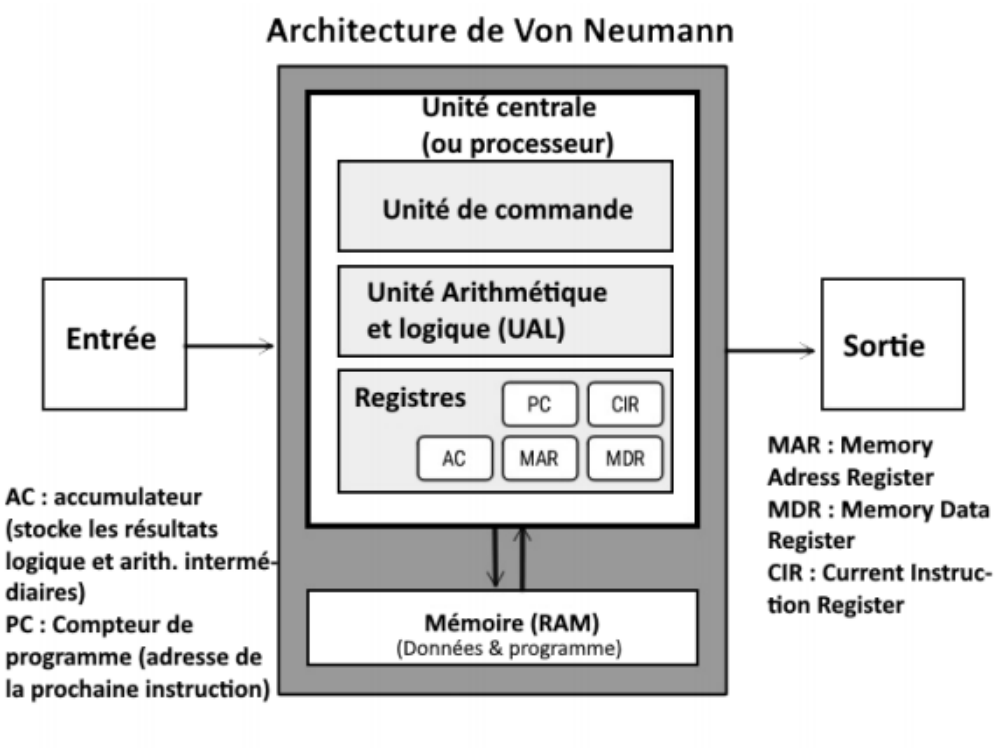
Initiation à l'assembleur 8086 sur système GNU/Linux

D'après Rémi HOARAU

Partie III : Registres et syscalls (Appels systèmes)

