R-222 Arquitectura del Computador

Trabajo Práctico Final

ARMando el Juego de la Vida

Tomás Fernández de Luco F-3443/6
Gianni Georg Weinand W-0528/2
Ignacio Sebastián Moliné M-6466/1

18 de Abril, 2018

Índice

1 Introducción **2**
1.1 El Juego de la Vida 2
1.2 Estructuración de código 2

2 Primera implementación: Espacio de usuario **4**
2.1 Cross-Compiling 6
2.2 Makefile 7

3 Segunda implementación: Emulación de Bare Metal **8**
3.1 Bare Metal 8
3.2 QEMU 9
3.3 UART 10
3.4 Compilación y Ejecución 12

4 Tercera implementación: Bare Metal en Raspberry Pi **14**
4.1 Booteo 14
4.2 Framebuffer 15
4.3 Compilación y Ejecución 20

5 Referencias **22**

1 Introducción

El objetivo de este trabajo es adquirir conocimientos sobre diversos fundamentos y características de la arquitectura ARM. Para ello, se busca implementar el Juego de la Vida de Conway en distintos entornos propios de la arquitectura. Los programas que se buscaron realizar fueron los siguientes:

- Implementación en espacio de usuario de un dispositivo ARM.
- Ejecución bare metal en un simulador.
- Ejecución bare metal en un dispositivo ARM.

Durante el informe, se explicarán los conceptos que se debieron investigar y los códigos realizados para cada objetivo. Si bien la ejecución en espacio de usuario es una consigna opcional, se procede a explicar esta antes de la consigna principal del trabajo. El motivo de este orden es debido a que la consigna principal tiene código que se construye a partir de la implementación en espacio de usuario.

1.1 El Juego de la Vida

El juego de la vida es un autómata celular diseñado por el matemático británico John Horton Conway en 1970.[16] Consiste en un tablero de filas con celdas contiguas, las cuales pueden asumir dos estados: viva o muerta. En cada turno, todas las células modifican su estado simultáneamente de acuerdo a las siguientes reglas:

- Una célula muerta con exactamente 3 células vecinas vivas "nace" (es decir, al turno siguiente estará viva).
- Una célula viva con 2 o 3 células vecinas vivas sigue viva, en otro caso muere (por "soledad" o "superpoblación").

1.2 Estructuración de código

Para las diferentes implementaciones del programa, la funcionalidad se mantendrá constante, teniendo que modificar las formas de entrada y salida de cada uno de acuerdo al entorno en el que se ejecute. Se procede a explicar la lógica básica detrás del código C, profundizando luego en cada caso en particular.

```
#define H 10
#define W 20
char T[2][H+2][W+2];
```

Las constantes H y W serán el alto y ancho del tablero, respectivamente.

T son dos arreglos bidimensionales de caracteres, los cuales usaremos para representar dos tableros. Se define un par para facilitar la actualización del estado: como todas las celdas deben actualizarse en simultáneo, es necesario tener una versión anterior del tablero mientras se van actualizando los valores en otro. La razón por la que se le suma dos al ancho y alto es porque se busca tener un borde alrededor del tablero real, lleno con celdas muertas que no se actualizarán, y que cumplen la función de simplificar la revisión de celdas de los bordes del tablero sin tener que definir casos específicos en el código. Las celdas con valor 0 estarán muertas; las que valgan 1, vivas.

```
int main(){

    initTablero();

    int p = 0;


    for(;;){
        evolTablero(p);
        p = (p ^ 1) & 1;
        printTablero(p);
        getchar();
    }

    return 0;
}
```

En el main se llaman a todas las funciones que hacen a la ejecución del Juego. En un principio se llama a initTablero, que se encargará de inicializar los valores de uno de los tableros de acuerdo al método de ingreso que se utilice.

