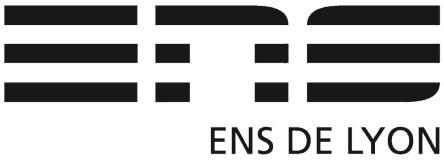


Compilation (MIF08)

Cahier de TD, janvier 2019

Contents

Chapter 1
SARUMAN Architecture, Lexical Analysis and Grammars

1.1 The SARUMAN architecture

We give you the "SARUMAN cheat sheet". The objective is to recall concepts from the architecture course and manipulate the assembly code of the architecture you will compile to.

Your teaching assistant will make a demo of the SARUMAN simulator during this session.

EXERCISE #1 ▶ **TD**
On paper, write (in SARUMAN assembly language) a program which initializes the r_0 register to 1 and increments it until it becomes equal to 8; using only one register.

Then, write a similar program that increments it until it becomes equal to 4242.

EXERCISE #2 ▶ **TD : sum**
Write a program in SARUMAN assembly that computes the sum of the 10 first positive integers.

EXERCISE #3 ▶ **Hand disassembling**
In Figure 1.1 we depicted a toy example with its corresponding assembly code.

Fill the first two rows of the table, and everywhere you find dots (...); read the rest of the solution, and answer the following questions:

- Which instruction is used to load data from memory?
- How is the pointer jumping done to create the loop?
- What happens to the labels in the disassembled program?
- What is the purpose of the "'jump -13"' instruction?
- In your own words describe what this program does.

EXERCISE #4 ▶ **Imperative code to SARUMAN- Skip if you are late**
Translate into SARUMAN code the following imperative-code:

```
x=5;
if (x>12) y=70; else y=x+12;
```

1.2 Lexical Analysis

A bit of ANTLR4 syntax is given as companion material.

EXERCISE #5 ▶ **Regular expressions for lexing**
Use the ANTLR4 syntax to define ANTLR4 macros to define:

1. Identifiers : any sequence of letters, digits and _ that do not begin by a digit nor _.
2. Floats like -3.96 (the sign is optional, but the dot is not).
3. Scientific notation like $-1.6E-12$.

EXERCISE #6 ▶ **Romans numbers**
Write an ANTLR4 lexical file that reads and interprets Roman numerals : $IV \rightarrow 4$ ...You can use the fact that the lexical analysis always takes the rule to match the longest subchain.

Labels	Binary	Instructions	pseudo-code
	`110011 000 00`	...	...
	`1110011 001 000 01`	...	...
	`11111101 010 001100111`	`lea r2 +88`	$R_2 \leftarrow$ &(label ...)
	`110110 10 010`	`setctr a0 r2`	$a_0 \leftarrow R_2$
	`10010 10 100 011`	`readze a0 8 r3`	$R_3 \leftarrow \text{mem}[a_0 : a_0 + 8]$
loop:	`0001 001 1000000010`	`add2i r1 2`	$R_1 \leftarrow R_1 + 2$
	`0011 011 01`	`sub2i r3 1`	$R_3 \leftarrow R_3 - 1$
	`0101 011 00`	`cmpi r3 0`	Compare R_3 and 0
	`1011 100 011001101`	`jumpif sgt -51`	if $R_3 > 0$ jump to ...
	`11111110 0001 001`	`print signed r1`	print r1
halt:	`1010 011110011`	`jump -13`	jump ...
data:	`00000110`	`.const ......`	...

Figure 1.1: A binary/hexadecimal program (`ex5.bin`)

1.3 Grammars

All grammars will use the ANTLR4 syntax.

EXERCISE #7 ▶ **Well-founded parenthesis**
Write a grammar and files to accept any text with well-formed parenthesis ')' and '['.

