

# Lab 1

## Warm-up : Python and the target machine : SARUMAN

### Objective

- Start with Python.
- Be familiar with the SARUMAN instruction set.<sup>1</sup>
- Understand how it executes on the SARUMAN processor with the help of a simulator.
- Write simple programs, assemble, execute.

Todo in this lab:

- Play and learn Python!
- Play and learn the SARUMAN ISA.
- Finish at home, nothing will be evaluated in this lab.

### 1.1 Quick intro to Python - 1h max

This part is strongly inspired by the Project 1 of ENSL (L3).

Please use a correct text editor ! We don't really care if it is SublimeText, Emacs, Atom or Vim, but please use a text editor made for programming.

<https://www.python.org/Official/tutorial/> An amazing interactive one <http://www.learnpython.org/en/Welcome>

#### 1.1.1 Inside the interpreter

And now, let's get to the heart of the matter.

##### EXERCISE #1 ▶ **Launch!**

Launch the Python interpreter (python3, in the terminal). Which version is it ? Use a version of Python not older than 3.5. Quit the interpreter with CTRL-D or quit().

##### EXERCISE #2 ▶

Launch the interpreter in interactive mode and **use it as a calculator** to solve these equations:

$$2 + 2 = x$$

$$11 = 3k + r$$

where  $k$  and  $r$  positive or null integers

$$27^{98} \bmod 97 = y$$

##### EXERCISE #3 ▶ **Strings**

Try the following code:

```
x = 'na'
'Ba' + 2 * x
```

Then write "j'aime les bons bonbons" with the same technique.

<sup>1</sup>SARUMAN stands for Stupid Address Registers, Useless Memory, Assembler Nonsense

**Lists****EXERCISE #4 ► Lists**

Create a list `li` of integers containing various éléments. Replace one of the elements with a new value. At last, use `+` or `+=` to add elements at the end of the list.

**EXERCISE #5 ► Sorts**

Sort a list using function `sorted`. What is the complexity in the worst case? In the best case? Use function `len()`; same questions.

**Print****EXERCISE #6 ► Formatting**

Give 3 different ways of building the following character string:

"2.21 Gigawatts !! 2.21 Gigawatts !! My godness !" using one variable `x = 2.21`, and another variable that uses `str()`, then the operator `%`, then the method `.format()`.

**1.1.2 Tiny programs**

Now, write your programs in `.py` files (with an editor). If you get encoding issue, add this at the beginning, but it shouldn't be needed with Python 3:

```
# -*- coding: utf-8 -*-
```

**EXERCISE #7 ► Hello**

Edit a file named `hello.py` with the following content:

```
print("Hello World")
```

Save, execute with: `python3 hello.py`.

**EXERCISE #8 ► If then else**

Write a program that initializes an int value to a number given by the user (use `input()`) and prints a different message according to its parity (odd/even).

**EXERCISE #9 ► While**

Write a program that declares two integer values `a` and `b`, then computes and prints their pgcd.

**EXERCISE #10 ► Imperative For**

Using the construction `for i in ...`, write a program that sums all even `i` from 2 to 42 (inclusive).

**EXERCISE #11 ► For expression / Lists**

- Write a program that declares and initialises a list, and computes the sum of all its elements.
- Write a 1-line code that, from a list `l`, returns a list whose elements are the squares of the elements in `l`.
- Write a 1-line code that, from a list `l`, returns a liste containing the even elements of `l.l`.

**EXERCISE #12 ► Dicts**

1. What are the types of `{}`, `{'a'}`, `{'a', 'b'}` and `{'a': 'b'}`?
2. What is the following code doing (where `t` is a dictionary):

```
while id in t:
    id = t[id]
print(id)
```

What is the problem?

3. Write a code doing the same operation but without the same drawback (*i.e.*: if needed, it doesn't print anything)

**EXERCISE #13 ► Functions**

1. Declare a function `fact` that computes the factorial of a number.
2. What returns `help(fact)`? If it is not done, document your function.