Luego, se realiza un bucle infinito para la evolución del juego. Primero se llama a evolTablero, la cual toma el tablero que describe el estado actual del juego, y se encarga de actualizar el otro. La sentencia p = (p ^ 1) & 1; alterna el valor de p entre 0 y 1, y luego se llama a printTablero para mostrar por pantalla el nuevo estado.

```
void initTablero(){
    int i, j;
    for(i = 0; i < H + 2; ++i)
        for(j = 0; j < W + 2; ++j)
            T[0][i][j] = 0;

    ingresarTablero();
}
```

initTablero inicializa todos los valores del primer tablero a 0, y luego llama a ingresarTablero() la cual tomará los valores de entrada dependiendo de la implementación.

```
const int HDIR[] = {-1, -1, -1, 0, 0, 1, 1, 1};
const int WDIR[] = {-1, 0, 1, -1, 1, -1, 0, 1};

void evolTablero(int p){
    int act = p;
    int sig = (p ^ 1) & 1;

    int i, j, k;
    for(i = 1; i < H+1; ++i){
        for(j = 1; j < W+1; ++j){
            int around = 0;
            for(k = 0; k < 8; ++k)
                around += T[act][ i + HDIR[k] ][ j +
                    WDIR[k] ];

            if(T[act][i][j] && around <= 3 && around >= 2)
                T[sig][i][j] = 1;

            else{
                if(around == 3)
                    T[sig][i][j] = 1;

                else
                    T[sig][i][j] = 0;
            }
        }
    }
}
```

evolTablero sigue las reglas del Juego para actualizar los valores del tablero siguiente en función del actual. Para ello, se definieron los arreglos HDIR y WDIR, que sirven para tener almacenadas todas las combinaciones de coordenadas relativas de las celdas de alrededor con respecto a la que se quiere actualizar. Se suman cuántas celdas vivas hay en la variable around, y en función de eso se determina el nuevo estado de la celda.

2 Primera implementación: Espacio de usuario

Para este caso en particular, la entrada y salida del código no poseen restricciones, por lo que se optó por los medios estándar por consola. Por lo tanto, las funciones de ingreso e impresión son las siguientes:

```
void ingresarCelda(){
    int x, y;
```

```
    while(1){
        printf("Ingrese la coordenada horizontal: ");
        scanf("%d", &x);
        if(x >= 1 && x <= W){
            printf("Ingrese la coordenada vertical: ");
            scanf("%d", &y);
            if(y >= 1 && y <= H){
                T[0][y][x] = (T[0][y][x] ^ 1) & 1;
                break;
            }

            else
                printf("\nERROR. Ingrese un valor valido
                    entre 1 y el alto.\n\n");

            }

        else
            printf("\nERROR. Ingrese un valor valido entre
                1 y el ancho.\n\n");

    }
}



void ingresarTablero(){
    char opcion;
    while(1){
        printf(" Desea encender o apagar otra celda? (S/N)
            : ");
        scanf("%c", &opcion);
        opcion = toupper(opcion);
        if (opcion == 'S'){
            ingresarCelda();
            getchar();
            printf("Estado actual del tablero:\n");
            printTablero(0);
        }
        else if (opcion == 'N')
            break;
        else
            printf("\nERROR. Ingrese una opcion valida.\n");

    }

    printf("Empezando simulacion. Presione una tecla para
        continuar.\n");
    getchar();
    getchar();
```

```
}
```

ingresarTablero le da al usuario la opción de modificar una cantidad indeterminada de celdas, alternando entre encenderlas y apagarlas. En caso de que quiera cambiar el estado de una, se llama a ingresarCelda, que se encarga de ello. El resto de las líneas hacen controles de la entrada e imprimen mensajes relacionados.

```
void clearTablero(){
    int i;
    for(i = 0; i < 10; ++i)
        putchar('\n');
}

void printTablero(int act){
    clearTablero();

    int i, j;
    for(i = 1; i < H+1; ++i){
        for(j = 1; j < W+1; ++j){
            if(T[act][i][j])
                putchar('*');
            else
                putchar(' ');
            }

        putchar('\n');
    }
}
```

printTablero imprime en pantalla una representación del tablero. Cada célula imprimirá un asterisco si está viva, y un espacio en caso contrario. Además, entre impresiones consecutivas del tablero, se llama a la función clear para que agregue líneas en blanco entre una y la otra.

2.1 Cross-Compiling

A lo largo del cursado se vieron dos arquitecturas de procesador diferentes: x86-64 y ARM. Las computadoras que se utilizan normalmente (en particular las utilizadas en la resolución de este trabajo) cuentan con procesadores x86-64. Consecuentemente, al compilar código fuente C utilizando el comando gcc se produce un ejecutable para la arquitectura nativa de la máquina. Es entonces necesario instalar un compilador que genere ejecutables para ARM. Un compilador capaz de generar código ejecutable para una plataforma diferente de la que ejecuta el compilador se llama cross-compiler.

