Informe Laboratorio Arquitectura de Computadoras II

Nombres: Juan Pablo Rojas, Nicolas Colman
Profesores: Delfina Velez,Agustin Laprovitta

Introduccion: Realizamos el siguiente informe con el objetivo de poder mostrar la trayectoria que tuvimos al momento de realizar los ejercicios, tambien con fines de agilizar la presentacion mostrando como nos fuimos desenvolviendo y que problemas se nos presentaron en medio.

Ejercicio 1:

Ejercicio 1

Escribir un programa en assembler ARMv8 sobre el código de ejemplo dado, que genere en la pantalla una paleta de colores continua, tal como se muestra en la siguiente figura:

Las condiciones que debe cumplir el ejercicio son las siguientes:

- Que el patrón dibujado utilice toda la extensión de la pantalla.
- Que el patrón sea continuo (la transición de colores no debe mostrar límites visibles).
- Que al menos se utilice una vez la gama de colores completa (es decir que se llegue al mismo color del que se partió al menos una vez).

-Luego de repasar las consignas del ejercicio uno, comenzamos a realizar la implementacion del mismo, primero que nada leyendo el documento y comprendiendo como es que se manejaba el framebuffer, pudimos empezar a trabajar.

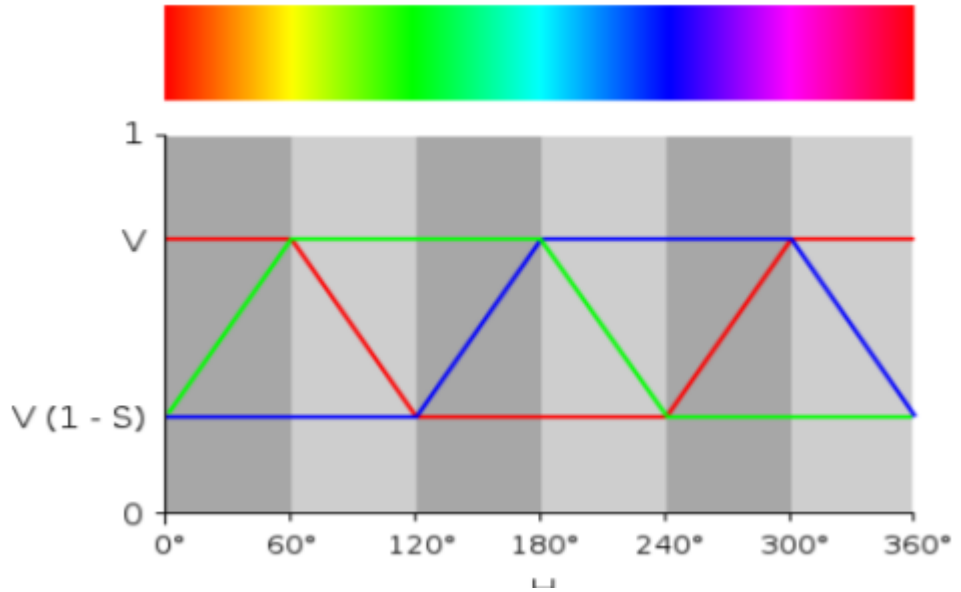
-lo primero que realizamos en el codigo es setear algunos colores para tener a mano, luego nos adherimos al recorrido del framebuffer que nos proveia el template.

```
    mov w16, 0xFFE0     // Amarillo
    mov w17, 0xF81F     // Rosa
    mov w18, 0xF800    // Rojo

    add x10, x0, 0      // X10 contiene la dirección base del framebuffer
loop2:
    mov x2,512          // Tamaño en Y
loop1:
        mov w3, 0xF800      // 0xF800 = ROJO
    mov x11, 1      // case
    mov x1,512          // Tamaño en X
    //mov x13, 63       // contador para color, arranco del maximo, el verde tiene mas bits, x eso el cont ma
```