**1.2 The SARUMAN processor, instruction set, simulator****EXERCISE #14 ► Lab preparation**

Clone the github repository for this year's labs:

```
git clone https://github.com/lauregonnord/mif08-labs18.git
```

Then:

- In the `saruman/emu/` directory, type `make` to compile the emulator. The assembler is `saruman/asm.py`. Some more documentation can be found in the SARUMAN ISA on the course webpage and in Appendix A.

<http://laure.gonnord.org/pro/teaching/compilM1.html>

- You may have issues to compile the graphical version of the simulator, which is not mandatory at all. This can be solved by compiling it with `make -B NO_SDL=1`
- On your personal machines you might have to install the `libncurses5-dev` package.
- The files you need for this lab are in TP01.

The assembly language for this year is a toy language called SARUMAN. We already played a bit with it in the exercise session.

**1.2.1 Assembling, disassembling****EXERCISE #15 ► Hand assembling, simulation of the hex code**

Assemble by hand the instructions :

begin:

```
and2i r0 0
cmpi r0 2
jumpif lt begin
print signed r0
```

You will need the set of instructions of the SARUMAN machine and their associated opcode. All the info is in the ISA documentation (and in the simulator README file for graphical instructions). Save your (hex) encoding in a file `dummy.bin`, and launch the SARUMAN simulator on it:

```
./<path/to/simulateur>/emu --text dummy.bin
```

The `--text` option is needed to read pseudo-binary files where 0 and 1's are actually written as text (ascii characters).

You may add the `--debug` option to run the program step-by-step in a debugger (use the `s` command).

Carefully follow each step of the execution.

From now on, we are going to write programs using an easier approach. We are going to write instructions using the SARUMAN assembly.

## 1.2.2 SARUMAN Simulator

### EXERCISE #16 ► Execution and debugging

1. First test assembling and simulation on the file `tp1-simple.s`:  
`$python3 <path/to/assembleur/>asm.py -b tp1-simple.s`  
`$</path/to/emu/simulateur/>emu ./tp1-simple.bin`
2. Check if your guess in the exercise sheet (Chapter 1, ex#3) was right by executing `ex5.bin`.  
 The simulator comes with a built-in debugger (option “-d” or “-debug”). The interface is divided into multiple parts, and should render like this:

```

+-----+-----+-----+
|           |           |           |
| Dissassembled code | Register Info | Memory view |
|           |           |           |
+-----+-----+-----+
|           |           |           |
| Interactive shell  |           |           |
|           |           |           |
+-----+-----+-----+

```

If it does not look like this, your terminal might be too small, or something might be broken.

3. Use the debugger to follow the execution of `tp1-simple.bin`.
4. Guess the output of the program `noha1.t.s`, then use the simulator to check if your assumption was right.  
 Use the built-in debugger to follow the execution of the program, and find out or confirm happened.

**Remark 1:** When displaying a binary file on a terminal (e.g. using the `cat` command), many characters do not print correctly. Check the `cat` manual (especially options “-v” and “-A”) to find out how to fix this. Be sure your text editor is not doing some funny stuff too with special characters.

**Remark 2:** You can use the “`xxd`” program to display files content in binary or hexadecimal, e.g “`xxd -b tp1-simble.bin`” (binary)

### EXERCISE #17 ► Algo in SARUMAN assembly

Write a program in SARUMAN assembly that computes the min and max of two integers, and stores the result in a precise location of the memory that has the label `min`. Try with different values.

### EXERCISE #18 ► (Advanced) Algo in SARUMAN assembly

Write and execute the following programs in assembly :

- Count the number of non-nul bits of a given integer.
- Draw squares and triangles of stars (character ‘\*’) of size  $n$ ,  $n$  being stored somewhere in memory.