En el transcurso de este trabajo se utilizó el toolchain arm-none-eabi, que se puede instalar desde Ubuntu con el comando:

```
Monika $> apt-get install gcc-arm-none-eabi
```

Además, para la compilación en espacio de usuario (que requiere libc) se utilizó arm-linux-gnueabi, instalado con los comandos:[1]

```
Monika $> sudo apt-get install libc6-armel-cross libc6-
    dev-armel-cross
Monika $>
    sudo apt-get install binutils-arm-linux-gnueabi
Monika $> sudo apt-get install libncurses5-dev
Monika $> sudo apt-get install gcc-arm-linux-gnueabi
```

2.2 Makefile

En cada carpeta del trabajo se encuentra un archivo makefile, que contiene los comandos de compilación para cada proyecto.[2] Basta con ejecutar el comando

```
Monika $> make
```

desde un shell para compilar todos los archivos del proyecto. Además al ejecutar

```
Monika $> make clean
```

se eliminan todos los archivos generados en el comando anterior (este comportamiento está especificado en cada makefile).

En particular el makefile del programa anterior es:

```
all: gameoflife

clean :
    rm -f *.o
    rm -f *.bin
    rm -f *.hex
    rm -f *.elf
    rm -f *.list
```

```
    rm -f *.img


gameoflife: gameoflife.c
    arm-linux-gnueabi-gcc gameoflife.c -o gameoflife
```

Una alternativa al cross-compiler en esta etapa del trabajo es compilar con

```
Monika $> gcc gameoflife.c -o gameoflife
```

desde el Raspberry Pi Zero con Raspbian. Al compilar el código en la misma plataforma donde se ejecutará basta con usar gcc de forma usual.

3 Segunda implementación: Emulación de Bare Metal

3.1 Bare Metal

En la siguiente parte del trabajo, el código ha de ser ejecutado en una computadora que no esté ejecutando su sistema operativo. A este tipo de programación se lo conoce como bare metal.[15] Para lograr ejecutar el código es necesario modificar algunos aspectos del encendido del procesador.

La arquitectura ARM comienza a ejecutar código en una dirección determinada, la cual depende del dispositivo que se esté utilizando, y puede estar almacenada en RAM o en ROM. En dicha dirección, se debe poner la tabla de vectores de interrupción. Así, se puede modificar el comportamiento que se realice cuando llegue una interrupción, como cuando el núcleo ARM se reinicie. Entonces, puede decirse que el handler de ese evento sea el código que se desea ejecutar, haciendo que se salte a la función main del Juego.[5]

Estos cambios se escriben en el archivo startup.s, que contiene lo siguiente:[12][11]

```
.global _Reset
_Reset:
    LDR sp, =stack_top
    BL c_entry
    B .
```

Primero inicializa el valor del tope de la pila, la cual será necesaria cuando se llame a funciones en el código C. El valor de stack_top se definirá en el momento de enlace.

Luego, le hace una llamada a la función c_entry. Este será el nuevo nombre que se le dará a la función main del código ya escrito. La razón por la que se cambia el nombre es porque los compiladores pueden insertar códigos binarios adicionales al encontrar una función main, los cuales pueden causar problemas en el entorno bare metal.[9]

El otro archivo que se debe escribir es el linker script, startup.ld , el que se encargará de enlazar el código assembler ya escrito con el que queremos ejecutar:

```
ENTRY(_Reset)
SECTIONS
{
 . = 0x10000;
 .startup . : { startup.o(.text) }
 .text : { *(.text) }
 .data : { *(.data) }
 .bss : { *(.bss COMMON) }
 . = ALIGN(8);
 . = . + 0x1000; /* 4kB of stack memory */
 stack_top = .;
}
```

El comando ENTRY define cuál es el punto de entrada del programa, es decir, la primera línea que se ejecutará. De esta forma, decimos que _Reset inicie el código. Luego, se define cómo serán las secciones del programa.

Se procede a definir el valor del símbolo especial . , que es el contador de locación (location counter). En este caso se le da la dirección 0x10000, que es la dirección sobre a la que QEMU irá a buscar la imagen del kernel para iniciar la emulación, y es donde necesitamos que se posicione el código que queremos ejecutar.[4] Se definen consecutivamente la sección de texto de startup.o , y luego las de texto, data y valores no inicializados. Por último, se alinea el valor del contador, se le aumenta en 4kB y se da el tope de la pila ahí, efectivamente dándole ese tamaño a la pila.