luego, comenzamos a pensar como es que ibamos a distribuir los distintos tramos para generar la paleta de colores, se nos ocurrio realizar una especie de switch case, que iba

comparando un registro con el numero del tramo y luego ibamos a las funciones donde realizabamos la transicion de los colores, guiandonos del siguiente grafico.




```
case:
    cmp x11, 1
    beq RA
    cmp x11, 2
    beq AV
    cmp x11, 3
    beq VC
    cmp x11, 4
    beq CA
    cmp x11, 5
    beq AR
    cmp x11, 6
    beq RR
    b loop0
```

```
RA:
    cmp w3,w16    // pasar de rojo a amarillo,
    beq incremento
    add w3,w3,#0x0020
    b loop0
```

-Como definimos en un principio, el primer tramo de la paleta iba a ser rojo,por lo cual, realizamos la comparacion entre el registro que contiene la direccion base del framebuffer y si es = a amarillo, me voy a una funcion que lo que hace es realizar un incremento del registro x11 que vimos en el case, unicamente para realizar el avanze a la siguiente funcion. luego si no es = a amarillo, lo que hacemos es sumarle a w3 que contiene la direccion base del fb un bit de verde y me voy a loop 0 que lo que hace es hacer el store de ese pixel, lo setea.

-En un principio habiamos trabajado de otra forma que funcionaba, pero creo que de esta forma resulto mas simple, la anterior consistia en por ejemplo si yo queria pasar de amarillo

a verde, le iba restando a w3 un bit de rojo y decrementaba un contador que tenia 31 o 63 dependiendo el tamaño, luego ese registro se restaura y se aplica el cambio de funcion.

-Adjunto capturas de las otras transiciones entre las paletas de colores, que funcionan igual que la detallada al principio, pero ibamos revisando dependiendo la transicion si realizamos un add o un sub de los bits correspondientes al color.

```
AV:
    cmp w3,0x07E0 //verde puro
    beq incremento
    sub w3,w3,0x0800 //decremento el rojo
    b loop0

VC:
        cmp w3,0x07FF
    beq incremento
    add w3,w3,0x0001
    b loop0

CA:
        cmp w3,0x001F
    beq incremento
    sub w3,w3,0x0020
    b loop0
AR:
        cmp w3,w17 //comparo directamente con el registro, porque a ltener los 32 bits,
    beq incremento
        add w3,w3,0x0800
    b loop0
RR:
cmp w3,w18
beq incremento
sub w3,w3,0x0001    // pasar de rosa a rojo
b loop0
```

-Aclaracion: comparo directamente con el registro que contiene el color, para que no me de error de los limites.

