
Polytech'Nice Sophia --- Département de sciences informatiques

# Simulation du jeu d'instruction d'un micro-processeur (très) simple

Jean-Paul RIGAULT

Année scolaire 2013-2014 — 6-13 juin 2014

## Table des matières



## Table des figures



## Liste des tableaux

# 1 Objectif

Le but de ce projet est d'écrire un simulateur du jeu d'instructions d'un processeur très simplifié [1].

Le processeur en question est décrit dans la section 2. Vous partirez d'un code complet et opérationnel fourni sous forme d'une bibliothèque binaire (`libsimul.a`) et des fichiers `.h` associés. Ce code est décrit dans la section 3.1.

Votre mission est alors de remplacer progressivement tous les modules de la bibliothèque `libsimul.a` par vos propres versions. La section 3.2 donne quelques indications et conseils sur la manière d'effectuer ce travail.

# 2 Présentation du processeur

## 2.1 Structure générale

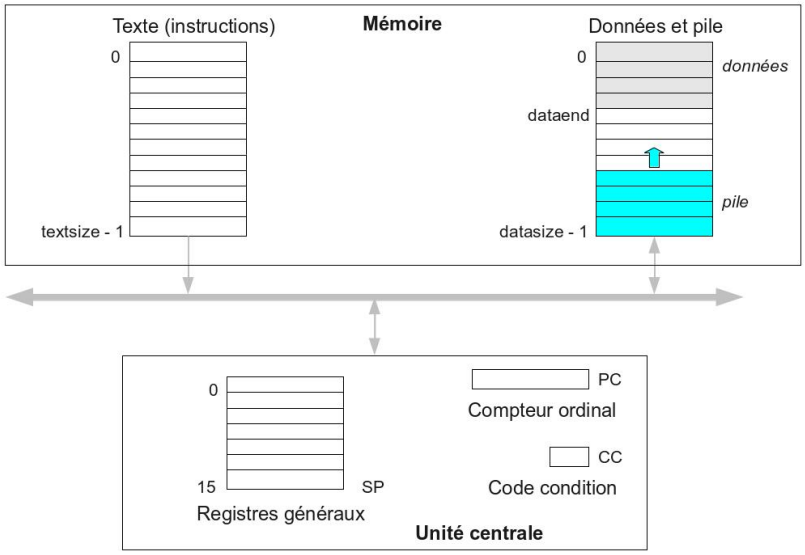


FIGURE 1 – Structure générale du processeur

L'architecture globale de notre processeur est décrite dans la figure 1. Il est simplement composé de mémoire et d'une unité centrale. La **mémoire** est composées de deux parties (deux « segments »), chacun addressable séparément à partir de l'adresse 0. Chaque segment est composé de mots de 32 bits. Une adresse mémoire adresse un mot tout entier [2].

1. Ce jeu d'instructions est tellement simplifié que le processeur en question serait difficilement utilisable pour des programmes sérieux !
2. et non pas un octet comme dans la plupart des machines actuelles.

### 2.1.1 Structure de la mémoire

Le premier segment, dit **segment de texte**, contient les **instructions** du programme en cours (notre machine n'exécute qu'un programme à la fois). Toutes les instructions sont codées sur 32 bits et leur format est décrit plus loin (section 2.2.1). La taille totale du segment d'instructions est notée `textsize`. L'instruction située à l'adresse 0 est la première instruction exécutable du programme.

Le deuxième segment, dit **segment de données**, est composée de deux parties distinctes. La taille totale de ce segment est désignée par `datasize`. Les adresses basses contiennent des **données à allocation statique**. Ces données sont allouées à partir de l'adresse 0, et en incrémentant les adresses. Les adresses hautes correspondent à la **pile d'exécution**. Cette pile commence à la dernière adresse du segment de données (`datasize-1`) et croit dans le sens des adresses décroissantes. Elle contient les données dynamiques nécessaires à l'appel et retour de sous-programmes ainsi qu'au rangement des variables locales des dits sous-programmes. L'adresse du sommet de la pile est contenue en permanence dans le registre $SP$ (voir section suivante).

Aucun mécanisme matériel ne permet de détecter que les deux parties du segment de données risquent d'empiéter l'une sur l'autre. (Il appartient au programmeur de s'assurer (ou de se convaincre !) que ce n'est pas le cas, sinon le programme risque d'avoir un comportement non défini.)

Toutes les données manipulées par le processeur sont des données entières (signées ou non) codées sur 32 bits.

Cette configuration du segment de données est représentée sur la figure 2.

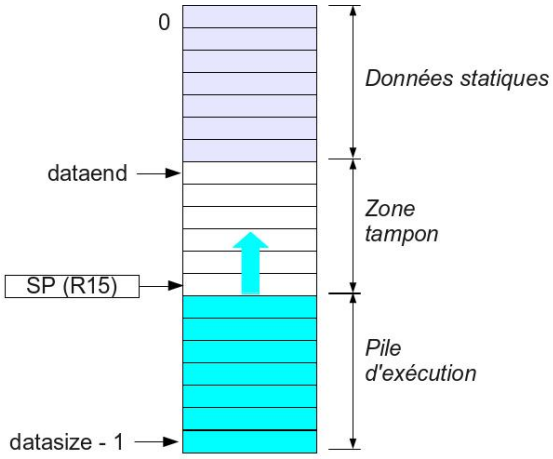


FIGURE 2 – Segment de données et pile d'exécution.

