Appendix A
LEIA Assembly Documentation (ISA)

Source:

- ISA: Florent de Dinechin, Nicolas Louvet, Antoine Plet, for ASR1, ENSL, 2016.

- Simulator: Pierre Oechsel and Guillaume Duboc, L3 students at ENSL, 2016.

A.1   Installing the simulator and getting started

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

A.2   The LEIA architecture

Here is an example of LEIA assembly code for 2017:

```
   letl r0 17     ; initialisation of  a register
loop:
   wmem r13 [r0]  ; write in memory
   rmem r13 [r2]  ; read in memory
   add r0 r10 r11 ; add
   snif r0 eq 3   ; test : if r0 = 3 skip next instruction
   jump loop  ; equivalent to jump -3, and this is a comment
   xor r0 r0 -1
```

Memory, Registers   The memory is shared into words of 16 bits, with address of size 16 bits (from $(0000)_H$ to $(FFFF)_H$).

The LEIA has 16 generalistic registers. Only R15[1] is reserved for the routine return address. They are also specific 16 bits registers: PC (*Program Counter*), IR (*Instruction Register*).

Constants: `leth` and `letl`   These expressions provide ways to initialize registers. The constant is encoded in the bits 0 to 7. For the `letl` instruction, bit 7 (sign bit) of the constant is replicated into the bits 8 to 15 of the destination register. Thus:

```
   letl r0 xx
```

stores the constant xx in register r0, provided xx between -128 and 127. The `leth` instruction stores the 8 bit constant in the bits 8 to 15 of the destination register, the other bits being unchanged. Thus:

```
   letl r0 2
   leth r0 3
```

stores in r0 the constant $2 + 3 * 2^8 = 770$. The LEIA assembler tool provides a macro:

```
   .let r0 770
```

to generate these two instructions automatically.

---

[1] registers are indifferently in capital letters or in lower case.

Table A.1: All LEIA instructions

15	14	13	12	mnemonic	class	description	ext(i)
0	0	0	0	wmem	wmem	write to memory	
0	0	0	1	add	ALU	addition	$z(i)$
0	0	1	0	sub	ALU	subtraction	$z(i)$
0	0	1	1	snif	snif	skip next if	$s(i)$
0	1	0	0	and	ALU	logical bitwise and	$s(i)$
0	1	0	1	or	ALU	logical bitwise or	$s(i)$
0	1	1	0	xor	ALU	logical bitwise xor	$s(i)$
0	1	1	1	lsl	ALU	logical shift left	$z(i)$
1	0	0	0	lsr	ALU	logical shift right	$z(i)$
1	0	0	1	asr	ALU	arithmetic shift right	$z(i)$
1	0	1	0	call	call	sub-routine call	
1	0	1	1	jump return	jump	relative jump if offset ≠ 1 return from call if offset = 1	
1	1	0	0	letl	letl	8-bit constant to Rd, sign-extended	
1	1	0	1	leth	leth	8-bit constant to high half of Rd	
1	1	1	0	print	print	print or refresh	
1	1	1	1	rmem copy	rmem	read from memory if i=0 register-to-register copy if i=1	

notation	meaning
d	a 3 or 4 bit number that specifies the destination register
i	a 4-bit number (bits 4 to 7 of the instruction word), the number of the first operand register
j	a 4-bit number

Table A.2: Notations

Arithmetical and logical instructions   Arithmetical and logical instructions have 3 operands:

```
add r1 r0 3     ; add immediate
add r1 r2 r1    ; add registers
```

The first operand is the destination register, and the two remaining operands are sources: either two registers (if the bit 11 is 0) or a register and an immediate constant j of 4 bits (if the bit 11 is 1). Because of the restricted number of bits to describe the first operand, the **destination register can only be one of the first eight registers** (from r0 to r7). If a constant is used then it is extended into a 16 bit constant before the operation. This is documented in the last column of table A.1:

- $z(j)$ means that j is extended with zeros. In other words j is interpreted as a *positive integer*.

- $s(j)$ means that the bit 3 (sign bit) of j is replicated into bits 4 to 15: j is interpreted as a *signed integer* and is transformed into a 16 bits integer of the same value.

Thus the result of the instruction:

```
add r1 r0 -1
```

is not really what is expected. The constant $j = -1$ is encoded as 1111, extended as $z(j) =$**0000000000001111**, thus the sum should be done with the 31 constant. **The assembler tool throws an error in that case:**

```
instruction add: Number, Not in bound: [0, 15]
```

Branching   Let a be the instruction's address, and c the integer encoded in the bits 0 to 11 of the instruction's word. The call instruction makes a copy of a + 1 into r_{15} then executes pc ← c × 16. **Thus procedures should have addresses that are multiple of 16.**

     The jump instruction considers the constant c as a signed integer (thus between -2048 and 2047) and executes pc ← a + c except if c = 1, in which case it executes pc ← r_{15}. In this case we can use the mnemonic return.

Table A.3: Encoding per instruction class

class	action	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ALU reg	$r_d \leftarrow r_i$ op r_j		opcode			0		d			i				j		
ALU imm4	$r_d \leftarrow r_i$ op ext(j)		opcode			1		d			i				j		
snif	skip next if	0	0	1	1	$c/\overline{r}$		condition			i				j		
letl	$r_d \leftarrow s(b)$	1	1	0	0		d					b					
leth	$r_d[15..8] \leftarrow b$	1	1	0	1		d					b					
call	jump to the routine	1	0	1	0					c							
jump	jump	1	0	1	1					c							
return	return to calling routine	1	0	1	1					0							1
wmem	mem$[r_j] \leftarrow r_i$	0	0	0	0	0	0	0	0		i				j		
rmem	$r_d \leftarrow$ mem$[r_j]$	1	1	1	1		d			0	0	0	0		j		
copy	$r_d \leftarrow r_j$	1	1	1	1		d			0	0	0	1		j		
print reg	print (the numerical content of) r_i	1	1	1	0	0	1	0	0		0				i		
print char	print c	1	1	1	0	1	0	0	0			$ascii(c)$					
refresh	wait	1	1	1	0					0							

Tests: `snif` "skip next if"   The `snif op1 <condition> op2` instruction deactivates the next instruction if the condition is true. Operands 1 and 2 are encoded like in the ALU instructions. In particular the second operand can be an immediate constant, which sign will be extended. The condition is encoded thanks to the following table:

10	9	8	mnemonic	description
0	0	0	eq	equal, op1 = op2
0	0	1	neq	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	gt	op1 > op2, unsigned
1	0	1	ge	op1 ≥ op2, unsigned
1	1	0	lt	op1 < op2, unsigned
1	1	1	le	op1 ≤ op2, unsigned

Let us illustrate the difference between `sgt` et `gt`: if R_0 contains 0, then:

   `snif r0 gt -1`

is false, but

   `snif r0 sgt -1`

is true. In fact, the -1 constant is extended as `ffff` (hexa), which is interpreted as 65535 by `gt`, and -1 by `sgt`.

Memory accesses   The memory address is always specified in the r_j register encoded in bits 0 to 3. The instruction `rmem rd [rj]` copies in the destination register (coded in bits 8 to 11) the content of the memory at address r_j. The instruction `wmem ri [rj]` copies the content of the register r_i (coded in bits 4 to 7) in the memory cell whose address is stored inside `rj`.

Register management   Some registers cannot be used with arithmetic and logical instructions, yet it is possible to use them to store a result thanks to the `copy` instruction. This instruction is also usefull before function calls to quickly save registers that are known to be used by the function.

Print   Two examples of use of the native print instruction:

```
print r1     ; prints the content of r1 (numerical value)
print 'z'    ; prints the character 'z'
```

Assembly directives A bit more of syntax:
- The assembly begins at address 0.
- Labels can be used for jumps. **Warning, for the compiler to work properly, do not type anything else than the label on its line, followed by a colon ':'.**
- The keyword `.word xxxx` reserves a memory cell initialized to the 16 bit constant xxxx.
- The keyword `.reserve xxxx` reserves n memory cells initialized to 0.
- 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).
- The keyword `.align16` pads memory cells in order for the next line to be at an address multiple of 16.
- The macro `.let r3 585` stores the constant 585 in register 3 (see paragraph A.2)
- The macro `.set r3 label` loads the address corresponding to label onto r3. For instance, the following program:

```
     .set r0 foo
foo:
3            .word 42
```

is assembled into:
```
c002 ; letl r0 2 (because 42 is stored at line 2)
d000 ; leth r0 0
002a ; the 42 constant
```

From Lab 5 we will be using a stack. The address of its top will be stored in r_7 and we will use the following macros:
- The macro `.push ri` that pushes the content of the r_i register into the memory. It is equivalent to:

```
  sub r7 r7 1
2 wmem ri [r7] ;
```

- The macro `.pop ri` that does the converse:

```
rmem ri [r7]
add r7 r7 1
```

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

A.4   The graphical library

Coordinates of the screen start on the bottom left corner of the screen $(_{(0,0)} \uparrow^x \rightarrow_y)$
- `cleanscr`: does what it is supposed to do. Uses register r_1.
- `putstr`: puts a string on the screen at coordinates (r_1, r_2) ; the string address is in register r3 ; if r_4 is not 1 then refresh between each letter. Uses registers 1, 2, 3, 6, 14, 15 and those of `putchar`. An example can be found in Lab 1.
- `putchar`: puts a char on the screen at coordinates (r_1, r_2). Uses registers r1 to r6. An example can be found in Lab 1.
- `refresh`: refreshes the screen.