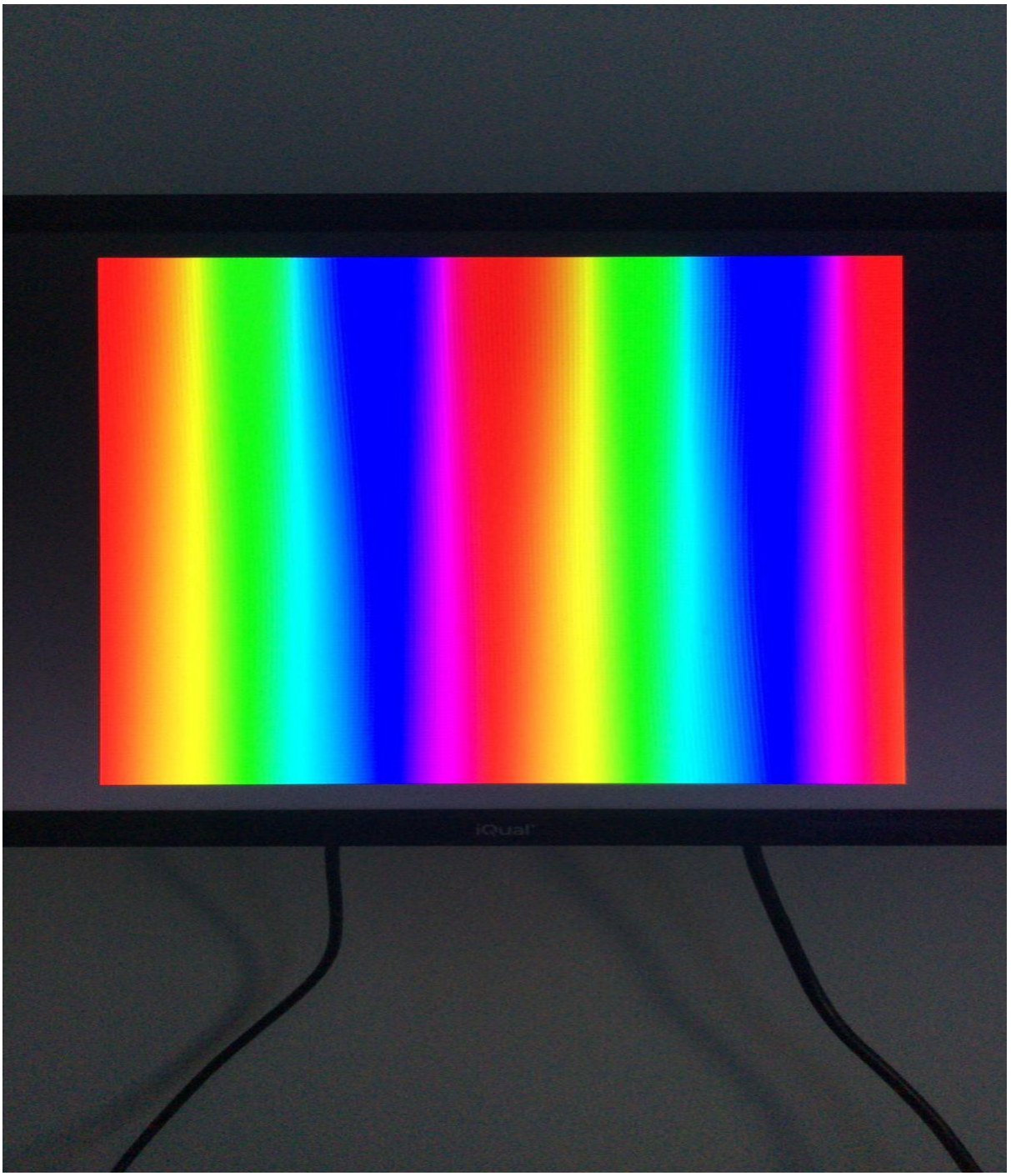
-Funcion incremento:

```
incremento:
add x11,x11,1
cmp x11,7
bne case
mov x11,1
b case
```

```
loop0:
    sturh w3,[x10]      // Setear el color del pixel N
    add x10,x10,2       // Siguiente pixel
    sub x1,x1,1         // Decrementar el contador X
    cbnz x1,case        // Si no terminó la fila, saltar
    sub x2,x2,1         // Decrementar el contador Y
    cbnz x2,loop1   // Si no es la última fila, saltar
```

-como habiamos nombrado antes, lo que hace esta parte del loop es setear el color del pixel con el que se este trabajando y luego hace el control de filas y columnas de un arreglo cuadrado.

-Conclusión del ejercicio 1:
 -En un principio nos costo comprender como es que funcionaba todo el manejo de los bits de los colores y como ir haciendo las transiciones del rgb, estuvimos con muchos errores en el sentido de no poder encontrar el color indicado, gracias a que en si no habiamos terminado de comprender lo necesario para implementarlo, luego de varias pruebas y de insistir, pudimos llegar al resultado esperado, seguramente haya otras formas de realizarlo.