### 2.1.2 Structure de l'unité centrale

L'**unité centrale** est composée de **registres** pouvant recevoir des données (entières, signées ou non) ou des adresses (codées sur 32 bits) et effectuer un nombre (assez limité) d'opérations dessus. Il y a deux registres spécialisés :

— le **compteur ordinal** --- noté $PC$ comme *Program Counter* --- contient en permanence l'adresse (dans le segment de texte) de la prochaine instruction à exécuter ;

— le **code condition** --- noté $CC$ --- contient l'information sur le signe du résultat de la dernière opération de calcul ou de transfert exécutée par l'unité centrale. Il a quatre valeurs entières possibles : $U$ (valeur 0) pour inconnu, $Z$ (valeur 1) si la dernière opération a donné un résultat nul, $P$ (2) si ce résultat était strictement positif, $N$ (3) s'il était strictement négatif.

Les 16 **registres généraux** --- nommés $R00, R01, ..., R15$ --- peuvent être utilisés pour des calculs sur les données ou sur les adresses. Ils sont tous équivalents à ceci près que le registre $R15$ sert de **pointeur sur la pile d'exécution** du programme ; il est donc aussi désigné par le nom $SP$ (*stack pointer*) et ne doit donc être utilisé que pour les manipulations concernant cette pile.

## 2.2 Jeu d'instructions

### 2.2.1 Format des instructions

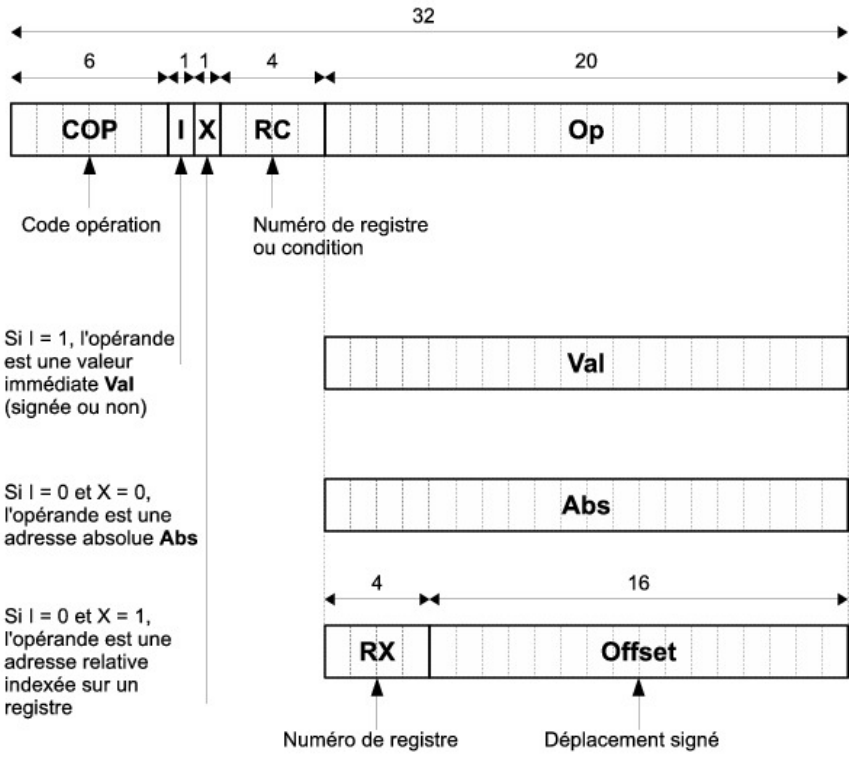


FIGURE 3 – Format des instructions machine.

Les instructions occupent toutes 32 bits et présentent toutes le même format (figure 3). Elles comportent 4 champs principaux :

— un **code opération** ($COP$) de 6 bits qui indique l'opération à effectuer ; la liste des opérations est fournie dans la table 2 ;
— deux **indicateurs** ($I$ et $X$) qui indiquent comment interpréter le champ $Op$ ;
— un champ « polymorphe » $RC$ qui, selon le code opération, est soit un **numéro de registre** général quelconque $R$ qui sera alors l'un des opérandes (source ou destination) de

TABLE 1 – Calcul de l'opérande (valeur immédiate ou adresse)

| I | X | Opérande |
|---|---|---|
| 1 | - | $Val$ |
| 0 | 0 | $Addr = Abs$ |
| 0 | 1 | $Addr = (RX) + Offset$ |

l'instruction, soit l'**indication d'une condition** $C$ à tester (relativement au code condition $CC$) --- voir page 7 ;

— un **opérande** $Op$ qui est soit une **valeur immédiate** entière signée $Val$ (si $I = 1$), soit une adresse (si $I = 0$) ; dans le second cas, cette adresse peut être **absolue** $Abs$ (si $X = 0$) ou **relative** ($X = 1$) ; une adresse relative (on dit aussi indexée) comporte un **déplacement** $Offset$ (valeur entière signée) et un numéro de **registre d'index** $RX$ qui peut être l'un quelconque des 16 registres généraux.

Il convient de noter que si le code opération est évidemment obligatoire, certaines instructions peuvent ignorer certains champs. Par ailleurs, toutes les combinaisons possibles ne sont pas autorisées, comme mentionné dans la table 2.

### 2.2.2 Calcul de l'opérande

Dans le cas d'une valeur immédiate ($I = 1$), l'opérande est simplement la valeur signée $Val$ présente dans l'instruction. Dans le cas d'une instruction référençant la mémoire ($I = 0$), l'opérande est sous forme d'une adresse $Addr$ telle que

$$Addr = Abs \text{ si } X = 0$$
$$Addr = (RX) + Offset \text{ si } X = 1$$

où la notation $(RX)$ désigne le **contenu du registre** de numéro $RX$. Ceci est résumé dans la table 1.

