NOVA SCHOOL OF SCIENCE & TECHNOLOGY

Arquitetura de Computadores 2021/22

Trabalho Individual 1 (Individual Assignment 1)

Due: April 7, 2022, 23:59

The goal of this assignment is to complete the implementation of a simulator, called CAOSS, for a very simple CPU.

The CPU's Instruction Set Architecture

The CPU features:

- A word of 16 bits, organized according to the *Little Endian* ordering.
- Two addressing spaces (that we will call segments):
  - *code* segment — a 2^{16}-byte memory with the code of the program. The first instruction is placed at address 0.
  - *data* segment — a 2^{16}-byte memory with the global data to be manipulated by the program. The first global variable is also placed at address 0.
- A set of 16-bit registers
  - *r1* to *r8* — 8 general purpose integer registers, internally numbered from 0 to 7. For example, register *r4* is internally identified with number 3.
  - *pc* — the program counter that contains the address (in the *code segment*) of the next instruction to execute.
  - *ir* — the instruction register that contains the representation of the next instruction to execute.

The CPU supports 4 types of instructions, with the following formats:

Memory Load/store	00	Operation (1 bit)	Address (10 bits)			Register (3 bits)
Immediate Load	01	Value (11 bits)				Register (3 bits)
ALU	10	Operation (4 bits)	Size (1 bit)	Register1 (3 bits)	Register2 (3 bits)	Register3 (3 bits)
Interrupt	11	Interrupt number (14 bits)				

Memory Instructions

Memory instructions may be of 2 kinds:

0 — Load a value from the *data segment* to a register. The address of the value to load is given by the *Address* field and the target register is given by the *Register* field.
Example: instruction 00 0 0000000010 111 represents the load of the value at *data*[2] to register r8 (with identifier 7)

1 — Store a value from a register to the *data segment*. The identifier of the source register is given by the *Register* field and the address where to save the value is given by the *Address* field.
Example: instruction 00 1 0000000010 111 represents the store of the value at register r8 (with identifier 7) to *data*[2].

Immediate Instruction

Loads an immediate value to a register. The value to load is given by the *Value* field and the target register is given by the *Register* field.
Example: instruction 01 00000000110 111 represents the load of the value 6 to register r8.

ALU Instructions

The ALU instructions provide for the execution of operations over the registers. If the operation is binary, the first two registers in the instruction denote the input registers, while the third denotes the output register, i.e., where the result is to be placed. If the operation is unary, the *Register2* field is ignored. The set of operations supported are:

0 ADD — Register3 = Register1 + Register2
1 SUB — Register3 = Register1 - Register2
2 AND — Register3 = Register1 *and* Register2, where *and* denotes the bitwise conjunction.
3 OR — Register3 = Register1 *or* Register2, where *or* denotes the bitwise disjunction.
4 NEG — Register3 = - Register1
5 NOT — Register3 = *neg* Register1, where *neg* denotes the one's complement.
6 SAR — Register3 = Register1 >> Register2, where >> denotes the arithmetic shift.
7 SAL — Register3 = Register1 << Register2, where << denotes the arithmetic shift.

The operations may be performed at word or byte level, given by the Size bit.

0 – Byte
1 – Word

Example: Suppose you have the following state of registers r1, r2 and r3:

r1 [4] [00000000 00000100]
r2 [34] [00000000 00100010]
r3 [60000] [11101010 01100000]
…

and that the operation code (*opcode*) of operation ADD is 0000. Instruction 10 0000 1 000 001 010 represents r3 = r1 + r2 at word level. Hence, the resulting value of r3 will be:

r3 [38] [00000000 00100110]

In turn, instruction 10 0000 **0** 000 001 010 represents r3 = r1 + r2 at byte level. Register r3 must now have only its least significant byte modified:

r3 [59942] [11101010 00100110]

Interrupt Instructions

An interrupt interrupts the execution of the current program to perform a task with higher priority or to the call the operating system. The identifier of the interruption is given in the instruction itself, in field *Interrupt number*. In the version provided, CAOSS only supports one type of interruption, with number 0, that enables the program to call the operating system. The call to be made (to the operating system) is also identified by a number that must be given in the *r1* register. Once again, only one system call is available. Its number is 0 and enables a program to terminate its execution by asking the operating system to do so.

The CAOSS Simulator

The CAOSS simulator is divided in several modules

- *memory*.h and *memory*.c — interface and implementation of the operations allowed over memory. See their description in file *memory*.h.
- *cpu*.h and *cpu*.c — interface and implementation of the CPU's behavior, namely the fetch-decode-execute cycle.
- *alu*.h and *alu*.c — interface and implementation of the ALU operations. See their description in file alu.h.
- *os*.h and os.c — interface and implementation of the system calls provided by the operation system. Currently there only two, load a program to memory and terminate a program's execution. See their description in file os.h.

To run the simulator run the command:

```
./caoss name_of_the_program_to_run [size_of_the_memory]
```

where `name_of_the_program_to_run` is the name of the binary file to execute, and `size_of_the_memory` is an optional parameter that defines the maximum size (of the *data* and *code* segments.

After processing the input parameter, the simulator calls function *cpu_run* (in file *cpu.c*) that has the following behavior:

```
void cpu_run() {
    while (1) {
        fetch();
        decode_and_execute();
        pc+=2;
    }
}
```

At each iteration, *cpu_run* first calls function *fetch* to obtain the next instruction to execute and, next, function *decode_and_execute* to decode and execute such instruction. The first step in the decoding process is to know the type of the instruction is to execute: MEMORY, LOAD_IMMEDIATE, ALU and INT. Once such information is retrieved, the function directs the execution flow to the respective function.

```
void decode_and_execute() {

    instruction_type inst_type = …; // TODO

    switch (inst_type) {
        case MEMORY:
            memory_operation();
            break;

        case LOAD_IMMEDIATE:
            load_immediate_operation();
            break;

        case ALU:
            alu_operation();
            break;
```

```
            case INT:
                interrupt_operation();
                break;

            default:
                error("instruction type not known.");
                exit(1);
        }
    }
```

In each of 3 of these 4 functions, you will have to proceed the decoding process. *interrupt_operation*, the simplest of the four is already implemented to serve as an example. Looking at *alu_operation* below

```
void alu_operation() {

    // TODO: decode the instruction.
    alu_opcode opcode = …; // TODO
    op_size size = …; // TODO
    byte reg_in_1 = …; // TODO
    byte reg_out = …; // TODO

    if (is_binary_operation(opcode)) {
       byte reg_in_2 = …; // TODO

       switch (opcode) {
         case ADD:
            add(size, reg_in_1, reg_in_2, reg_out);
            break;
         …
       }
    }
    …
}
```

you will have to retrieve all the components of the instruction, for the simulation to call the ALU function that carries out the execution of the operation.

The Assignment

In your assignment you must complete the implementation of CAOSS. Concretely, you must:

1. Implement the *read_data* and *write_data* and *read_code* functions in file ***memory.c***.

```
/**
 * Read a value from data[address] and copy it to register reg
 */
void read_data(caoss_address address, caoss_word* reg);

/**
 * Write the given register value to data[address]
 */
void write_data(caoss_word value, caoss_address address);

/**
 * Read a value from code[address] and copy it to register reg
 */
void read_code(caoss_address address, caoss_word* reg);
```

2. Implement function *fetch* in file ***cpu*.c**

```
/**
 * Uses function read_code to load instruction at code[pc] to ir (the
instruction register)
 */
void fetch()
```

3. Decode an instruction by completing the implementation of functions *decode_and_execute*, *memory_operation*, *load_immediate_operation* and *alu_operation* in file ***cpu.c***.
   - *In* function *decode_and_execute* you must retrieve the type of the instruction to execute: *memory*, *immediate*, *ALU* or *interrupt*, by looking at the 2 most significant bits of the instruction. Consider, once again, the 10 0000 1 000 001 010 instruction introduced above.

     We have than $1000001000001010_{(2)} = 33290_{(10)}$
     So, after executing the *fetch* function, the value of the *ir* register will be 33290.
     The type of instruction will have to be 2, which maps to value ALU in *instruction_type* enumeration used to define the types of instructions supported (see file cpu.h).

   The code given will call function *alu_operation*, where you will have to retrieve the operation's opcode, size, and registers, according to the instruction format table at the beginning of this document. In the given example
   opcode = 0, size = 1 (WORD), reg1=0, reg2 = 1 and reg3 = 2

   The DEBUG information will produce a link such as:

```
ALU instruction 0x820a --> type = 2, opcode = 0, size = 1, reg_in_1 =
2 (r3), reg_in_2 = 2 (r3), reg_out = 6 (r7)
```

   To check if your decoding is correct, you may compare it against the DEBUG output of the assembler (presented in a section below). **The outputs must match**.

4. Implement all the ALU operations in the *alu.c* file. To access a register, you must access the registers array. For example, to access register r1 (with identifier 0), you must access registers[0].

   A partial implementation of operation ADD may be:

```
void add(op_size size, byte reg1, byte reg2, byte reg_result) {
   if (size == WORD)
     registers[reg_result] = registers[reg1] + registers[reg2];
   else
     // figure it out ☺
}
```

The Assembly Language

To aid in the programming of our CPU, we supply a basic assembly language with two pseudo-operations with the following syntax:

`.data`
Global data initialization
`.code`
Code of the program

Global Data Initialization

The initialization of global data follows the syntax:

size value

where *size* may be "`.b`", to denote byte, or "`.w`" to denote word, and *value* is the value to assign to the memory location with a given address (starting at 0, and computed by the assembler).

Example:

```
.data
  .w 300
  .w 20
  .b 2
  .w 4400
```

will produce a *data segment* with contents:

```
0:  [44]   [0010 1100]
1:  [ 1]   [0000 0001]  // these two bytes represent value "w 300" in Little Endian
2:  [20]   [0001 0100]
3:  [ 0]   [0000 0000]  // these two bytes represent value "w 20" in Little Endian
4:  [ 2]   [0000 0010]  // this byte represent value "b 2"
5:  [48]   [0011 0000]
6:  [17]   [0001 0001]  // these two bytes represent value "w 4400" in Little Endian
```

Assembly instructions

The assembly instructions represent the instructions supported by the CPU.

mov (number), register	Load a word from data[number] to the given register
mov register, (number)	Store the contents (a word) of the register in data[number]
mov number, register	Load value number (a word) to the given register
add size reg1, reg2, reg3	reg3 = reg1 + reg2
sub size reg1, reg2, reg3	reg3 = reg1 - reg2
and size reg1, reg2, reg3	reg3 = reg1 and reg2
or size reg1, reg2, reg3	reg3 = reg1 or reg2
not size reg1, reg2	reg2 = not reg1
neg size reg1, reg2	reg2 = neg reg1
sar size reg1, reg2, reg3	reg3 = reg1 >> reg2 (arithmetic shift)
sal size reg1, reg2, reg3	reg3 = reg1 << reg2 (arithmetic shift)
int number	Trigger the execution of the behavior associated to the interrupt with the given number.

Where applicable, `size` is optional and may be "`.b`" or "`.w`". If no size is given, the operation will be performed at word-level.

The assembly does not feature named variables. Thus, data items will have to be referred by their addresses, as in exemplified in the following example that adds data[0] (with value 10) to data[2] (with value 20) and stores the result in data[4] (initially 0).

```
.data
    .w 10
    .w 20
    .w 0
.code
    mov (0), r1          # r1 ← data[0]  r1: 10
    mov (2), r2          # r2 ← data[2]  r2: 20
    add r1, r2, r3       # r3 ← r1 + r3          r3: 30
    mov r3, (4)          # data[4] ← r3

# terminate program
    mov 0, r1            # r1 ← 0                r1: 0
    int 0                # system call 0
```

The Assembler

The assembler, named CAAS, transforms a program in the assembly language in a binary program understandable by CAOSS. To execute CAAS, run command:

```
./caas name_of_example_to_assemble.caoss name_of_the_output_file
```

For example:

```
./caas examples/add.caoss add.out
```

To, subsequently, execute CAOSS run command:

```
./caoss add.out
```

Compiling CAOSS and CAAS

To compile CAAS, you will first need to install Flex and Bison

```
sudo apt install flex bison
```

Having the software installed,

1. download ti1.zip
2. unzip the file
3. cd ti1
4. run command `make`

You should now have the `caoss` and `caas` executables in the current directory. The code that you need to modify is in folder src/*caoss*. Take the time to read it and place your doubts in Piazza.

The Output of the CAOSS simulator

At each iteration of the fetch-decode-execute loop, the simulator outputs the following information:

- The decoding of the instruction to execute
- The resulting state of the data segment
- The resulting state of the registers.

Example: Consider the first lines of example ex1.caoss assembled to file ex1.out

```
mov (0), r5
mov 22, r2
add .b r5, r2,r3
```

The output with some comments to help its understanding is:

```
Loading program examples/ex1.out
Number of global variables 1
Data segment:
     0: [   10]   [00001010]
     1: [    0]   [00000000]  // Value 10 in Little endian
Registers:
 pc [    0]       [00000000 00000000] // Address of the next instruction to execute
 ir [    0]       [00000000 00000000]

 r1 [    0] [    0] [00000000 00000000]
 r2 [    0] [    0] [00000000 00000000]
 r3 [    0] [    0] [00000000 00000000]
 r4 [    0] [    0] [00000000 00000000]
 r5 [    0] [    0] [00000000 00000000]
 r6 [    0] [    0] [00000000 00000000]
 r7 [    0] [    0] [00000000 00000000]
 r8 [    0] [    0] [00000000 00000000]

Memory instruction 0x4 --> type = 0, opcode = 0, address = 0, register = 4 (r5)
       // Decoding of instruction mov (0), r5
Data segment:
     0: [   10]   [00001010]
     1: [    0]   [00000000]
Registers:
 pc [    2]       [00000000 00000010] // Address of the next instruction to execute
 ir [    4]       [00000000 00000100] // Instruction executed: 0x4

 r1 [    0] [    0] [00000000 00000000]
 r2 [    0] [    0] [00000000 00000000]
 r3 [    0] [    0] [00000000 00000000]
 r4 [    0] [    0] [00000000 00000000]
 r5 [   10] [   10] [00000000 00001010] // data[0]=10 is copied to register r5.
                                           Note that the Little Endian ordering no longer
                                           applies
 r6 [    0] [    0] [00000000 00000000]
 r7 [    0] [    0] [00000000 00000000]
 r8 [    0] [    0] [00000000 00000000]

Load immediate instruction 0x40b1 --> type = 1, value = 22, register = 1 (r2)
       // Decoding of instruction mov 22, r2
Data segment:
     0: [   10]   [00001010]
     1: [    0]   [00000000]
Registers:
 pc [    4]       [00000000 00000100] // Address of the next instruction to execute
 ir [16561]       [01000000 10110001] // Instruction executed: 0x40b1

 r1 [    0] [    0] [00000000 00000000]
 r2 [   22] [   22] [00000000 00010110] // Value 22 loaded to register r2
 r3 [    0] [    0] [00000000 00000000]
 r4 [    0] [    0] [00000000 00000000]
 r5 [   10] [   10] [00000000 00001010]
 r6 [    0] [    0] [00000000 00000000]
 r7 [    0] [    0] [00000000 00000000]
 r8 [    0] [    0] [00000000 00000000]
```

```
ALU instruction 0x810a --> type = 2, opcode = 0, size = 0, reg_in_1 = 4 (r5), reg_in_2 = 1
 (r2), reg_out = 2 (r3)
         // Decoding of instruction add .b r5, r2, r3
  Data segment:
    0: [   10]    [00001010]
    1: [    0]    [00000000]
 Registers:
  pc [    6]      [00000000 00000110] // Address of the next instruction to execute
  ir [33034]      [10000001 00001010] // Instruction executed: 810a

  r1 [    0] [    0] [00000000 00000000]
  r2 [   22] [   22] [00000000 00010110]
  r3 [   32] [   32] [00000000 00100000]  // Least significant byte of register r3
                                             is filled with the result of r5+r2
  r4 [    0] [    0] [00000000 00000000]
  r5 [   10] [   10] [00000000 00001010]
  r6 [    0] [    0] [00000000 00000000]
  r7 [    0] [    0] [00000000 00000000]
  r8 [    0] [    0] [00000000 00000000]
```

GRADING

- Functions *read_code* and *fetch*: 20%
- Decode the instruction type in function *decode_and_execute*: 15%
- Decode the instruction in function *mem_operation* and implement *read_data* and *write_data*: 25%
- Decode the instruction in *alu_operation* plus implementation of ALU operations for size WORD: 25%
- Implementation of ALU operations for size BYTE: 15%