Una vez guardados estos archivos, falta modificar el código fuente del Juego, que ya puede ser compilado.

3.2 QEMU

Para ejecutar un programa compilado para ARM en una computadora con arquitectura x86-64, es necesario un emulador. En este trabajo se utilizó QEMU con la opción VersatilePB (que tiene un procesador ARM926EJ-S)[17][6]. Se puede instalar en Ubuntu con el comando:

```
Monika $> apt-get install qemu
```

3.3 UART

UART significa Universal Asynchronous Reciever-Transmitter[18]. Es un dispositivo de hardware que permite comunicación asíncrona serial. En QEMU, el primer puerto UART, llamado UART0 funciona como terminal si se ejecuta con la opción -nographic.[4] Los puertos están mapeados a direcciones de memoria. En particular, a UART0 le corresponde la dirección 0x101f100 en el VersatilePB. Esto se ve reflejado en la línea:

```
pl011_T * const UART0 = (pl011_T *)0x101f1000;
```

En C se implementó mediante la siguiente estructura (el manual explicita los campos y la memoria que le corresponde a cada uno):[3]

```
typedef volatile struct {
 uint32_t DR;
 uint32_t RSR_ECR;
 uint8_t reserved1[0x10];
 const uint32_t FR;
 uint8_t reserved2[0x4];
 uint32_t LPR;
 uint32_t IBRD;
 uint32_t FBRD;
 uint32_t LCR_H;
 uint32_t CR;
 uint32_t IFLS;
 uint32_t IMSC;
 const uint32_t RIS;
 const uint32_t MIS;
 uint32_t ICR;
 uint32_t DMACR;
} pl011_T;
```

En particular se utilizó UARTDR (o DR en la estructura), que es el registro designado para enviar y recibir bytes. Aparece en consecuencia la función uart_puts, que envía una cadena de caracteres s a través del puerto que recibe como argumento. La función recorre el string y envía cada caracter a través de UARTDR (con un casting previo para satisfacer el tamaño del registro DR):

```
static void uart_puts(pl011_T *uart, char *s){
    while(*s != '\0') {
        uart->DR = (unsigned int)(*s);
```

```
        s++;
    }
}
```

El programa recibe entrada de varios tipos: cadenas de caracteres, números, caracteres individuales. Se implementó una función uart_gets que lee una cadena de caracteres de la entrada serial que recibe como argumento. Para leer desde UART se usó el Flag Register (FR en la estructura). Para acceder a un flag específico basta con hacer el and lógico entre FR y el bit correspondiente al flag deseado. Son de interés RXFE (encendido cuando el recieve FIFO está vacío) y TXFF (encendido cuando el transmit FIFO está lleno). Esto se ve reflejado en las líneas:

```
enum {
 RXFE = 0x10,
 TXFF = 0x20,
};
```

Utilizando esto, la función gets lee caracteres hasta recibir un retorno de carro. Soporta el borrado de caracteres con Backspace y Delete, pero no lo muestra en tiempo real en la terminal.

```
static void uart_gets(pl011_T *uart, char *s){
    char* posicionActual = s;
    int lineend = 0;
    char c;
    while(!lineend){
        //Entra al if cuando el recieve FIFO no esta vacio
        if ((uart->FR & RXFE) == 0) {
            //Se queda esperando mientras el transmit FIFO
                este lleno
            while(uart->FR & TXFF)
                ;
            c  = uart->DR;
            switch(c){
                //Carriage return
                case(13):
                    lineend = 1;
                    break;
                //Backspace
                case(8):
                    if(posicionActual > s)
                        --posicionActual;
                    break;
                //Delete
                case(127):
                    if(posicionActual > s)
```

```
                        --posicionActual;
                    break;
                default:
                    *posicionActual = uart->DR = c;
                    ++posicionActual;
            }
        }
    }

    *posicionActual = 0;
}
```

Las funciones ingresarTablero y printTablero mantienen el mismo código, salvo que tuvieron que reemplazarse las distintas llamadas de ingreso e impresión por funciones propias que utilizen UART.

