

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.

FASE b. SISTEMA DE GESTIÓN DE MEMORIA VIRTUAL

La Memoria y los espacios de direccionamiento

La Memoria es direccionable a **byte**, está organizada en **palabras** de **4 B** y tiene un tamaño de **página** es de **256 bytes**.

El Espacio de Direccionamiento Virtual es de **24 bits**, lo que direcciona 2^{24} posiciones. Es decir, los programas utilizan un direccionamiento directo o absoluto de **24 bits**.

Sin embargo, el Espacio de Direccionamiento Físico es variable. El tamaño de la Memoria Física (evidentemente mayor que el tamaño de página) debe ser parametrizable con un número de bits, **size_bits**, lo cual nos dice que debemos crear una memoria física de tamaño $2^{\text{size_bits}}$ bytes.

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 como la tabla de páginas, el PCB y aquellas estructuras que consideres necesarias para su gestión.

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>
...
```

Donde:

- **<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.e1f**. Donde **XXX** serán números consecutivos empezando por el **000**.

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). Hasta ahora se limitaba a consumir el tiempo de vida de los procesos asociados a cada hilo hardware (pudiendo haber hilos hardware ociosos). Pero 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.

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).

Un ejemplo de entrada con un programa sería:

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

Os proporcionaremos un programa que cree los ficheros de entrada (**prometeo**) y otro que los desensambla y vuelca (**heracles**).