MOD 7 - Computer Architecture

Final Assignment - Architecture Simulator

Specification

The objective of this assignment is to create an architecture simulator that follows the ISA defined next. The simulator can be developed using **any programming language or script** students feel more comfortable with, as long as it is a *description language* (i.e., C, Java, Python, etc).

Students are expected to create a simulation environment that accepts as **input a file describing an assembly** program and, based on the description, the simulator performs the architectural steps to execute the program. The ISA is composed of **20 instructions**, four general purpose registers (***t0, t1, t2* and *t3***), a 4096 bytes **stack**, a shared **memory to store code and data**, and a **program counter** to fetch the current instruction in memory.

The ALU implements the following instructions, with the following behavior and limitations:

1. LDA <reg1> <reg2>/<var>/<const>
   Load register *reg1* with the contents of either the contents of *reg2,* or the memory *var* or a constant *const*. Memory regions loads (load into a variable, for instance) are NOT ALLOWED.

2. STR <var> <reg>/<const>
   Store in the memory position referred by *var* the value of register *reg* or a constant *const.* Register stores (store into register t0, for instance) are NOT ALLOWED.

3. PUSH <reg>/<var>/<const>
   Push to the top of the *stack* the contents of *reg* or *var* or a constant *const*

4. POP <reg>
   Pop from the top of the *stack* and store the value on *reg. S*toring in a memory region is NOT ALLOWED.

5. AND <reg1> <reg2>/<var>/<const>
   Performs a logical AND operation between *reg1* and a register reg2, a variable *var* or a constant *const,* and store the result on register *reg1.* Memory regions stores (store result into a variable, for instance) are NOT ALLOWED.

6. OR <reg1> <reg2>/<var>/<const>
   Performs a logical OR operation between *reg1* and a register reg2, a variable *var* or a constant *const,* and store the result on register *reg1.* Memory regions stores (store result into a variable, for instance) are NOT ALLOWED.

7. NOT <reg>
   Performs a logical NOT operation on register *reg* and store the result on register *reg.* Memory regions stores (store result into a variable, for instance) are NOT ALLOWED.

8. ADD <reg1> <reg2>/<var>/<const>
   Performs the addition operation of *reg1* and a register reg2, a variable *var* or a constant *const,* and store the result on register *reg1.* Memory regions stores (store result into a variable, for instance) are NOT ALLOWED.

9. SUB <reg1> <reg2>/<var>/<const>
   Performs the subtraction operation of *reg1* and a register reg2, a variable *var* or a constant *const,* and store the result on register *reg1.* The operation is given by second argument minus the first argument (i.e., reg2 – reg1). Memory regions stores (store result into a variable, for instance) are NOT ALLOWED.

10. DIV <reg1> <reg2>/<var>/<const>
    Performs the integer division operation of *reg1* and a register reg2, a variable *var* or a constant *const,* and store the result on register *reg1.* The operation is given by second argument divided by the first argument (i.e., reg2 / reg1). Memory regions stores (store result into a variable, for instance) are NOT ALLOWED.

11. MUL <reg1> <reg2>/<var>/<const>
    Performs the integer multiplication operation of *reg1* and a register reg2, a variable *var* or a constant *const,* and store the result on register *reg1.* Memory regions stores (store result into a variable, for instance) are NOT ALLOWED.

12. MOD <reg1> <reg2>/<var>/<const>
    Performs the integer modulo operation of *reg1* and a register reg2, a variable *var* or a constant *const,* and store the result on register *reg1.* The operation is given by second argument modulo the first argument (i.e., reg2 mod reg1). Memory regions stores (store result into a variable, for instance) are NOT ALLOWED.

13. INC <reg>
    Increments the value of a register *reg.* Memory increments (incrementing a variable, for instance) are NOT ALLOWED.

14. DEC <reg>
    Decrements the value of a register *reg.* Memory increments (decrementing a variable, for instance) are NOT ALLOWED.

15. BEQ <reg1>/<var1>/<const1> <reg2>/<var2>/<const2> <LABEL>
    Performs a comparison between two values, given by registers, variables or constants. Any combination is permitted. If they are equal, jump to the address defined by the label *LABEL*

16. BNE <reg1>/<var1>/<const1> <reg2>/<var2>/<const2> <LABEL>
    Performs a comparison between two values, given by registers, variables or constants. Any combination is permitted. If they are different, jump to the address defined by the label *LABEL*

17. BBG <reg1>/<var1>/<const1> <reg2>/<var2>/<const2> <LABEL>
    Performs a comparison between two values, given by registers, variables or constants. Any combination is permitted. If the first parameter is bigger than the second parameter, jump to the address defined by the label *LABEL*

18. BSM <reg1>/<var1>/<const1> <reg2>/<var2>/<const2> <LABEL>
    Performs a comparison between two values, given by registers, variables or constants. Any combination is permitted. If the first parameter is smaller than the second parameter, jump to the address defined by the label *LABEL*

19. JMP <LABEL>
    Jump to the address defined by the label *LABEL*

20. HLT

    End the program execution.

The assembly program follows the given structure:

Lines starting with **!** are commentaries and can be discarded. A line starting with **#data** indicates that this is the memory (variables) region. Variables are declared using two entries: <VAR_NAME> <INITIAL_VALUE>. Variables are listed right after the #data macro. A line starting with **#code** indicates that the program code is starting. From this point on, only code is allowed (i.e., no variable definitions).

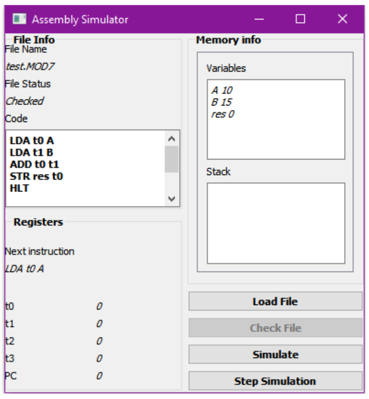
This is a simple example of a MOD7 assembly file that loads into registers the values of two variables, adds them and store the result into a third variable, before halting its execution.

```
!FELIPE GM - MOD7 COMPUTER ARCHITECTURE CONCORDIA CONTINUING EDUCATION
!FINAL ASSIGNMENT FOR THE FALL CLASS 2022
!EXAMPLE OF A SIMPLE ASSEMBLY PROGRAM USING A FEW INSTRUCTIONS. ANY LINE STARTING WITH '!' IS A COMMENT

!THE FIRST PART OF THE FILE IS THE DATA DECLARATION, STARTING WITH #DATA. ANY VARIABLE HAS A UNIQUE TYPE OF SIZE
32BITS AND SHOULD BE INITIALIZED TO A VALUE
#DATA
A 10
B 15
RES 0

!NEXT, WE START THE CODE WITH THE #CODE MACRO
#CODE
LDA T0 A
LDA T1 B
ADD T0 T1
STR RES T0
HLT
```

The above program is expected to execute as it follows:

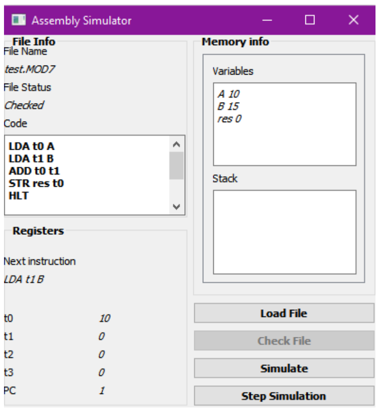


```
File Name
test.MOD7
File Status
Checked
Code
LDA t0 A
LDA t1 B
ADD t0 t1
STR res t0
HLT

Variables
A 10
B 15
res 0
Stack

Registers
Next instruction
LDA t0 A

t0   0
t1   0
t2   0
t3   0
PC   0
```

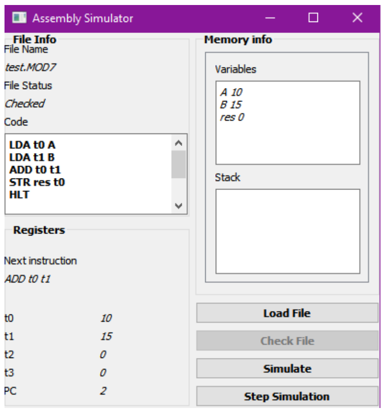


```
File Name
test.MOD7
File Status
Checked
Code
LDA t0 A
LDA t1 B
ADD t0 t1
STR res t0
HLT

Variables
A 10
B 15
res 0
Stack

Registers
Next instruction
LDA t1 B

t0   10
t1   0
t2   0
t3   0
PC   1
```

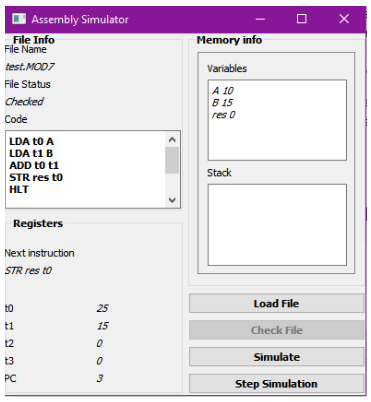


```
File Name
test.MOD7
File Status
Checked
Code
LDA t0 A
LDA t1 B
ADD t0 t1
STR res t0
HLT

Variables
A 10
B 15
res 0
Stack

Registers
Next instruction
ADD t0 t1

t0   10
t1   15
t2   0
t3   0
PC   2
```

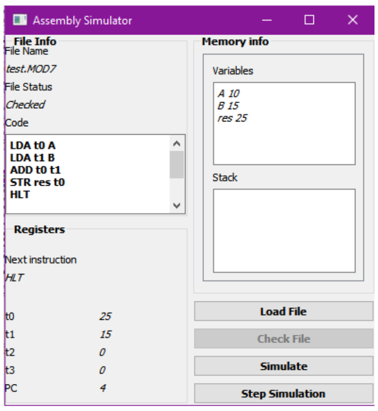


```
File Name
test.MOD7
File Status
Checked
Code
LDA t0 A
LDA t1 B
ADD t0 t1
STR res t0
HLT

Variables
A 10
B 15
res 0
Stack

Registers
Next instruction
STR res t0

t0   25
t1   15
t2   0
t3   0
PC   3
```

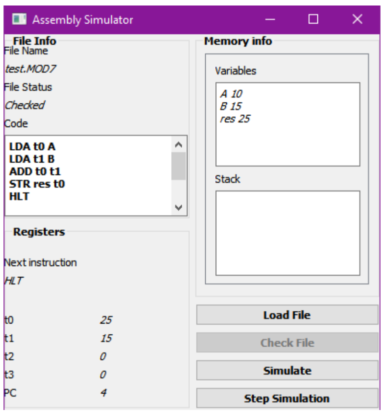


```
File Name
test.MOD7
File Status
Checked
Code
LDA t0 A
LDA t1 B
ADD t0 t1
STR res t0
HLT

Variables
A 10
B 15
res 25
Stack

Registers
Next instruction
HLT

t0   25
t1   15
t2   0
t3   0
PC   4
```



```
File Name
test.MOD7
File Status
Checked
Code
LDA t0 A
LDA t1 B
ADD t0 t1
STR res t0
HLT

Variables
A 10
B 15
res 25
Stack

Registers
Next instruction
HLT

t0   25
t1   15
t2   0
t3   0
PC   4
```

Expected work

For this assignment, students can have groups of two, three or four students each. As the number of group members rises though, more things are expected for the students to accomplish. This way, groups of two are expected to complete the first part of the assignment, groups of three the first and second parts and the groups of four are expected to complete parts one, two and three.

Part 1

Students should:

1. create the environment to parse the assembly file
2. create the memory structure to store the parsed file as well as variables
3. create the stack
4. create the ALU operations (20 in total, as listed before)
5. manipulate the memory to load and store values in the variables addresses (direct access is permitted)
6. enable step-by-step simulation as well as full program simulation
7. show up-to-date information for each register, variable and stack of the system, at any time

Part 2

Students should:

1. Add two new instructions to the list of the ISA (22 in total)
   a. SRL <reg> <const>
      This operation takes the value in *reg* and performs a logical shift left of the number of bits defined by the constant *const.* For instance, the value *0001* left shifted 1 time becomes *0010*.

   b. SRR <reg> <const>
      This operation takes the value in *reg* and performs a logical shift right of the number of bits defined by the constant *const.* For instance, the value *1000* right shifted 1 time becomes *0100*.

2. Add the possibility to perform LOAD and STORE operations in indirect addresses. For instance, LDA t0, var loads into register t0 the value stored in the memory region allocated for variable *var.* What is asked is to allow operations such as LDA t0, var+2. In this case, register *t0* is to loaded with the value of the memory address given by the address of the variable *var* + 2 positions. For example, if *var* is stored at address 0x15, *t0* is to be loaded with the contents of address 0x17.

Part 3

Students should:

1. Add a syntax checking tool to the file parser. The tool should check for the correctness of each assembly line and operation types. For instance, using LDA <var> <var> should trigger an error, as LDA is not allowed to load variables.
2. Enable the declaration of arrays in the #data section. The arrays should be directly accessed in the #code area (i.e., LDA t0, var[2]).

Submission

The following deliverables are expected:

- A written report in PDF format
- The project containing all files needed to execute the solution, as well as any instruction to compile and run it.

This assignment should be submitted through Moodle, in the assigned space.

Grading

The evaluation is done ***individually***.

Questions?

Use any channel to contact the professor before the submission date.

Due date

8th of May 2023