Ejercicio 2:

Ejercicio 2

Escribir un programa en assembler ARMv8 sobre el código de ejemplo dado, que genere un laberinto que ocupe toda la pantalla y un jugador que se desplace hacia arriba, abajo, izquierda y derecha cuando se presione el pulsador correspondiente (GPIO14=arriba, GPIO17=abajo, GPIO18=izquierda, GPIO15=derecha).

Utilizar los leds para indicar eventos en el juego. Por ejemplo: al iniciar el juego encender el led rojo (GPIO3) y mantenerlo encendido hasta terminar el recorrido del laberinto. Cuando se alcance la meta, encender el led verde (GPIO2) durante unos segundos.

Las condiciones que debe cumplir el ejercicio son las siguientes:

-Respecto al ejercicio 2, el planteamiento antes de empezar fue como hacer que el programa dibuje en el pixel que yo quiero, leyendo el documento

segunda línea. La estructura resultante se muestra en el siguiente diagrama:

	Columna						
linea	0	1	2	...	509	510	511
0	0	1	2	..	509	510	511
1	512	513	514	...	1021	1022	1023
2	1024	1025	...				
...				...			
509					...		
510						...	
511							...

Debido a que la palabra que contiene el estado de cada pixel es de 16 bits, la dirección de memoria que contiene el estado del pixel N se calcula como:

Dirección = Dirección de inicio + (2 * N)

Si se quisiera calcular la dirección de un píxel en función de las coordenadas x e y, la fórmula quedaría como:

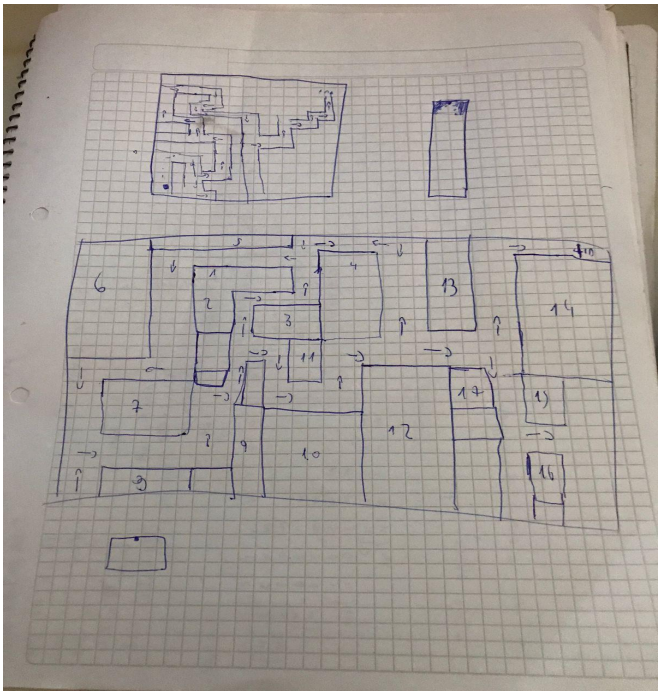
Dirección = Dirección de inicio + 2 * [x + (y * 512)]

-como vemos en la segunda formula podemos calcular la direccion del pixel con las coordenadas de x e y, ya con el concepto un poco mas fresco, nos largamos a la implementacion.

-primero que nada implementamos el recorrido del frame buffer de un array cuadrado, luego empezamos a pensar la forma de diseñar el laberinto, en este caso, lo que planteamos es

un boceto en la hoja de el laberinto que queriamos plasmar.

```
inicio2:
    mov x2,512          // Tamaño en Y
inicio1:
    mov x1,512          // Tamaño en X
inicio0:
    sturh w3,[x10]      // Setear el color del pixel N
    add x10,x10,2       // Siguiente pixel
    sub x1,x1,1     // Decrementar el contador X
    cbnz x1,inicio0     // Si no terminó la fila, volver a pintar
    sub x2,x2,1     // Decrementar el contador Y
    cbnz x2,inicio1     // Si no es la última fila, saltar
```



