SISTEMAS OPERATIVOS

CURSO 2020-2021

SIMULADOR DE UN SISTEMA INFORMÁTICO PRIMITIVO

MANUAL - V0

Introducción

Construir un sistema operativo (SO) sobre hardware real es una tarea muy compleja, por muy simple que sea el SO a construir. La razón de ello es que el hardware es complejo y diverso y, por tanto, toda pieza de software que aspire a controlarlo y gestionarlo adecuada y eficientemente, tiene que ser, a su vez, muy compleja. A pesar de ello, no renunciamos a poder estudiar los detalles e interioridades de un SO desde el punto de vista de su diseño y programación y, si no podemos hacerlo de una manera muy realista, podemos simplificar (quedándonos únicamente con los aspectos relevantes) y simular.

Los primeros sistemas informáticos (SI) eran muy primitivos. Cada uno de sus componentes, tanto hardware como software, era muy simple, en comparación con los que existen en la actualidad. Desde el punto de vista de la simulación, nuestro SI primitivo estaría compuesto de:

- Un procesador (primitivo).
- Una memoria principal.
- Los buses del sistema.
- Un SO (primitivo).

Explicándolo muy rápidamente, el SO se encargaría de cargar en la memoria principal un programa ejecutable, que sería posteriormente ejecutado por el procesador. Veamos a continuación el detalle de los componentes anteriores.

DISEÑO

El procesador

El procesador es muy elemental. Desde el punto de vista de estas prácticas no nos interesa hacer una disección tradicional del mismo (registros, Unidad de Control, Unidad Aritmético y Lógica) sino que realizaremos un diseño basado más en aspectos funcionales del mismo. Por tanto, el procesador estará compuesto de:

- Un conjunto de registros:
    - Registro acumulador, que se usará, en general, en operaciones aritméticas y de manipulación de la memoria principal.
    - Registro contador de programa (Program counter - PC).
    - Registro de instrucción (Instruction register - IR).
    - Registro de palabra de estado del procesador (Processor state word - PSW).
    - Registro de direcciones de memoria (Memory address register - MAR).
    - Registro intermedio de memoria (Memory buffer register - MBR).
    - Registro de control del bus (ConTRLol register – CTRL).
- La funcionalidad necesaria para la ejecución iterativa del ciclo de instrucción (etapas del ciclo de instrucción):
    - Búsqueda de instrucción.
    - Decodificación de instrucción.
    - Ejecución de la instrucción.
    - Almacenamiento del resultado.
    - Tratamiento hardware de interrupciones.

Evidentemente, todo procesador está construido para procesar un cierto conjunto de instrucciones definidas a priori (su juego de instrucciones). El juego de instrucciones del procesador es muy simple:

- `ADD op1 op2`: suma los valores enteros `op1` y `op2` y deja el resultado dentro del registro acumulador.
- `SHIFT op1`: realiza un desplazamiento aritmético sobre el registro acumulador las posiciones indicadas por el valor de `op1`. Si `op1` es negativo el desplazamiento será hacia la izquierda y si es positivo hacia la derecha. Si el desplazamiento es hacia la derecha, rellena usando el bit más significativo (esto implica que, si hay un valor negativo, seguirá siendo negativo). Deja el resultado en el registro acumulador.
- `NOP`: típica instrucción *NO-OPERATION* de los procesadores, que consume un ciclo de instrucción sin realizar acción alguna.
- `JUMP dirRelativa`: modifica el valor del registro PC sumándole el valor indicado en `dirRelativa`.
- `ZJUMP dirRelativa`: funciona igual que `JUMP` si el valor actual del registro acumulador es 0.
- `READ dir`: lee el contenido de la posición de memoria principal indicada por `dir`, depositando el valor leído en el registro acumulador.
- `WRITE dir`: almacena el valor contenido el registro acumulador en la posición de memoria principal indicada por `dir`.
- `INC op1`: incrementa el valor actual del acumulador con el valor indicado por `op1`.
- `HALT`: provoca la detención, definitiva, del funcionamiento del procesador.