Chapter 2
AST, Attributions, Types

2.1 Derivation trees and attributions

EXERCISE #1 ▶ **Arithmetic expressions**
Let us consider the following grammar (the end of an expression is a semicolon):

$$
\begin{aligned}
Z &\rightarrow E; \\
E &\rightarrow E + T \\
E &\rightarrow T \\
T &\rightarrow T * F \\
T &\rightarrow F \\
F &\rightarrow (E) \\
F &\rightarrow i
\end{aligned}
$$

- What are the derivation trees for $1+2+3;$, $1+2*3;$, $(1+2)*3;$?
- Attribute the grammar to evaluate arithmetic expressions.

EXERCISE #2 ▶ **Declarations of variables**
Write a grammar that accepts declarations of variables like:

```
int x=1;
float y,z;
int t;
float u,v=0;
```

and rejects:

```
int x, int y;
```

Then write an attribution that prints individual declarations (of the first case) like:

```
int x=1; float y; float z; int t; float u; float v=0;
```

EXERCISE #3 ▶ **XML Files**

We give the following grammar:

```
L ->E L
|
E -> A L B
|    ident
A -> < ident >
B -> </ ident >
```

1. Give the derivation tree for the chain <html><head>toto</head>titi</foo>.
2. Attribute this grammar to verify that opening and closing tags refer to the same identifiers.

2.2 The Mu-language

The objective here is to be familiar with the grammar of the language we will compile.

EXERCISE #4 ▶ **Mu-grammar**
Here is the (simplified) grammar for the Mu language (expr are numerical or boolean expressions):

```
grammar Mu;

prog: vardecl_l block EOF #progRule;

vardecl_l: vardecl* #varDeclList;

vardecl: VAR id_l COL typee SCOL #varDecl;

id_l
    : ID              #idListBase
    | ID COM id_l     #idList
    ;

block: stat*   #statList;

stat
    : assignment
    | if_stat
    | while_stat
    | log
    | OTHER {print("unknown␣char:␣{}".format($OTHER.text))}
    ;

assignment: ID ASSIGN expr SCOL #assignStat;

if_stat: IF condition_block (ELSE IF condition_block)* (ELSE stat_block)? #
   ifStat;

condition_block: expr stat_block  #condBlock;

stat_block
    : OBRACE block CBRACE
    | stat
    ;

while_stat: WHILE expr stat_block #whileStat;

log: LOG expr SCOL #logStat;

expr
    : <assoc=right> expr POW expr          #powExpr
    | MINUS expr                           #unaryMinusExpr
    | NOT expr                             #notExpr
```

Write two valid programs for this grammar.

2.3 Typing

We recall (some of) the rules of the course for Typing:

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \qquad \frac{\Gamma(x) = t \quad t \in \{\texttt{int}, \texttt{bool}\}}{\Gamma \vdash x : t}$$

$$\frac{\Gamma \vdash e : t \quad \Gamma \vdash x : t \quad t \in \{\texttt{int}, \texttt{bool}\}}{\Gamma \vdash x := e : \texttt{void}} \qquad \frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma \vdash S : \texttt{void}}{\Gamma \vdash \texttt{while}\ b\ \texttt{do}\ S\ \texttt{done} : \texttt{void}}$$

EXERCISE #5 ▶ **Well typed**
Type the mini-while program:

```
var x1:int;
var x3:bool;
x1 := 3 ;
while (not x3) do
  x1 := x2 + 1 ;
  x3 := x3 and true
done
```

Chapter 3
3-address Code Generation, Liveness

3.1 Code generation with temporaries

The code we generate will have an unbounded number of temporaries (`tmp0`, `tmp1`, ...) but actual SARUMAN instructions (`add`, `and`, `let`, ...) except for `jumpif`

The instruction set and documentation for the SARUMAN machine can be found in Appendix A.

The code generation functions (see Appendix 4.1) have the following signatures:

`GenCodeExpr` : AExp $\rightarrow$ Code$^* \times \mathbb{N}$
`GenCodeSmt` : Inst $\rightarrow$ Code*

where $Code*$ is a sequence of 3-address instructions (SARUMAN with temporaries). As a side effect, the code generation for statements might update a map $Var \rightarrow \mathbb{N}$ (program variable to a temporary where to find its current value).

