Lab 1

Warm-up : the target machine : TARGET18

Objective

- Be familiar with the TARGET18 instruction set.
- Understand how it executes on the TARGET18 processor with the help of a simulator.
- Write simple programs, assemble, execute.

1.1 The TARGET18 processor, instruction set

EXERCISE #1 ▶ **Lab preparation**
Clone the github repository for this year's labs:

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

Then:

- In the `target18/emu/` directory, type make to compile the emulator. The assembler is `target18/asm.py`. Some more documentation can be found in the TARGET18 ISA on the course webpage and in Appendix A.

  `http://laure.gonnord.org/pro/teaching/capM1.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`.

In the architecture course, you already saw a version of the target machine TARGET18. The instruction set is depicted in Appendix A.

1.1.1 Hand exercises

EXERCISE #2 ▶ **TD**
On paper, write (in TARGET18 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 #3 ▶ **TD : sum**
Write a program in TARGET18 assembly that computes the sum of the 10 first positive integers.

1.1.2 Assembling, disasembling

EXERCISE #4 ▶ **Hand assembling, simulation of the hex code**
Assemble by hand the instructions :

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

You will need the set of instructions of the TARGET18 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 TARGET18 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.

EXERCISE #5 ▶ **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.

Labels	Binary	Instructions	pseudo-code
	`110011 000 00`	...	...
	`1110011 001 000 01`	...	...
	`11111101 010 001001011`	`lea r2 +75`	$R_2 \leftarrow$ mem [+75] (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 ...
halt:	`11111110 0001 001`	`jump -13`	jump ...
data:	`00000110`	`.const ......`	...

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

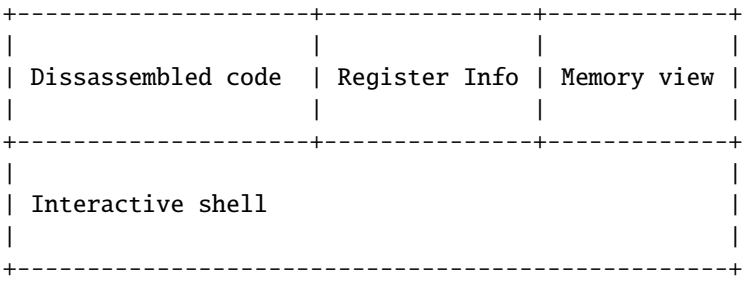
From now on, we are going to write programs using an easier approach. We are going to write instructions using the TARGET18 assembly.

1.2 TARGET18 **Simulator**

EXERCISE #6 ▶ **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 previous exercise 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:



   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 `nohalt.s`, then use the simulator to check if your assumtion 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 #7 ► **Algo in** TARGET18 **assembly**
Write a program in TARGET18 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 #8 ► **(Advanced) Algo in** TARGET18 **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:
  ```
    *
   * *
  * * *
  ```

1.2.1 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
TARGET18 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 TARGET18 assembler and simulator, follow instructions of the first lab (git pull on the course lab repository).

A.2 The TARGET18 architecture

Among others, the TARGET18 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 TARGET18 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 adressed by bits (and not words), from address 0.

The TARGET18 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/target18/doc/emu_flag_management.md` for details.

[1] Registers are in lower case.

Table A.1: TARGET18 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

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 TARGET18 emulator's memory layout is documented in the `cap-labs18/target18/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 $(-3)_{10} = \texttt{111101}_{\bar{2}}$.