Las dos funciones restantes ingresan un string llamando a uart_gets, sólo que modifican el resultado de manera que se ajuste al tipo de dato que requiere ingresarse. uart_getint realiza una conversión de una cadena de caracteres que contiene un número al int correspondiente. Por otro lado, uart_getc toma el primer caracter de una cadena, y es utilizado cuando se busca ingresar un único valor char o para esperar que se ingrese una tecla para continuar la ejecución del programa.

```
static void uart_getint(pl011_T *uart, int* n){
    char s[9];
    uart_gets(UART0, s);
    *n = 0;
    int i;
    for(i = 0;;++i){
        if(s[i] == 0)
            break;
        (*n) *= 10;
        (*n) += s[i]-'0';
    }
}


static void uart_getc(pl011_T *uart, char *c){
    char s[20];
    uart_gets(uart, s);
    *c = s[0];
}
```

3.4 Compilación y Ejecución

Primero se compila el código fuente en C:

```
Monika $> arm-none-eabi-gcc -c -mcpu=arm926ej-s test.c
     -o test.o
```

La opción mcpu sirve para optimizar el ejecutable para un procesador específico (se puede hacer esto pues se eligió el ARM926EJ-S) y la -c genera código objeto (pues hay un linkeo posterior). Con las mismas opciones se compila el código en assembly:

```
Monika $> arm-none-eabi-as -mcpu=arm926ej-s startup.s
    -o startup.o
```

Luego se linkean los archivos compilados de la forma especificada en test.ld:

```
Monika $> arm-none-eabi-ld -T test.ld test.o startup.o
     -o test.elf
```

Y se convierte el programa .elf a un .bin, para poder ejecutarlo en QEMU:

```
Monika $> arm-none-eabi-objcopy -O binary test.elf
    test.bin
```

Todas estas líneas forman parte del makefile.

Se procede a ejecutar el archivo test.bin con QEMU:

```
Monika $> qemu-system-arm -M versatilepb -m 128M
     -nographic -kernel test.bin
```

La opción -M sirve para especificar el sistema a emular (en este caso VersatilePB), -m 128M para tener 128 megabytes de RAM, -nographic para no tener una ventana que simule el display y que UART0 funcione como terminal.

4 Tercera implementación: Bare Metal en Raspberry Pi

4.1 Booteo

Cuando el Raspberry Pi se enciende, el núcleo del procesador está apagado, y el GPU comienza a ejecutar un código de inicio almacenado en la ROM. Se lee la tarjeta SD buscando un archivo bootcode.bin, el cual se almacena en una caché y es ejecutado. Luego se habilita la SDRAM, se carga allí el archivo loader.bin, y se ejecuta. Este se encarga de leer el firmware de la GPU en otro archivo, start.elf, y recién en ese momento se busca el kernel.

Ya que la imagen del kernel determina el código que se ejecuta, se necesita tener allí el programa bare metal. Sin embargo, como se explicó previamente, también son necesarios ciertos otros archivos. Este problema se resolvió de una manera sencilla: se tomó una instalación de Raspbian en la tarjeta, la cual ya posee todos los archivos necesarios para inicializar el Raspberry, y se sobreescribió el archivo de kernel con el código del Juego de la Vida.

De manera similar al programa de la segunda implementación, se crearon los archivos memmap y vectors.s que se encargan de configurar el inicio de la ejecución del programa, inicializando la pila, definiendo la cantidad de RAM necesaria y llamando a la función principal del Juego, llamada notmain.

memmap

```
MEMORY
{
    ram : ORIGIN = 0x8000, LENGTH = 0x1000000
}

SECTIONS
{
    .text : { *(.text*) } > ram
    .bss : { *(.bss*) } > ram
}
```

vectors.s

```
.globl _start
_start:
    mov sp,#0x8000
    bl notmain
hang: b hang
```

4.2 Framebuffer

Al estar trabajando sobre un Raspberry Pi físico, ya no se tiene acceso a alguna de las interfaces que la emulación facilitaba. Es decir, ya no se posee una terminal UART fácilmente accesible, ni se reconocen los periféricos USB de manera automática. Por estas razones, se debió buscar una alternativa para mostrar la salida del programa por pantalla.

Se optó por utilizar el framebuffer, el cual es un buffer de memoria en RAM que proporciona un mapa de bits que controla una salida de video. Dicho mapa se convierte en una señal que puede ser transmitida por un puerto HDMI para mostrarse en un monitor.