Examples:

`n=3 square:`

```
***
```

```
***
```

```
***
```

`n=3 triangle:`

```
*
```

```
* *
```

```
* * *
```

## 1.2.3 Finished?

If you’re done with the lab, do the python tutorial at the following address:

<https://docs.python.org/fr/3.5/tutorial/>

# Appendix A

## SARUMAN Assembly Documentation (ISA)

### About

- ISA: Florent de Dinechin for ASR1, ENSL, 2017-18.
- Simulator and Assembler code: Maxime Darrin, Alain Delaët-Tixueil, Antonin Dudermel, Sébastien Michelland, Alban Reynaud, L3 students at ENSL, 2017-18.
- Document: Remy Grüblatt, Laure Gonnord, Sébastien Michelland, and Matthieu Moy, for CAP and MIF08.

This is a simplified version of the machine, which is (hopefully) conform to the chosen simulator.

### A.1 Installing the simulator and getting started

To get the SARUMAN assembler and simulator, follow instructions of the first lab (git pull on the course lab repository).

### A.2 The SARUMAN architecture

Among others, the SARUMAN architecture has two particular features:

- The number of bits used to encode instructions is non constant. But for compilation, we do not care!
- Read and write instructions use special registers.

Here is an example of SARUMAN assembly code for 2018:

---

```
leti r0 17 ; initialisation of a register to 17
loop:
sub2i r0 1 ; subtraction of an immediate
jumpif nz loop ; equivalent to jump xx
```

---

**Memory, Registers** The memory is addressed by bits (and not words), from address 0.

The SARUMAN has 8 registers from r0 to r7. Only r7<sup>1</sup> is reserved for the routine return address. There are specific registers ("counters") for manipulating memory, namely a1 and a0. Finally, we have special registers sp (*Stack Counter*) and pc (*Program Counter*). Accesses to registers are direct, and Section A.2 explains how to access memory.

**Shifts** The directions for the shift are either "left" or "right".

**Flags** Each instruction may update carry flags (last column of A.1). Flags represent informations about the last operation that modified them:

- **z**: The result of the previous operation was a zero.
- **c**: A carry happened during the previous operation.
- **v**: An overflow happened during the previous operation.
- **n**: The result of the previous operation is strictly negative (< 0).

Check the file `mi f08-labs18/saruman/doc/emu_flag_management.md` for details.

---

<sup>1</sup>Registers are in lower case.

Table A.1: SARUMAN instructions. For constants, padding is done with zeros (z) or sign extension (s).

opcode	mnemonic	operands	description	ext.	Flags update
0000	add2	<i>reg reg</i>	addition		zcvn
0001	add2i	<i>reg const</i>	add immediate constant	z	zcvn
0010	sub2	<i>reg reg</i>	subtraction		zcvn
0011	sub2i	<i>reg const</i>	subtract immediate constant	z	zcvn
0100	cmp	<i>reg reg</i>	comparison		zcvn
0101	cmpi	<i>reg const</i>	comparison with immediate constant	s	zcvn
0110	let	<i>reg reg</i>	register copy		
0111	leti	<i>reg const</i>	fill register with constant	s	
1000	shift	<i>dir reg shiftval</i>	logical shift		zcn
10010	readze	<i>ctr size reg</i>	read <i>size</i> memory bits (zero-extended) to <i>reg</i>		
10011	readse	<i>ctr size reg</i>	read <i>size</i> memory bits (sign-extended) to <i>reg</i>		
1010	jump	<i>addr</i>	relative jump		
1011	jumpif	<i>cond addr</i>	conditional relative jump		
110000	or2	<i>reg reg</i>	logical bitwise or		zcn
110001	or2i	<i>reg const</i>	logical bitwise or	z	zcn
110010	and2	<i>reg reg</i>	logical bitwise and		zcn
110011	and2i	<i>reg const</i>	logical bitwise and	z	zcn
110100	write	<i>ctr size reg</i>	write the lower <i>size</i> bits of <i>reg</i> to mem		
110101	call	<i>addr</i>	sub-routine call	s	
110110	setctr	<i>ctr reg</i>	set one of the four counters to the content of <i>reg</i>		
110111	getctr	<i>ctr reg</i>	copy the current value of a counter to <i>reg</i>		
1110000	push	<i>reg</i>	push value of register on stack		
1110001	return		return from subroutine		
1110010	add3	<i>reg reg reg</i>			zcvn
1110011	add3i	<i>reg reg const</i>		z	zcvn
1110100	sub3	<i>reg reg reg</i>			zcvn
1110101	sub3i	<i>reg reg const</i>		z	zcvn
1110110	and3	<i>reg reg reg</i>			zcn
1110111	and3i	<i>reg reg const</i>		z	zcn
1111000	or3	<i>reg reg reg</i>			zcn
1111001	or3i	<i>reg reg const</i>		z	zcn
1111010	xor3	<i>reg reg reg</i>			zcn
1111011	xor3i	<i>reg reg const</i>		z	zcn
1111100	asr3	<i>reg reg shiftval</i>			zcn
1111101	sleep		sleep		
1111100	rand		rand		
1111101	lea	<i>reg addr</i>	load effective address <i>addr</i>		
1111110	print	<i>type reg</i>	print		
1111111	printi	<i>type const</i>	print		