-Donde fui enumerando cada pieza para tener registro de cual modificar en caso de que no quede como esperamos, (en el laberinto final hay algunos cambios con respecto a este primer boceto), entonces, lo primero que hicimso es definir 4 registros, 2 para x y 2 para y, x4=inicio de x, x5= final de x, x6=incio de y, x7=final de y.

```
formas:


    mov x4,50           //  inicio x 1
    mov x6,50        // inicio y
    mov x7,100      // final y
    mov x5,250      // argumento final x
    bl rectangulo      // llama a la funcion que dibuja la figura
```

como se puede observar, tenemos un bl al rectangulo, que es la funcion que aplica la formula mostrada anteriormente.

```
rectangulo:

vuelta2:
    mov w8,0xFFE0
    add x10,x0,0
    mov x16,1024
    mov x17,2
    mul x9,x6,x16
    mul x14,x4,x17
    add x9,x9,x14
    add x10,x10,x9
    add x14,x6,0
```

donde defino en un registro, el color que quiero que tenga, luego a otro registro auxiliar le paso la direccion base del frame buffer asi no modifico el original, luego continuo con la metodologia del calculo de la formula plasmada anteriormente, hago el loop donde como siempre definimos el contador en x e y para el recorrido y hacemos el seteo del color del pixel n que se hace el store, luego paso al siguiente pixel y hago los controles de filas y columnas y sigo loopeando en base a los parametros de las coordenadas pasados

```
vuelta1:
    add x9,x4,0    /
vuelta0:
    sturh w8,[x10]
    add x10,x10,2
    add x9,x9,1
    cmp x9,x5
    b.ne vuelta0
    add x14,x14,1
    sub x15,x5,x4
    mul x15,x15,x17
    sub x15,x16,x15
    add x10,x10,x15
    cmp x14,x7
    b.ne vuelta1

    br x30
```

-Con respecto al manejo de otros colores dentro del laberinto, aplicamos la misma tecnica nombrada, pasando otro color, me dio la sensacion que no fue de la mas optima porque redundaba el loop de nuevo, tuvimos que aplicarlo en 2 funciones mas, una para formar el triangulo que basicamente hacia el mismo procedimeinto de pintura, en negro (porque era el color del fondo) y en base a la figura que se plasma al principio, cuadrado o rectangulo, iba pintando con la funcion triangulo, ubicando pixel por pixel para formarlo, lo que tambien creo que se puede optimizar bastante, pero fue la forma en la que se nos ocurrio en base a la estructura de pintado que elegimos.

-Bien, hasta aca pudimos explicar como es que realizamos la parte del diseño y pintado, lo cual una vez que pudimos aplicar correctamente el calculo de pintado, se nos va a facilitar un poco la implementacion del jugador con las colisiones