Por ejemplo, el programa siguiente recorre los números enteros desde el 10 hasta el 0, momento en el que se detiene el simulador:

```
ADD 10 0
INC -1
ZJUMP 2 // Si el acumulador==0 en la última operación, saltar dos posiciones hacia adelante
JUMP -2  // Saltar dos posiciones de memoria hacia atrás
HALT
```

La memoria principal

Desde el punto de vista del diseño, la memoria principal puede ser considerada como un vector (*array*) de celdas de memoria, todas ellas con la misma capacidad de almacenamiento. Las operaciones básicas que se pueden hacer sobre la memoria son dos:

- Almacenar un cierto valor en una posición (celda) de memoria dada.
- Recuperar el valor almacenado en una cierta posición (celda) de memoria.

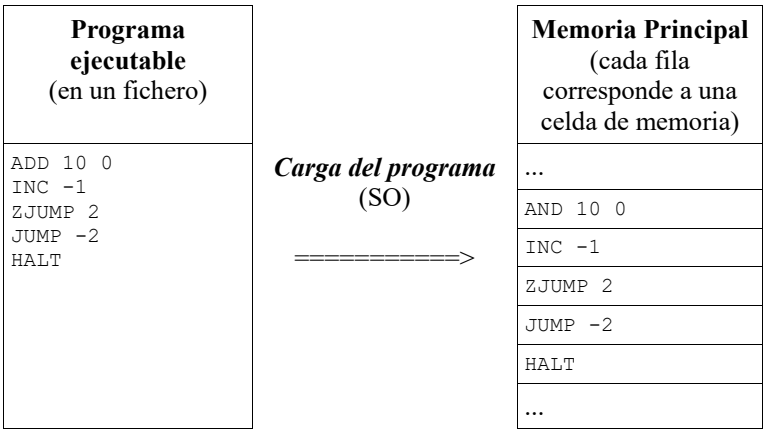
En el apartado anterior vimos que el procesador dispone de instrucciones para realizar las acciones anteriores. No podemos perder de vista, tampoco, que el procesador también accede a la memoria, por ejemplo, para obtener una instrucción o para almacenar el resultado de su ejecución (si es el caso).

Los buses del sistema

Son las típicas "vías de comunicación" entre diferentes componentes hardware. En el SI primitivo comunicarán únicamente al procesador con la memoria principal.

El sistema operativo

El SO de este SI primitivo se restringe a ser capaz de cargar en memoria principal un programa ejecutable almacenado en un fichero. El procesador sólo es capaz de ejecutar instrucciones almacenadas en memoria principal, por lo que la carga realizada por el SO es indispensable.



IMPLEMENTACIÓN

Consideraciones generales sobre la implementación

El simulador de SI está escrito en lenguaje C. En general, se han repartido en ficheros de código fuente diferentes los diferentes componentes del SI que resulta razonable e interesante separar. Además, cada componente del SI está implementado utilizando al menos dos ficheros:

- Un fichero cuyo nombre termina en "`.c`", que define las propiedades básicas de ese componente e implementa su funcionalidad.
- Un fichero cuyo nombre termina en "`.h`", que define tipos de datos propios de ese componente y los prototipos de las funciones implementadas en el correspondiente fichero "`.c`" que deben ser conocidos desde otros ficheros.

El procesador

Está implementado en los ficheros `Processor.h` y `Processor.c`. Define el comportamiento de un procesador elemental, ya comentado anteriormente. El conjunto de propiedades de este componente hardware se limita a sus registros:

```
int registerPC_CPU; // Program counter
int registerAccumulator_CPU; // Accumulator
BUSDATACELL registerIR_CPU; // Instruction register
unsigned int registerPSW_CPU; // Processor state word
int registerMAR_CPU; // Memory Address Register
BUSDATACELL registerMBR_CPU; // Memory Buffer Register
int registerCTRL_CPU; // CoNTRol bus register
```

En cuanto a su funcionalidad, define cuatro funciones:

- Una función para dar valores iniciales a los registros del procesador:

```
void Processor_InitializeRegisters(int regPC, int regAcum, unsigned int regPSW) {
        registerPC_CPU=regPC;
        registerAccumulator_CPU=regAcum;
        registerPSW_CPU=regPSW;
}
```