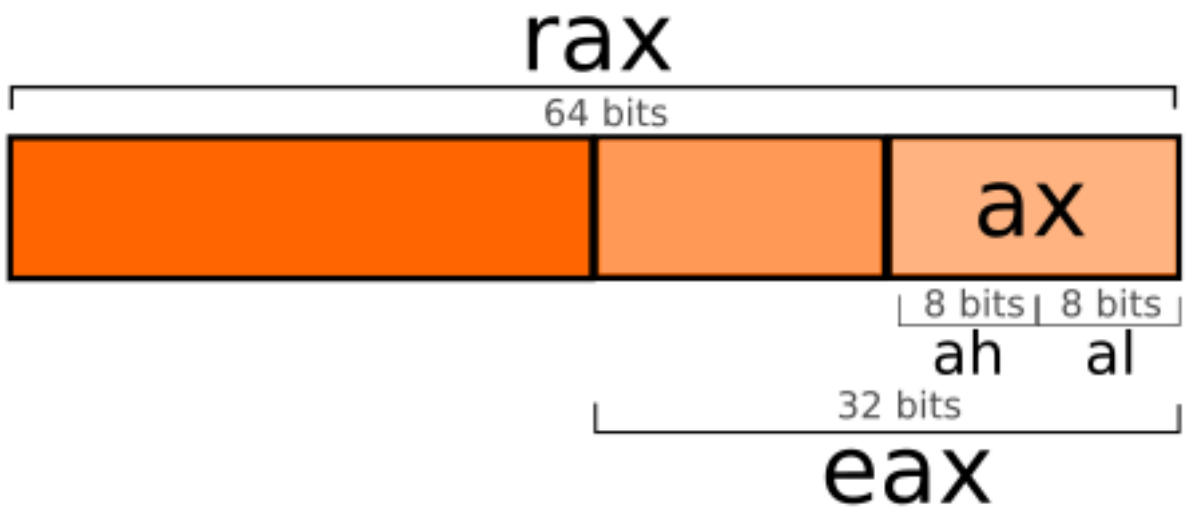
I. Notions de registres

Souvenons-nous de ce schéma :



Les registres sont situés directement dans le processeurs. Ce sont des petits espaces mémoires que l'on peut considérer comme "espaces de travail". Chaque registre a sa propre fonction, mais certains de ces registres peuvent être utilisés un peu n'importe comment...

Voici la structure d'un registre :



En prenant l'exemple du registre accumulateur, nous avons:

AL : la partie basse du registre **AX** représentant 8 bits (1 octet)

AH : la partie haute du registre **AX** représentant 8 bits (1 octet)

AX : le registre de 16 bits (2 octets => 1 word => **AH+AL**)

Attention, **AX** n'est pas disponible en 8 bits.

EAX : le registre "étendu" de 32 bits (4 octets => 1 dword => **AX+AX**).

Attention, **EAX** n'est pas disponible en 16 bits ni 8 bits.

RAX : le registre de 64 bits (8 octets => 1 qword => **EAX+EAX**).

Attention, **RAX** n'est pas disponible en 32 bits, ni 16 bits, ni 8 bits

Voici les registres les plus importants :

Register	16 bits	32 bits	64 bits	Type
Accumulator	AX	EAX	RAX	General
Base	BX	EBX	RBX	General
Counter	CX	ECX	RCX	General
Data	DX	EDX	RDX	General
Source Index	SI	ESI	RSI	Pointer
Target Index	DI	EDI	RDI	Pointer
Base Pointer	BP	EBP	RBP	Pointer
Top Pointer	SP	ESP	RSP	Pointer
Instruction Pointer	IP	EIP	RIP	Pointer
Code Segment	CS	-	-	Segment

AX : Registre d'accumulateur, il charge en mémoire le numéro des appels systèmes, il contient la valeur de retour d'une fonction appelée et il est utilisé comme résultat de n'importe quelle opération mathématique, tels que la division ou la multiplication.

BX : Registre de base, il est souvent utilisé comme un pointeur de données

CX : Registre compteur, il est souvent utilisé comme compteur de boucle

DX: Registre de données, il est utilisé dans des opérations mathématiques tel que le reste de la division euclidienne, dans des opérations d'entrée et de sorties ou encore comme registre étendu à **AX** pour former le double registre **DX:AX**

SI: Registre d'indice de source, il est utilisé comme pointeur d'une source dans des opérations de flux, et contient souvent l'adresse d'une chaîne de caractères

DI : Registre d'indice de destination, il est utilisé comme pointeur d'une destination dans des opérations de flux.

BP : Registre pointeur de base, il est utilisé comme "ancre" dans la pile afin de pouvoir par exemple sélectionner les variables locales d'une fonction et permet à la machine de savoir où se situe la fin de la pile d'une fonction.

SP : Registre pointeur de pile, il est utilisé comme pointeur du haut de la pile et permet à la machine de savoir quelle est la dernière valeur de cette dernière. À chaque push, **SP** est décrémenté d'un certain nombre d'octets. Incrémenter **SP** revient à diminuer la taille de la pile.

IP : Registre d'instructions, il est aussi souvent appelé **PC** (Program Counter) et il contient l'adresse de la prochaine instruction à exécuter. Lorsqu'une fonction est appelée, l'adresse de l'instruction suivant l'appel de la fonction est push dans la pile. Une fois la sortie de la fonction, l'adresse qui a été push dans la pile est pop par IP pour pouvoir exécuter la suite des instructions suivant l'appel.

