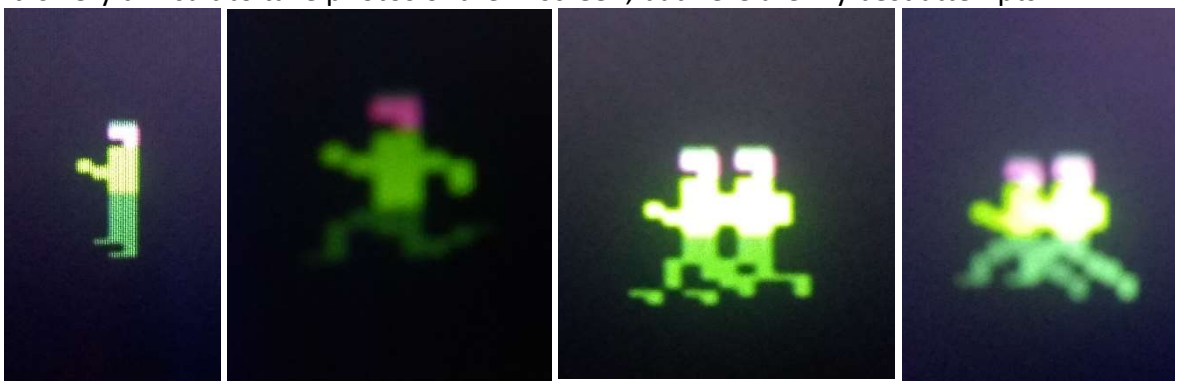
By far, the largest problem was inconsistencies when using the screen output. In both Lab 10 and this assignment, I found that drawing to the screen often resulted in artefacts, duplicate pixels and random crashing. This made early testing (before I was 100% sure of the methods) very difficult, and in the end detracts a lot from the program as the output often shows these inconsistencies very noticeably. Due to this, I found the need to restart the Pi multiple times before being able accurately test my program.

Once I got over this, I attempted to make use of arrays and structs. As many may have noticed, our provided resources on these are not supported by FASMARM, as it uses its own syntax. It took me some time to find accurate information and fully understand how FASM does these, and then to work out the changes to this that FASMARM makes. Confusion between data definitions and the align command leaded to a huge loss of time, where I was trying to use the load register (ldr) command to load from unaligned intervals (it loads 32-bits at a time, and thus must start aligned to this interval). I don't think these ideas (especially arrays) were expressed well in the tasks or lecture, and this damaged my assignment and many others.

Final Results

As a final result, I was able to setup a strong basis for continuing my Pitfall! project. In the submission I have successfully created a moving character with animated sprites that reacts to button inputs. I've also set up the controller for implementation of more complex behaviours (the other states), as well as for other entities (such as the log or scorpion).

It is very difficult to take photos of the Pi screen, but here are my best attempts:



Here is also a photo of all of the sprites implemented in the assignment (not displayed on the Pi). Try calling DrawSprite using their respective names (see graphics.asm) to see them on the Pi. Do note that a config.txt file matching that used for Lab 10 would be required.