- Una función que simula el comportamiento iterativo de todo procesador: la ejecución de su ciclo de instrucción. La ejecución iterativa del ciclo de instrucción llega a su fin cuando en el registro PSW se anota el valor `POWEROFF`. Dicha anotación tiene lugar cuando se ejecuta la instrucción `HALT`. Las etapas habituales del ciclo de instrucción se han separado en tres funciones:

```
void Processor_InstructionCycleLoop() {
	while (registerPSW_CPU!=POWEROFF) {
		Processor_FetchInstruction();
		Processor_DecodeAndExecuteInstruction();
		Processor_ManageInterrupts();
	}
}
```

- Una función que simula la etapa de búsqueda de instrucción:

```
void Processor_FetchInstruction() {

	// The instruction must be located at the memory address pointed by the PC register
	registerMAR_CPU=registerPC_CPU;
	// Send to the main memory controller the address in which the reading has to take
place: use the address bus for this
	Buses_write_AddressBus_From_To(CPU, MAINMEMORY);
	// Tell the main memory controller to read
	registerCTRL_CPU=CTRLREAD;
	Buses_write_ControlBus_From_To(CPU,MAINMEMORY);
	// All the read data is stored in the MBR register. Because it is an instruction
	// we have to copy it to the IR register
	memcpy((void *) (&registerIR_CPU), (void *) (&registerMBR_CPU), sizeof(BUSDATACELL));
…
}
```

- Una función que simula las etapas de decodificación de instrucción, búsqueda de operandos, ejecución de instrucción y almacenamiento del resultado (lo que sigue es sólo un extracto):

```
void Processor_DecodeAndExecuteInstruction() {
	// Decode
	int operationCode= Processor_DecodeOperationCode(registerIR_CPU);
	int operand1=Processor_DecodeOperand1(registerIR_CPU);
	int operand2=Processor_DecodeOperand2(registerIR_CPU);

	// Execute
	switch (operationCode) {

		// Instruction ADD
		case ADD_INST: registerAccumulator_CPU= operand1 + operand2;
			registerPC_CPU++;
			break;

		// Instruction SHIFT
		case SHIFT_INST:
			operand1<0 ? (registerAccumulator_CPU <<= (-operand1)) :
					(registerAccumulator_CPU >>= operand1);
			registerPC_CPU++;
			break;
…
```

La memoria principal

Está implementada en los ficheros `MainMemory.h` y `MainMemory.c`. Define el comportamiento básico de la memoria principal: leer una celda de memoria y escribir en una celda de memoria.

La memoria principal está definida como un array de `MAINMEMORYSIZE` celdas de memoria:

```
// Main memory size (number of memory cells)
#define MAINMEMORYSIZE 256
```

```
// A memory cell is capable of storing a structure of the MEMORYCELL type
typedef int MEMORYCELL;
```

```
// Main memory can be simulated by a memory cell array
MEMORYCELL mainMemory[MAINMEMORYSIZE];

// Main memory has a MAR register whose value identifies
//      where the next read/write operation will take place
int registerMAR_MainMemory;

// It also has a register that plays the rol of a buffer for the mentioned operations
MEMORYCELL registerMBR_MainMemory;
```

En una celda de memoria, además de almacenar un valor entero, se puede almacenar una instrucción con dos operandos de forma codificada. Los 8 bits más significativos almacenen el código de operación, y los 24 restantes se reparten entre los dos posibles operandos (12 bits para cada uno).

Cuando describamos la implementación del SO entenderemos la razón que está detrás de esta decisión.

Además, la memoria principal tiene tres registros:

- Registro de direcciones de memoria (Memory Address Register - MAR): contiene la dirección de memoria que indica dónde va a tener lugar la siguiente operación de acceso a la misma.
- Registro de control (ConTRLol register - CTRL): Contiene codificada la operación que se quiere realizar, y también los resultados codificados.
- Registro intermedio de memoria (Memory Buffer Register - MBR): contiene la información a ser escrita o la información ya leída en una operación de acceso a la memoria.