**Constants: let and leti** These expressions provide ways to initialize or copy registers.

The constants are encoded according to A.2 (encoding of ALU constants). For the `leti` instruction, padding is done with sign extension. Thus:

```
1 leti r0 -17
```

stores the constant -17 in register r0, and the encoding of the instruction is:

```
0111 000 1011101111
```

Register copy is done with:

```
let r0 r1
```

**Arithmetical and logical instructions** Arithmetical and logical instructions have 2 or 3 operands:

```
add3i r1 r0 3 ; r1 ← r0+3
```

```
add2i r1 15 ; r1 ← r1+15
```

```
add3 r1 r2 r3 ; r1 ← r2+r3
```

```
4 add2 r1 r2 ; r1 ← r1+r2
```

Table A.2: Constant encoding

<i>addr</i> : prefix-free encoding for addresses and moves	
0 + 8 bits	value of move on 8 bits
10 + 16 bits	same on 16 bits
110 + 32 bits	same on 32 bits
111 + 64 bits	same on 64 bits
<i>shiftval</i> : prefix-free encoding of shift constants	
0 + 6 bits	constant between 0 and 63
1	constant value 1
<i>const</i> : prefix-free encoding of ALU constants	
0 + 1 bit	constant 0 ou 1
10 + 8 bits	byte
110 + 32 bits	
111 + 64 bits	
<i>size</i> : prefix-free encoding of memory sizes	
00	1 bit
01	4 bits
100	8 bits
101	16 bits
110	32 bits
111	64 bits

The first operand is always the destination register, and the two remaining operands are sources, registers or constants. If a constant is used then its value is encoded in the instruction following the encoding depicted in Table A.2. For instance:

```
1  add2i r1 15      ; r1 <- r1+15
```

is encoded as:

```
0001 001 10 00001111 ;
add2i, register 1, 1 byte constant (*addr* prefix code), value 15 and padding with 0
```

Be careful, add only uses positive constants:

```
add3i r1 r0 -12
```

Throw the following error:

```
couldn't read UCONSTANT : The value is not in the right range
```

**Branching (jump jumpif)** Let *a* be the address of the instruction following the `jump` or `call` instruction, and *c* the integer encoded in a constant of type *addr* (see Table A.2), and signed.

The `jump` instruction executes  $pc \leftarrow a + c$ .

The `jumpif` instruction does the same, but only if the condition is true (see Section A.2).

The `call` instruction stores *R7* in PC and jumps to the called address.

The `return` instruction does  $pc \leftarrow R7$ .

