Module 4: Langage assembleur (suite 1)

1 Introduction

Nous avons fait, dans la leçon précédente, une introduction au langage assembleur, et nous avons présenté les différents registres du processeur 8086. Nous avons également présenté les caractéristiques d'un programme assembleur avec quelques instructions relatives au transfert de données. Nous avons terminé la leçon par présenter les instructions relatives aux opérations arithmétiques dans le langage assembleur.

Dans cette leçon, nous allons continuer avec les instructions du langage assembleur. Nous allons aborder les instructions relatives aux ruptures de séquence ou branchements, les décalages et rotations, et finalement les instructions d'entrées/sorties.

2 Ruptures de séquence

Nous avons vu précédemment que les processeurs sont dotés d'instructions permettant les exécutions différenciées en fonction des contions. Ces instructions sont appelées les sauts conditionnels. Les conditions sont souvent des tests effectués sur les valeurs de registres.

Les sauts conditionnels impliquent généralement un changement dans les registres d'état comme le compteur ordinal (PC). Nous allons voir à travers les instructions, ci-dessous, comment les sauts conditionnels sont traduits en assembleur.

Notez que nous avons également les sauts inconditionnels pour lesquels nous n'avons pas de tests à effectuer. Vous allez, donc, voir dans les points suivants des exemples de sauts conditionnels et inconditionnels.

- JMP adr : saut inconditionnel vers l'adresse adr (JMP : JuMP). Ici l'adresse de saut est indiquée explicitement. Puisque la prochaine instruction à exécuter se trouve à l'adresse adr, par conséquent le compteur ordinal PC reçoit la valeur adr (PC = adr).
- JZ AX, adr : saut conditionnel vers l'adresse adr si le registre AX est nul (JZ : Jump if Zero).
- JNZ AX, adr : saut conditionnel vers l'adresse adr si le registre AX est non nul (JNZ : Jump if Not Zero).
- JGT AX, adr : saut conditionnel vers l'adresse adr si le registre AX est strictement positif (JGT : Jump if Greater Than zero).

— JLT AX, adr : saut conditionnel vers l'adresse adr si le registre AX est strictement négatif (JLT : Jump if Less Than zero).
— JGE AX, adr : saut conditionnel vers l'adresse adr si le registre AX est positif ou nul (JGE : Jump if Greater or Equal to zero).
— JLE AX, adr : saut conditionnel vers l'adresse adr si le registre AX est négatif ou nul (JLE : Jump if Less or Equal to zero).

Il existe également des instructions de manipulation du registre d'état, ces instructions sont aussi appelées des instructions de transfert de bit de condition. Ces instructions sont présentées comme suit [1] :

— CCR AX : le bit C (bit de retenue) du registre d'état est mis dans le bit de poids faible de AX et ses autres bits sont mis à 0 (CCR : Copy bit C into Register). Nous pouvons donc écrire : AX = bit C.
— CZR AX : le bit Z (Zero) du registre d'état est mis dans le bit de poids faible de AX et ses autres bits sont mis à 0 (CZR, Copy bit Z into Register). Nous pouvons donc écrire : AX = bit Z.
— CNR AX : le bit N (Negative) du registre d'état est mis dans le bit de poids faible de AX et ses autres bits sont mis à 0 (CNR, Copy bit N into Register). Nous pouvons donc écrire : AX = bit N.
— CVR AX : le bit V (oVerflow) du registre d'état est mis dans le bit de poids faible de AX et ses autres bits sont mis à 0 (CVR, Copy bit V into Register). Nous pouvons donc écrire : AX = bit V.

Les instruction de branchement s'exécutent de la façon suivante :

— Une comparaison du registre avec la valeur 0.
— Si le résultat est vrai, le processeur effectue le saut et exécute l'instruction se trouvant dans l'adresse indiquée par l'instruction de branchement.
— Si le résultat est faux, le processeur continue donc l'exécution avec l'instruction qui se trouve juste après le saut.

Un exemple de saut conditionnel est présenté ci-dessous.

```
; Exemple de branchement conditionnel
MOV     AL, 25      ; mettre le registre AL à 25.
MOV     BL, 10      ; mettre le registre BL 10.

CMP     AL, BL      ; comparer AL et BL en effectuant la ↩
    soustraction AL - BL.

JE      egale       ; effectuer un saut si AL = BL.

PUTC    'N'         ; si le programme est rendu là, alors AL ↩
    <> BL, imprimer 'N'.

egale:              ; si le programme est rendu là,
PUTC    'Y'         ; alors AL = BL, donc imprimer 'Y'.

END
```

1. Ces instructions sont tirées du livre d'architecture de l'ordinateur d'Emmanuel Lazard

3 Décalages et rotations

Nous avons vu précédemment que le décalage et la rotation sont des opérations de grande importance et qui peuvent s'exécuter de façon très rapide sur les registres pour des opérations de multiplication ou de division sur 2. Dans ce qui suit, nous allons voir comment ces opérations sont traduites en langage assembleur.