La unidad multimedia a la que se le transmitirá la información para mostrar se conoce como VideoCore. La comunicación entre él y el procesador se realiza a través de un sistema de mailbox, dos canales entre los cuales se transmite información entre ambos. La utilización del mismo es bastante similar a UART, escribiendo en una dirección para transmitir y leyendo de otra para recibir.

Como punto inicial del código, definimos un conjunto de estructuras y constantes que resultarán útiles más tarde.

```
static volatile uint32_t * const MB0_READ   = (uint32_t *)
    0x2000B880;
static volatile uint32_t * const MB0_WRITE  = (uint32_t *)
    0x2000B8A0;
static volatile uint32_t * const MB0_STATUS = (uint32_t *)
    0x2000B898;

static const uint32_t MAIL_FULL   = 0x80000000;
static const uint32_t MAIL_EMPTY  = 0x40000000;

static const uint32_t BUFF_ADDR_MASK = 0x40000000;

static const uint32_t CHANNEL_MASK = 0xF;

typedef enum {
  POWER_MANAGEMENT,
  FRAMEBUFFER,
  UART,
  VCHIQ,
  LED,
  BUTTON,
  TOUCH,
  __reserved__,
  ARM_TO_VC,
  VC_TO_ARM
}MB0_CHANNEL;
```

```
typedef struct {
  uint32_t width;
  uint32_t height;
  uint32_t vWidth;
  uint32_t vHeight;
  uint32_t pitch;
  uint32_t depth;
  uint32_t xOffset;
  uint32_t yOffset;
  uint32_t fb;
  uint32_t fbSize;
} fb_info_t __attribute__((aligned(16)));
```

Cada mailbox cuenta con un conjunto de registros con offsets fijos a la dirección base, los cuales se utilizan en la comunicación. MB0_READ, MB0_WRITE y MB0_STATUS son las direcciones de los registros de lectura, escritura y estado, respectivamente, del mailbox 0.

MAIL_FULL, MAIL_EMPTY, BUFF_ADDR_MASK, CHANNEL_ MASK son máscaras que resultarán útiles para la interpretación de mensajes. La razón para usarlas es que la interacción con el mailbox sigue cierto formato en particular.

MB0_CHANNEL describe los canales con los que cuenta el mailbox0, los cuales son:

1. Power management
2. Framebuffer
3. Virtual UART
4. VCHIQ
5. LEDs
6. Buttons
7. Touch screen
8. -
9. Property tags (ARM -> VC)
10. Property tags (VC -> ARM)

fb_info_t es una estructura que contiene la información que se usa en la negociación con el framebuffer. Los valores son los siguientes:[14]

- uint32_t width : Ancho pedido del display físico.
- uint32_t height : Alto pedido del display físico.
- uint32_t vWidth : Ancho pedido del framebuffer virtual.
- uint32_t vHeight : Alto pedido del framebuffer virtual.
- uint32_t pitch : Número de bytes en cada fila de la pantalla.
- uint32_t depth : Profundidad de color.
- uint32_t xOffset : Desplazamiento en X pedido al framebuffer virtual.
- uint32_t yOffset : Desplazamiento en Y pedido al framebuffer virtual.
- uint32_t fb : Dirección del buffer alocado por el VideoCore.
- uint32_t fbSize : Tamaño del framebuffer.

Los valores deben estar alineados a 16 bytes para facilitar la construcción de valores que se transmitan.

Para poder mandar información al framebuffer, se debe negociar con el VideoCore para que se aloje uno, y a partir de ese momento se puede enviar los colores de píxeles determinados que se deseen modificar en el buffer. Esto se realiza al principio del programa, mediante las siguientes funciones:

```
uint32_t readMB0 (MB0_CHANNEL channel)
{
  uint32_t response = 0;
  while (1) {
    while (*MB0_STATUS & MAIL_EMPTY) {};
    response = *MB0_READ;
    if ((response & CHANNEL_MASK) == channel) {
      return (response & (~CHANNEL_MASK));
    }
  }
}

void writeMB0 (fb_info_t * data, MB0_CHANNEL channel)
{
  uint32_t intData = (uint32_t) data | BUFF_ADDR_MASK |
      channel;
  while (*MB0_STATUS & MAIL_FULL) {};
  *MB0_WRITE = intData;
}
```

Las funciones readMB0 y writeMB0 se encargan de leer y escribir del mailbox, siguiendo las instrucciones especificadas en la documentación.[13]

Para leer:

1. Leer el registro de estado hasta que la bandera de vacío no esté activada.
2. Leer información del registro de lectura.
3. Si los 4 bits más bajos no corresponden al número de canal esperado, repetir desde el paso 1. (Considerar que la información se transmite en formato little-endian)
4. Los 28 bits más altos son el mensaje devuelto.