In:

---

loop:

```
sub2i r0 1      ; subtraction of an immediate
jumpif nz loop ; equivalent to jump -25
```

---

is assembled into

```
0011 000 01          ; 9 bits
1011 001 011100111  ; 16 bits
jump, nz, 0 (mv on 8 bits), -25 bits jump
```

Table A.3: Tests

			mnemonic	description (after <code>cmp op1 op2</code> )
0	0	0	<code>eq, z</code>	equal, $op1 = op2$
0	0	1	<code>neq, nz</code>	not equal, $op1 \neq op2$
0	1	0	<code>sgt</code>	signed greater than, $op1 > op2$ , two's complement
0	1	1	<code>slt</code>	signed smaller than, $op1 < op2$ , two's complement
1	0	0	<code>sge</code>	$op1 \geq op2$ , signed
1	0	1	<code>ge, nc</code>	$op1 \geq op2$ , unsigned
1	1	0	<code>lt, c</code>	$op1 < op2$ , unsigned
1	1	1	<code>sle</code>	$op1 \leq op2$ , signed

Table A.4: Counters (special registers).

encoding	mnemonic	description
00	<code>pc</code>	program counter
01	<code>sp</code>	stack pointer
10	<code>a0</code>	generic address counter
11	<code>a1</code>	generic address counter

**Tests** Operands 1 and 2 are encoded like in the ALU instructions. In particular the second operand can be an immediate constant. The condition is encoded thanks to Table A.3.

In this class, we will use only the signed version of comparisons (`sgt/slt/sle/sge`, and `eq/neq/z/nz` which work for both signed and unsigned). Not all unsigned comparisons are available, and they are misleading: don't use them here.

**Memory accesses** Special registers `a0`, `a1` are used to access memory.

The instructions `readze`, `readse` and `write` read or write the specified number of bits and also increment the associated (address) registers:

```
readze a0 4 r1
```

reads 4 bits of memory content from the address stored in `a0` and store them in `r1` (with a zero padding). In addition, `a0` is incremented by 4.

```
write a1 2 r1
```

writes the lower 2 bits of register `r1`.

We can emulate the classical read operation in memory from an address stored in a register  $r_2 \leftarrow Mem[r_1]$ :

```
setctr a0 r1
```

```
readse a0 xxx r2 ; xxx the number of bits to read
```

The instruction `lea r3 label` loads the address corresponding to label onto `r3`. For instance, the following program:

```
lea r0 foo
```

```
3 foo:
```

```
  .const 5 #10101
```

loads the address of the constant. The `#` prefix is used to introduce a binary constant (10101, i.e. 21), and works only for the `.const` directive. It is assembled into:

```
11111101 000 000000000
10101
```

The SARUMAN emulator's memory layout is documented in the `cap-labs18/saruman/doc/emu_memory_layout.md` file.



**Print** Two examples of use of the native print instruction:

```

1  let r0 126
   print char r0 ; "~"
   print char '\n' ; newline
   print signed r0 ; "126"
   print unsigned r0 ; "0x7e"
6  print unsigned '0' ; "0x30"

```

You can also print a string at a given label with:

```

lea r0 str
print string r0 ; "Hello, World!"

4 str:
.string "Hello, World!"

```

**Assembly directives** A bit more of syntax:

- The assembly begins at address 0.
- Labels can be used for jumps.
- The keyword `.const n xxxx` reserves a memory cell initialized to the  $n$  bits constant `xxxx`.
- The keyword `.string "Hello"` reserves 6 memory cells and store the ascii numbers corresponding to all the characters of the message (ending it with a Null character).
- Hexadecimal constants are prefixed by `0x`, for instance `0xff` is decimal 255.
- Comments begin with a semicolon;

The assembly implements a stack in memory, from an address stored in the special register `sp`. We will use it in Lab5.

**Stopping execution** When instructions terminate, the emulator halts the execution. But as it has no way of differentiating instructions from data (like strings or constants), the emulator provides a way to stop execution by detecting infinite self loops, such as this one:

```

halt:
  jump halt

```

### A.3 Help to encode constants

hex to binary	a	b	c	d	e	f
	1010	1011	1100	1101	1110	1111

**2's complement** Let us code  $n = (-3)_{10}$  in 2's complement on 6 bits, with the recipe: "code  $-n$  in base 2, then negate bitwise, then add one". First, 3 is encoded as `000011` on 6 bits. Its negation is `111100`, thus  $(-3)_{10} = 111101_2$ .