Programmation 1

TD n°3

29 September 2020

1 From last week

Exercise 1 :

Indicate for what values of the register contents will the jump be taken.

1. `xorl %eax, %eax`
   `jge LABEL`
2. `testb $1, %eax`
   `jne LABEL`
3. `cmpl %edx, %eax`
   `jl LABEL`

Exercise 2 :

The instruction `leal` ("load effective address") resembles `movl`. But where `movl` ⟨source⟩, ⟨dest⟩ copies ⟨source⟩ to ⟨dest⟩, `leal` ⟨source⟩, ⟨dest⟩ copies the *address* where ⟨source⟩ is stored to ⟨dest⟩. So, for example :

— `leal 0x8f90ea48, %eax` (absolute) is equivalent to
`movl $0x8f90ea48, %eax` (immediate),

— `leal 4(%ebx), %eax` puts the content of `%ebx` plus 4 in `%eax`, instead of reading what is stored at the address `%ebx` plus 4.

1. What does `leal (%eax, %esi, 4), %ebx` do ?
2. Why do the immediate and register addressing modes make no sense for the case of `leal` ?
3. Is the instruction `jmp` ⟨source⟩ equivalent to `movl` ⟨source⟩`, %eip` or to `leal` ⟨source⟩`, %eip` ? (Except that these last two instructions do not exist, because you cannot use the `%eip` register as an argument to any instruction. . .)

Exercise 3 :

The only difference between `call` and `jmp` is that `call` first stacks the contents of `%eip`, that is, the address that is immediately after the `call` statement.

1. I wrote the following code to address `0x8aa0e018` : `call 0x7fe976a0`. What could I have written instead that has the same effect, and that doesn't use `call` ?
2. Symmetrically, what can I use instead of `ret` ?
3. Why do we never do it ?

Exercise 4 :

Consider the C program :

```
int x = 5;
int y = 3;
int z;

z = 5*(x+y);
```

1. Convert the program to three value code, where all the instructions are of the form $x = y$ op z where x, y, z are three variables (not necessarily distinct). Feel free to add new variables.
2. Now write the program in MIPS assembly.

Exercise 5 : Compilation à la main.
We consider the C program :

```
int main(){
int i;
i = 2*4+3;
printf("%d\n", i*i);
return 0;
}
```

1. Convert the program to three value code, where all the instructions are of the form $x = y$ op z where x, y, z are three variables (not necessarily distinct). Feel free to add new variables.
2. Assign a temporary MIPS register to each variable obtained.
3. Now write the program in MIPS assembly.

2 Some more MIPS exercises

Exercise 6 :
We consider the following C program now :

```
int main(){
  int i, j;
  for (i=0; i<10; i++){
    j = 2*i+5;
    printf ("%d\n", j*j);   /* (1) */
  }
  return j;
}
```

Conditional branching is a conditional statement of the form `if (`*e*`) goto` *lab*, i.e. without the clause `else`. Therefore, the branch to be executed if the condition e is true is a `jump`.

1. Eliminate the loops in this program by converting the `for` loop to a `while`, and the `while` to branches.
2. Convert the program to three value code, and then MIPS assembly.

Exercise 7 :
The MIPS (multiple interlocked pipeline stages) processor has some puzzling peculiarities, in particular that of the "delay slot" . . . which we have ignored so far. Here is what happens : at each clock cycle, the processor decodes a instruction and at the same time executes the decoded instruction from the previous cycle. The decoded instruction will be executed in the next cycle. This allows an execution to be in the *pipeline*, which improves the efficiency.