Il existe bien évidemment plusieurs instructions de décalage et rotation qui sont résumées comme suit :

- LRT AX, BX, CX : le registre AX reçoit la valeur du registre BX décalée par rotation CX fois vers la gauche. Nous avons vu dans les cours précédents comment le décalage et la rotation sont effectués. (LRT, Left RotaTe). Nous pouvons donc écrire : AX = bit V. Nous pouvons également écrire cette instruction en spécifiant le nombre de fois (i fois) de décalage par rotation comme suit : LRT AX, BX, #i
- LLS AX, BX, CX : le registre AX reçoit la valeur du registre BX décalée CX fois vers la gauche. Les nouveaux bits seront donc remplacés par des 0 lors du décalage. (LLS, Logical Left Shift).
- ALS AX, BX, CX : le registre AX reçoit la valeur du registre BX décalée CX fois vers la gauche. Les nouveaux bits sont remplacés par des 0 lors d'un décalage vers la gauche (comme l'instruction LLS) our par le bit de poids fort lors d'un décalage à droite. (ALS, Arithmetical Left Shift).

Rappelons que les instructions de décalage et de rotation s'exécutent de la même façon que les autres instructions. Si la valeur du registre CX (ou la valeur immédiate i) est positive, le décalage ou la rotation s'effectue vers la gauche. Sinon, le décalage ou la rotation s'effectue vers la droite.

Rappelons que le décalage vers la gauche (valeur de CX ou i positive) correspond à une multiplication par 2, et donc les bits de poids faible sont remplacés par des 0. Dans ce cas de figure, les deux instructions LLS et ALS seront équivalentes.

Cependant, le décalage logique vers la droite (valeur de CX ou i négative) consiste donc à remplacer les bits de poids fort par des 0. Cela ne correspond pas à une division par 2 à cause des nombres signés. Car nous devons conserver le bit de poids fort lors d'un décalage arithmétique vers la droite pour garder le signe du nombre (instruction ALS).

4 Instructions d'entrées/sorties

Afin de communiquer avec un périphérique, l'accumulateur est AL pour le transfert d'un octet, AX pour le transfert d'un mot, et EAX pour un double mot. Deux scénarios se présentent :

1. Lorsque le numéro de port est une valeur octet (0 à 255), elle peut être spécifiée comme une valeur immédiate dans l'instruction. Nous pouvons à ce moment là écrire : IN AL, 250 qui permet de lire le port numéro 250.

Pour écrire, nous pouvons écrire : OUT PORT_P, AX qui permet d'écrire un mot de donnée contenu dans le registre AX dans le port PORT_P.

2. Si c'est une valeur mot (0 à 64K), elle doit être chargée au préalable dans le registre DX, c'est toujours plus commode de la changer facilement. Donc, nous pouvons écrire : IN AL, DX qui permet de lire le port dont le numéro se trouve dans le registre DX, et OUT DX, AX qui permet d'écrire un mot au port dont le numéro se trouve dans le registre DX.

Dans les instructions d'entrées/sorties, nous devons spécifier d'abord le nom de l'accumulateur suivi par le nom du port où écrire dans le cas de lecture (IN Accumulateur, port), et le nom du port suivi par l'accumulateur dans le cas d'écriture (OUT port, Accumulateur).

4.1 Instructions de boucle

Les instructions de boucle sont très importantes lors des tâches répétitives par exemple, le parcours d'un tableau. Il existe plusieurs instructions de boucle qui peuvent être utilisées dans des contextes différents.