CS : Segment pointeur de code, il pointe tout simplement vers le début du code (attention c'est un segment, pas un registre)

DS : Segment pointeur de données, il pointe tout simplement vers le début des données (attention c'est un segment, pas un registre)

C’est peut-être assez flou pour nous au début, mais il faut savoir que l’on peut utiliser ces registres un peu n’importe comment… à quelques exceptions près.

La liste que nous allons voir représente les registres préservés qui ont besoin de restaurer leurs anciennes valeurs après les avoir utilisés, et des registres qui n’ont pas besoin de restaurer leurs anciennes valeurs :

Name	**Notes**	Type	**64-bit long**	**32-bit int**	**16-bit short**	**8-bit char**
rax	Values are returned from functions in this register.	scratch	rax	eax	ax	ah and al
rcx	Typical scratch register. Some instructions also use it as a counter.	scratch	rcx	ecx	cx	ch and cl
rdx	Scratch register.	scratch	rdx	edx	dx	dh and dl
rbx	*Preserved register: don't use it without saving it!*	*preserved*	*rbx*	*ebx*	*bx*	*bh and bl*
rsp	*The stack pointer. Points to the top of the stack (details coming soon!)*	*preserved*	*rsp*	*esp*	*sp*	*spl*
rbp	*Preserved register. Sometimes used to store the old value of the stack pointer, or the "base".*	*preserved*	*rbp*	*ebp*	*bp*	*bpl*
rsi	Scratch register. Also used to pass function argument #2 in 64-bit Linux	scratch	rsi	esi	si	sil
rdi	Scratch register. Function argument #1 in 64-bit Linux	scratch	rdi	edi	di	dil
r8	Scratch register. These were added in 64-bit mode, so they have numbers, not names.	scratch	r8	r8d	r8w	r8b
r9	Scratch register.	scratch	r9	r9d	r9w	r9b
r10	Scratch register.	scratch	r10	r10d	r10w	r10b
r11	Scratch register.	scratch	r11	r11d	r11w	r11b
r12	*Preserved register. You can use it, but you need to save and restore it.*	*preserved*	*r12*	*r12d*	*r12w*	*r12b*
r13	*Preserved register.*	*preserved*	*r13*	*r13d*	*r13w*	*r13b*
r14	*Preserved register.*	*preserved*	*r14*	*r14d*	*r14w*	*r14b*
r15	*Preserved register.*	*preserved*	*r15*	*r15d*	*r15w*	*r15b*

Comment sauvegarder une valeur et la retourner ? C’est simple, en utilisant la pile. Avant de modifier un registre qui est préservé, on push ce registre dans la pile, on fait toutes nos modifications et une fois terminé, on rend l’ancienne valeur du registre en poppant le registre de la pile. Nous verrons comment utiliser la pile un peu plus tard…

II. Syscalls

Le syscall (system call => appel système) est le moment où le programme s'interrompt pour demander au système d'exploitation d'accomplir une certaine tâche. Parmi les plus utilisés on retrouve write (qui permet d'écrire un certain nombre d'octets dans un fichier), open (qui permet d'ouvrir un fichier selon certains modes de lecture/écriture) ou encore read (qui lit des octets dans un fichier).

Dans notre code, souvenons-nous, nous lisons à la ligne 14 et 17 :

```
int 0x80
```

En fait, le mot clé '**int**' (interrupt) représente une interruption système. Une interruption possède un numéro entre 0 et 255 et chaque interruption s'occuppe d'une fonction particulière. Par exemple **INT 0x00** se charge de traiter l'erreur survenue lors d'une division par 0, ou encore **INT 0x03** correspond à un point d'arrêt (breakpoint) dans le code. Eh bien, **INT 0x80**, c'est une interruption pour appeler un appel système.