Auxiliary functions:

`newTemp()` : $\rightarrow \mathbb{N}$
`newLabel()` : $\rightarrow \mathbb{N}$

EXERCISE #1 ▶ **By hand!**
Using the code generation rules for the SARUMAN machine, generate the three-address SARUMAN code for the following (mini-while) program:

```
var x1,x2: int;
x1 := 13;
x2 := 7 + x1;
while (x2>x1) do
   x2 := x2-1;
   log (x2);
done
```

EXERCISE #2 ▶ **A new operator for expressions**
Write a code generation rule for the `xor` boolean operator without using the native `xor`

EXERCISE #3 ▶ **A new langage construction**
Write a code generation rule for the `repeat S until e` statement.

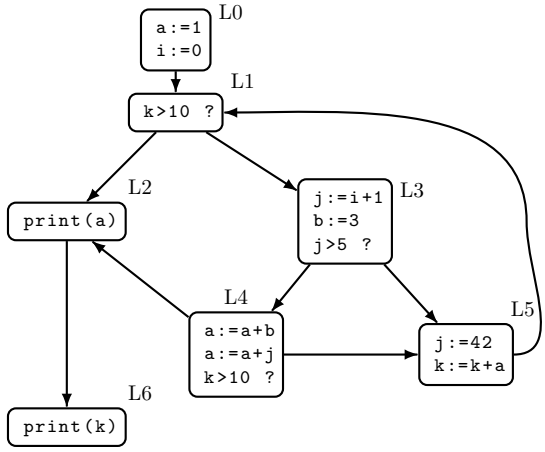
3.2 Liveness analysis

3.2.1 Liveness by hand

EXERCISE #4 ▶ **Liveness by hand - CC 2016**

In Figure 3.1, we give a CFG and we recall that a *variable is alive after a block if there exists a path from this block to one use of this variable that do not contain a definition of it.*

1. (by hand) Fill the array with "out"-alive variables for each block.
2. Remove dead code.



bloc	variables vivantes en sortie du bloc
L0	
L1	
L2	
L3	
L4	
L5	
L6	$\emptyset$

Figure 3.1: CFG and alive variables to complete

3.2.2 Liveness with fixpoint!

Let us recall the notations here: A variable at the left-hand side of an assignement is *killed* by the block. A variable whose value is used in this bloc (before any assignment) is *generated*.

$$LV_{exit}(\ell) = \begin{cases} \emptyset & \text{if } \ell = \text{final} \\ \bigcup\{LV_{entry}(\ell') | (\ell, \ell') \in flow(G)\} \end{cases}$$

$$LV_{entry}(\ell) = \left(LV_{exit}(\ell) \backslash kill_{LV}(\ell)\right) \cup gen_{LV}(\ell)$$

The sets are initialised to $\emptyset$ and computed iteratively, until reaching a fixpoint.

EXERCISE #5 ▶ **Live variables**
Generate the CFG for the following program:

```
while d>0 then {
    a:=b+c;
    d:=d-b;
    e:=a+f;
    if e>0 then {
        f:=a+d;
        b:=d+f;
    }
    else{
        e:=a-c;
        }
    b:=a+c;
}
```

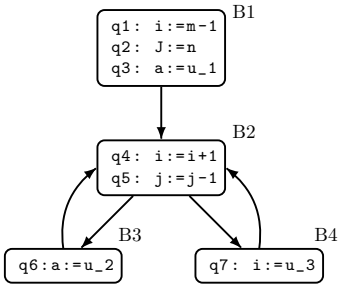
On this CFG:
- Compute Gen, $Kill$ for each block ℓ
- Compute $In(\ell) = LV_{entry}(\ell)$ and $Out(\ell) = LV_{exit}(\ell)$ iteratively.
- Suppress the dead code.

