OPERATING SYSTEMS

2024-2025

SIMULATOR OF A PRIMITIVE COMPUTER SYSTEM

V0

Introduction

Building an operating system (OS) that runs over real hardware is a very complex task, even if the OS is a very simple one. The main reason is that hardware is complex and diverse, and so, any software piece design to control and manage it in an efficient way must be, in turn, very complex. Despite that, we do not give up the possibility of studying the details of an OS from the design and implementation points of view. If things get very complex, we can simplify (retaining the most relevant aspects) and simulate.

The early computer systems (CS) were very primitive. Each of their components, hardware and software, were very simple, in comparison with the ones we use nowadays. From the simulation point of view, our primitive CS would be composed of:

- A processor (primitive).
- Main memory.
- The system buses.
- An OS (primitive).

So, the OS would be in charge of loading an executable program in main memory. Afterwards, the program would be executed by the processor.

DESIGN

The processor

The processor is a very basic one. We will not distinguish all its components (registers, Control Unit, Arithmetic Logic Unit) but we will make a design based in its functionality. Thus, the processor is composed of:

- A set of registers:
    - Accumulator register, generally used in arithmetic and memory access operations.
    - Program counter (PC).
    - Instruction register (IR).
    - Processor state word (PSW).
    - Memory address register (MAR).
    - Memory buffer register (MBR).
    - Bus control register (CTRL).
- The necessary functionality to execute iteratively its instruction cycle (instruction cycle stages):
    - Instruction fetching.
    - Instruction decoding.
    - Instruction execution.
    - Result storing.
    - Interrupts processing.

Every processor is built to process a given set of instructions (its instruction set). The instruction set of the processor is very simple:

- `ADD op1 op2`: adds the integer values of `op1` and `op2` and stores the result in the accumulator

register.

- `SHIFT op1`: carries out an arithmetic shift over the accumulator register the number of bit positions specified by `op1`. If `op1`is negative, it will be a left shift; if it is positive, a right shift. In the latter case, the most significant bit is used to fill bit position, so if a negative value were stored in the accumulator, it will continue to be negative after the shift. The result is stored in the accumulator.
- `NOP`: *NO-OPERATION* typical instruction that consumes a processor cycle without doing any task.
- `JUMP relativeAddr`: modifies the value of the PC register incrementing it with the value of `relativeAddr`.
- `ZJUMP relativeAddr`: has the same functionality as the `JUMP` instruction but only if the current value of the accumulator register is equal to 0.
- `READ addr`: reads the contents of the memory cell addressed by `addr`, putting the read value in the accumulator register.
- `WRITE addr`: stores the value of the accumulator register in the memory cell addressed by `addr`.
- `INC op1`: increments the current value of the accumulator register with `op1`.
- `HALT`: causes the processor to stop.

For example, next program goes through the list of integer numbers 10 to 0 and finally, stops the simulator.

```
ADD 10 0
INC -1
ZJUMP 2  // If accumulator register==0, jump two memory positions forwards
JUMP -2  // Jump two memory positions backwards
HALT
```

Main memory

From the design point of view, main memory can be considered as a memory cell vector (array). The two basic operations defined for main memory are:

- Storing a given value in a given address (of a memory cell).
- Retrieving a value from a given memory address.

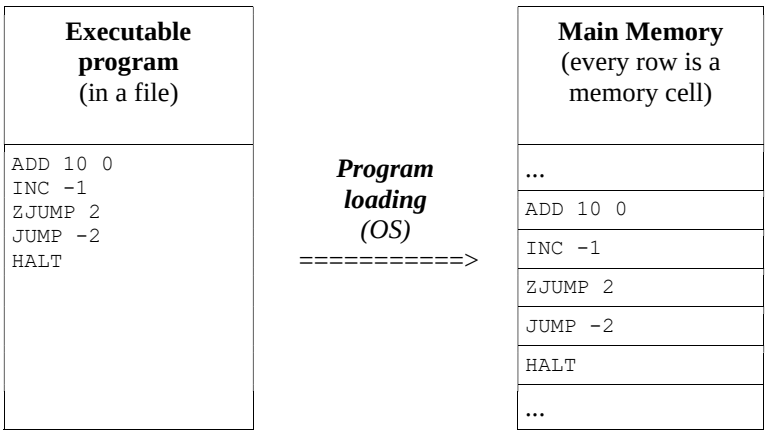
Some of the processor instructions are designed to make those actions. Besides them, the processor also accesses main memory to, e.g., retrieve an instruction or to store the result of its execution (just in case).

The system buses

They must be considered the “communication lines” among hardware components. In the primitive CS they only communicate the processor with main memory.

The operating system

The OS of this primitive CS is only capable of loading a program, stored in a file, in main memory. This preliminary task is essential, because the processor is only capable to execute instructions stored in main memory.



IMPLEMENTATION

General considerations of the implementation

The CS simulator is written using the C language. Separate source files contain different CS components, as far as we have considered to separate. Moreover, each CS component is implemented using two files:

- A file whose name ends with "`.c`", that defines the basic features of the component and implements its functionality.
- A file whose name ends with "`.h`", that defines data types related to this component and the function prototypes implemented in the corresponding "`.c`", file.

The processor

It is implemented in the files `Processor.h` and `Processor.c`. Defines the behaviour of a simple processor, as described before. The set of properties of this component is given by its registers:

```
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
```

From the point of view of its functionality, it defines four functions:

- A function to give initial values to some of its registers:

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

- A function that simulates the iterative behaviour of every processor: the execution of its instruction cycle. This iterative execution ends when the PSW register contains the `POWEROFF` value. The `HALT` instruction is the responsible of storing that value in the register. The common instruction cycle stages have been implemented by three functions:

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

```
        }
}
```

- A function that simulates the instruction fetching:

```
    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));
    }
```

- A function that simulates the decoding, operands fetching, execution and result storing stages (below, an extract of the function):

```
void Processor_DecodeAndExecuteInstruction() {
        // Decode
        char 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=
                        registerIR_CPU.operand1 + registerIR_CPU.operand2;
                        registerPC_CPU++;
                        break;

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

Main memory

It is implemented in the files `MainMemory.h` and `MainMemory.c`. Defines the behavior of the main memory: reading from a memory cell and writing in a memory cell.

Main memory is defined as an array of `MAINMEMORYSIZE` memory cells:

```
// 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;
```

As you can see in the previous definitions, a memory cell is capable of storing a whole instruction

composed of an operation code and two operands. The 8 most significant bits store the operation code and the remaining 24 bits, the two possible operands (12 bits for each one).

Furthermore, main memory has three registers:

- Memory address register (MAR): contains the memory address where the next access operation will take place.
- Control register (CTRL): contains the operation to be performed coded and the result of its execution, coded as well.
- Memory buffer register (MBR): contains the information to be written or the information just read by a memory access operation.

From the point of view of its functionality, it defines functionality to read and write its registers, given that the information in the control register will start the requested operation:

- A function to read the contents of a memory cell:

```
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);
}
```

The system buses

They are implemented in the files `Buses.h` and `Buses.c`. The functionalty of this component is restricted to simulate the behaviour of the data, addresses and control buses: read the contents of a given hardware component (the one that sends the information along the bus) and write the appropriate register of the recipient hardware component.

```
int Buses_write_AddressBus_From_To(int fromRegister, int toRegister) {
…
     data=Processor_GetMAR(); //  if  fromRegister is CPU
…
     MainMemory_SetMAR(data); //  and toRegister is MAINMEMORY
…
}

int Buses_write_DataBus_From_To(int fromRegister, int toRegister) {
…
     MainMemory_GetMBR(data);  // if fromRegister is MAINMEMORY
…
     Processor_SetMBR(data);  // if toRegister is CPU
…
}

int Buses_write_ControlBus_From_To(int fromRegister, int toRegister) {
…
     control=MainMemory_GetCTRL();  // if fromRegister is MAINMEMORY
…
```

```
    Processor_SetCTRL(control);  // if toRegister is CPU
...
}
```

The operating system

It is implemented in the files `OperatingSystem.h` and `OperatingSystem.c`. Its functionality is restricted to loading an executable program into main memory, having in mind that:

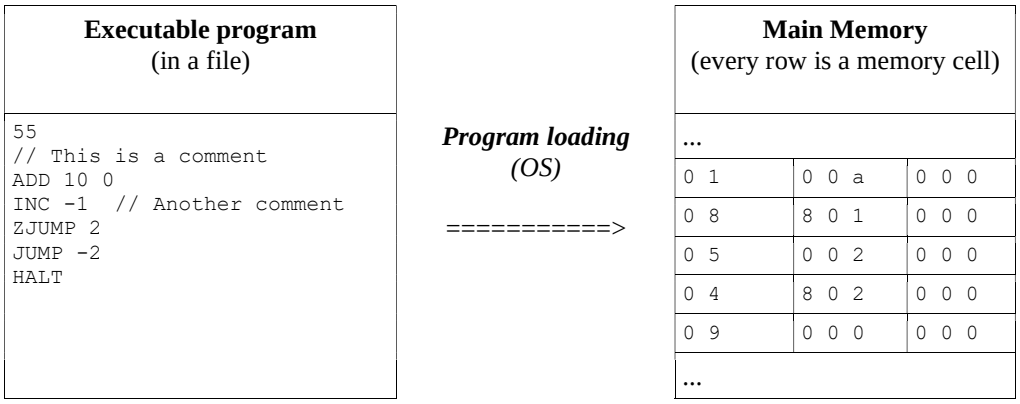
- The executable program must contain, as the first value, an integer, that indicates the number of necessary memory cells it declares.
- Then, a set of lines, containing the instructions of the program:
  - Each instruction in a different line.
  - The program can contain comments.
- During program loading, the OS processes the information read and convert it to a more suitable format for memory cells internal arrangement:
  - Operation codes are reduced to one character (first character of the instruction, in lowercase) and use the 8 most significant bits.
  - Each memory cell contains, separately, operation code, first operand (a 0, if it does not exist) and second operand (a 0, if it does not exist). The most significant bit of negative operands is set.
  - This way, a whole instruction fits in one memory cell, make the system more simple.

Loading operation is implemented by the `OperatingSystem_LoadProgram(…)` function:

```
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;
}
```

Next figure shows how a program is stored in main memory (memory contents are hexadecimal values).



Operation codes are obtained by means of an X-macro over the contents of `Intructions.def` file:

```
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)
```

The computer system

It is a necessary component from the implementation point of view. Its main role is as a container for the rest of the components of the CS. It is implemented in the files `ComputerSystem.h` and `ComputerSystem.c`. Its main functions are powering on and off the system and providing a function that shows the internal working of the simulator.

- The function that powers on the system is in charge of initializing the processor registers, loading the debug messages, requesting the OS to load the user program and, finally, requesting the processor to begin the execution of its instruction cycle.

```
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
   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();
}
```

- The system power off is simple: it finishes the C program.

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

- Finally, the CS component provides the programmer with a function to show the internal working of the simulator. It is not necessary to understand the implementation of this function, but it is important to know how to invoke it.

  The static part of the messages, as well as the type and the colour code "@?" are specified inside the "`messages.txt`" file. Each line includes the number of the message and, separated by a comma, its format.

  Several conversion characters can be included in the format: `%s`, `%d`, `%x`, `%f` and `%c`. They will be replaced by the value of the arguments with a type of *string*, *decimal number*, *hexadecimal number* (8 hexadecimal characters, filled with zeroes, if necessary) *float number* and *character*.

  The colour codes are: `@R` (red), `@G` (green), `@B` (blue), `@M` (magenta), `@C` (cyan), `@W` (white) and `@@` (monochrome). Once a colour has been set, this colour will be used until another colour code appears or the message ends.

  Assuming that the "messages.txt" file has the following content:

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

  - For example, if invoking like this:

```
ComputerSystem_DebugMessage(3,HARDWARE,100,2122,2122);
```

    we should get this (on the monitor):

```
(PC: 100, Accumulator: 2122 [0000084A])
```

  - The parameters of the function are:
    - The message number.
    - The simulator section the message belongs to (`HARDWARE`, in the example). It is a `char` parameter. All predefined sections can be found in the file `ComputerSystem.h`.
    - Finally, the arguments corresponding to the message itself (`100`, `1920` and `1920` in the example). The types of these arguments must be compatible with the conversion characters specified for the message in the messages file.

Compiling the simulator

Compiling the simulator is a simple task with the help of a `Makefile`. By using the make command, the executable code corresponding to the simulator will be generated, if the compilation succeeds. As expected, only the necessary parts of the project will be compiled, taking into account the last made changes in the source code.

```
PROGRAM = Simulator

# Compilation Details
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
…
```

Simulator execution

The simulator is executed, from the command line, as in the following example:

```
$ ./Simulator A
```

where `Simulator` is the name of the executable program resulting from the compilation and argument `A` indicates the sections of the simulator the user is interested to see messages of. If we execute that command, the displayed result would be like this:

File `programToBeExecuted`	Displayed result when executing `./Simulator A`
55 // This is a comment ADD 10 0 INC -1 // Another comment ZJUMP 2 JUMP -2 HALT	{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])

The displayed information corresponds to the result of the (repeated) execution of the function `ComputerSystem_DebugMessage(HARDWARE, …)`, invoked from the processor (`Processor.c`). The command line argument `a` specifies that all messages of all simulator sections must be displayed. We would have got the same result with this command:

```
$ ./Simulator H
```

where the `H` argument corresponds to the `HARDWARE` section (see `ComputerSystem.h`).

If we do not want any message to be displayed, we would use the argument `N` (`N`one).

```
$ ./Simulator N
```

No message would be displayed if we specify as an argument a section for which there are not

invocations to `ComputerSystem_DebugMessage` in the source code. For example, the command line:

```
$ ./Simulator MF
```

does not produce any message, because there are not invocations to the mentioned function for sections `SYSMEM` and `SYSFILE`.

If you use capital letters to specify a section, messages will be displayed using different colours; if you use lowercase letters, they will be displayed in black and white.

Messages in the section ERROR (E) are always displayed.

Operating Systems: Simulator V0