1. LOOP : L'instruction LOOP décrémente le registre CX par 1 à chaque boucle et ne passe à l'instruction suivante que si CX = 0. Par exemple :

```
1        ; Exemple de LOOP
2        MOV CX, 10       ; charger le compte de ↩
             bouclage dans CX.
3    BOUCLE: ...          ; ici on met les instructions ↩
         à répéter.
4        LOOP BOUCLE      ; si CX est différent de 0 ↩
             sauter à BOUCLE,
5        ...              ; sinon continuer.
6        END
```

2. LOOPE/LOOPZ : Cette instruction décrémente le registre CX par 1 et saute si CX est différent de 0 et ZF = 1. Elle continue à boucler jusqu'à ce que CX = 0 ou ZF = 0. Elle sert généralement à trouver un résultat non nul dans une série d'opérations.
3. LOOPNE/LOOPNZ : Cette instruction décrémente le registre CX par 1 et saute si CX est différent de 0 et ZF = 0. Elle continue à boucler jusqu'à ce que CX = 0 ou ZF = 1. Elle sert généralement à trouver un résultat nul dans une série d'opérations.
4. LOOPW/LOOPWE/LOOPWNE/JCXZ : (décrémentation de CX). LOOPD/-LOOPDE/LOOPDNE/JECXZ : (décrémentation de ECX) : De façon générale, LOOP/LOOPE/LOOPNE s'utilisent de façon implicite avec le registre CX comme compteur de boucle dans les segments 16 bits et avec

le registre ECX comme compteur de boucle dans les segments 32 bits. Sinon, on pourrait spécifier explicitement CX en adjoignant à LOOP soit W soit D. Par exmeple :

```
1         MOV DX, PORT_ENT
2         MOV CX, 100H
3 BOUCLE: IN AL, DX
4         CMP AL, 0AH       ; 0AH en hexadécimal est 10 ↩
              en décimal.
5         LOOPDNE BOUCLE    ; teste CX et n'affecte pas ↩
              les flags.
6         JNZ SAUT1         ; teste le résultat de la ↩
              comparaison.
7         ...
8 SAUT1:  ...
9         END
```

4.2 Instructions de chaîne

Les instructions de chaîne permettent de travailler sur des blocs d'octets ou de mots, allant jusqu'à 64 Koctets et pouvant être des valeurs numériques ou alphanumériques. Les instructions de chaîne permettent cinq sortes d'opérations de base (appelées aussi primitives) qui sont : le transfert, le chargement, la comparaison, la lecture et l'écriture.

Notez que l'adresse de la source est lue à partir des registres DS :ESI ou DS :SI en fonction de l'attribut de la taille d'adresse dans l'instruction (32 ou 16 bits respectivement). L'adresse de la destination est lue à partir des registres ES :EDI ou ES :DI en fonction de l'attribut de la taille d'adresse dans l'instruction (32 ou 16 bits respectivement).

— MOVS : move string, MOVSB : move byte string, MOVSW : move word string, MOVSD : move double word string. Ces instructions permettent de copier respectivement des données octet, mot ou double mot de la chaîne source vers la chaîne destination. Ces instructions demandent que la source (DS :SI) et l'arrivée (ES :DI) soient configurées. Il ne faut pas diriger les données vers n'importe quel emplacement de la mémoire. Par exemple dans DS :SI, nous pouvons avoir les données d'une image et dans ES :DI, l'adresse de la mémoire vidéo (A000h :0000). Si nous voulons copier 10000 bytes, nous emploierons directement le MOVSD.
Dans le cas où plusieurs MOVS, MOVSB, MOVSW, ou MOVSD sont prévus, il est inutile d'utiliser les boucles. L'instruction REP est faite pour ça. Cependant, il faut utiliser conjointement avec REP, le registre CX qui contient le nombre d'itérations (le nombre de fois que l'on répète

l'instruction MOVS). Par exemple, Pour les 10000 bytes, nous avons [2] :

```
1         CLD     ; Cette instruction, clear direction flag, met l'indicateur d'état DF à 0 (voir note en bas).
2         MOV CX, 10000H
3         REP MOVSB
4         ;Si nous utilisons les Words (plus rapide), c'est :
5         MOV CX, 5000 (un word = 2 bytes)
6         REP MOVSW
7         ; Finalement, avec le 32 bits, c'est :
8         MOV ECX,2500 (un dword = 2 words = 4 bytes) - utilisation d'un registre étendu
9         REP MOVSD
```

L'indicateur d'état DF contrôle la direction du traitement d'une chaîne de gauche à droite ou de droite à gauche.

Notez qu'à chaque instruction de transfert de chaîne, le registre DI augmente de 1, 2 ou 4 bytes selon l'instruction utilisée.