Para escribir:

1. Leer el registro de estado hasta que la bandera de lleno no esté activada.
2. Escribir la información (desplazada a los 28 bits superiores) combinada con el canal (puesto en los 4 bits inferiores).

Se define una variable fbInfo, utilizada para negociar con el VideoCore, y una vez que un buffer esté disponible, se definirá la siguiente variable:

```
pix = (volatile uint32_t *) (fbInfo.fb & (~BUFF_ADDR_MASK));
```

pix almacena la dirección de memoria del buffer. Luego, para poder modificar el valor de un pixel específico, basta con escribir en su dirección específica el valor de 32 bits correspondiente al color que se desea que adopte.

Dado que ya no se posee una interfaz UART, o cualquier otro acceso por periféricos, se opta por modificar la entrada del código, utilizando la inclusión de un archivo gameoflife.h donde se encuentra un arreglo con las coordenadas de las celdas que inician vivas. Este cambio genera que haya que volver a compilar ante la necesidad de cambiar el estado inicial.

```
void ingresarTablero(){
    int i;
    for(i = 0; i < sizeof(coor)/sizeof(int); i += 2){
        T[0][coor[i+1]][coor[i]] = 1;
    }
}
```

La función ingresarTablero ahora considera los pares de valores establecidos en el arreglo coor para darle un valor inicial al tablero.

```
#define RGB32(red, green, blue) (red | (green << 8) | (blue
    << 16) | 0xFF000000)

void setCasilla(int x, int y, int R, int G, int B){
    int tamanioCasilla = fbInfo.vWidth/W;
    int offsetX = (x - 1) * tamanioCasilla;
    int offsetY = (y - 1) * tamanioCasilla;
    int i, j;
    for(i = 0; i < tamanioCasilla; i++)
        for(j = 0; j < tamanioCasilla; j++)
            pix[(offsetY + i) * fbInfo.vWidth + (offsetX +
                j)] = RGB32(R,G,B);
}
```

Se definió un tablero de tamaño proporcional a la pantalla, pero más pequeño. Por lo tanto, a cada casilla le corresponde un conjunto de pixeles que la representa. La función setCasilla se encarga de tomar las coordenadas de una celda, junto con las componentes RGB del color del que se desea pintarla, y escribe en el framebuffer dicha información en los píxeles del bloque. Esta función es la que se utilizará en aquellas que requieran cambiar el estado de un pixel en pantalla.

```
void printTablero(int act){
    int i, j;
    for(i = 1; i < H+1; ++i){
        for(j = 1; j < W+1; ++j){
            if(T[act][i][j])
                setCasilla(j, i, 255, 255, 255);
            else
                setCasilla(j, i, 0, 0, 0);
            }
    }
}
```

La función printTablero mantiene la misma estructura que en códigos anteriores, salvo las modificaciones pertinentes para utilizar framebuffer al momento de mostrar por pantalla. Sin embargo, resulta muy costoso refrescar toda la pantalla luego de cada evolución del tablero. En consecuencia, se modificó evolTablero para que únicamente se actualicen en pantalla los valores de las celdas que modificaron su estado entre un turno y el siguiente.

```
void evolTablero(int p){
    int act = p;
    int sig = (p ^ 1) & 1;

    int i, j, k;
```

```
    for(i = 1; i < H+1; ++i){
        for(j = 1; j < W+1; ++j){
            int around = 0;
            for(k = 0; k < 8; ++k)
                around += T[act][ i + HDIR[k] ][ j +
                    WDIR[k] ];

            if(T[act][i][j] && around <= 3 && around >= 2)
                T[sig][i][j] = 1;

            else{
                if(around == 3)
                    T[sig][i][j] = 1;

                else
                    T[sig][i][j] = 0;
            }

            if(T[act][i][j] != T[sig][i][j]){
                if(T[sig][i][j])
                    setCasilla(j, i, 255, 255, 255);
                else
                    setCasilla(j, i, 0, 0, 0);
            }
        }
    }
}
```

