Parte 3. Gestor de Memoria
Fase b

En la primera fase hemos completado el entorno para poder simular el sistema convirtiendo el **Process Generator** en un cargador (**Loader**), ampliando la máquina de cómputo (**CPUs**) y creando una memoria física (**Physical**). En la segunda fase, construiremos el sistema de gestión de memoria virtual.

Los programas de entrada

El **Loader** leerá programas, cada uno en un fichero de texto (p.e. **`prog000.elf`**), simulando la memoria secundaria (**Secondary Memory**). Deberá cargar estos programas en memoria física y actualizar las estructuras de datos necesarias.

El formato de cada programa en su fichero de texto sólo dispondrá de dos segmentos, código (**`.text`**) y datos (**`.data`**), y será el siguiente:

```
.text <code start address>
.data <data start address>
    ...
<code line>
    ...
<data>
    ...
```

- `<code start address>`: Dirección de comienzo del correspondiente segmento del código del espacio de direccionamiento virtual. Normalmente valdrá **`000000`**.
- `<data start address>`: Dirección de comienzo del correspondiente segmento de datos del espacio de direccionamiento virtual. Comenzará a continuación del segmento de datos.
- `<code line>`: Una línea de código por cada instrucción, en hexadecimal, y cuyo formato se explica más adelante. Siempre terminarán con una instrucción **`exit`**.
- `<data>`: Una línea por cada dato. Será un valor entero, también representado en hexadecimal. El rango de valores es el clásico de un **`int`** (32 bits).

Cada instrucción ocupará 32 bits, el mismo tamaño de una palabra de memoria (4 bytes consecutivos). El juego de instrucciones tiene cinco formatos (en hexadecimal):

```
CRAAAAAA
CRRR----
CRR-----
C-AAAAAA
C-------
```

Donde:

- **`C`**: Es el código de operación (ver Tabla 1).
- **`R`**: Un registro general, del 0 al 15.
- **`AAAAAA`**: Dirección absoluta o directa (virtual). Son 24 bits.

Código	Formato	Ensamblador	Comportamiento
0[0]	**CRAAAAAA**	**ld rd,addr**	**rd = [addr]**
1[1]	**CRAAAAAA**	**st rf,addr**	**addr = [rf]**
2[2]	**CRRR----**	**add rd,rf1,rf2**	**rd = [rf1] + [rf2]**
3[3]	CRRR----	sub rd,rf1,rf2	rd = [rf1] - [rf2]
4[4]	CRRR----	mul rd,rf1,rf2	rd = [rf1] * [rf2]
5[5]	CRRR----	div rd,rf1,rf2	rd = [rf1] / [rf2]
6[6]	CRRR----	and rd,rf1,rf2	rd = [rf1] & [rf2]
7[7]	CRRR----	or rd,rf1,rf2	rd = [rf1] | [rf2]
8[8]	CRRR----	xor rd,rf1,rf2	rd = [rf1] ^ [rf2]
9[9]	CRR-----	mov rd,rf	rd = [rf]
10[A]	CRR-----	cmp rf1,rf2	cc = [rf1] - [rf2]
11[B]	C-AAAAAA	b addr	branch to addr
12[C]	C-AAAAAA	beq addr	branch to addr, if cc == 0
13[D]	C-AAAAAA	bgt addr	branch to addr, if cc > 0
14[E]	C-AAAAAA	blt addr	branch to addr, if cc < 0
15[F]	**C-------**	**exit**	**stop**
			cc: condition code

Tabla 1. *Juego de instrucciones.*

Los programas vendrá en ficheros de texto independientes. Los ficheros se denominarán de forma similar: **`progXXX.elf`**. Donde **`XXX`** serán números consecutivos empezando por el **`000`**. Un ejemplo de entrada con un programa sería:

prog000.elf	Interpretación
.text 000000 .data 000034 0700005C 08000060 29780000 19000064 06000064 07000034 28670000 18000038 0E00003C 0F000040 20EF0000 10000044 F0000000 FFFFFF76 FFFFFFC3 00000021 FFFFFFEF 00000043 FFFFFFD4 FFFFFF80 00000049 0000004F 000000C6 00000072 FFFFFF6F FFFFFF4E	.segmento de código **0x000000**: [0700005C] ld r7,0x00005C 0x000004: [08000060] ld r8,0x000060 0x000008: [29780000] add r9,r7,r8 0x00000C: [19000064] st r9,0x000064 0x000010: [06000064] ld r6,0x000064 0x000014: [07000034] ld r7,0x000034 0x000018: [28670000] add r8,r6,r7 0x00001C: [18000038] st r8,0x000038 0x000020: [0E00003C] ld r14,0x00003C 0x000024: [0F000040] ld r15,0x000040 0x000028: [20EF0000] add r0,r14,r15 0x00002C: [10000044] st r0,0x000044 0x000030: [F0000000] exit .segmento de datos **0x000034**: [FFFFFF76] -138 0x000038: [FFFFFFC3] -61 0x00003C: [00000021] 33 0x000040: [FFFFFFEF] -17 0x000044: [00000043] 67 0x000048: [FFFFFFD4] -44 0x00004C: [FFFFFF80] -128 0x000050: [00000049] 73 0x000054: [0000004F] 79 0x000058: [000000C6] 198 0x00005C: [00000072] 114 0x000060: [FFFFFF6F] -145 0x000064: [FFFFFF4E] -178

Os proporcionaremos un programa C que cree los ficheros de entrada (**`prometheus`**) y un ejecutable que los desensambla y vuelca (**`heracles`**).

Observaciones

Vamos a considerar que:

- No habrá errores en la entrada de los datos.
- Sólo se van a utilizar cuatro instrucciones: **`ld`**, **`st`**, **`add`** y **`exit`**.
- Si has implementado una política con prioridades, genéralas como lo hayas considerado (probablemente al azar).

El motor de la ejecución

Como ya sabemos, el **Clock** es el hilo encargado de "mover" la máquina (la estructura de CPUs, cores e hilos hardware). A partir de ahora debe hacer que cada hilo hardware ejecute el correspondiente programa del proceso asignado.

Como hemos comentado anteriormente los programas sólo tendrán cuatro tipo de instrucciones: **`ld`**, **`st`**, **`add`** y **`exit`**. Los programas ya están ensamblados correctamente. Esto quiere decir que cada unidad de cómputo ejecutará instrucciones hasta encontrar una instrucción **`exit`**. En este caso quedará ociosa. El comportamiento general de los programas será: cargar valores de memoria en registros, sumar contenidos de registros y almacenar datos de registro a memoria.

Convendría volcar los trozos afectados de memoria física (segmento de datos) antes y después de cada ejecución.