-jugador: con respecto al jugador, lo que hicimos fue pintar un pixel de rojo, donde le movemos a un registro el color y le seteamos la posicion inicial donde debe arrancar, luego para realizar la logica del pintado donde se lo setea, aplicamos la misma formula que nombramos en reiteradas ocasiones y luego se realiza el store de el registro del jugador en la direccion base del framebuffer, luego de hacer el store y que el jugador esta posicionado en pantalla, procedemos a implementar el movimiento, donde hacemos un bl a la funcion de inputRead que va a leer el boton del gpio que se este presionando y al hacer un br x30, (branch register a x30, direccion del Program Counter), volvemos a la instruccion que le sigue, luego pensamos en comparar con la direccion del gpio17, es decir si el bit de ese gpio esta en 1 esta apretado y sino liberado, realizamos el cmp a esa dirección, pero en un principio no nos funcionaba, luego de hacer consultas con la profe, nos explico que debíamos usar la mascara, que básicamente realizamos un and del registro w22, que es de los gpio y le paso el valor de dicho gpio, si en el caso de tener un valor muy grande, es decir en un bit mas arriba, le paso la mascara del valor que le quiero pasar según el gpio, si no esta ese valor se queda el original o 0 y si esta borra los bits y se queda con el valor que se le pasa, luego de aplicar la mascara realizo la comparacion que se nombro anteriormente, despues continuamos con un branch que verifica que si son iguales entonces vaya ala funcion que implementa el movimiento.

```
    jugador1:
    mov w3, 0xF800
    mov x8,15        //
    mov x9,500
jugador:
    add x10, x0, 0
    mov x16,1024
    mov x17,2
    mul x19,x9,x16
    mul x14,x8,x17
    add x19,x19,x14
    add x10,x10,x19
    add x14,x5,0


loop0:
//la mascara  del and

    sturh w3,[x10]
    bl inputRead
    and w23,w22,#0x200
    cmp w23,#0x20000
    b.eq abajo      //
    and w23,w22,#0x400
    cmp w23,#0x4000
    b.eq arriba     //d
    and w23,w22,#0x400
    cmp w23,#0x40000
    b.eq izquierda
    and w23,w22,#0x800
    cmp w23,#0x8000
    b.eq derecha
    b delay2       //Si
```

-Una vez que saltamos a la función de movimiento, lo que hacemos es hacer add o sub dependiendo donde me quiera mover y luego le paso la cantidad de pixeles que me quisiera desplazar

-Ahora uno de los temas en los cuales mas dificultar nos represento, fueron las colisiones donde en un principio lo pensamos de la forma que cargue el pixel siguiente donde se iba a realizar el movimento y mover en un registro el color de la pared y si son iguales que no mueva y si son distintos que si mueva, fue exitoso luego de muchos intentos para el movimiento de la derecha e izquierda .

```
    b delay2
derecha:
        ldurh w4,[x10,#10]
        mov w19,#0xFFE0
        cmp w4,w19
        beq delay2
        mov w29,#0x0000
        sturh w29,[x10]//borro rastro una vez que se mueve
    add x8,x8,5   // mover el jugador hacia la derecha
    mov w28,#0x07E0
    cmp w4,w28
    beq ledverde
    b delay2
izquierda:

        ldurh w4,[x10,#-10]
        mov w19,#0xFFE0
        cmp w4,w19
        beq delay2
        mov w29,#0x0000
        sturh w29,[x10]
    sub x8,x8,5  // mover el jugador hacia la izquierda
    b delay2
```

-Luego al querer implementar la misma lógica pero para arriba y abajo nos encontramos que el offset era muy grande y causaba overflow, por lo que nuevamente tuvimos que ir haciendo el calculo del pixel al que me debería mover segun coordenadas, donde le paso a un registro la dirección base del frame buffer, luego muevo el valor de 512 a un registro y 2 a otro,le paso cuantos pixeles me quiero mover y realizo el calculo de la formula

Dirección = Dirección de inicio + 2 * [x + (y * 512)]

-Tambien se implemento que una vez que cumpla la condicion, borre el rastro del pixel del jugador

- Luego realizo un load en un registro de la dirección base del framebuffer y a otro registro le paso el color de la pared para luego comparar y realizar el control de colisión, básicamente, lo que se propone con esto es fijarme en el pixel que me quiera mover según el movimiento y controlar que cumpla con la condición.
-Para arriba y abajo es lo mismo pero cambia en arriba se usa sub y abajo un add para el movimiento.

```
abajo:
// esta implementacion s

        add x25,x0,0
    mov x16,512          //
    mov x17,2             /
    add x24,x9,5
    mul x19,x24,x16
    add x19,x19,x8
    mul x19,x19,x17       /
    add x25,x25,x19       /
    ldurh w19,[x25]    //c
    mov w25,#0xFFE0    //
    cmp w25,w19        //
    beq delay2      //si c
    mov w29,#0x0000
        sturh w29,[x10]
    add x9,x9,5   // muev
    b delay2


arriba:
        add x25,x0,0
    mov x16,512          //
    mov x17,2             /
    sub x24,x9,5
    mul x19,x24,x16
    add x19,x19,x8
    mul x19,x19,x17       /
    add x25,x25,x19       /
    ldurh w19,[x25]    //c
    mov w25,#0xFFE0    //
    cmp w25,w19        //
    beq delay2      //si c
    mov w29,#0x0000
        sturh w29,[x10]
    sub x9,x9,5   // muev
    b delay2
```

