PARCIAL 2

Condiciones

- Realizar el parcial individualmente.
- Entregar el parcial resuelto por mail hasta el **viernes 12 de Junio** a las 11hs (de la mañana). Se enviará confirmación de la recepción. Los parciales entregados después de esa fecha/hora se consideran desaprobados.
- En caso de tener algún problema para resolver el parcial (falta de hardware, conectividad, etc.) comunicarse por email a: **odc.famaf@gmail.com** antes del **miércoles 27 de Mayo**.

Formato de entrega

Deben entregar una carpeta comprimida (tarball o zip) con el nombre: `Parcial2_Apellido_Nombre`. La carpeta debe contener un directorio “Laberinto1” con los archivos main.s, Makefile, memmap, gdb.sh que resuelvan el ejercicio 1, un directorio “Laberinto2” con los archivos main.s, Makefile, memmap, gdb.sh que resuelvan el ejercicio 2 y un archivo pdf con la resolución de los ejercicios 3 y 4 (ver ejemplo). Los archivos deben seguir el estilo de código de los programas dados y contener comentarios que ayuden a comprender la manera en que solucionaron el problema.

```
joe@zoidberg:~/Organizacion del Computador$ find Parcial2_Miapellido_Minombre/
Parcial2_Miapellido_Minombre/
Parcial2_Miapellido_Minombre/Ejercicios3y4.pdf
Parcial2_Miapellido_Minombre/Laberinto2
Parcial2_Miapellido_Minombre/Laberinto2/memmap
Parcial2_Miapellido_Minombre/Laberinto2/gdb.sh
Parcial2_Miapellido_Minombre/Laberinto2/.gdb_history
Parcial2_Miapellido_Minombre/Laberinto2/main.s
Parcial2_Miapellido_Minombre/Laberinto2/Makefile
Parcial2_Miapellido_Minombre/Laberinto1
Parcial2_Miapellido_Minombre/Laberinto1/memmap
Parcial2_Miapellido_Minombre/Laberinto1/gdb.sh
Parcial2_Miapellido_Minombre/Laberinto1/.gdb_history
Parcial2_Miapellido_Minombre/Laberinto1/main.s
Parcial2_Miapellido_Minombre/Laberinto1/Makefile
```

La carpeta comprimida debe enviarse por email a: **odc.famaf@gmail.com** con el asunto: `Parcial2_Apellido_Nombre`.

Ejercicio 1 *(Laberinto ASCII)*:

Compilar el código base dado en el archivo comprimido “laberinto.tar.gz” y correr el programa base siguiendo los pasos indicados en el archivo “HowTo: Debug AArch64 GDB”. El programa tiene cargado un arreglo en hexadecimal, correspondiente en código ASCII a un laberinto en el que el “Personaje X” quiere llegar al “Tesoro #”.

El código ASCII (acrónimo inglés de *American Standard Code for Information Interchange* — Código Estándar Estadounidense para el Intercambio de Información), al igual que otros códigos de formato de representación de caracteres, es un método para una correspondencia entre cadenas de bits y una serie de símbolos (alfanuméricos y otros), permitiendo de esta forma la comunicación entre dispositivos digitales así como su procesado y almacenamiento. En este formato se hace una correspondencia de un símbolo gráfico cada 8 bits (1 byte). En nuestro caso, cargaremos en la memoria el siguiente código:

Código Hexadecimal	**Código ASCII**
2b 2d 2b 2d 2d 2d 2b 2d 2d 2d 2d 2d 2d 2d 2d 2b 7c 58 7c 20 20 20 20 20 7c 20 20 20 20 20 2d 7c 7c 20 7c 20 2d 2d 2b 20 7c 20 2b 2d 2d 20 20 7c 7c 20 7c 20 20 20 7c 20 7c 20 7c 20 20 20 20 7c 7c 20 2b 2d 2d 20 7c 20 7c 20 7c 20 20 2d 2d 2b 7c 20 20 20 20 20 7c 20 20 20 7c 20 20 20 23 7c 2b 2d 2d 2d 2d 2d 2b 2d 2d 2d 2b 2d 2d 2d 2d 2b	+-+---+--------+ |X| | -| | | --+ | +-- | | | | | | | | +-- | | | --+ | | | #| +-----+---+----+

Símbolos a tener en cuenta:

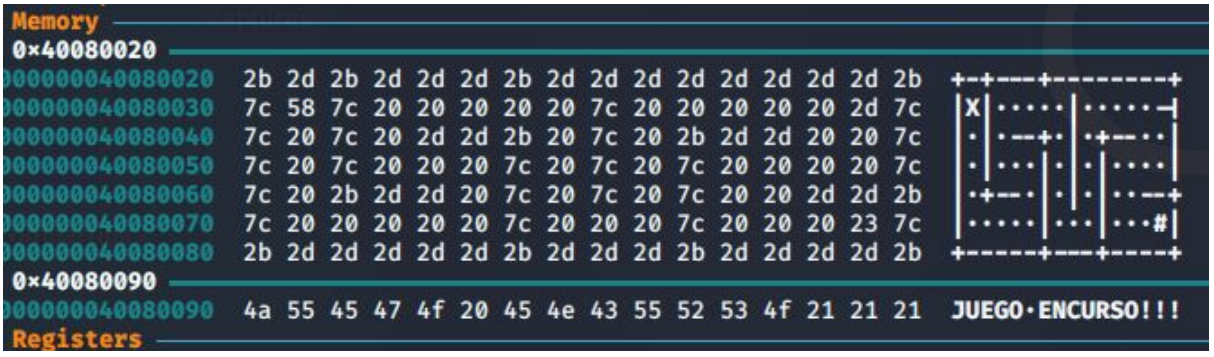
X	Personaje principal	0x58
#	Objetivo	0x23
|	Pared vertical	0x7c
+	Esquinas	0x2b
-	Pared horizontal	0x2d

En el programa base se podrán encontrar dos arreglos que se modificarán durante la ejecución:

- laberinto: contiene el laberinto, personaje y tesoro.
- estado: cadena de caracteres que representa el estado del juego.

Vistos en memoria se podrán visualizar usando el dashboard de GDB de la siguiente manera:

```
>>> dashboard memory watch 0x40080020 112
>>> dashboard memory watch 0x40080090 16
```



Se deberá modificar el programa base para que el Personaje “X” resuelva el laberinto cargado en la memoria. Además, si el Personaje X toca una pared (‘|’, ‘+’, ‘-’), pierde.

Para resolver este problema se deberá definir “funciones” arriba, abajo, derecha, izquierda. Y utilizar una solución fija (por ejemplo llamarlas en una secuencia: abajo, abajo, derecha, derecha, derecha, arriba, izquierda, arriba, ...).

Una vez implementadas dichas funciones, en caso de haber perdido, se deberá modificar el contenido del arreglo “estado” por la palabra: “PERDISTE :(”.
Por el contrario, en caso de llegar al tesoro se deberá modificar el estado del arreglo “estado” por la cadena: “GANASTE! B-) ”.

Vale aclarar que la modificación a implementar debe “mostrar” todo el recorrido que el personaje realiza para resolver el laberinto. Esto puede realizarse mediante la incorporación de “breakpoints” en el QEMU a fin de permitir la visualización del personaje con cada movimiento dentro de la estructura del laberinto. Para esto, se pide MARCAR mediante comentarios en el código generado, las instrucciones donde colocar el/los break points que permitan dicha visualización.

Observaciones:

- La instrucción “`LDR X0, =laberinto`” del `main.s` guarda la dirección del arreglo laberinto. Su programa debe recorrer este arreglo y modificar las posiciones del personaje, almacenando el símbolo X en la nueva posición y reemplazando a la posición anterior con un carácter espacio (`0x20`).

- Recuerden que la representación de las palabras en el Dashboard corresponde al formato **little-endian**. Es decir, si la palabra cargada en el arreglo es:

  `0x0123456789ABCDEF`

  En el Dashboard se muestra:

  `EF CD AB 89 67 45 23 01`

- Las direcciones de los arreglos laberinto y estado cambian dependiendo del tamaño del programa en ARM.

Ejercicio 2:

Reutilizar el programa anterior, en uno nuevo (Laberinto2), y modificarlo para poder solucionar mediante un programa el mismo laberinto sin conocer de entrada donde estará el Personaje X.

Es decir, el programa a entregar debe poder solucionar el laberinto con los siguientes Laberintos iniciales sin requerir cambios en el código:

```
+-+---+--------+
| |     |     -|
| | --+ | +--  |
| |   | | |    |
| +-- | | |  --+
|  X  |   |   #|
+-----+---+----+
```

```
+-+---+--------+
| |     |     -|
| | --+ | +--  |
| |   | | |   X|
| +-- | | |  --+
|     |   |   #|
+-----+---+----+
```

```
+-+---+--------+
| |    X|     -|
| | --+ | +--  |
| |   | | |    |
| +-- | | |  --+
|     |   |   #|
+-----+---+----+
```

Ejercicio 3:

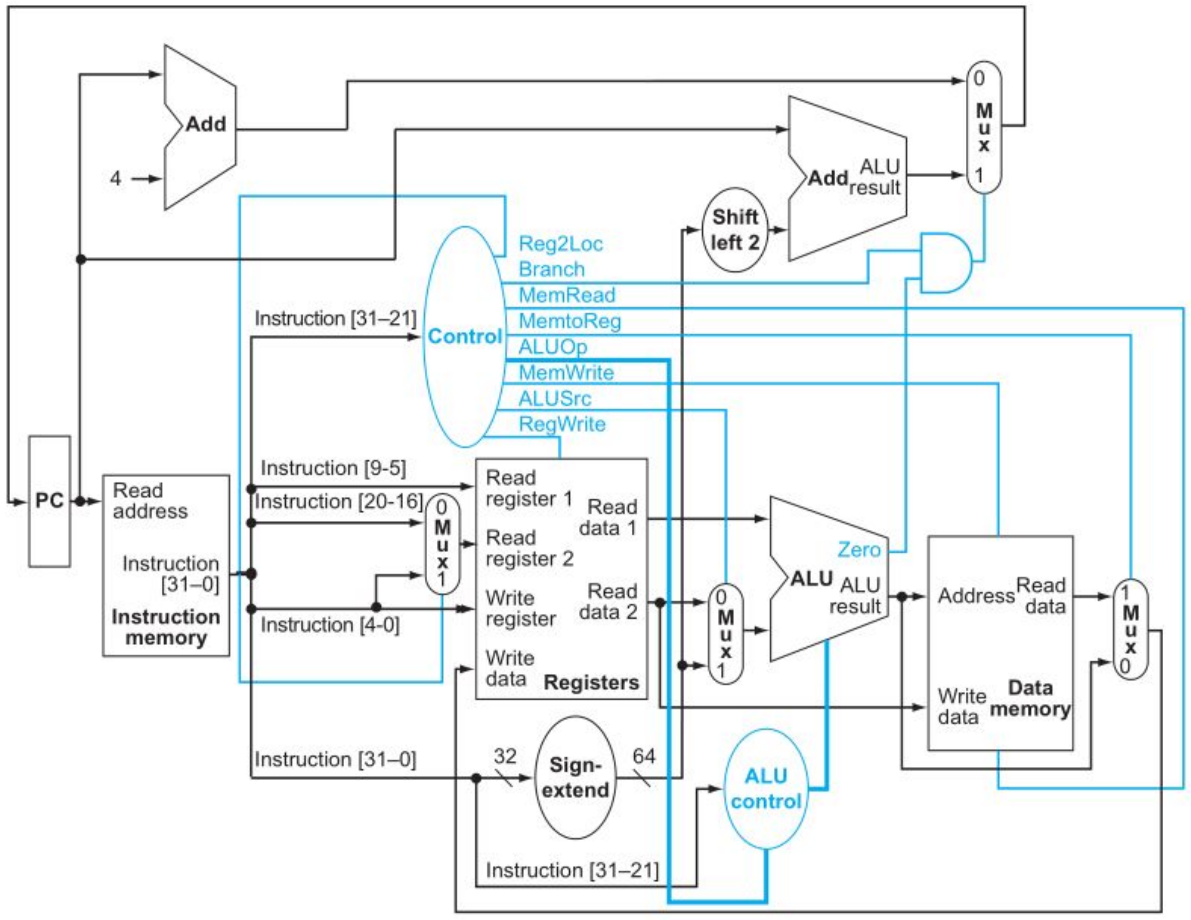
Tomar del código implementado para resolver el ejercicio anterior una instrucción de cada formato (R, I, D, B, CB e IM). En caso de no estar utilizando alguna de ellas en su código, modificarlo de manera que contenga al menos una instrucción de cada formato.
Luego ensamblar cada una de ellas a código de máquina LEGv8, mostrando las representaciones en binario y en hexadecimal de cada uno de los campos (opcode, inmediatos, registros, etc.).

Ejercicio 4:

El microprocesador de la figura (abajo) permite ejecutar las instrucciones LEGv8: LDUR, STUR, CBZ, ADD, SUB, AND y ORR. Realizar sobre el diagrama las modificaciones que considere necesarias para añadir alguna de las otras instrucciones LEGv8 vistas en la materia.

- Agregar los módulos y las señales de control que considere necesarias.
- Si se requieren, describir en pocas palabras las modificaciones internas de los módulos existentes.
- En la evaluación se tendrá en cuenta la cantidad, pertinencia y funcionamiento de las modificaciones propuestas.
- No serán válidos los ejemplos dados en clases.
- Completar la siguiente tabla, determinando qué valor toma cada una de las señales de control al ejecutar la nueva instrucción:

Reg2Loc	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp



Ejemplo: Agregar la instrucción Branch (no condicional)

Como en CBZ, para calcular la cantidad de posiciones de memoria a saltar se debe tomar el campo de inmediato, extender el signo y agregar 2 ceros al final, obteniendo como resultado un número de 64 bits. Para esto, el módulo *sign-extend* debe reconocer el opcode de la instrucción B, tomar los 25 bits menos significativos y replicar 36 veces el bit 25.
Además, se deberá incorporar en el módulo *Control* la decodificación del opcode de B para generar la señal de control “Uncondbranch”, que tomará valor ‘1’ en caso de ejecutar la instrucción B y ‘0’ en cualquier otro caso.

Señales de control al ejecutar esta nueva instrucción:

Reg2Loc	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp	Uncondbranch
X	X	X	0	X	0	0	XX	1

Diagrama del microprocesador modificado: