Lab 4
Syntax-Directed Code Generation

Objective

During the previous lab, you have written your own evaluator of the Mu language. In this lab the objective is to generate *valid* SARUMAN codes from Mu programs:

- Generate 3-address code for the Mu language.
- Generate executable "dummy" SARUMAN from programs in Mu via two simple allocation algorithms.
- **Please follow instructions and COMMENT YOUR CODE!**

Student files are in the Git repository.

You may have to install some additional Python libs:

```
pip3 install networkx graphviz --user
```

And on your personal machines:

```
apt-get install graphviz-dev
```

4.1 Preliminaries

This section must be **carefully** read.

Important remark From now on, we add some restrictions to the Mu language:

- Values (variables, argument of `log`) are of type (signed) `int` or `bool` only (no float, no string, **no char**). Thus all values can be stored in regular registers or in one cell (16 bits) in memory. You can let your program crash if another type of variable is provided.
- The `log` statement only supports printing (the content of) a register, or an integer constant.

Note that real compilers would perform the code generation from a decorated AST (with type annotations attached to nodes). For simplicity, we will work on the non-decorated AST: our language is simple enough to generate code without decorations.

Structure of the compiler's code

- In `APISaruman.py` we provide you with utility functions to encode 3-address SARUMAN instructions. Instruction classes are in `Instruction3A.py` and `Operands.py`. An Instruction is either a Comment, a Label, or a `Instru3A`; it has arguments which can be immediate numbers (of type `Immediate`), temporaries (of type `Temporary`), regular registers (`Register`), offsets in memory (`Offset`).
- A SARUMAN program contains a list of instructions, and also a temporary pool (temporary variables).
- In Section 4.2, you will use an instance of the `SARUMANProg` class in order to construct a list of such instructions via calls to `addInstructionXXX` methods. A call to the `printCode` method will dump this code into a text file.
- File `Allocations.py` is responsible for the allocation part. From a `SARUMANProg` with temporaries (instructions formed with temporaries), producing an actual SARUMAN program (instructions with regular registers or memory accesses) is done by:
  - First, compute an allocation for each temporary (in the current `SARUMANProg` instance). In Section 4.3, we provide you with `SARUMANProg.naive_alloc()` which computes such a (naive) allocation, you will have to design your own allocation function in Section 4.4.

  - For each instruction of the program, if the instruction contains a read or write access to a temporary, replace operands with the corresponding actual registers/memory location (and possibly add some instructions before and after). This is done by the use of the `SARUMANProg.iter_instructions` iterator on instructions and `Allocations.replace_reg` methods. In Section 4.4 you will have to write such a "replacement" function.

- The file `Main.py` launches the chain: production of 3-address code with temporaries, allocation, replacement, print.
- The script `test_codegen.py` will help you to test your code. We will use it in Section 4.3.
- A `README.md` file to be completed progressively during the lab.

EXERCISE #1 ▶ SARUMAN **Simulator - test**

Re-test the command-line version of the saruman simulator:

```
$ cd saruman
$ ./asm.py -b prog/hello.s
$ ./emu/emu prog/hello.bin
Hello, world!
```

4.1.1 Conventions used in the assembly code

- All data items are stored on 16 bits. Integers are short integers, and we don't use the full power of SARUMAN which would be able to address booleans at the bit level.
- Registers `r0` and `r1` are reserved for temporary computations (e.g. to compute an address before a `write` or a `readse`, or to store a value between a memory access and an arithmetic operation).
- The address counters `a0` an `a1` are used for read/write accesses, but may be overwritten at any time (in practice only `a0` is useful).
- The stack pointer is `sp`. **The stack is growing with increasing addresses**. Thus data in the stack is accessed by adding a **positive offset** to this register.

4.2 First step: three-address code generation

In this section you have to implement the course rules (Figures 4.2 and 4.3) in order to produce SARUMAN code with temporaries.

Here is an example of the expected output of this part. From the following Mu program:

```
var a,n:int;
n=1;
a=7;
while (n<a) {
 n= n+1;
}
log(n);
```

the following code is supposed to be generated:

```
;;Automatically generated TARGET code, MIF08 & CAP 2018
;;non executable 3-Address instructions version
        ;; (stat (assignment n = (expr (atom 1)) ;))
        leti temp_2 1
        let temp_0 temp_2
        ;; (stat (assignment a = (expr (atom 7)) ;))
```

```
        leti  temp_3 7
        let  temp_1 temp_3
        ;;  (stat  (while_stat while (expr (atom ( (expr (expr (atom n)) < (expr (atom a))) )))  (stat_block {  (
    block (stat  (assignment n = (expr (expr (atom n)) + (expr (atom 1))) ;))  (stat  (log log  (expr (atom ( (expr (
    atom n)) )))  ;) ))  }) ))
lbl_l_while_begin_0:
        leti  temp_4 0
        ;;  cond_jump lbl_end_relational_1 temp_0 sge temp_1
        cmp temp_0 temp_1
        jumpif  sge lbl_end_relational_1
        ;;  end cond_jump lbl_end_relational_1 temp_0 sge temp_1
        leti  temp_4 1
lbl_end_relational_1:
        ;;  cond_jump lbl_l_while_end_0 temp_4 neq 1
        cmp temp_4 1
        jumpif  neq lbl_l_while_end_0
        ;;  end cond_jump lbl_l_while_end_0 temp_4 neq 1
        ;;  (stat  (assignment n = (expr (expr (atom n)) + (expr (atom 1))) ;))
        leti  temp_5 1
        add3 temp_6 temp_0 temp_5
        let  temp_0 temp_6
        ;;  (stat  (log log  (expr (atom ( (expr (atom n)) )))  ;) )
        print signed temp_0
        print char '\n'
        jump lbl_l_while_begin_0
lbl_l_while_end_0:
        ;;  (stat  (log log  (expr (atom ( (expr (atom n)) )))  ;) )
        print signed temp_0
        print char '\n'


;;postlude
end:
        jump end
```

EXERCISE #2 ▶ 3-address code generation

In the archive, we provide you a main and an incomplete `MuCodeGen3AVisitor.py`. To test it, type

```
make TESTFILE=tests/step1/test01.mu
```

and observe the generated code in `<samepath>/test01.s`[1]. You now have to implement the 3-address code generation rules seen in the course. Code and test incrementally [2]:

- log(n) was given in 2018-2019(you may want to extend it for string arguments later, but it is not as simple as it seems, so keep it for extensions) (we recall that there is a native `print` instruction in the SARUMAN assembly).
- numerical expressions without variables (constants are expected to hold on 16 bits).
- then (numerical) assignments and expressions with variables; MultiplicativeExpr is bonus, implement it only if after everything else is working.

At this step, the code generation is not finished, but we will do some allocation to be able to test properly. All examples in `tests/step1` directory should generate code without any error at this point:

```
for i in tests/step1/*.mu; do echo "file="$i; python3 Main.py --reg-alloc=none $i > /dev/null; done
```

[1] We generated SARUMAN comments with Mu statements for debug.

[2] Using files in the `TP04/tests/*` directories. All the test files you use will have to be in your archive.

4.3 Testing with the trivial allocator (and real SARUMAN instructions), then end of 3@ code generation

The former code is not executable since it uses temporaries. We provide you with an allocation method which allocates temporaries in registers as long as possible, and fails if there is no available registers. The process takes as input the former 3-address code and transforms each instruction according to the allocation function.

EXERCISE #3 ▶ **Testing the trivial allocator**
Open, read, understand the `prog.naive_alloc()` implementation in `APISaruman` and `Allocations.py` and how it is used to perform the actual SARUMAN code generation. Then, intensively test your former code generation with this allocator [3]:

1. Have a look at the `test_codegen.py` script: comment or uncomment files to test, and what to test.
2. Test with:

   ```
   python3 test_codegen.py
   ```

   This script tests all files in the `test/*` directories:

   - if the pragma `# EXPECTED` is present in the file, it compares the actual output after assembling and simulating with the list of expected values. For instance:

     ```
     var x,y:int;
     x = 42;
     log(x);
     y = x + 8;
     log(y);
     # EXPECTED
     # 42
     # 50
     ```

     is a great test case to test assignments.
   - If the `AllocationError` exception is raised by the naive allocator, the test is skipped.
   - If the compilation succeeded, it compares the actual output after assembling and simulating to the output given by your evaluator of the Mu Language (Lab 3). **If your evaluator is buggy, you can decide either to correct your bugs or to comment appropriate lines in the Python script.**
   - For debugging, you can obviously launch your compiler manually with e.g.

     ```
     python3 Main.py --reg-alloc naive --stdout tests/step1/test00.mu
     ```

     Run `python3 Main.py -help` or see `Main.py` for more options.

At this step, the tests should be OK or SKIPPED for all files given in directory `tests/step1/`:

```
make tests
[...]
============================== xx passed, xx skipped in xx seconds ========
```

"skipped" here means that we cannot compare the output to the ideal output since some of our 3 adress-codes cannot be allocated with registers only. That's life !

Now that we have a way to test our code generation for tiny Mu codes, we can come back to it.

EXERCISE #4 ▶ **End of 3-address code generation for Mu**
Implement the 3-address code generation rules:

- for boolean expressions and numerical comparison: compute 1 (true) or 0 (false) in the destination register;

[3] Be careful, this allocator crashes if there is more than 8 temporaries !

- while loops;
- if then else. **Be careful with nested ifs and their labels!**.

At this point all the tests should be ok for all files in directory `tests/step2/` (You should modify the test script pathes). However these tests are not sufficient, you should add some other ones (in the directory `tests/mine/`).

About `if` and `while` For tests (and boolean expressions), make sure you generate "conditional jumps" with:

```
self._prog.addInstructionCondJUMP(label, op1, cond, op2)
```

where `op1` (resp `op2`) is the left operand (resp right operand), ie a register or a value of the boolean condition (`Condition('eq')` for equality, for instance) [4], and `label` is a label to jump to if the condition evaluates to true. Later on (while printing), this instruction will expand itself to a regular list of SARUMAN instructions.

About nested if-then-else (a bit more difficult) There is an issue with nested ifs. Indeed, how can we remember where to jump after one CondBlock (in `visitCondBlock(self, ctx)`)? We propose to use a label stack called `self.ctx_stack`: each time we enter `visitIfStat`, we push the end label. This label is used in all `visitCondBlock` (at some point you have to insert a jump instruction to the `cond_if` label). At the end of the `visitIfStat` function this label is popped out.

4.4 SARUMAN code with "all-stack" allocation of temporaries

As the number of registers is only 8, the naive allocator cannot deal with more than 8 temporaries (or even 6 considering that we reserved r0 and r1): we have to find a way to store the results elsewhere. In this particular lab, we will use the following solution:

- the generated code will use memory locations in the stack, and will not use registers r2 to r7 at all (r2 to r7 will be used to store some temporaries in the clever version of the allocator);
- but all values that are propagated from one rule to another (sub-expressions, ...) must be stored in the stack, whose address will be stored in sp (as defined in SARUMANProg.printCode).
- r_0 will be used to compute the actual addresses from the base stack register sp.
- r_1 will be used to compute the value to store or as a destination register for the value to read.
- a_0 and a_1 will be used to compute actual addresses.

Figure 4.1 depicts the stack implementation for the SARUMAN machine.

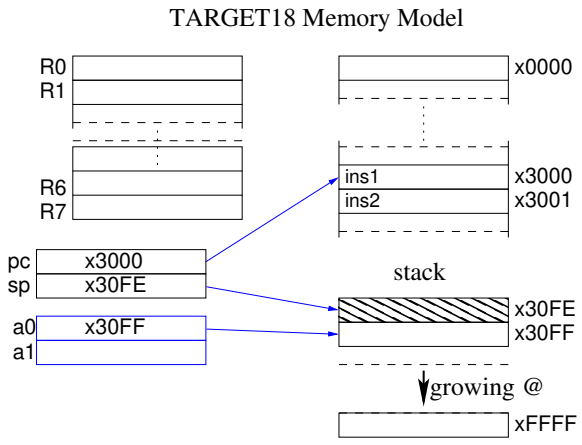

Figure 4.1: Memory model for SARUMAN

Following the convention that sp always stores the "begining of stack address", pushing[5] the content of r_1 in the stack will be done following the steps:

[4] We suggest to use `grep` and find this class definition and this method somewhere in the code we provide.
[5] Please do not use the assembly macros push and pop that do not follow our conventions!

- compute a new offset (call to the `new_offset` method of the class `SARUMANProg`).
- generate the following instructions:

```
GETCTR  sp r0
ADD     r0 r0 <valueofoffset*16>
SETCTR  a0 r0
WRITE   a0 16 r1
```

`r0` is used to compute the address, `r1` holds the value to write, and is the register to use instead of the temporary in the final instruction.

Be careful with the size of the offsetvalue!

EXERCISE #5 ▶ **Manual translation**
Complete the expected output for the following two statements (13/15 lines of SARUMAN code):

```
var x,y:int;
x=4;
y=12+x
```

Listing 4.1: 'all in mem alloc for test00b.mu'

```
1  ;;Automatically generated TARGET code, MIF08 & CAP 2018
   ;;all-in-memory allocation version
         ;; (stat (assignment x = (expr (atom 4)) ;))
         ;; leti temp_2 4
         leti r1 4
6        getctr sp r0
         add r0 r0 64
         setctr a0 r0
         write a0 16 r1
         ;; end leti temp_2 4
11       ;; let temp_1 temp_2
         getctr sp r0
         add r0 r0 64
         setctr a0 r0
         readse a0 16 r0
16       let r1 r0
         getctr sp r0
         add r0 r0 16
         setctr a0 r0
         write a0 16 r1
21       ;; end let temp_1 temp_2
         ;; (stat (assignment y = (expr (expr (atom 12)) + (expr (atom x))) ;))
         ;; leti temp_3 12
   ;;  5 LINES HERE
   ;;  <TODO>
26       ;; end leti temp_3 12
         ;; add3 temp_4 temp_3 temp_1
   ;;  13 LINES HERE
   ;;  <TODO>
         ;; let temp_0 temp_4
31 ;;  <NOT TODO>


   ;;postlude
```

```
end:
    jump end
```

Update 21/1: here is the output of our compiler for the TODO:

Listing 4.2: 'output of the teaching staff compiler'

```
    ;; add3 temp_4 temp_3 temp_1
    getctr sp r0
    add r0 r0 0
    setctr a0 r0
    readse a0 16 r0
    getctr sp r1
    add r1 r1 16
    setctr a0 r1
    readse a0 16 r1
    add3 r1 r0 r1
    getctr sp r0
    add r0 r0 32
    setctr a0 r0
    write a0 16 r1
    ;; end add3 temp_4 temp_3 temp_1
```

EXERCISE #6 ▶ **Implement**

Now you are on your own to implement this code generation. Here are the main steps (less than 50 locs of PYTHON):

1. We have implemented for you an `alloc_to_mem(self)` method in `APISaruman.py`. This method only maps each temporary ("temporary") to a new offset in memory (in a PYTHON `dict`), then iterates the `replace_mem` function on all instructions of the three adress program to perform the actual allocation.
2. In `Allocations.py`, implement a `replace_mem(old_i)` that takes as input a "3-address with temporaries" SARUMAN code and outputs a list of instructions as a replacement. For instance, each time we access a source operand, we have to load it from memory before, thus the `replace_mem` should contains lines like:

```
after.append(Instru3A('getctr', SP, Indirect(R0)))
after.append(Instru3A('add', R0, R0, offset * 16))
after.append(Instru3A('setctr', A0, Indirect(R0)))
after.append(Instru3A('write', A0, 16, R1))
```

where offset is strongly linked with the temporary id (`temp0` has an offset of value 0, ...)

The files you generate have to be tested with the SARUMAN simulator with the same script as before. **Of course, with "all-in-mem" allocation, there should not be any "skipped" test any more.**

EXERCISE #7 ▶ **Tests, Tomuss**

We provide you the same test infra than in Lab 3. Same instructions as the former Lab for the archive deposit on Tomuss. **Please to not modify the Makefile, nor the Grammar, nor the code filenames, its structure.**

c	`dr <-newTemp()`
	`code.add(InstructionLETI(dr, c))`
	`return dr`

x	`#get the place associated to x.`
	`regval<-getTemp(x)`
	`return regval`

e_1+e_2	`t1 <- GenCodeExpr(e_1)`
	`t2 <- GenCodeExpr(e_2)`
	`dr <- newTemp()`
	`code.add(InstructionADD(dr, t1, t2))`
	`return dr`

e_1-e_2	`t1 <- GenCodeExpr(e_1)`
	`t2 <- GenCodeExpr(e_2)`
	`dr <- newTemp()`
	`code.add(InstructionSUB(dr, t1, t2))`
	`return dr`

true	`dr <-newTemp()`
	`code.add(InstructionLETI(dr, 1))`
	`return dr`

$e_1 < e_2$	`dr <- newTemp()`
	`t1 <- GenCodeExpr(e1)`
	`t2 <- GenCodeExpr(e2)`
	`endrel <- newLabel()`
	`code.add(InstructionLETI(dr, 0))`
	`#if t1>=t2 jump to endrel`
	`code.add(InstructionCondJUMP(endrel, t1, 'sge' , t2)`
	`code.add(InstructionLETI(dr, 1))`
	`code.addLabel(endrel)`
	`return dr`

Figure 4.2: 3@ Code generation for numerical or Boolean expressions (t1 and t2 are already defined)

x = e	`dr <- GenCodeExpr(e)`
	`#a code to compute e has been generated`
	`find loc the location for var x`
	`code.add(instructionLET(loc,dr))`

S1; S2	`#concat codes`
	`GenCodeSmt(S1)`
	`GenCodeSmt(S2)`

`if` *b* `then` *S*1 `else` *S*2	`lelse,lendif <-newLabels()`
	`t1 <- GenCodeExpr(b)`
	`#if the condition is false, jump to else`
	`code.add(InstructionCondJUMP(lelse, t1, "eq", 0))`
	`GenCodeSmt(S1) #then`
	`code.add(InstructionJUMP(lendif))`
	`code.addLabel(lelse)`
	`GenCodeSmt(S2) #else`
	`code.addLabel(lendif)`

`while` *b* `do` *S* `done`	`ltest,lendwhile <-newLabels()`
	`code.addLabel(ltest)`
	`t1 <- GenCodeExpr(b)`
	`code.add(InstructionCondJUMP(lendwhile, t1, "eq", 0))`
	`GenCodeSmt(S) #execute S`
	`code.add(InstructionJUMP(ltest)) #and jump to the test`
	`code.addLabel(lendwhile) #else it is done.`

Figure 4.3: 3@ Code generation for Statements

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:

```
1   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 adressed by bits (and not words), from address 0.

The SARUMAN has 8 registers from r0 to r7. Only r7[1] 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 **z**ero.
- **c**: A **c**arry happened during the previous operation.
- **v**: An o**v**erflow happened during the previous operation.
- **n**: The result of the previous operation is strictly negative (< 0).

Check the file `mif08-labs18/saruman/doc/emu_flag_management.md` for details.

[1] 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`	*reg reg*	addition		zcvn
0001	`add2i`	*reg const*	add immediate constant	z	zcvn
0010	`sub2`	*reg reg*	subtraction		zcvn
0011	`sub2i`	*reg const*	subtract immediate constant	z	zcvn
0100	`cmp`	*reg reg*	comparison		zcvn
0101	`cmpi`	*reg const*	comparison with immediate constant	s	zcvn
0110	`let`	*reg reg*	register copy		
0111	`leti`	*reg const*	fill register with constant	s	
1000	`shift`	*dir reg shiftval*	logical shift		zcn
10010	`readze`	*ctr size reg*	read *size* memory bits (zero-extended) to *reg*		
10011	`readse`	*ctr size reg*	read *size* memory bits (sign-extended) to *reg*		
1010	`jump`	*addr*	relative jump		
1011	`jumpif`	*cond addr*	conditional relative jump		
110000	`or2`	*reg reg*	logical bitwise or		zcn
110001	`or2i`	*reg const*	logical bitwise or	z	zcn
110010	`and2`	*reg reg*	logical bitwise and		zcn
110011	`and2i`	*reg const*	logical bitwise and	z	zcn
110100	`write`	*ctr size reg*	write the lower *size* bits of *reg* to mem		
110101	`call`	*addr*	sub-routine call	s	
110110	`setctr`	*ctr reg*	set one of the four counters to the content of *reg*		
110111	`getctr`	*ctr reg*	copy the current value of a counter to *reg*		
1110000	`push`	*reg*	push value of register on stack		
1110001	`return`		return from subroutine		
1110010	`add3`	*reg reg reg*			zcvn
1110011	`add3i`	*reg reg const*		z	zcvn
1110100	`sub3`	*reg reg reg*			zcvn
1110101	`sub3i`	*reg reg const*		z	zcvn
1110110	`and3`	*reg reg reg*			zcn
1110111	`and3i`	*reg reg const*		z	zcn
1111000	`or3`	*reg reg reg*			zcn
1111001	`or3i`	*reg reg const*		z	zcn
1111010	`xor3`	*reg reg reg*			zcn
1111011	`xor3i`	*reg reg const*		z	zcn
1111100	`asr3`	*reg reg shiftval*			zcn
1111101	`sleep`		sleep		
11111100	`rand`		rand		
11111101	`lea`	*reg addr*	load effective address *addr*		
11111110	`print`	type *reg*	print		
11111111	`printi`	type *const*	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

addr : *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

shiftval : *prefix-free* encoding of shift constants	
0 + 6 bits	constant between 0 and 63
1	constant value 1

const : *prefix-free* encoding of ALU constants	
0 + 1 bit	constant 0 ou 1
10 + 8 bits	byte
110 + 32 bits	
111 + 64 bits	

size : *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` ← `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` ← *R*7.

In:

```
loop:
  sub2i r0 1      ; substraction 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 `cmp` op1 op2)
0	0	0	`eq`, `z`	equal, op1 = op2
0	0	1	`neq`, `nz`	not equal, op1 ≠ op2
0	1	0	`sgt`	signed greater than, op1 > op2, two's complement
0	1	1	`slt`	signed smaller than, op1 < op2, two's complement
1	0	0	`sge`	op1 ≥ op2, signed
1	0	1	`ge`, `nc`	op1 ≥ op2, unsigned
1	1	0	`lt`, `c`	op1 < op2, unsigned
1	1	1	`sle`	op ≤ op2, signed

Table A.4: Counters (special registers).

encoding	mnemonic	description
00	`pc`	program counter
01	`sp`	stack pointer
10	`a0`	generic address counter
11	`a1`	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 adress 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

foo:
    .const 5 #10101
```

loads the adress 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:

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

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

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

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 semicolum `;`

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 differenciating 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 $(-3)_{10} = \texttt{111101}_{\bar{2}}$.