Voici le lien vers un site qui regroupe tous les appels sytèmes ainsi que leurs arguments :

https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86-32_bit

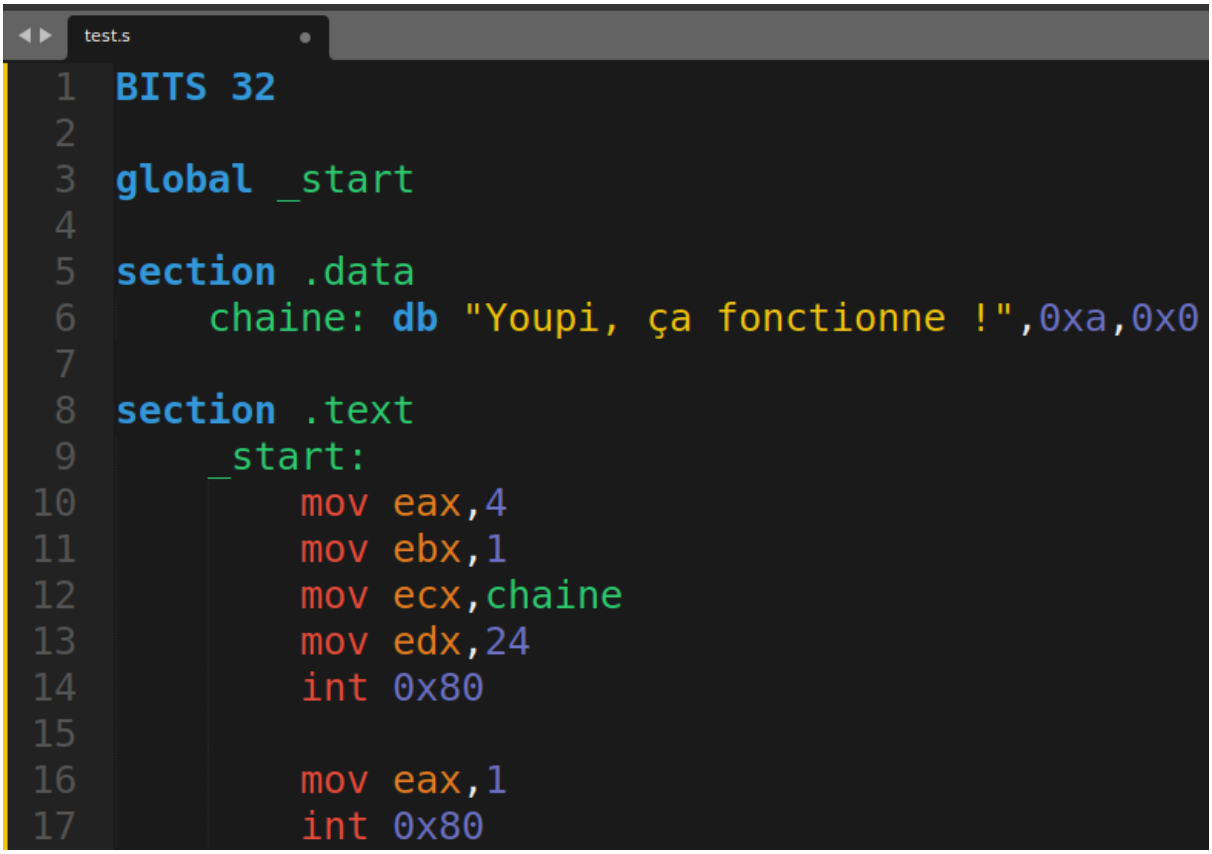
Nous noterons qu'en fonction de l'architecture choisie, les numéros des appels systèmes (placés dans AX) ne sont pas les mêmes. Il en va de même pour la position des arguments à charger dans les registres...

Nous allons voir très rapidement une instruction permettant de placer des valeurs dans des registres ou dans la mémoire. C'est l'instruction **mov**.

Si j'écris **mov A,B** c'est exactement comme si j'écrivais **A=B**.

Il y a d'autres subtilités mais j'en parlerai au prochain chapitre.

Eh bien voilà, nous avons tous les éléments nécessaires pour comprendre le code que nous avons écrit précédemment !



