SAE - TP1
Structures de contrôle en Assembleur 32 bits

Votre Nom

Version 1.0

Introduction

Ce TP a pour objectif de vous familiariser avec les structures de contrôle en assembleur GNU 32 bits (if/else, boucles, switch). Vous travaillerez uniquement avec les registres et la mémoire, sans utiliser de fonctions ni d'appels systèmes complexes.

Utilisation de SASM

Vous utiliserez le logiciel **SASM** (Simple ASM) pour tester vos programmes. SASM est un IDE qui vous permet de :

- Écrire et compiler votre code assembleur
- Placer des **breakpoints** (points d'arrêt) en cliquant sur la marge gauche
- Exécuter le code pas à pas (Step Over / Step Into)
- Observer en temps réel les valeurs des **registres** (%eax, %ebx, %ecx, etc.)
- Inspecter le contenu de la **mémoire**

Méthode de travail recommandée :

1. Écrivez votre code dans SASM
2. Placez un breakpoint au début de la boucle ou après un calcul important
3. Lancez le programme en mode débogage (F5)
4. Avancez pas à pas (F10) et observez les registres
5. Vérifiez que les valeurs correspondent à vos attentes

Configuration SASM :

- Mode : **GAS (GNU Assembler)**
- Architecture : **x86 (32 bits)**
- Build : **gcc -m32 -no-pie**

Important : Dans chaque exercice, des commentaires indiquent quels registres observer et à quel moment. Utilisez les breakpoints pour vérifier vos résultats à chaque étape !

Exercice 1 : Les conditions (if/else)

1.1 Maximum de deux nombres

Écrire un programme qui compare deux nombres stockés en mémoire (`x` et `y`) et place le maximum dans le registre `%eax`.

Données :

```
.data
x:  .int 15
y:  .int 23
```

Résultat attendu : `%eax` doit contenir 23

Test avec SASM :

— Placez un breakpoint sur la ligne `cmpl`
— Observez : `%eax = 15` et `%ebx = 23`
— Après le saut, vérifiez que `%eax = 23` (le maximum)

Solution :

```
.data
x:  .int 15
y:  .int 23

.text
.global main
main:
    movl %esp, %ebp          # for correct debugging

    movl x, %eax             # charger x dans eax
    movl y, %ebx             # charger y dans ebx
    # BREAKPOINT ICI : observer eax=15, ebx=23

    cmpl %ebx, %eax          # comparer eax et ebx
    jge fin                  # si eax >= ebx, aller à fin

    movl %ebx, %eax          # sinon, mettre ebx dans eax

fin:
    # BREAKPOINT ICI : observer eax=23 (le maximum)
    ret
```

1.2 Valeur absolue

Écrire un programme qui calcule la valeur absolue d'un nombre signé stocké dans la variable `nombre`.
Données :

```
.data
nombre:  .int -42
```

Résultat attendu : `%eax` doit contenir 42

Solution :

```
.data
nombre:  .int -42

.text
.global main
main:
    movl %esp, %ebp

    movl nombre, %eax    # charger le nombre

    cmpl $0, %eax        # comparer avec 0
    jge positif          # si >= 0, c'est déjà positif

    negl %eax            # sinon, inverser le signe

positif:
    ret
```

1.3 Nombre pair ou impair

Écrire un programme qui teste si un nombre est pair ou impair. Le résultat doit être :
— 0 dans `%eax` si le nombre est pair
— 1 dans `%eax` si le nombre est impair
Astuce : Utilisez l'instruction `andl` pour tester le bit de poids faible.

Solution :

```
.data
nombre:  .int 17

.text
.global main
main:
    movl %esp, %ebp

    movl nombre, %eax
    andl $1, %eax        # garder seulement le bit de poids faible
                         # si 0 -> pair, si 1 -> impair
    ret
```

1.4 Maximum de trois nombres

Écrire un programme qui trouve le maximum de trois nombres stockés dans `a`, `b` et `c`.
Données :

```
.data
a:  .int 45
b:  .int 78
c:  .int 23
```

Solution :

```
.data
a:  .int 45
b:  .int 78
c:  .int 23

.text
.global main
main:
    movl %esp, %ebp

    # Comparer a et b
    movl a, %eax
    movl b, %ebx
    cmpl %ebx, %eax
    jge comp_c           # si a >= b, comparer avec c
    movl %ebx, %eax      # sinon, eax = b

comp_c:
    # Maintenant eax contient max(a,b)
    movl c, %ecx
    cmpl %ecx, %eax
    jge fin              # si max(a,b) >= c, terminé
    movl %ecx, %eax      # sinon, eax = c

fin:
    ret
```

Exercice 2 : Les boucles

2.1 Somme de 1 à N

Écrire un programme qui calcule la somme de 1 à N (N=10). Le résultat doit être dans `%eax`.
Formule : $1 + 2 + 3 + ... + 10 = 55$