— STOS : store string, STOSB : store byte, STOSW : store word, STOSD : store double word : ces instructions permettent de placer des données dans un emplacement de la mémoire. Comme pour les instruction de transfert (MOVSx), ces instructions stockent un byte, un word et un double word. A la différence des déplacements, les STOSx utilisent AL, AX (EAX 32bits) comme valeur à stocker. La valeur de AL, AX, EAX sera stockée dans ES :DI. À chaque STOSx, DI est augmenté de 1,2 ou 4 bytes. Ces instructions sont utiles pour effacer l'écran par exemple ou remplir un bloc mémoire avec la même valeur. Par exemple,

```
1         MOV CX, 1000
2         MOV AX, 0
3         REP STOSB
```

Dans cette exemple, CX équivaut à 1000, pour 1000 répétitions. Ensuite, le MOV AX, 0 qui est la valeur qui sera placée dans le bloc mémoire. Pour finir, le REP STOSB qui dit que nous plaçons 1000 bytes de valeur 0 à l'emplacement de ES :DI. En fait, ici nous plaçons AL, on ne tient pas compte de AH dans un STOSB.

— CMPS : compare string, CMPSB : compare byte, CMPSW, compare word, CMPSD : compare double word : ces instructions permettent d'effectuer des comparaisons d'un élément (octet, mot, double-mot) du segment de données pointé par le registre SI à l'élément correspondant pointé

2. Exemple tiré de `https://repo.zenk-security.com/Reversing%20.%20cracking/Assembleur.pdf`

par le registre DI dans le segment appelé segment extra. Comme nous l'avons vu précédemment, afin de comparer plus de 2 éléments entre eux, on devra appliquer l'instruction REPEAT (REP) à l'instruction CMPS. Puisque REP retourne seulement l'état des flags des 2 derniers éléments, il faudra utiliser soit REPE/REPZ ou REPNE/REPNZ. Par exemple :

```
1    REPE CMPS DESTINATION, SOURCE
```

Dans cet exemple, on compare en décrémentant CX jusqu'à 0, tant que ZF = 1, et dès qu'un élément de DEST ne coïncide pas (ZF = 0) avec l'élément correspondant de SOURCE, on sort.

```
1    REPNE CMPS DESTINATION, SOURCE
```

Dans cet exemple, on compare élément à élément jusqu'à ce qu'on trouve que 2 éléments coïncident (ZF = 1) alors on sort, sinon on continue jusqu'à CX = 0.
Comme une comparaison répétée peut s'arréter sur l'une des 2 conditions CX=0 ou ZF = 0/1 (REPE/REPNE), il faudrait faire suivre l'instruction COMPSx par une instruction de branchement JE/JZ ou JNE/JNZ, afin de tester ZF. Par exemple,

```
1    REPNE   CMPS DESTINATION, SOURCE
2            JNE NON_TROUVE
3            ...
4    NON_TROUVE: ...
5                ...
```

— LODS : load string, LODSB : load byte, LODSW, load word, LODSD : load double word : ces instructions permettent de transférer une chaîne source (pointée par SI/ESI) du segment de donnée à l'accumulateur AL (pour un octet), AX (pour un mot) et EAX (pour un double mot) et d'incrémenter SI/ESI respectivement de 1, 2 et 4.

— INS/INSB/INSW/INSD : INput from port to String, OUTS/OUTSB/OUTSW/OUTSD : OUTput String to port : les instructions INS et OUTS sont des instructions extrêment rapides qu'il faut obligatoirement utiliser lorsque la machine le permet. INS fait entrer plusieurs octets/mots/double mots à partir d'un port d'entrées/sorties pointé par le registre DX pour les ranger dans un tableau mémoire pointé par ES :DI ; soit par incrémentation (DF=0) de DI par 1/2/4 (octet/mot/double mot) soit par décrémentation (DF=1). Donc, on peut spécifier les octets avec INSB/OUTSB, les mots avec INSW/OUTSW, et double mots avec INSD/OUTSD comme le montre l'exemple du transfert, via TABLE, des données de PORT_IN à PORT_OUT :

```
1    CLD
2    MOV AX, SEG TABLE    ; l'adresse du segment ↩
```

```
        contenant TABLE.
    MOV DS, AX
    MOV ES, AX
    MOV DX, PORT_IN
    LEA DI, TABLE   ; transférer l'adresse effective ↩
        de TABLE dans le registre DI (LEA: Load ↩
        Effective Address).
    MOV CX, SIZE TABLE
    REP INSB        ; itérer jusqu'à ce que CX = 0 (↩
        lecture).
    MOV DX, PORT_OUT
    LEA SI, TABLE
    MOV CX, SIZE TABLE
    REP OUTSB       ; itérer jusqu'à ce que CX = 0 (é↩
        criture).
```