-Luego para marcar la meta, realizo la misma implementacion de rectangulo pero con color verde

```
   rectangulo1:


vueltaa2:
    mov w8,0x07E0
    add x10,x0,0
    mov x16,1024
    mov x17,2
    mul x9,x6,x16
    mul x14,x4,x17
    add x9,x9,x14
    add x10,x10,x9
    add x14,x6,0
vueltaa1:
    add x9,x4,0
vueltaa0:
    sturh w8,[x10]
    add x10,x10,2
    add x9,x9,1
    cmp x9,x5
    b.ne vuelta0
    add x14,x14,1
    sub x15,x5,x4
    mul x15,x15,x17
    sub x15,x16,x15
    add x10,x10,x15
    cmp x14,x7
    b.ne vuelta1

     br x30
```

-lo mismo para el triangulo.

```
triangulo:
vueltaaa2:
    mov w8,0x0000
    add x10,x0,0
    mov x16,1024
    mov x17,2
    mul x9,x6,x16
    mul x14,x4,x17
    add x9,x9,x14
    add x10,x10,x9
    add x14,x6,0
vueltaaa1:
    add x9,x4,0
vueltaaa0:
    sturh w8,[x10]
    add x10,x10,2
    add x9,x9,1
    cmp x9,x5
    b.ne vuelta0
    add x14,x14,1
    sub x15,x5,x4
    mul x15,x15,x17
    sub x15,x16,x15
    add x10,x10,x15
    cmp x14,x7
    b.ne vuelta1

    br x30
```

-Para terminar, implementamos el control de los leds, donde mientras estoy en recorrido, no estan prendida la led verde, pero cuando se llega a la meta se prende.

-primero que nada se define que cuando se arranque el programa solo este la luz roja prendida

```
    mov w21, 0x4
str w21, [x20, 0x1C]
```