Test avec SASM :

- Placez un breakpoint dans la boucle (ligne `addl`)
- Avancez pas à pas et observez :
  - Itération 1 : `%eax = 1`, `%ecx = 1`
  - Itération 2 : `%eax = 3`, `%ecx = 2`
  - Itération 3 : `%eax = 6`, `%ecx = 3`
  - ...
  - Itération 10 : `%eax = 55`, `%ecx = 10`

Solution :

```
.data
n:  .int 10

.text
.global main
main:
    movl %esp, %ebp

    xorl %eax, %eax          # eax = 0 (somme)
    movl $1, %ecx            # ecx = 1 (compteur)
    movl n, %edx             # edx = n

boucle:
    cmpl %edx, %ecx          # comparer compteur avec n
    jg fin_boucle            # si compteur > n, sortir

    # BREAKPOINT ICI : observer eax (somme) et ecx (compteur)
    addl %ecx, %eax          # somme += compteur
    incl %ecx                # compteur++
    jmp boucle

fin_boucle:
    # BREAKPOINT ICI : observer eax=55
    ret
```

2.2 Factorielle

Écrire un programme qui calcule la factorielle de N (N=5).
Rappel : $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

Solution :

```
.data
n:  .int 5

.text
.global main
main:
    movl %esp, %ebp

    movl n, %ecx         # ecx = n (compteur)
    movl $1, %eax        # eax = 1 (résultat)
```

```
boucle:
    cmpl $1, %ecx        # si compteur <= 1, terminer
    jle fin_boucle

    imull %ecx, %eax     # résultat *= compteur
    decl %ecx            # compteur--
    jmp boucle

fin_boucle:
    ret                  # eax = 120
```

2.3 Puissance

Écrire un programme qui calcule $base^{exposant}$ (exemple : $2^8 = 256$).
Données :

```
.data
base:       .int 2
exposant:   .int 8
```

Solution :

```
.data
base:       .int 2
exposant:   .int 8

.text
.global main
main:
    movl %esp, %ebp

    movl exposant, %ecx # ecx = exposant (compteur)
    movl $1, %eax       # eax = 1 (résultat)
    movl base, %ebx     # ebx = base

    cmpl $0, %ecx       # cas spécial : exposant = 0
    je fin_boucle       # résultat = 1

boucle:
    cmpl $0, %ecx
    jle fin_boucle

    imull %ebx, %eax    # résultat *= base
    decl %ecx
    jmp boucle

fin_boucle:
    ret                 # eax = 256
```

2.4 Compter les multiples

Écrire un programme qui compte combien de multiples de 3 existent entre 1 et 30.
Résultat attendu : 10 (les multiples sont : 3, 6, 9, 12, 15, 18, 21, 24, 27, 30)

Solution :

```
.data
limite:     .int 30
diviseur:   .int 3
```

```
.text
.global main
main:
    movl %esp, %ebp

    xorl %eax, %eax      # eax = 0 (compteur de multiples)
    movl $1, %ecx        # ecx = 1 (nombre à tester)
    movl limite, %edi    # edi = limite

boucle:
    cmpl %edi, %ecx      # comparer avec la limite
    jg fin_boucle

    # Tester si ecx est multiple de diviseur
    movl %ecx, %ebx      # sauvegarder ecx
    xorl %edx, %edx      # préparer pour division
    movl %ecx, %eax
    divl diviseur        # eax = ecx / diviseur, edx = reste

    cmpl $0, %edx        # si reste = 0, c'est un multiple
    jne pas_multiple

    movl %ebx, %eax      # restaurer le compteur de multiples
    incl %eax            # incrémenter
    jmp suite

pas_multiple:
    movl %ebx, %eax      # restaurer le compteur

suite:
    movl %ebx, %ecx      # restaurer ecx
    incl %ecx            # prochain nombre
    jmp boucle

fin_boucle:
    # Note: cette solution est complexe, voici une version simplifiée
    ret
```

Solution simplifiée :

```
.text
.global main
main:
    movl %esp, %ebp

    xorl %eax, %eax      # compteur de multiples
    movl $3, %ecx        # commencer à 3

boucle:
    cmpl $30, %ecx
    jg fin_boucle

    incl %eax            # c'est un multiple de 3
    addl $3, %ecx        # passer au prochain multiple
    jmp boucle

fin_boucle:
    ret                  # eax = 10
```

2.5 PGCD (Plus Grand Commun Diviseur)

Écrire un programme qui calcule le PGCD de deux nombres en utilisant l'algorithme d'Euclide.
Données :

```
.data
a:  .int 48
b:  .int 18
```

Algorithme :

```
Tant que b != 0 :
    temp = b
    b = a mod b
    a = temp
PGCD = a
```

Résultat attendu : 6

Solution :

```
.data
a:  .int 48
b:  .int 18

.bss
temp: .lcomm 4

.text
.global main
main:
    movl %esp, %ebp

    movl a, %ebx        # ebx = a
    movl b, %ecx        # ecx = b

boucle:
    cmpl $0, %ecx       # si b = 0, terminé
    je fin_boucle

    # Calculer a mod b
    xorl %edx, %edx     # edx = 0
    movl %ebx, %eax     # eax = a
    divl %ecx           # eax = a/b, edx = a mod b

    # temp = b, b = a mod b, a = temp
    movl %ecx, %ebx     # a = b
    movl %edx, %ecx     # b = reste

    jmp boucle

fin_boucle:
    movl %ebx, %eax     # résultat dans eax
    ret                 # eax = 6
```

Exercice 3 : Les switch/case

En assembleur, un switch/case peut être implémenté de deux manières :
— Par des comparaisons successives (if/else if)
— Par une table de sauts (plus efficace pour de nombreux cas)

3.1 Jours de la semaine

Écrire un programme qui, étant donné un numéro de jour (1 à 7), retourne un code :
— 1-5 (lundi à vendredi) : retourner 1 (jour de travail)
— 6-7 (samedi, dimanche) : retourner 0 (weekend)
— Autre : retourner -1 (invalide)
Données :

```
.data
jour:  .int 3   # Mercredi
```

Solution :

```
.data
jour:  .int 3

.text
.global main
main:
    movl %esp, %ebp

    movl jour, %eax

    # Vérifier si jour valide (1-7)
    cmpl $1, %eax
    jl invalide
    cmpl $7, %eax
    jg invalide

    # Vérifier si weekend (6 ou 7)
    cmpl $6, %eax
    jge weekend

    # Sinon c'est un jour de travail
    movl $1, %eax
    jmp fin

weekend:
    movl $0, %eax
    jmp fin

invalide:
    movl $-1, %eax

fin:
    ret
```

3.2 Calculatrice simple

Écrire un programme qui effectue une opération selon un code opération :
— 1 : addition
— 2 : soustraction
— 3 : multiplication
— Autre : retourner 0
Données :

```
.data
operation:  .int 3   # multiplication
operande1:  .int 7
operande2:  .int 6
```

Solution :

```
.data
operation:  .int 3
operande1:  .int 7
operande2:  .int 6

.text
.global main
main:
    movl %esp, %ebp

    movl operation, %ecx
    movl operande1, %eax
    movl operande2, %ebx

    cmpl $1, %ecx
    je addition
    cmpl $2, %ecx
    je soustraction
    cmpl $3, %ecx
    je multiplication

    # Opération invalide
    xorl %eax, %eax
    jmp fin

addition:
    addl %ebx, %eax
    jmp fin

soustraction:
    subl %ebx, %eax
    jmp fin

multiplication:
    imull %ebx, %eax
    jmp fin

fin:
    ret                 # eax = 42 (7 * 6)
```

3.3 Table de sauts

Réécrire la calculatrice précédente en utilisant une table de sauts.

Solution :

```
.data
operation:  .int 2      # soustraction
operande1:  .int 15
operande2:  .int 8

# Table de sauts
jump_table:
    .long operation_invalide    # cas 0
    .long addition              # cas 1
    .long soustraction          # cas 2
    .long multiplication        # cas 3
```

```
.text
.global main
main:
    movl %esp, %ebp

    movl operation, %ecx
    movl operande1, %eax
    movl operande2, %ebx

    # Vérifier si opération valide (1-3)
    cmpl $1, %ecx
    jl operation_invalide
    cmpl $3, %ecx
    jg operation_invalide

    # Utiliser la table de sauts
    jmp *jump_table(,%ecx,4)    # sauter à l'adresse jump_table[ecx]

addition:
    addl %ebx, %eax
    jmp fin

soustraction:
    subl %ebx, %eax
    jmp fin

multiplication:
    imull %ebx, %eax
    jmp fin

operation_invalide:
    xorl %eax, %eax

fin:
    ret                 # eax = 7 (15 - 8)
```

Exercice 4 : Exercices de synthèse

4.1 Somme des nombres pairs

Écrire un programme qui calcule la somme de tous les nombres pairs de 1 à N.
Données :

```
.data
n:  .int 20
```

Résultat attendu : 2+4+6+8+10+12+14+16+18+20 = 110

4.2 Nombre de chiffres

Écrire un programme qui compte le nombre de chiffres dans un nombre.
Exemple : 12345 a 5 chiffres
Données :

```
.data
nombre:  .int 12345
```

Astuce : Diviser successivement par 10 jusqu'à ce que le nombre soit 0.

4.3 Somme des chiffres

Écrire un programme qui calcule la somme des chiffres d'un nombre.
Exemple : 1234 → 1+2+3+4 = 10
Données :

```
.data
nombre:  .int 1234
```

Solution :

```
.data
nombre:  .int 1234
dix:     .int 10

.text
.global main
main:
    movl %esp, %ebp

    movl nombre, %ebx    # nombre à traiter
    xorl %edi, %edi      # somme = 0

boucle:
    cmpl $0, %ebx        # si nombre = 0, terminé
    je fin_boucle

    # Extraire le dernier chiffre
    xorl %edx, %edx
    movl %ebx, %eax
    divl dix             # eax = nombre/10, edx = nombre%10

    addl %edx, %edi      # ajouter le chiffre à la somme
    movl %eax, %ebx      # nombre = nombre/10

    jmp boucle

fin_boucle:
    movl %edi, %eax      # résultat dans eax
    ret                  # eax = 10
```

4.4 Inverser un nombre

Écrire un programme qui inverse les chiffres d'un nombre.
Exemple : 1234 → 4321
Données :

```
.data
nombre:  .int 1234
```

Solution :

```
.data
nombre:  .int 1234
dix:     .int 10

.text
.global main
main:
    movl %esp, %ebp
```

```
    movl nombre, %ebx    # nombre à inverser
    xorl %edi, %edi      # résultat = 0

boucle:
    cmpl $0, %ebx        # si nombre = 0, terminé
    je fin_boucle

    # Extraire le dernier chiffre
    xorl %edx, %edx
    movl %ebx, %eax
    divl dix             # eax = nombre/10, edx = chiffre

    # Ajouter le chiffre au résultat
    imull dix, %edi      # résultat *= 10
    addl %edx, %edi      # résultat += chiffre

    movl %eax, %ebx      # nombre = nombre/10
    jmp boucle

fin_boucle:
    movl %edi, %eax      # résultat dans eax
    ret                  # eax = 4321
```

4.5 Nombre de bits à 1

Écrire un programme qui compte le nombre de bits à 1 dans un nombre.
Exemple : 13 = 0b1101 → 3 bits à 1
Données :

```
.data
nombre:  .int 13
```

Astuce : Utiliser un décalage à droite et tester le bit de poids faible.

Solution :

```
.data
nombre:  .int 13

.text
.global main
main:
    movl %esp, %ebp

    movl nombre, %ebx    # nombre à traiter
    xorl %ecx, %ecx      # compteur = 0

boucle:
    cmpl $0, %ebx        # si nombre = 0, terminé
    je fin_boucle

    # Tester le bit de poids faible
    movl %ebx, %eax
    andl $1, %eax        # isoler le bit de poids faible
    addl %eax, %ecx      # ajouter au compteur

    shrl $1, %ebx        # décalage à droite de 1 bit
    jmp boucle

fin_boucle:
    movl %ecx, %eax      # résultat dans eax
```

```
    ret                 # eax = 3
```

Exercice 5 : Défis

5.1 Nombre parfait

Un nombre parfait est un nombre égal à la somme de ses diviseurs (excluant lui-même). Exemple : $6 = 1 + 2 + 3$

Écrire un programme qui teste si un nombre est parfait. Retourner 1 si oui, 0 si non.

5.2 Suite de Fibonacci

Écrire un programme qui calcule le n-ième terme de la suite de Fibonacci.
Rappel : F(0)=0, F(1)=1, F(n)=F(n-1)+F(n-2)
Exemple : F(8) = 21

Solution :

```
.data
n:  .int 8

.text
.global main
main:
    movl %esp, %ebp

    movl n, %ecx        # ecx = n

    # Cas de base
    cmpl $0, %ecx
    je cas_zero
    cmpl $1, %ecx
    je cas_un

    # Calculer Fibonacci
    movl $0, %ebx       # F(n-2) = 0
    movl $1, %eax       # F(n-1) = 1
    movl $2, %edi       # compteur = 2

boucle:
    cmpl %ecx, %edi     # si compteur > n, terminé
    jg fin_boucle

    movl %eax, %edx     # sauvegarder F(n-1)
    addl %ebx, %eax     # F(n) = F(n-1) + F(n-2)
    movl %edx, %ebx     # F(n-2) = ancien F(n-1)

    incl %edi
    jmp boucle

cas_zero:
    xorl %eax, %eax
    jmp fin

cas_un:
    movl $1, %eax
    jmp fin

fin_boucle:
```

```
fin:
    ret                 # eax = 21
```

5.3 Conversion binaire vers décimal

Écrire un programme qui convertit un nombre stocké comme une suite de bits (0 et 1) en décimal.
Exemple : Si on stocke les bits 1, 0, 1, 1 (représentant 0b1011), le résultat doit être 11.