En cuanto a su funcionalidad, define operaciones para leer y escribir en sus registros; teniendo en cuenta que la información que se ponga en el registro de control, iniciará la operación solicitada:

```
void MainMemory_SetCTRL(int ctrl) {

  registerCTRL_MainMemory=ctrl&0x3;
  switch (registerCTRL_MainMemory) {
     case CTRLREAD:
        memcpy((void *) (&registerMBR_MainMemory),
              (void *)(&mainMemory[registerMAR_MainMemory]), sizeof(MEMORYCELL));
        Buses_write_DataBus_From_To(MAINMEMORY, CPU);
        break;
     case CTRLWRITE:
        memcpy((void *) (&mainMemory[registerMAR_MainMemory])
              ,(void *) (&registerMBR_MainMemory), sizeof(MEMORYCELL));
        break;
     default:
        registerCTRL_MainMemory |= CTRL_FAIL;
        Buses_write_ControlBus_From_To(MAINMEMORY,CPU);
        return;
        break;
     }
  registerCTRL_MainMemory |= CTRL_SUCCESS;
  Buses_write_ControlBus_From_To(MAINMEMORY,CPU);
}
```

Los buses del sistema

Están implementados en los ficheros `Buses.h` y `Buses.c`. Su funcionalidad se limita a simular el funcionamiento de los buses de datos, direcciones y control: leer el contenido del registro apropiado del componente hardware que envía la información a través del bus y escribir en el registro apropiado del componente hardware destinatario de la información.

```
int Buses_write_AddressBus_From_To(int fromRegister, int toRegister) {
...
    data=Processor_GetMAR(); //  si fromRegister es CPU
...
    MainMemory_SetMAR(data); //  y toRegister es MAINMEMORY
...
}

int Buses_write_DataBus_From_To(int fromRegister, int toRegister) {
...
    MainMemory_GetMBR(data);  // si fromRegister es MAINMEMORY
...
    Processor_SetMBR(data);  // si toRegister es CPU
...
}

int Buses_write_ControlBus_From_To(int fromRegister, int toRegister) {
...
    control=MainMemory_GetCTRL();  // si fromRegister es MAINMEMORY
...
    Processor_SetCTRL(control);  // si toRegister es CPU
...
}
```

El sistema operativo

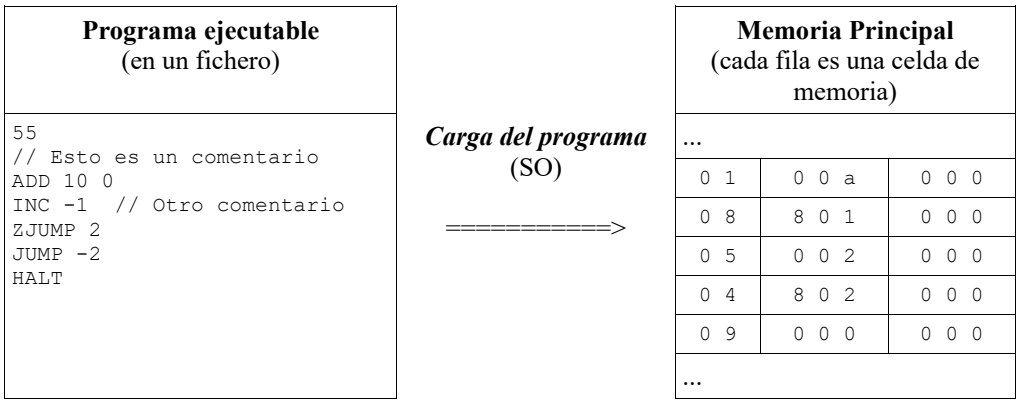
Está implementado en los ficheros `OperatingSystem.h` y `OperatingSystem.c`. Su funcionalidad se limita a la carga de un programa ejecutable en memoria principal, teniendo en cuenta que:

- El programa ejecutable contiene un primer valor entero, que indica el número de posiciones de memoria principal que necesita el programa.
- A continuación, viene un conjunto de líneas que contienen las instrucciones del programa:
    - Cada instrucción ocupa una línea del fichero.
    - El programa ejecutable puede contener comentarios.
- Durante la carga del programa, el SO codifica la información leída al formato en el que se codifica una instrucción en una celda de memoria:
    - Los códigos de operación se reducen a un sólo byte (correspondientes a los valores de un enumerado), que ocupa los 8 bits más significativos
    - En cada celda de memoria quedan codificados: código de operación, primer operando (0 si no existe) y segundo operando (0 si no existe). Los operandos negativos tiene a 1 el bit más significativo de los 12 que lo forman.
    - La codificación se hace para que una instrucción quepa en una única posición de memoria, y así simplificar el sistema.

La carga se hace dentro de la función `OperatingSystem_LoadProgram(…)`, que se muestra parcialmente a continuación:

```
int OperatingSystem_LoadProgram (FILE *programFile, int initialAddress) {

    char lineRead[LINEMAXIMUMLENGTH];
    char *token0, *token1, *token2;
    BUSDATACELL data;
    int opCode, op1, op2;

...

    Processor_SetMAR(initialAddress);
    while (fgets(lineRead, LINEMAXIMUMLENGTH, programFile) != NULL) {
    // REMARK: if lineRead is greater than LINEMAXIMUMLENGTH in number of characters,
    // the program loading does not work
            opCode=op1=op2=0;
            token0=strtok(lineRead," \n\t\r");
            if ((token0!=NULL) && (token0[0]!='/')&& (token0[0]!='\n')) {
                // I have an instruction with, at least, an operation code
                opCode=Processor_ToInstruction(token0);
                token1=strtok(NULL," ");
                if ((token1!=NULL) && (token1[0]!='/')) {
                            // I have an instruction with, at least, an operand
                            op1=atoi(token1);
                            token2=strtok(NULL," ");
                            if ((token2!=NULL) && (token2[0]!='/')) {
                                    // The read line is similar to 'sum 2 3 //coment'
                                    // I have an instruction with two operands
                                    op2=atoi(token2);
                            }
                    }
                    data.cell=Processor_Encode(opCode,op1,op2);
                    Processor_SetMBR(&data);
                    // Send data to main memory using the system buses
                    Buses_write_DataBus_From_To(CPU, MAINMEMORY);
                    Buses_write_AddressBus_From_To(CPU, MAINMEMORY);
                    // Tell the main memory controller to write
                    Processor_SetCTRL(CTRLWRITE);
                    Buses_write_ControlBus_From_To(CPU,MAINMEMORY);

                    Processor_SetMAR(Processor_GetMAR()+1);
            }
    }
    return SUCCESS;
}
```

La siguiente figura muestra cómo se almacena en memoria un programa escrito en el ensamblador de nuestra máquina (el contenido de la memoria se muestra en hexadecimal)



Los códigos de las instrucciones se obtienen mediante una X-macro del fichero `Intructions.def`, que en la V0 son los siguientes:

```
INST(ADD)       // decimal 1 = Hex (0x01)
INST(SHIFT)     // decimal 2 = Hex (0x02)
INST(NOP)       // decimal 3 = Hex (0x03)
INST(JUMP)      // decimal 4 = Hex (0x04)
INST(ZJUMP)     // decimal 5 = Hex (0x05)
INST(WRITE)     // decimal 6 = Hex (0x06)
INST(READ)      // decimal 7 = Hex (0x07)
INST(INC)       // decimal 8 = Hex (0x08)
INST(HALT)      // decimal 9 = Hex (0x09)
```

El sistema informático

Es un componente necesario desde el punto de vista de la implementación y juega, fundamentalmente, el papel de "contenedor" del resto de componentes. Está implementado en los ficheros `ComputerSystem.h` y `ComputerSystem.c`. Sus funciones principales consisten en arrancar y apagar el sistema y proveer una función para la visualización de mensajes de funcionamiento interno del simulador.

- La función que arranca el sistema se encarga de inicializar los registros del procesador, , cargar los mensajes de depuración, solicitar al SO que cargue en memoria el programa indicado y, finalmente, indicar al procesador que se disponga a ejecutar, de manera indefinida, su ciclo de instrucción:

```
void ComputerSystem_PowerOn(int argc, char *argv[]) {
…
   // Initial values for the processor registers. In the old days, the
   // CS operator had to initialize them in a similar way
   const int initialValueForPCRegister=230;
   const int initialValueForAccumulatorRegister=0;
   const int initialValueForPSWRegister=128;

   // Load debug messages
   nm=Messages_Load_Messages();
   printf("%d Messages Loaded\n",nm);

// To remember the simulator sections to be message-debugged and if messages must be coloured
   // If parameter exists is debugLevel. Default "A"
   if (argc==2)
          debugLevel = argv[1];
   …
   // Initialize processor registers
   Processor_InitializeRegisters(initialValueForPCRegister, initialValueForAccumulatorRegister,
                                 initialValueForPSWRegister);

   // If PROGRAM_TO_BE_EXECUTED exists, is executed
   programFile= fopen(PROGRAM_TO_BE_EXECUTED, "r");

   // Check if programFile exists, if not, poweroff system
   if (programFile==NULL)
          ComputerSystem_PowerOff();

// Load the program in main memory, beginning at the address given by the second argument
   OperatingSystem_LoadProgram(programFile, initialValueForPCRegister);

   // Tell the processor to begin its instruction cycle
   Processor_InstructionCycleLoop();
}
```

- El apagado del sistema, por su parte, es simple: consiste únicamente en hacer finalizar el programa C.

```
void ComputerSystem_PowerOff() {
  exit(0);
}
```

- Finalmente, el SI proporciona la función `ComputerSystem_DebugMessage(…)` cuyo fin es dar la posibilidad al programador de realizar una visualización ordenada del funcionamiento interno

del simulador. No es necesario comprender la implementación de esta función, pero sí cómo invocarla y crear los mensajes.

La parte constante de los mensajes, los tipos de las partes variables (caracteres de conversión) y los códigos de colores "@?" se especifican en el fichero "`messages.txt`".

Cada línea tiene el número de mensaje, seguido del formato del mensaje (separado por una coma).

Los posibles caracteres de conversión son: `%s`, `%d`, `%x`, `%f` y `%c`, que se sustituirían por los parámetros de tipo: *cadena de caracteres*, *número decimal*, *número hexadecimal (8 caracteres hexadecimales completados con ceros por la izquierda si fuese necesario)*, *número real* y *carácter* respectivamente.

Los posibles códigos de colores son: `@R` (rojo), `@G` (verde), `@B` (azul), `@M` (magenta), `@C` (cian), `@W` (blanco) y `@@` (monocromo). Se sigue usando el color definido hasta que aparezca otro código de color o se termine el mensaje.

```
1,%c %d %d [%s]
3,(PC: @R%d@@, Accumulator: @R%d@@ [%x])\n
```

- Por ejemplo, la invocación:

  ```
  ComputerSystem_DebugMessage(3,HARDWARE,100,1819,1819);
  ```

  Mostraría en pantalla en color rojo el 100 y el 1920):

  ```
   (PC: 100, Accumulator: 1819 [0000071B])
  ```

- Los parámetros de la función son, por este orden:
  - El número del mensaje
  - La sección del simulador en la que se circunscribe el mensaje (`HARDWARE`, en el ejemplo). Es un parámetro de tipo `char`. Se pueden encontrar diferentes secciones predefinidas en el fichero `ComputerSystem.h`.
  - Finalmente vienen los argumentos que se corresponden con el mensaje propiamente dicho (`100`, `1819` y `1819`, en el ejemplo). Los tipos de estos argumentos tienen que ser compatibles con los caracteres de conversión del mensaje del fichero de mensajes.

Compilación del simulador

La tarea de compilación se ha simplificado al máximo, con la utilización de un fichero `Makefile`. Basta, por tanto, ejecutar la orden `make` para que se genere el código ejecutable si es necesario (el fichero `Makefile` contiene una descripción de las dependencias que existen entre los distintos ficheros de código fuente, de manera que sólo se recompilará aquella parte del proyecto software que haya sufrido cambios).

```
PROGRAM = Simulator

# Detalles de Compilation
SHELL = /bin/sh
CC = cc
STDCFLAGS = -g -c -Wall
INCLUDES =
LIBRARIES =

${PROGRAM}: Simulator.o ComputerSystem.o MainMemory.o OperatingSystem.o /
                Processor.o Buses.o Messages.o
   $(CC) -o ${PROGRAM} Simulator.o ComputerSystem.o MainMemory.o /
                OperatingSystem.o Processor.o Buses.o Messages.o $( LIBRARIES)

Simulator.o: Simulator.c Simulator.h ComputerSystem.h
   $(CC) $(STDCFLAGS) $(INCLUDES) Simulator.c

Messages.o: Messages.c Messages.h
   $(CC) $(STDCFLAGS) $(INCLUDES) Messages.c

ComputerSystem.o: ComputerSystem.c ComputerSystem.h
   $(CC) $(STDCFLAGS) $(INCLUDES) ComputerSystem.c

MainMemory.o: MainMemory.c MainMemory.h
   $(CC) $(STDCFLAGS) $(INCLUDES) MainMemory.c

OperatingSystem.o: OperatingSystem.c OperatingSystem.h
   $(CC) $(STDCFLAGS) $(INCLUDES) OperatingSystem.c

Processor.o: Processor.c Processor.h
   $(CC) $(STDCFLAGS) $(INCLUDES) Processor.c


Buses.o: Buses.c Buses.h
   $(CC) $(STDCFLAGS) $(INCLUDES) Buses.c

clean:
   rm -f $(PROGRAM) *.o *~ core
…
```

Ejecución del simulador

El simulador se invoca, desde línea de comandos, tal y como se puede ver en el siguiente ejemplo:

```
$ ./Simulator A
```

Donde `Simulator` es el nombre del programa ejecutable resultante de la compilación y el argumento `A` indica las secciones del simulador de las que el usuario está interesado obtener mensajes por pantalla. Si no se pone el argumento, el valor por defecto es A.

Si ejecutamos la orden anterior (con o sin argumento A) obtendríamos un resultado similar al siguiente, con los valores del PC y del Accumulator en color rojo:

Fichero del programa

```
55
// Esto es un comentario
ADD 10 0
INC -1  // Otro comentario
ZJUMP 2
JUMP -2
HALT
```

Resultado (en pantalla) de ejecutar la orden: `./Simulator`

```
{01 00A 000} ADD 10 0 (PC: 231, Accumulator: 10 [0000000A])
{08 801 000} INC -1 0 (PC: 232, Accumulator: 9 [00000009])
{05 002 000} ZJUMP 2 0 (PC: 233, Accumulator: 9 [00000009])
{04 802 000} JUMP -2 0 (PC: 231, Accumulator: 9 [00000009])
{08 801 000} INC -1 0 (PC: 232, Accumulator: 8 [00000008])
{05 002 000} ZJUMP 2 0 (PC: 233, Accumulator: 8 [00000008])
{04 802 000} JUMP -2 0 (PC: 231, Accumulator: 8 [00000008])
…
…
{08 801 000} INC -1 0 (PC: 232, Accumulator: 0 [00000000])
{05 002 000} ZJUMP 2 0 (PC: 234, Accumulator: 0 [00000000])
{09 000 000} HALT 0 0 (PC: 234, Accumulator: 0 [00000000])
```

El texto que aparece en pantalla se corresponde con el resultado de la ejecución (repetida) de la función `ComputerSystem_DebugMessage (…,HARDWARE, …)` presente en el fichero `Processor.c`. El argumento `A` de la línea de comandos anterior especifica que se desean mostrar todos (*All*) los mensajes de todas las secciones del simulador. Igual resultado hubiésemos obtenido con la siguiente orden:

```
$ ./Simulator H
```

donde el argumento `H` se corresponde con la sección `HARDWARE` (ver `ComputerSystem.h`).

Si no quisiésemos que apareciese en pantalla ningún mensaje (excepto los de ERROR), podríamos utilizar el carácter `N` (N*one*, ninguno).

```
$ ./Simulator N
```