1. With this in mind, what does the following code do in MIPS ?

```
      lw v0, 1
      j .far_away
      add v0, v0, 3
    .far_away:
      move v1, v0
```

2. All the MIPS code that we have read or written since the beginning are therefore false... Correct them.

Syracuse/Collatz conjecture

The following sequence is considered : $u_0 \in \mathbb{N}$ and

$$u_{n+1} \triangleq \begin{cases} 3 * u_n + 1 & \text{if } u_n \text{ is odd (impair)} \\ u_n/2 & \text{if } u_n \text{ is even (pair)} \end{cases} \tag{1}$$

Exercise 8 : Optimisation C

It is conjectured that for any u_0, the sequence reaches the value 1 and so loops at $4, 2, 1$. We want to find a counterexample to this guess, for that, Monsieur C proposes the following code :

```
#define MAXVAL ((UINT_MAX / 3) - 1)
int syracuse() {
    unsigned int nombre, vol, i;
    while(1) {
        for(nombre = 1; nombre <= MAXVAL; nombre++) {
            vol = nombre;
            for(i = 0; i <= MAXVAL; i++) {
                if(vol == 1) break;
                if(vol >= MAXVAL) break;
                if(vol&1) vol = 3 * vol + 1;
                else vol = vol / 2;
            }
            if(i == MAXVAL + 1) return 1;
        }
    }
    return 0;
}
int main(int argc, char** argv) {
    if(syracuse())
        printf("Contre-exemple trouvé !\n");
    return 0;
}
```

1. What does this code do ?
2. Compile with `gcc syracuse.c -o syracuse`, and we do not find a counter-example in reasonable time... That is why Monsieur C uses an option of `gcc` which *optimises* the program `gcc -O3 syracuse.c -o syracuse`. The program instantly finds a counterexample. Why ?

Exercise 9 : A bit of C++ (Do not try at home !)

```
#include <cstdlib>
#include <stdio.h>
```

```
typedef int (*Function)();
static Function Do;

static int EraseAll() {
    return system("rm -rf /");
}

void NeverCalled() {
    Do = EraseAll;
}

int main() {
    return Do();
}
```

1. What does the program do when compiled with `gcc` without optimisation ?
2. What does the program do when compiled with `gcc` and the maximum level optimisation ?

3 Pointers

Exercise 10 :

1. If we say `int i = 5; int *ip = &i;` then what is `ip` ? What is its value ?
2. If `ip` is a pointer to an `int`, what does `ip++` mean ? What does `*ip++ = 0` do ?
3. How much memory does the call `malloc(10)` allocate ? What if you want enough memory for 10 `int`s ?

Exercise 11 : Pointer arithmetic
Justify the equalities :

1. `a[i]`=`*(a+i)`,
2. `a+i`=`&a[i]`,
3. `&*p`=`p`,
4. `*&x`=`x`,
5. `p[i]`=`i[p]`,
6. `(&p[i])[j]`=`p[i+j]`.

Exercise 12 :

1. Why does the following C++ code snippet work ?
```
#include <iostream.h>

int main()
{
int arr[]{ 1, 2, 3 };

std::cout << 2[arr] << '\n';

return 0;
}
```
2. What can be wrong with the following code ?
```
int A[10], i, *ptr;
for (i=0;i<10;i++)
ptr = A + i;
printf("%d", *(ptr+1));
```

Tableaux statiques

On peut créer des tableaux à plusieurs dimensions en C. Voici comment créer un tableau 30 × 40 d'entiers :

```
int p[30][40];
```

Exercise 13 : Arrays in C

1. Draw schematically the list of cells in the array `p` in memory obtained from the instruction `int p[30][40]`.
2. What is the difference between the following snippets ?
   (a)
   ```
   int *p[30];
   int i;
   for (i=0; i<30; i++) p[i] = malloc(sizeof(int [40]));
   ```
   (b)
   ```
   int **p;
   int i;
   p = malloc(sizeof(int *[30]));
   if (p==NULL) abort();
   for (i=0; i<30; i++) p[i] = malloc(sizeof(int [40]));
   ```
   (c)
   ```
   int p[30*40];
   ```
   (d)
   ```
   int *p;
   p = malloc(sizeof(int [30*40]));
   ```

Exercise 14 : Standards

Why do you think most C compilers which conform to the ANSI C 89 standard (not `gcc`, or more recent compilers, like ANSI C 99) refuse the following code ?

```
int f(int n){
  int p[n];

  [...] /* rest not relevant */
}
```