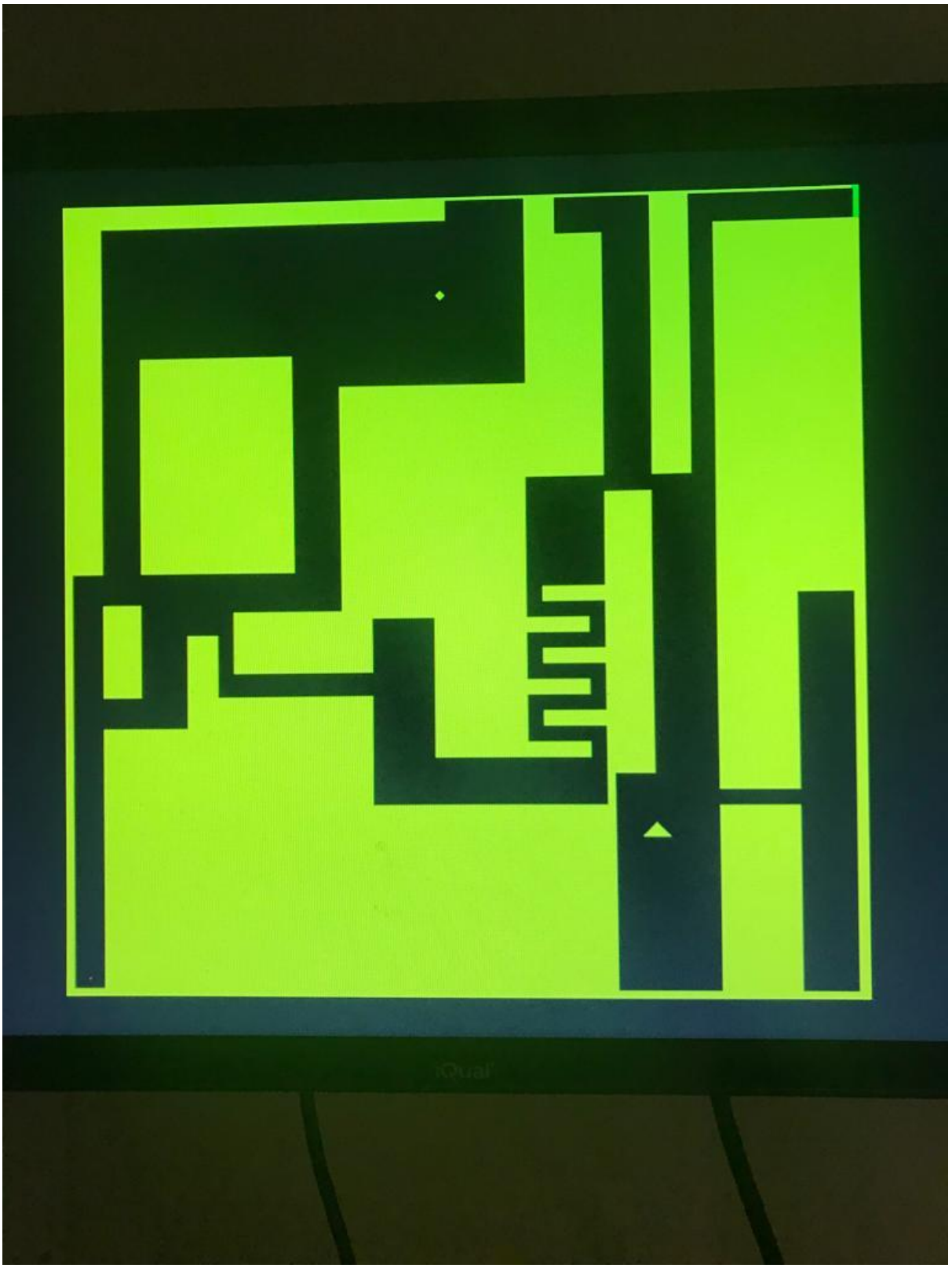
-Como se que en el diseño del mapa la meta esta desplazándose hacia la derecha, hago el control sobre esa función, donde al igual que el control de la colision, aplico la misma tecnica donde comparo el registro con la direccion de memoria del framebuffer con el offset siguiente y si es = a color verde salte a una funcion ledverde, que lo que hace es prender el led.

```
    mov w28,#0x07E0
    cmp w4,w28
    beq ledverde
    b delay2

ledverde:
    mov w21, 0x4
    str w21, [x20, 0x28]
    b delay2
```

-Conclusión del ejercicio 2: Nos costo en si darnos cuenta bien el funcionamiento de la formula para calcular los pixeles en base a x e y, creo que fue el punto fundamental entender esa parte, porque nos sirvio de puntapie para realizar los métodos, tuvimos muchos problemas con las colisiones en un principio, porque teniamos la idea pero no estabamos tan finos a la hora de implementarlo, por suerte probando y probando, conseguimos llegar al resultado esperado, con respecto al diseño del laberinto, por ahi nos confundimos en algunos numeros de las coordenadas y no se fijaban donde queriamos, pero era mas intuitivo aplicar la solucion y con respecto al movimiento y jugador, la mascara

fue algo que nos fue fundamental conocer, porque de no ser asi no podriamos haber llegado al a solucion.



-Este es el laberinto finalizado, se implementaron 3 figuras, rectangulo, circulo y triangulo, que se fueron formando en base a la logica de las coordenadas.