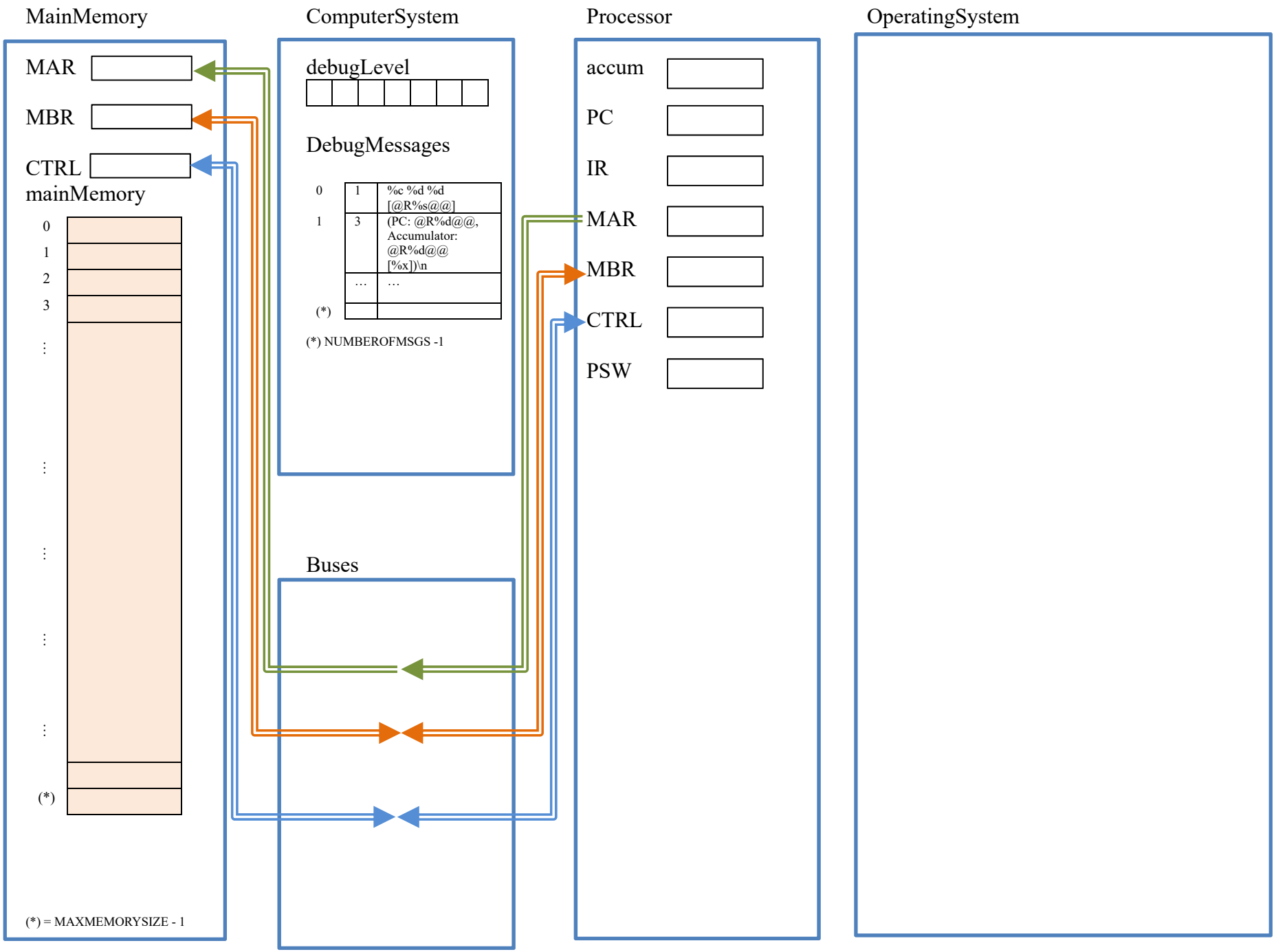
Tampoco aparecería en pantalla ningún mensaje si desde línea de comandos especificásemos una o varias secciones del simulador para las que no hay llamadas a `ComputerSystem_DebugMessage(...)` dentro del código fuente del simulador. Por ejemplo, si ejecutamos la orden:

```
$ ./Simulator MF
```

No aparecerá mensaje alguno en la pantalla, dado que no hay llamadas a la función indicada en el código actual del simulador para las secciones `SYSMEM` y `SYSFILE`.

Si alguna de las letras correspondientes a las secciones está en mayúsculas, se muestran los mensajes con colores; si son en minúsculas, en monocolor.

Los mensajes de la sección `ERROR` (`E`) se muestran siempre, aunque no se ponga la sección.


MainMemory
MAR
MBR
CTRL
mainMemory
0
1
2
3
(*)
(*) = MAXMEMORYSIZE - 1
ComputerSystem
debugLevel
DebugMessages
0
1
(*)
1
%c %d %d [@R%s@@]
3
(PC: @R%d@@, Accumulator: @R%d@@ [%x])\n
(*) NUMBEROFMSGS -1
Buses
Processor
accum
PC
IR
MAR
MBR
CTRL
PSW
OperatingSystem