

MINI-PROJET AVEC L'ARCHITECTURE Y86

Nathan VANBESELAERE, Sacha DUPERRET

30 avril 2023

Résumé

Rapport détaillant le travail effectué en binôme pour adapter l'architecture Y86 aux consignes.

Table des matières

1	Exercice 1	2
1.1	Question 1	2
1.2	Question 2	2
2	Exercice 2	2
2.1	Question 1	2
2.2	Question 2	3
2.3	Question 3	3
2.4	Question 4	3
3	Exercice 3	4
3.1	Question 1	4
3.2	Question 2	4
4	Exercice 4	5
4.1	Question 1	5
4.2	Question 2	5
4.3	Question 3	5

1 Exercice 1

1.1 Question 1

Nous supprimons l'instruction :

```
intsig MRMOVL                                'instructionSet.get("mrmovl").icode'
```

Nous modifions également l'instruction set pour que RMMOVL ait un icode = 4 et ifun = 0, ainsi que MRMOVL ait un icode = 4 avec un ifun = 1. Le code source Y86 compile bien, le code hexadécimal est bien de 40 et 41 pour RMMOVL (ifun = 0) et MRMOVL (ifun = 1) respectivement.

1.2 Question 2

Nous supprimons l'ensemble des occurrences de MRMOVL. Dans les cas où MRMOVL était dissocié de RMMOVL (dans le code HCL), nous ajoutons l'instruction :

```
|| icode == RMMOVL && ifun == 1
```

permettant ainsi d'exécuter correctement le programme.

2 Exercice 2

2.1 Question 1

Nous modifions le instruction set en ajoutant STRGL avec un icode = 14 et un ifun = 0. Nous testons cette nouvelle instruction en utilisant ce code :

```
.pos 0
init:  irmovl stack,%esp
      call  test
      halt

test:  irmovl 3, %ecx
      strgl %ecx,%edx
      strgl %ecx,%ecx
      andl  %ecx,%ecx
      je   end
end:   ret

      .align 4
n:     .long 4
s:     .long 0
t:     .long 0x000a
      .long 0x00b0
      .long 0x0c00
      .long 0xd000

      .pos 0x100
stack:
```

```
.long 0
```

L'exécution se déroule conformément à nos attentes.

2.2 Question 2

Nous ajoutons l'instruction

```
intsig STRGL                                'instructionSet.get("lodsl").icode'
```

permettant de donner un icode à l'instruction STRGL. Nous testons le code avec les mêmes instructions que précédemment. Les valeurs des signaux et les opérations réalisées sont conformes à nos attentes.

2.3 Question 3

Suivant la même technique que pour la question 2 de l'exercice 1, nous factorisons les cas communs à STRGL ifun = 0 || ifun = 1.

2.4 Question 4

Nous codons un clone de strcpy en y86. Nous la testons dans le simulateur, avec le code ci-dessous :

```
.pos 0
irmovl stack,%esp

main : mrmovl size, %eax      #Taille du tableau
      pushl %eax

      irmovl t, %eax         #Source
      pushl %eax

      irmovl r, %eax         #Destination
      pushl %eax

      call strcpy

      iaddl 12, %esp
      halt

strcpy: mrmovl 4(%esp),%edi
      mrmovl 8(%esp),%esi
      mrmovl 12(%esp),%ecx

boucle: lodsl %eax
      stosl %eax
      isubl 1, %ecx
      jne boucle
      ret
```

```

        .pos 0x100
size:    .long 5

t:       .long 2           #Source
        .long 3
        .long 5
        .long 7
        .long 11

r:                               #Destination

        .pos 0x200
stack:   .long 0

```

Le comportement du code correspond aux attendus.

3 Exercice 3

3.1 Question 1

Nous ajoutons un icode de 15 avec ifun de 4 pour le code LOOP. Le code compile avec cette nouvelle instruction.

3.2 Question 2

Nous déclarons les signaux LOOP et RECX. Sous la forme :

```

intsig LOOP           'instructionSet.get("loop").icode'
intsig RECX           'registers.ecx'

```

Dans new_pc nous modifions la *#Taken branch : Use immediate value*.

```

icode in { JXX } && Bch : valC;
icode in { LOOP } && Bch : valC;

```

Nous testons nos modifications avec ce code :

```

        .pos 0
irmovl t, %esi
irmovl r, %edi
mrmovl s, %ecx

boucle: lodsl %eax
        stosl %eax
        loop boucle
        halt

        .pos 0x100
s:      .long 5

```

```

t:  .long 2
    .long 3
    .long 5
    .long 7
    .long 11
r:

```

Le code compile et s'exécute sans erreur.

4 Exercice 4

4.1 Question 1

Nous ajoutons `loope` et `loopne` au jeu d'instruction (`icode = 15`, identique à celui de `loop`). Pour les ifun, nous attribuons :

- `loop` : 0
- `loope` : 3
- `loopne` : 4

Ce qui positionne ainsi le signal `Bch` conformément au comportement souhaité.

4.2 Question 2

Dans le code HCL, nous ajoutons le support de `loope` et `loopne` via la même procédure que dans les exercices précédents.

4.3 Question 3

Nous codons un clone de la fonction `strcpy` en Y86. Pour cela, nous utilisons les fonctions codées précédemment.

Nous testons la fonction avec le code suivant :

```

.pos 0
irmovl stack,%esp

main :  mrmovl limite, %eax      #Limite
        pushl %eax

        irmovl t, %eax         #Source
        pushl %eax

        irmovl r, %eax         #Destination
        pushl %eax

        call strcpy            #Test premier cas
        call strcpy_bis       #Test deuxième cas
        iaddl 12, %esp
        halt

```

```

strcpy: mrmovl 4(%esp),%edi      #Test premier cas
        mrmovl 8(%esp),%esi
        mrmovl 12(%esp),%ecx

boucle: lodsl %eax
        stosl %eax
        isubl 0, %eax           #On vérifie si le tableau est vide
        jne nonvide            #Si le tableau n'est pas encore vide
        rmmovl %eax,-4(%edi)    #Si le tableau est vide,
                                #on ajoute un 0 à la suite de notre tableau
                                #de destination

nonvide:loopne boucle
        ret

strcpy_bis: irmovl b, %esi      #Test deuxième cas
            mrmovl 4(%esp),%edi
            mrmovl 12(%esp),%ecx
            jmp boucle          #Renvoi à notre boucle avec notre 2nd tableau

.pos 0x100
limite: .long 4                 #Limite

t: .long 2                      #Tableau premier cas
   .long 3

.pos 0x120
b: .long 2                      #Tableau deuxième cas
   .long 3
   .long 5
   .long 7
   .long 11

r:                                #Destination

.pos 0x200
stack: .long 0

```

La valeur sentinelle est bien trouvée avant que le nombre de mots maximum à copier ne soit atteint. Si la valeur sentinelle se situe après le nombre de mots à copier nous trouvons le nombre total de mots en premier.