Computer Architecture 2023/24

TPC 2

Delivery: 9am59m on April 16, 2024

This homework consists of a programming exercise **to be carried out in a group of no more than two students**. You can clarify general doubts with colleagues but the solution and writing of the code should be strictly carried out by the members of the group. All resolutions will be automatically compared, and cases of plagiarism will be punished in accordance with the regulations in force. **If you use tools such as CoPilot or ChatGPT, you must include a comment in the source code reporting this use.**

Exercise

In this exercise you must complete the code of a simulator (written in the C language) of an architecture composed of a very simple CPU and a memory. This includes, already implemented, a console where it is possible for a user to give commands to read and write to the central memory, including reading to memory the values represented in a text file. You can also have the instructions in the memory executed. The implemented commands are as follows:

> `poke` *EEE X* – writes the value X to the *EEA* address.
>
> `peek` *EEE* – reads the value at the *EEE* memory address. The value is displayed in hexadecimal.
>
> `dump` *X* – shows the contents of the accumulator, flags, and *X* words from memory starting from address 0.
>
> `load` *FILE* – reads the file named *FILE,* interpreting each line of text as a value that is placed sequentially in program memory.
>
> `run` – executes the instructions from address 0 of memory.

The *X* and *EEA* values can be given in base 10 or in base 16 if starting with `0x`.

The code to complete is in the `dorun.c` file and you should implement the *fetch*, *decode,* and *execute* loop to simulate the execution of the instructions in memory. Execution should always run from memory address 0 until the HALT statement is executed (see next section).

Processor Machine Instructions:

The processor uses 8-bit words and has one 8-bit register which is the accumulator. It has two memories with 256 positions each:

- Program memory with 256 16-bit positions.
- Data memory with 256 8-bit positions.

There are the following registers:

- *Program Counter* (PC) with 8 bits that contains the address of the instruction to be executed.
- *Instruction Register* with 16 bits that contains the instruction under execution.

There are 3 flags associated with the arithmetic and logical unit:

- ZERO contains the value 1 when the last arithmetic operation produced a zero result and 0 in the opposite case.
- CARRY: contains 1 when, in the last arithmetic operation performed, there was transport in the most significant bit and 0 in the opposite case.
- OVERFLOW contains 1 when, in the last arithmetic operation performed, there was a positive or negative overflow and 0 in the opposite case.

The instructions are fixed in size of 16 bits distributed as follows.

15 12	11 8	7 0
Instruction code (4 bits)	ALU Operation (4bits)	Address or constant (8-bits)

Bits 15-12: *Instruction code*: specifies the instruction to be executed and how the remaining bits should be interpreted.

Bits 11-8: *Specifies the operation to be performed by the ALU.* Only sequences 0000 (ADD) and 0001 (SUB) are used.

7-0 bits: *Constant with 8 bits*: can be an address (unsigned integer) or a value (signed integer)

In some instructions there are bits that are not considered. The supported instructions and their machine code in hexadecimal representation are described below. In the *Statement column* , the values displayed are sequences of digits in binary. The following conventions are used:

Symbol	Meaning
x	1 bit to ignore
aaaa	4 bits indicating the operation to be performed by the ALU
eeeeeeee	8 bits specifying an address (unsigned integer 0 to 255)
vvvvvvvv	8 bits specifying an integer constant with 8-bit sign (-128 to +127)

Instruction	**Mnemonic**	**Description**
0000 xxxx xxxxxxxx	noop	It doesn't do anything
0001 0000 vvvvvvvv 0001 0001 vvvvvvvv	addi value subi value	Accumulator ← Accumulator + vvvvvvvv Accumulator ← Accumulator - vvvvvvvv
0010 0000 eeeeeeee 0010 0001 eeeeeeee	add address sub address	Accumulator ← Accumulator + MemoryData[eeeeeeee] Accumulator ← Accumulator - MemoryData[eeeeeeee]
0011 xxxx xxxxxxxx	clac	Accumulator ← 0; set the flag ZERO to 1
0100 xxxx eeeeeeee	stor Address	MemoryData[eeeeeeee] = Accumulator
0101 xxxx eeeeeeee	beqz address	if (Acumuldor == 0) PC← eeeeeeee
1111 xxxx xxxxxxxx	halt	End of program and simulation

The ADD, SUB, ADDI, SUBI, and CLAC instructions must correctly set the flags.

Example of a program for multiplying two unsigned integers (for example, to calculate 2x3). Irrelevant bits are set to 0. Note that in the file to be uploaded to the simulator only the middle column can be there, without the header.

Code

```
Address     code            Mnemonic
0x00:       0x3000 clac     // ac ← 0
0x01:       0x4002 stor 0x02    // result ←0
0x02:       0x4003 stor 0x03    // counter ←0
0x03:       0x3000 clac     // ac ← 0
0x04:       0x2001 add 0x01     // ac ← multiplier
0x05:       0x2103 sub 0x03     // ac ← multiplier - counter
0x06:       0x5011 beqz 0x11    // se counter == multiplier termina
0x07:       0x3000 clac
0x08:       0x2002 add 0x02     //  ac ← result
0x09:       0x2000 add 0x00     //  ac ← result + multiplicand
0x0A:       0x4002 stor 0x02    // result ← ac
0x0B:       0x3000 clac
0x0C:       0x2003 add 0x03     // ac ← counter
0x0D:       0x1001 addi 0x01    // ac ← ac + 1
0x0E:       0x4003 stor 0x03    // counter ← counter + 1
0x0F:       0x3000 clac
```

```
0x10:        0x5003 beqz 0x03    // jump to instruction at program memory address 0x03
0x11:        0xf000 halt
```

Data

```
0x00:  2     // multiplicand initialized with poke
0x01:  3     // multiplier initialized with poke
0x02:  ?     // result, non-initialized, read with peek in the end
0x03:  ?     // counter, non-intialized
```

The supplied `p.code` file contains the code and data of this program in hexadecimal notation, occupying one word of memory per line. Running the simulator, we can load the program by doing `load p.code` and running it. We can check the result by seeing what is in the address variable 0x02 with `peek`. We can change the variables corresponding to the multiply and multiplier using the `poke` command. Example of a 4x2 calculation session:

```
cmd> load p.code

cmd> poke 0x00 4
cmd> poke 0x01 2
cmd> run

HALT instruction executed
cmd> peek 0x02
8
cmd>
```

Delivery

The delivery mode will be announced via CLIP.

The delivery must be made on time, submitting only your *dorun.c file.* Compliance with the specification by the delivered code is not the only criterion for defining the grade. Other criteria, such as the quality of the code and completeness of the solution, are also considered.

The program will be compiled with the following command:

```
gcc -Wall -std=c99 -o sim sim.c dorun.c
```