			Step		Step		Step		Step	
ℓ	$kill(\ell)$	$gen(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$

			Step		Step		Step		Step	
ℓ	$kill(\ell)$	$gen(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$

EXERCISE #6 ▶ **Live Variables**

After code generation, we obtain the following graph:



On this graph, perform liveness analysis and suppress the dead code.

			Step		Step		Step		Step	
ℓ	$kill(\ell)$	$gen(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$

			Step		Step		Step		Step	
ℓ	$kill(\ell)$	$gen(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$

Chapter 4
Register allocation and final code generation

In the following exercises we are looking for compiler independant optimisations (on the 3-address code). The goal here is to allocate registers with as few "spilled variables" as possible.

4.1 Register Allocation

For all these exercises, we do not consider the exact list of instructions that can access a given address in memory, but rather a "macro" instruction `LOAD Ri [`ℓ`]` that puts the content of address ℓ in the register r_i. (symetrically a `STORE` instruction).

EXERCISE #1 ▶ **Code production and register allocation**
Consider the expression $E = ((n * (n+1)) + (2 * n)$. n is supposed to be an integer stored at label n.

1. Generate a 3 address-code with temporaries and *LOAD* instruction to load n. Do it as blindly as possible (no temporary recycling).
2. (Without applying liveness analysis) Draw the liveness intervals. How many registers are sufficient to compute this expression?
3. Draw the interference graph (nodes are variables, edges are liveness conflicts).
4. Color this graph with three colors using the algorithm seen in the course.
5. Split one of the non colored variables in two and generate the spilling code.

We recall the following algorithm for final register allocation after coloring:
- For non-spilled variable: replace the temporary with its associated color/register.
- For a spilled variable (say, $temp5$ here, assigned to color 2):
  `ADD temp6 temp1 temp5`
  becomes (we use $R0, R1, R2$ to make load and stores for spilled variables):
  `SUB R0, R6, 2`
  `<code to read in memory at adress R0 -> R1>`
  `ADD alloc(temp6), alloc(temp1), R1`
  where $alloc(temp1)$ denotes the allocation of $temp1$ (if it is a spilled variable, we have to first load its value in $R2$).

EXERCISE #2 ▶ **Register allocation, adapted from Exam, 2016**

We consider (in two columns) the following SARUMAN code. The t_i are temporaries to be allocated (in registers, in memory). For this exercise, we consider that we have two novel instructions that are capable to directly read/write at memory labels (`ld , st`).

```
LOAD  t1 [label1]
LOAD  t2 [label2]
sub   t3 t1 t2
LOAD  t4 [label3]
LOAD  t5 [label4]
sub t6 t4 t5
add t7 t6 0
add t8 t3 t7
STORE  t8  [label5]
```

```
end:
 jump end
```

```
;;données/résultats
label1 : .word 2
label2 : .word 3
label3 : .word -1
label4 : .word 7
label5 : .reserve 1

;;gestion de la pile
[...]
```

1. What is the computed expression ? Where will it be stored ?
2. Fill the following table with stars: put a star for a given temporary at a given line if and only if it is alive *at the entry of the instruction*. After the last store, all temporaries are supposed to be dead.

code	t1	t2	t3	t4	t5	t6	t7	t8
`LOAD t1 [label1]`								
`LOAD t2 [label2]`								
`sub t3 t1 t2`								
`LOAD t4 [label3]`								
`LOAD t5 [label4]`								
`sub t6 t4 t5`								
`add t7 t6 0`								
`add t8 t3 t7`								
`STORE t8 [label5]`								

3. Draw the interference graph.
4. Color the graph with the algorithm from the course with 3 colors (green, blue, red, in this order).
5. We decide to spill the t_3 register and place it in memory. Color the rest of the graph with 2 colors (green, blue).
6. Generate the final code with two registers (r_3,r_4), r_6 for the stack, r_0, r_1, r_2 for the spill management.

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 4.1: 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 4.2: 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:

```
  leti r0 17      ; initialisation of a register to 17
loop:
3 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 `cap-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		
1111110	`rand`		rand		
1111111	`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

3   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:

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