4.3 Compilación y Ejecución

Al usar las mismas opciones de compilación varias veces, o para hacer el makefile más legible, resulta conveniente agruparlas como se ve en la línea:

```
COPS = -Wall -O2 -nostdlib -nostartfiles -ffreestanding
    -mcpu=arm1176jzf-s -std=c99
```

Al usar O2 son necesarias las precauciones previamente explicadas (como no tener una función main). [9]

Basta entonces con ejecutar:

```
Monika $> $(ARMGNU)-as vectors.s -o vectors.o
Monika $> $(ARMGNU)-gcc $(COPS) -c gameoflife.c -o
    gameoflife.o -std=c99
Monika $> $(ARMGNU)-ld vectors.o gameoflife.o -T memmap
    -o gameoflife.elf
Monika $> $(ARMGNU)-objcopy gameoflife.elf -O binary
    gameoflife.bin
Monika $> $(ARMGNU)-objcopy gameoflife.elf -O binary
    kernel.img
```

Donde vectors.s cumple la misma función que startup.s en la versión anterior y memmap es nuestro linker script. El proceso de compilación es similar a la implementación con UART, lo único que cambia son los archivos involucrados.

5 Referencias

[1] Install the arm cross compiler toolchain on your linux ubuntu pc. https://www.acmesystems.it/arm9_toolchain.

[2] Makefile tutorial. http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/.

[3] ARM. Primecell uart (pl011) technical reference manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0183f/DDI0183.pdf. La sección relevante a la estructura de C del trabajo es la 3.3.

[4] Balau. Hello world for bare metal arm using qemu. https://balau82.wordpress.com/2010/02/28/hello-world-for-bare-metal-arm-using-qemu/.

[5] Balau. Simplest bare metal program for arm. https://balau82.wordpress.com/2010/02/14/simplest-bare-metal-program-for-arm/. Primer acercamiento al problema. De aquí se pueden descargar las herramientas que se utilizaran para desarrollar el trabajo.

[6] Balau. Using ubuntu arm cross-compiler for bare metal programming. https://balau82.wordpress.com/2010/12/05/using-ubuntu-arm-cross-compiler-for-bare-metal-programming/. Segundo acercamiento al problema. En este punto, ya se deberían tener todas las herramientas necesarias para lograr el objetivo principal.

[7] Raspberry Compote. Low-level graphics on raspberry pi. http://raspberrycompote.blogspot.com.ar/2012/12/low-level-graphics-on-raspberry-pi-part_9509.html. Entendimiento basico de Framebuffer.

[8] dwelch67. https://github.com/dwelch67/raspberrypi/tree/master/boards/pizero. Ejemplos de prueba de Bare Metal en PiZero.

[9] dwelch67. Raspberry pi bare metal readme. https://github.com/dwelch67/raspberrypi/blob/master/baremetal/README. Explicación del boot en PiZero y algunas cosas a tener en cuenta al momento de programar en Bare Metal.

[10] gusc. https://github.com/gusc/raspberrypi/blob/master/README.md. Explicacion de Framebuffer en PiZero.

[11] Red Hat. Simple linker script commands. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/4/html/Using_ld_the_GNU_Linker/simple-commands.html.

[12] Red Hat. Simple linker script example. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/4/html/Using_ld_the_GNU_Linker/simple-example.html.

[13] Raspberri Pi. Accessing mailboxes. https://github.com/raspberrypi/firmware/wiki/Accessing-mailboxes.

[14] Raspberri Pi. Mailbox framebuffer interface. https://github.com/raspberrypi/firmware/wiki/Mailbox-framebuffer-interface.

[15] Wikipedia. Bare machine. https://en.wikipedia.org/wiki/Bare_machine. Explicación bsica de programación en Bare metal.

[16] Wikipedia. Juego de la vida. https://es.wikipedia.org/wiki/Juego_de_la_vida. Explicación del Juego de la Vida.

[17] Wikipedia. Qemu. https://en.wikipedia.org/wiki/QEMU. Explicación de QEMU.

[18] Wikipedia. Uart. https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter. Explicación de UART.