### 2.2.3 Liste des instructions

La table 2 liste toutes les instructions de la machine, leur paramètres et leur signification. Les colonnes y ont les significations suivantes :

**COP**
: code opération sous forme symbolique

**VOP**
: la valeur entière (décimale) du code opération

**I**
: l'indicateur *I* de l'instruction ; une case vide indique qu'il peut prendre toutes les valeurs possibles (0 ou 1) ; une case contenant 0 indique qu'il doit être nul (l'instruction est purement à référence mémoire, les valeurs immédiates ne sont pas autorisées) ; un tiret (--) indique que ce champ est ignoré

TABLE 2 – Liste des instructions

| COP | VOP | I | X | RC | Synoptique | Exécution | CC |
|---|---|---|---|---|---|---|---|
| ILLOP | 0 | - | - | - | Opération illégale | Arrêt intempestif de l'exécution | - |
| NOP | 1 | - | - | - | Opération vide | Rien | -- |
| LOAD | 2 | | | $R$ | Chargement d'un registre | si $I = 0 : R \leftarrow$ **Data**$[Addr]$<br>si $I = 1 : R \leftarrow Val$ | $R$ |
| STORE | 3 | 0 | | $R$ | Rangement d'un registre | **Data**$[Addr] \leftarrow R$ | -- |
| ADD | 4 | | | $R$ | Addition à un registre | si $I = 0 : R \leftarrow (R) +$ **Data**$[Addr]$<br>si $I = 1 : R \leftarrow (R) + Val$ | $R$ |
| SUB | 5 | | | $R$ | Soustraction à un registre | si $I = 0 : R \leftarrow (R) -$ **Data**$[Addr]$<br>si $I = 1 : R \leftarrow (R) - Val$ | $R$ |
| BRANCH | 6 | 0 | | $C$ | Branchement à une adresse | si $C$ est vrai : $PC \leftarrow Addr$ | -- |
| CALL | 7 | 0 | | $C$ | Appel d'un sous programme | si $C$ est vrai :<br>**Data**$[(SP)] \leftarrow (PC)$<br>$SP \leftarrow (SP) - 1$<br>$PC \leftarrow Addr$ | -- |
| RET | 8 | - | - | - | Retour sous programme | $SP \leftarrow (SP) + 1$<br>$PC \leftarrow$ **Data**$[(SP)]$ | -- |
| PUSH | 9 | | | - | Empilement | si $I = 0$ : **Data**$[(SP)] \leftarrow$ **Data**$[Addr]$<br>si $I = 1$ : **Data**$[(SP)] \leftarrow Val$<br>dans tous les cas : $SP \leftarrow (SP) - 1$ | -- |
| POP | 10 | 0 | | - | Dépilement | $SP \leftarrow (SP) + 1$<br>**Data**$[Addr] \leftarrow$ **Data**$[(SP)]$ | -- |
| HALT | 11 | - | - | - | Fin du programme | Arrêt normal de l'exécution | - |

**X**

l'indicateur *X* de l'instruction ; il n'a de sens que si *I* = 0 et indique alors si l'adresse est indexée ou non ; une case vide indique qu'il peut prendre toutes les valeurs possibles (0 ou 1) ; un tiret (--) indique que ce champ est ignoré

**RC**

un numéro de registre général (*R*) ou une condition (*C*, voir ci-après) ; un tiret (--) indique que ce champ est ignoré

**Synoptique**

résumé de l'effet de l'instruction

**Exécution**

l'exécution détaillée de l'instruction en utilisant le petit langage de transfert de registres décrit ci-après

**CC**
: la manière dont l'exécution de l'instruction modifie le code condition ; un tiret (--) indique que CC reste inchangé ; *R* indique que le code condition reflète le signe du contenu du registre de numéro *R* après l'exécution de l'opération.

Les notations suivantes sont utilisées dans les cellules de la table :

$SP$
: le pointeur de pile d'exécution ; c'est en fait le registre général 15 ($R15$) ; il pointe en permanence sur le sommet de cette pile (en fait le premier élément libre de la pile)

$PC$
: le compteur ordinal qui contient l'adresse de la prochaine instruction à exécuter

$R$
: un numéro de registre général (entre 0 et 15 inclus)

$Val$
: une valeur immédiate (un entier signé)

$Addr$
: une adresse, calculée après indexation, comme indiqué en 2.2.2

$C$
: une condition à tester par rapport au code condition $CC$ ; les valeurs (entières) possibles de $C$ sont les suivantes :

  $NC$ (0)
  : pas de condition, donc toujours vrai

  $EQ$ (1)
  : le code condition indique un résultat précédent nul (*i.e.*, $CC = Z$)

  $NE$ (2)
  : le code condition indique un résultat précédent non nul

  $GT$ (3)
  : le code condition indique un résultat précédent strictement positif

  $GE$ (4)
  : le code condition indique un résultat précédent positif ou nul

  $LT$ (5)
  : le code condition indique un résultat précédent strictement négatif

  $LE$ (6)
  : le code condition indique un résultat précédent négatif ou nul

$(R)$, $(PC)$, $(SP)$
: le contenu d'un registre (la valeur qu'il contient)

Dans ce petit langage de transfert de registres, **Data**$[Addr]$ (resp. **Text**$[Addr]$) désigne le contenu de la mémoire de données (resp. d'instructions) à l'adresse $Addr$, alors que **Data**$[(R)]$ désigne le contenu de la mémoire de données à l'adresse contenue dans le registre $R$. Quant à $Source \leftarrow Destination$, cela correspond au transfert de la valeur $Source$ (une valeur d'opérande, le contenu d'un registre, d'une mémoire) dans le registre ou la mémoire désigné par $Destination$.

## 2.3 Fonctionnement du processeur

### 2.3.1 Initialisation

1. Allouer les segments de texte (taille `textsize`) et de données (taille `datasize`) ;
2. Remplir le segment de texte avec les instructions du programme ; l'adresse 0 doit contenir la première instruction à exécuter ; l'exécution du programme se termine avec l'instruction HALT ;
3. Remplir le début du segment de données avec les données statiques initiales ; calculer `dataend` ; s'assurer qu'il reste à la fin du segment de données suffisamment de place libre pour la pile d'exécution ;
4. Initialiser les registres du processeur. Il est bon d'initialiser les registres généraux $R00$ à $R14$ (ainsi que $CC$) à 0, mais ce n'est pas indispensable. En revanche les initialisations suivantes sont obligatoires :

   $PC \leftarrow 0$ (adresse de la première instruction)
   $SP \leftarrow$ `datasize` $- 1$ (base de la pile d'exécution)

### 2.3.2 Boucle d'exécution

1. Recherche de l'instruction courante : c'est le contenu du mot mémoire **Text**$[(PC)]$ ;
2. Incrémentation du compteur ordinal : $PC \leftarrow (PC) + 1$ ;
3. Décodage de l'instruction : en particulier, vérification du format et des paramètres de l'instruction ;
4. Recherche de l'opérande de l'instruction qui peut être une valeur immédiate ou une adresse possiblement indexée ;
5. Exécution de l'instruction selon les spécifications de la table 2 ;
6. Reboucler en 1.

La boucle d'exécution s'arrête normalement quand on rencontre une instruction HALT. Elle peut aussi s'arrêter en cas d'erreur comme indiqué dans la section suivante.

### 2.3.3 Exceptions

La boucle d'exécution précédente doit en effet s'arrêter en cas d'erreur. Parmi les erreurs à détecter, on trouve :

— exécution de l'instruction illégale ILLOP ;
— instruction mal formée selon les spécifications de la table 2 : par exemple, STORE avec un opérande immédiat ;
— paramètres de l'instruction incorrects : par exemple, valeur impossible de la condition $C$ d'un branchement ;
— erreur de « segmentation » c'est-à-dire tentative d'accès à une adresse (de données, de texte ou de pile) supérieure à la taille du segment correspondant ;
— et tout autre situation qui vous paraitra significative...

### 2.3.4 Exemple simple

Le programme suivant multiplie la valeur à l'adresse 0 du segment de données (valeur `0xa`, soit 10) par celle à l'adresse 1 (valeur `0x5`, soit 5) du même segment ; le résultat (on espère que c'est `0x32`, soit 50) est rangé à l'adresse 2, toujours dans le segment de données. La multiplication est bien entendu obtenue par une boucle sur l'addition. Le programme est donné ici sous forme « désassemblée » :

```
0x0000: LOAD R00, @0000          chargement du premier opérande
0x0001: LOAD R01, @0001          chargement du second opérande
0x0002: SUB R01, #1              R01 est l'indice de boucle
0x0003: BRANCH LE, @0007         sortie de boucle si (R01) ≤ 0
0x0004: ADD R00, @0000           ajoute le premier opérande à R00
0x0005: SUB R01, #1              décrémentation de l'indice de boucle R01
0x0006: BRANCH NC, @0003         branchement inconditionnel au test de début de boucle
0x0007: STORE R00, @0002         sortie de boucle et rangement du résultat
0x0008: HALT                     fin de programme
0x0009: ILLOP
0x000a: ILLOP
0x000b: ILLOP
```

Le caractère `#` précède un opérande à **valeur immédiate** en base 10 : ainsi `#1` est la valeur 1 et `#-127` la valeur décimale -127 ; le caractère `@` précède une **adresse absolue** exprimée en base 16 ; enfin, dans le cas d'une **adresse relative**, on aurait une notation comme

```
STORE R05, +4[R09]
```

où +4 est le déplacement signé (*offset*, en base 10) et 9 le numéro du registre d'index (donc $Addr = (R09) + 4$). Les autres notations devraient être évidentes. Le segment de données initial a la forme suivante (les valeurs des données sont présentées à la fois en base 16 et 10) :

```
0x0000: 0x0000000a 10            premier opérande
0x0001: 0x00000005 5             second opérande
0x0002: 0x00000000 0             résultat futur
0x0003: 0x00000000 0
...
```

La trace d'exécution est la suivante (on l'a décorée en couleurs et avec quelques commentaires pour faire apparaitre clairement les différentes itérations de la boucle) :

```
*** Execution trace ***
TRACE: Executing: 0x0000: LOAD R00, @0000
TRACE: Executing: 0x0001: LOAD R01, @0001
TRACE: Executing: 0x0002: SUB R01, #1
TRACE: Executing: 0x0003: BRANCH LE, @0007 --- itération 1
TRACE: Executing: 0x0004: ADD R00, @0000
TRACE: Executing: 0x0005: SUB R01, #1
TRACE: Executing: 0x0006: BRANCH NC, @0003
TRACE: Executing: 0x0003: BRANCH LE, @0007 --- itération 2
TRACE: Executing: 0x0004: ADD R00, @0000
TRACE: Executing: 0x0005: SUB R01, #1
TRACE: Executing: 0x0006: BRANCH NC, @0003
```

```
TRACE: Executing: 0x0003: BRANCH LE, @0007 --- itération 3
TRACE: Executing: 0x0004: ADD R00, @0000
TRACE: Executing: 0x0005: SUB R01, #1
TRACE: Executing: 0x0006: BRANCH NC, @0003
TRACE: Executing: 0x0003: BRANCH LE, @0007 --- itération 4
TRACE: Executing: 0x0004: ADD R00, @0000
TRACE: Executing: 0x0005: SUB R01, #1
TRACE: Executing: 0x0006: BRANCH NC, @0003
TRACE: Executing: 0x0003: BRANCH LE, @0007 --- fin de boucle
TRACE: Executing: 0x0007: STORE R00, @0002
TRACE: Executing: 0x0008: HALT
WARNING: HALT reached at address 0x8
```

Le segment de données est de taille 20. L'état final de la partie utile de ce segment apparait alors comme :

```
0x0000: 0x0000000a 10          premier opérande
0x0001: 0x00000005 5           second opérande
0x0002: 0x00000032 50          résultat final : 10 × 5
0x0003: 0x00000000 0
...
```

On note que le résultat est bien correct (à l'adresse 2).

### 2.3.5 Exemple un peu plus compliqué

Voici un second example, un peu plus complexe puisqu'il effectue le même travail mais en faisant de la multiplication un sous-programme :

```
0x0000: PUSH @0002             empilement du premier opérande
0x0001: PUSH @0003             empilement du second opérande
0x0002: CALL NC, @000a         appel du sous-programme à l'adresse 0xa (10)
0x0003: ADD R15, #2            réajustement du pointeur de pile en sortie de sous-programme
0x0004: STORE R00, @0001       rangement du résultat à l'adresse 1
0x0005: HALT                   fin du programme (principal)
0x0006: NOP                    inutilisé
0x0007: NOP                    inutilisé
0x0008: NOP                    inutilisé
0x0009: NOP                    inutilisé
0x000a: LOAD R00, +3[R15]      début du sous-programme : chargement du premier opérande dans R00
0x000b: LOAD R01, +2[R15]      chargement du second opérande dans l'indice de boucle R01
0x000c: SUB R01, #1            la même boucle que dans l'exemple précédent
0x000d: BRANCH LE, @0011           test de début de boucle
0x000e: ADD R00, 3[R15]            corps de boucle
0x000f: SUB R01, #1                décrémentation de l'indice de boucle
0x0010: BRANCH NC, @000d           rebouclage
0x0011: RET                    retour du sous-programme
```

Ce programme est composée de deux parties. Dans la première, on empile les deux opérandes (qui sont cette fois-ci aux adresses 2 et 3), puis on appelle le sous-programme, ce qui a pour

effet d'empiler aussi l'adresse de retour. Le sous-programme lui-même commence à l'adresse `0xa` (10). Il est identique à celui du premier exemple, excepté pour la manière de récupérer les paramètres qui se fait par addressage relatif (indexé) par rapport au registre de pile $R15$. L'instruction RET dépile l'adresse de retour et revient juste après l'instruction CALL. Il faut alors réajuster la pile afin « d'oublier » la zone d'empilement des deux paramètres.

Les adresses basses du segment de données initial (dont la taille totale est 20) se présentent ainsi :

```
0x0000: 0x00000000 0
0x0001: 0x00000000 0          futur résultat
0x0002: 0x00000014 20         premier opérande
0x0003: 0x00000005 5          second opérande
0x0004: 0x00000000 0
0x0005: 0x00000000 0
...
```

À la fin du programme, on obtient :

```
0x0000: 0x00000000 0
0x0001: 0x00000064 100        résultat : 20 × 5
0x0002: 0x00000014 20
0x0003: 0x00000005 5
0x0004: 0x00000064 0
0x0004: 0x00000000 0
0x0005: 0x00000000 0
...
```

Quand aux adresses hautes du segment du segment de données, qui correspondent à la pile d'exécution, elles se présentent ainsi

```
Ox000f: 0x00000000 0
Ox0010: 0x00000000 0
Ox0011: 0x00000014 3          trace résiduelle de l'adresse de retour
Ox0012: 0x00000005 5          trace résiduelle du second opérande
Ox0013: 0x00000003 20         trace résiduelle du premier opérande
```

On note que le résultat correct (100) est bien à l'adresse `0x01` (en rouge). On constate aussi la trace des opérations sur la pile (en bleu) ; en revanche, en examinant l'état final des registres (également affiché par le simulateur) on peut vérifier que le pointeur de pile ($SP$ ou $R15$) est revenu à la valeur décimale 19, la dernière adresse possible pour un segment de taille 20.

```
PC: 0x00000006 CC: P

R00: 0x00000064 100 R01: 0x00000000 0 R02: 0x00000000 0
R03: 0x00000000 0 R04: 0x00000000 0 R05: 0x00000000 0
R06: 0x00000000 0 R07: 0x00000000 0 R08: 0x00000000 0
R09: 0x00000000 0 R10: 0x00000000 0 R11: 0x00000000 0
R12: 0x00000000 0 R13: 0x00000000 0 R14: 0x00000000 0
R15: 0x00000013 19
```

# 3 Travail à effectuer

## 3.1 Présentation du code de départ

Le code de départ de ce projet est à télécharge depuis mon Wiki à cette adresse.

Vous allez partir d'un code déjà existant et (en principe) opérationnel mais qui vous est fourni sous forme d'une bibliothèque binaire (**libsimul.a**). Votre mission sera donc de remplacer progressivement les modules de cette bibliothèque par les votres, jusqu'à obtenir un code qui vous sera complètement propre mais qui devra avoir les mêmes fonctionnalités --- éventuellement dégradées ou améliorées --- que le code de départ.

Ce code de départ vous est fourni sous forme d'un fichier **tar** compressé. Vous devez choisir parmi les trois possiblitiés suivantes en fonction de l'architecture de votre propre ordinateur :

— `Simul_Proc-linux-m32.tgz` : pour Linux 32 bits ;
— `Simul_Proc-linux--m34.tgz` : pour Linux 64 bits ;
— `Simul_Proc-mac-osx.tgz` : binaire « universel » [3] (32 et 64 bits) pour MacOS X.

Un fois extrait, par exemple par la commande **shell**

```
tar xvzf Simul_Proc-linux-m64.tgz
```

vous devez obtenir un répertoire (par exemple ici `Simul_Proc-linux-m64` dans lequel vous pouvez vous rendre et travailler. Ce répertoire contient

— la bibliothèque **libsimul.a** : dont la commande **shell**
`ar tv libsimul.a`
vous indiquera son contenu.
— les fichiers d'entête (`.h`) contenant l'interface des modules de la bibliothèques (il y en a 5).
— le fichier `test_simul.c` contenant le source de la fonction `main()` du simulateur.
— une `Makefile`.
— un répertoire `Examples` contenant des version binaires des deux exemples de programmes décrits dans le présent document (en 2.3.4 et 2.3.5) ainsi que d'autres exemples dans différents formats (voir 4).
— un répertoire `doc` contenant une documentation du code fourni générée par **doxygen** ; ouvrez à l'aide de votre butineur Web favori le fichier `doc/html/index.html` et vous trouverez une description de ce code et de la manière d'utiliser la `Makefile` pour faire ce que l'on vous demande.
— le fichier `Simul_Proc_IS.pdf` qui contient une copie du présent document.
— un exécutable (statiquement lié) nommé `asm` qui contient le code d'un assembleur symbolique (voir 4.3).

Compilez (il suffit de faire **make**) et exécutez le code (l'exécutable est **test_simul**) avec différentes options (**test_simul -h** vous listera les quelques options possibles) pour comprendre comment il fonctionne et est organisé. La documentation de `doc/html/index.html` vous donnera également plus d'information.

**Compatibilité du code Linux** Le code Linux (32 et 64 bits) a été généré sous FEDORA 19 avec **gcc**-4.9.0. L'édition de liens en a été totalement statique, ce qui signifie que les exécutables

3. Cependant, voir plus loin la note spécifique à l'utilisation du Mac.

(`test_simul`, `asm`) et la bibliothèque (`libsimul.a`) générés sont autonomes, ne dépendant d'aucune bibliothèque partagée. Leur utilisation a été vérifiée avec succès sous Fedora (17 et 19, 64 bits) et avec différentes versions d'Ubuntu 32 et/ou 64 bits (de 10.10 à 14.04LTS) [4].

**Note sur l'utilisation du Mac** L'utilisation du Mac est ici découragée. Si une partie seulement des membres du groupe de projet utilise un Mac, l'intégration du code risque d'être (un peu plus) compliquée. Par ailleurs, si le code du simulateur et de la bibliothèque (`test_simul` et `libsimul.a`) sont fournis sous forme de binaires universels (32 et 64 bits) et si la Makefile fournie génère bien du binaire universel, *il n'en est pas de même pour l'assembleur* `asm` (voir 4.3) . Ce dernier est écrit en C++ et seule une version binaire 64 bits est fournie.

De toutes façons, pour faire le projet sur Mac, il vous faut installer (si ce n'est pas encore fait) le logiciel de développement XCODE version 5 (pour avoir une version suffisamment récente de C++) *ainsi que ses outils en ligne de commande*. L'édition de liens statique n'esp pas (bien) supportée sous MacOS, et donc les binaires produits risquent d'être dépendant des versions des bibliothèques partagées. Bref... évitez le Mac, si vous pouvez.

Pour information, le code MacOS a été généré avec le compilateur de XCODE 5.1 (**clang**-3.4), la bibliothèque C++ standard d'Apple **libc++**, sous OSX 10.9.3 (Mavericks).

## 3.2 Ce qu'il faut faire

Votre mission est de remplacer les modules de code fournis dans la bibliothèque `libsimul.a` par les vôtres jusqu'à obtenir un code qui vous est complètement personnel mais qui correspond aux spécifications du code de départ.

La documentation de `doc/html/index.html` vous indique comment faire ce remplacement sur le plan pratique. Ce qui suit présente un certain nombre de suggestions et de conseils, voire d'impératifs catégoriques :

- Vous devez impérativement travailler sous Linux (ou éventuellement MaxOS X) [5], en ANSI C99 (option `-std=c99` de **gcc**).
- Vous devez rendre un code complet et propre : fichiers sources `.c` et `.h`, `Makefile` et documentation (par exemple en utilisant doxygen).
- Votre code doit au minimum respecter les spécifications établies (implicitement) par ce qui vous est fourni ; vous avez cependant le droit d'améliorer le résultat, l'affichage, l'interaction, etc.
- Ne modifiez pas les fichiers d'entête (`.h`) fournis ; les modules de la bibliothèque en dépendent. Il va sans dire que vous devez respecter les noms des modules (fichiers `.o`), des variables, des fonctions et des types de la bibliothèque `libsimul.a`. Vous n'avez pas non plus à modifier ni à reconstruire la bibliothèque binaire `libsimul.a`.
- En revanche vous pouvez ajouter vos propres fichiers `.h` et/ou `.c` si vous le désirez (bien que cela ne soit pas strictement nécessaire) ;
- Répartissez vous clairement les tâches, synchronisez vous souvent et développez incrémentalement, en remplaçant un module à la fois ; de cette manière vous pourrez toujours avoir quelque chose « qui tourne » en partie grâce à votre propre code et en partie à celui qui est fourni ;

4. Seules les versions 32 bits sont utilisables sur les Ubuntu 32 bits ; en général, les Ubuntu 64 bits s'accomodent à la fois de 32 et 64 bits.
5. Avec la manière dont le code de départ est fourni vous n'avez de toutes façons guère le choix.

TABLE 3 – Longueur (en nombre de lignes source) des différents modules.

| Fichier | Lignes totales | Lignes utiles |
|---|---|---|
| `debug.c` | 56 | 51 |
| `debug.h` | 25 | 6 |
| `error.c` | 54 | 30 |
| `error.h` | 69 | 27 |
| `exec.c` | 263 | 171 |
| `exec.h` | 30 | 6 |
| `instruction.c` | 139 | 87 |
| `instruction.h` | 139 | 73 |
| `machine.c` | 219 | 169 |
| `machine.h` | 176 | 37 |
| `prog.c` | 47 | 30 |
| `test_simul.c` | 122 | 78 |
| Total | 1339 | 765 |

— Testez souvent ; évidemment votre simulateur doit pouvoir exécuter correctement les deux exemples de code (en 2.3.4 et 2.3.5) donnés dans ce document mais c'est loin d'être un test suffisant : vous devez développer vos propres cas de test et vos propres programmes afin de vérifier que toutes les instructions sont interprétées correctement.

Finalement, pour vous donner un idée du volume de travail, la table 3 indique le nombre de ligne total et le nombre de lignes utiles (sans les commentaires ni les lignes vides) des différents fichiers du code fourni :

Peu de code donc, mais plus délicat qu'il n'y parait à faire proprement...

# 4 Comment écrire des programmes de test pour votre simulateur ?

Il est clair que les deux exemples de programmes donnés en 2.3.4 et 2.3.5 ne suffisent pas à tester votre programme. Il faut donc pouvoir générer d'autres programmes. Pour cela trois solutions s'offrent à vous.

## 4.1 En binaire

Écrire code et données en binaire n'est pas le plus pratique et n'est envisageable que pour des examples simples.

Lorsque vous lancez **test_simul**, cela simule l'exécution du programme défini dans le fichier source `prog.c` (par défaut le programme de 2.3.5 dont le binaire `prog.o` fait partie de

la bibliothèque). Il vous suffit donc de remplacer ce `prog.c` par votre propre version (voir 3.2 pour savoir comment faire) et de recompiler **test_simul**.

À titre d'exemple voici le contenu par défaut de `prog.c` (en 2.3.5). Ce code correspondant est dans le fichier `Examples/prog_subroutine-bin.c`).

```
Instruction text[] = {
    0x00002f09, 0x00003f09, 0x0000a007, 0x00002f44,
    0x00002003, 0x0000000b, 0x00000001, 0x00000001,
    0x00000001, 0x00000001, 0x0003f082, 0x0002f182,
    0x00001145, 0x00011606, 0x0003f084, 0x00001145,
    0x0000d006, 0x00000008,
};
int textsize = 18;

Word data[] = {
    0x00000000, 0x00000000, 0x00000014, 0x00000005,
    0x00000000, 0x00000000, 0x00000000, 0x00000000,
    0x00000000, 0x00000000, 0x00000000, 0x00000000,
    0x00000000, 0x00000000, 0x00000000, 0x00000000,
    0x00000000, 0x00000000, 0x00000000, 0x00000000,
};
int datasize = 20;
```

**Attention !** Lisez bien la section 4.4 pour comprendre l'ordre des champs avant d'envisager de vous lancer dans ce codage binaire.

## 4.2 En utilisant l'initialisation des structures de C99

Le format des instructions du processeur est défini dans le fichier `instruction.h` sous forme d'une union de structures C99. On peut alors utiliser la syntaxe « sophistiquée » d'initialisation des structures de C99. Cela clarifie et simplifie l'écriture des instructions, mais les données restent à coder en binaire.

Grâce à cette facilité, le fichier `prog.c` de la section précédente prend la forme suivante. Le code correspondant est dans le fichier `Examples/prog_subroutine.c`).

```
Instruction text[] = {
//   type                 cop    imm     ind     regcond operand      adresse
//-------------------------------------------------------------------------
    {.instr_absolute =  {PUSH,   false, false,  0,      2        }},  // 0
    {.instr_absolute =  {PUSH,   false, false,  0,      3        }},  // 1
    {.instr_absolute =  {CALL,   false, false,  NC,     10       }},  // 2
    {.instr_immediate = {ADD,    true,  false,  15,     2        }},  // 3
    {.instr_absolute =  {STORE,  false, false,  0,      1        }},  // 4
    {.instr_generic =   {HALT,                                    }},  // 5
    {.instr_generic =   {NOP,                                     }},  // 6
    {.instr_generic =   {NOP,                                     }},  // 7
    {.instr_generic =   {NOP,                                     }},  // 8
    {.instr_generic =   {NOP,                                     }},  // 9
    {.instr_indexed =   {LOAD,   false, true,   0,      15, +3   }},  // 10
    {.instr_indexed =   {LOAD,   false, true,   1,      15, +2   }},  // 11
    {.instr_immediate = {SUB,    true,  false,  1,      1        }},  // 12
    {.instr_absolute =  {BRANCH, false, false,  LE,     17       }},  // 13
    {.instr_indexed =   {ADD,    false, true,   0,      15, +3   }},  // 14
    {.instr_immediate = {SUB,    true,  false,  1,      1        }},  // 15
```

```
    {.instr_absolute =  {BRANCH, false, false,  NC,     13      }},  // 16
    {.instr_generic =   {RET,                                   }},  // 17
};

//! Taille utile du programme
const unsigned textsize = sizeof(text) / sizeof(Instruction);

//! Premier exemple de segment de données initial
Word data[20] = {
    0,  // 0
    0,  // 1: résultat
    20, // 2: premier opérande
    5,  // 3: second opérande
};

//! Fin de la zone de données utile
const unsigned dataend = 10;

//! Taille utile du segment de données
const unsigned datasize = sizeof(data) / sizeof(Word);
```

## 4.3 Grâce à un assembleur symbolique

Il est encore plus simple d'utiliser un assembleur symbolique. Un tel assembleur vous est fourni sous forme de l'exécutable `asm`. Pour l'utiliser, il suffit d'écrire un fichier source [6], d'extension `.asm`, disons `foo.asm`, et d'exécuter la commande **shell** :

```
asm foo.asm foo.bin
```

qui analysera le code et produira un fichier purement binaire `foo.bin`. Ce dernier pourra être directement lu par le simulateur grâce à l'option **-b** :

```
test_simul -b foo.bin
```

En outre l'assembleur vous produira sur la sortie standard un listing comportant l'affichage de la table des symboles ainsi que des instructions et des données de votre programme.

Toujours à titre d'exemple, voici le texte source assembleur du programme de 2.3.5 contenu dans le fichier `Examples/prog_subroutine.asm`).

```
//-----------------
// Instructions
//-----------------
        TEXT 30

        // Programme principal
main    EQU *
        PUSH @op1
        PUSH @op2
        CALL NC, @subprog
        ADD R15, #2
        STORE R00, @result
        HALT
```

6. Cet assembleur ne traite que des programmes tenant dans *un seul* fichier source

```
        NOP
        NOP
        NOP
        NOP

        // Sous-programme
subprog EQU *
        LOAD R00, 3[R15]
        LOAD R01, 2[R15]
        SUB R01, #1
loop    BRANCH LE, @return
        ADD R00, 3[R15]
        SUB R01, #1
        BRANCH NC, @loop
return  RET

        END

//-----------------
// Données et pile
//-----------------
        DATA 30

        WORD 0
result  WORD 0
op1     WORD 20
op2     WORD 5

        END
```

Le fichier `Examples/systax.asm` décrit par des exemples la syntaxe de cet assembleur.

**Attention !** Cet assembleur est fourni « tel quel », à titre de commodité. Vous n'êtes pas obligés de l'utiliser mais il peut vous simplifier la vie [7]. En revanche soyez conscients que ses messages d'erreur sont (très) incomplets et qu'il peut même générer du code illégal (qui se plantera donc à la simulation !).

## 4.4 Note sur l'ordre des champs de bits

Lorsque l'on compare, sur l'architecture INTEL, la valeur binaire générée (sous forme entière) pour une instruction et la description du format, on peut être surpris par l'ordre des octets.

Prenons l'instruction suivant : `LOAD R01, +3[R06]` (charger le registre 1 avec le contenu de la mémoire situé au déplacement 3 par rapport au contenu du registre 6). Son initialisation C99 a la forme suivante :

7. Inutile de dire que contrairement au simulateur, vous n'avez pas à le réécrire !

```
// type              cop     imm    index  regcond  rx        offset
{.instr_indexed =   {LOAD,   false, true,  1,       6,        +3   }}
```

où `LOAD` vaut 2. La transcription en entier non signé 32 bits de cet instruction peut vous surprendre: c'est `0x00036182`.

En effet, l'architecture INTEL étant « petit boutiste » (*little endian*, poids faibles en tête), les bits de cet entier sont **rangés à l'envers** en mémoire. Les champs de bits de l'instruction sont aussi alloués dans le même ordre que les adresses et les bits. On obtient donc

```
010000 0 1 1000 0110 1100000000000000
  COP  I X   R   RX      offset
```

Ici par exemple, le code opération `010000` a bien la valeur 2, codée en binaire avec les bits de poids faible en tête.

Si on regroupe les octets, on obtient

```
01000001 10000110 11000000 00000000
```

C'est donc cette dernière chaine de bits qui est rangée en mémoire.

Pour en afficher la valeur, if faut renverser l'ordre de tous ces bits (pour passer les poids forts en tête comme il est normal lorsqu'on affiche un entier) et passer, par exemple, en hexadécimal :

```
00000000 00000011 01100001 10000010
  0x00     0x03     0x61     0x82
```

On retrouve bien la valeur entière `0x00036182`. Ouf !

Donc, l'INTEL semble mettre tout à l'envers ! Nous survivrons à cette bizarrerie car c'est évidemment une question de pure convention. Mais il faut le savoir, car cela peut surprendre.

En fait, le problème ne se pose vraiment que

— pour écrire directement des instructions en binaire (voir 4.1) ; autant essayer d'éviter cela :

— pour interpréter les instructions du processeur qui sont affichées sous forme d'entiers mais après le regroupement des champs de bits décrits ci-dessus.

Pour les données entières elles-même (segment **Data**, valeurs des champs des instructions après déassemblage), les affichages hexadécimal ou décimal présentent toujours la valeur correctement.