À partir de la ligne 10, on retrouve ces instructions :

```
mov eax,4
mov ebx,1
mov ecx,chaine
mov edx,24
int 0x80
```

On sait que l'on peut directement traduire ces instructions par :

```
EAX = 4
EBX = 1
ECX = chaine                ; (attention, ECX vaut l'adresse de la chaine, qui
                            ; représente un tableau de caractères)
```

EDX = 24

Pour **EAX = 4**, le tableau nous indique que c'est le syscall write :

x86 (32-bit)

Compiled from Linux 4.14.0 headers.

NR	syscall name	references	%eax	arg0 (%ebx)	arg1 (%ecx)	arg2 (%edx)	arg3 (%esi)	arg4 (%edi)	arg5 (%ebp)
0	restart_syscall	man/ cs/	0x00	-	-	-	-	-	-
1	exit	man/ cs/	0x01	int error_code	-	-	-	-	-
2	fork	man/ cs/	0x02	-	-	-	-	-	-
3	read	man/ cs/	0x03	unsigned int fd	char *buf	size_t count	-	-	-
4	write	man/ cs/	0x04	unsigned int fd	const char *buf	size_t count	-	-	-
5	open	man/ cs/	0x05	const char *filename	int flags	umode_t mode	-	-	-

Donc on écrit quelque chose. Mais quoi ? Où ? Combien d'octets on écrit ? Tout est indiqué sur la table.

EBX = 1 donc le file descriptor (descripteur de fichier) vaut 1, ce qui représente la sortie standard (stdout), c'est à dire le shell. (0 => stdin / entrée standard, 1 => stdout / sortie standard, 2 => stderr / erreur standard)

ECX = chaine donc **ECX** est chargé par l'adresse du tableau de caractères à afficher.

EDX = 24, on affiche alors 24 octets (en comptant les deux derniers octets 0xa ('\n') et 0x0 ('\0'))

On peut écrire le syscall comme suit :

```
write(stdout, &chaine, 24);                // Langage C / C++
```

En suivant le même procédé, nous remarquons que les lignes 16 et 17 représentent le syscall **exit**, qui se charge de quitter convenablement le programme.

Remarque : L'appel système **exit** est primordial dans un programme puisqu'il permet d'indiquer à la machine que le programme est censé s'arrêter. Essayons d'enlever l'appel système **exit** de notre code, de tout réassembler et d'exécuter le programme... *Erreur de segmentation* ! Bah oui ! Puisqu'on a jamais indiqué à la machine quand est ce que le programme s'arrêtait, le pointeur d'instructions (**IP** ou **PC**) va pointer vers la prochaine instruction à exécuter.... sans jamais s'arrêter... et peut rencontrer n'importe quelle instruction : par exemple il suffit d'un déréférencement d'un pointeur sur une adresse invalide et tout crash...

Ben voilà, on sait charger les registres avec différentes données et on sait utiliser les appels systèmes : on peut déjà faire beaucoup de choses !

III. Exercices

Afin de savoir si tout a été saisi, voici 3 exercices à réaliser sur les syscalls :

1) Demander à l'utilisateur de saisir une chaîne de caractères et réafficher cette chaîne de caractères juste en dessous de la saisie. Il faut veiller à ce que le programme se ferme convenablement et aussi veiller à ce que le message soit affiché sans bugs, avec des retours à la ligne.

   Astuces :

   - Il faut utiliser les appels systèmes **read** et **write**.
   - Il est possible d'utiliser la même zone mémoire pour lire et écrire la chaîne de caractères.
   - La saisie de l'utilisateur demande des octets d'une zone mémoire non initialisée

2) Ecrire la phrase "L'assembleur c'est trop cool !" dans le fichier "output.txt" et lire ce fichier.

Astuces :

- Il faut utiliser les appels systèmes **open**, **read** et **write**.
- Aucune donnée non initialisée n'est à prévoir

3) Exécuter le shell "/bin/sh"

Astuces :

- Il faut utiliser l'appel système **execve**.