HE2B ESI

MICL : TD01 : Prise en main de l'environnement de développement

ABS – BEJ – DBO – HAL – NVS – SRE – YVO *

Année académique 2021 – 2022

Les laboratoires de technique des microprocesseurs et assembleur micro (MICL [1]) se consacrent à l'étude du microprocesseur par le biais de la réalisation de programmes en langage d'assemblage. Ce premier TD (travail dirigé) est consacré à la prise en main des outils qui sont utilisés tout au long des laboratoires et à un premier contact intime avec le microprocesseur.

Il existe une multitude de familles de microprocesseurs [2]. Au cours des labos micro, nous étudions uniquement le microprocesseur qui occupe les ordinateurs personnels (PC). Il appartient à la famille de processeurs x86 [3]. Nous travaillons en mode 64 bits.

Chaque famille de processeurs se distingue par son architecture, son modèle mémoire et son jeu d'instructions. Nous découvrons ceux de la famille x86 au fur et à mesure du déroulement du cours théorique (MIC [4]), des laboratoires MICL et du cours consacré au systèmes d'exploitation (SYSI [5]).

*Et aussi, lors des années passées : DHA – DWI – EGR – ELV – FPL – JDS – MBA – MCD – MHI – MWA.

1. https://heb-esi.github.io/he2besi-web/online/cours/ac2021_mic2_mic-l.html (consulté le 26 janvier 2021).
2. https://fr.wikipedia.org/wiki/Microprocesseur#Familles (consulté le 28 janvier 2020).
3. https://en.wikipedia.org/wiki/X86 (consulté le 28 janvier 2020).
4. https://heb-esi.github.io/he2besi-web/online/cours/ac2021_mic2_mic-l.html (consulté le 25 janvier 2021).
5. https://heb-esi.github.io/he2besi-web/online/cours/ac2021_sys2_sys.html (consulté le 25 janvier 2021).

1 Registres

Lors de ce premier TD, le seul aspect de l'architecture du processeur abordé concerne ses registres [6]. Chaque registre est identifié par un nom.

Certains sont accessibles au programmeur directement via leur nom, comme `rax`, `rsp` ou `xmm5`, par exemple. D'autres sont accessibles indirectement, par le biais de certaines instructions, comme `rflags`, par exemple. Comme ces registres sont accessibles à tout code, quel que soit son niveau de privilège [7], et donc en particulier au *code utilisateur*, on les appelle *registres utilisateur*.

D'autres registres sont utilisés par le système d'exploitation ou sont réservés, comme `cr3`, `cs` ou `idtr`, par exemple. En général, seul du code privilégié, comme le noyau du système d'exploitation, peut les atteindre. C'est pourquoi on les appelle *registres système*. Nous ne les détaillons pas dans la suite des labos microprocesseur, par contre ils sont abordés en théorie micro et en système.

1.1 Registres utilisateur

1.1.1 Registres généraux

Dire qu'un processeur est un processeur 64 bits, c'est indiquer que la majorité de ses registres ont une taille de 64 bits.

Les processeurs 64 bits de la famille x86 possèdent seize registres d'utilité générale [8]. Ils sont nommés [9] : `rax`, `rcx`, `rdx`, `rbx`, `rsp`, `rbp`, `rsi`, `rdi`, `r8`, `r9`, `r10`, `r11`, `r12`, `r13`, `r14` et `r15`. Les huit premiers tirent leur nom du premier processeur de la famille x86 [10]. Il est également possible d'y accéder par le nom `rx` où `x` prend une valeur de 0 à 7.

Les registres `rbx`, `rcx` et `rdx` ont une structure interne identique à celle de `rax`, illustrée à la Fig. 1. Les registres `rdi`, `rsp` et `rbp` ont la même structure interne que `rsi`, illustrée à la Fig. 2. Enfin, les registres `r8` à `r15` partagent la structure interne de `r8`, représentée à la Fig. 3. Les suffixes `b`, `w` et `d` signifient *byte* [11], *word* [12] et *double word*, respectivement.

Notez que bien que qualifiés de généraux, certains de ces registres sont associés et utilisés implicitement par diverses instructions.

Semblablement, bien que placés dans la liste des registres généraux, `rsp` et `rbp` ne sont pas réellement utilisés comme tels. Ils sont utilisés en conjugaison avec la gestion de l'accès à la pile, dont l'étude est détaillée dans le TD07.

---

6. https://wiki.osdev.org/CPU_Registers_x86-64 (consulté le 28 janvier 2020).
7. https://fr.wikipedia.org/wiki/Mode_prot%C3%A9g%C3%A9#Niveaux_de_privil%C3%A8ge (consulté le 28 janvier 2020).
8. http://www.sandpile.org/x86/gpr.htm (consulté le 28 janvier 2020).
9. https://www.nasm.us/doc/nasmdo11.html#section-11.1 (consulté le 28 janvier 2020).
10. https://en.wikipedia.org/wiki/Intel_8086#Registers_and_instructions (consulté le 28 janvier 2020).
11. https://fr.wikipedia.org/wiki/Byte (consulté le 28 janvier 2020).
12. https://fr.wikipedia.org/wiki/Mot_(architecture_informatique) (consulté le 28 janvier 2020).

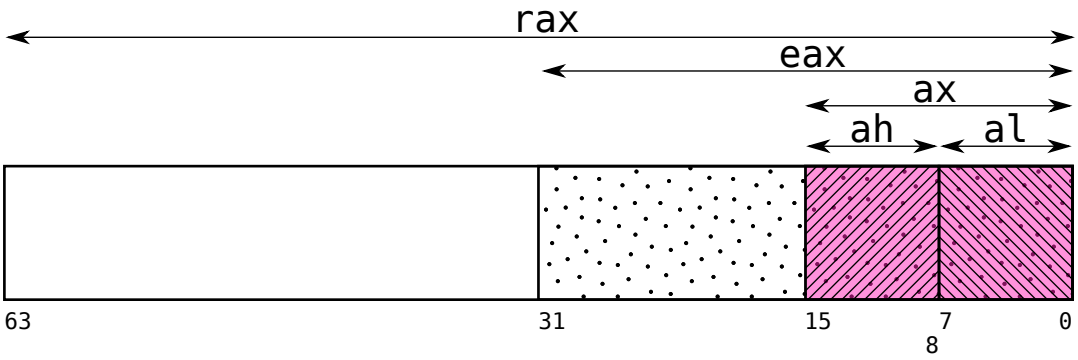

FIG. 1 – Structure interne du registre `rax`.

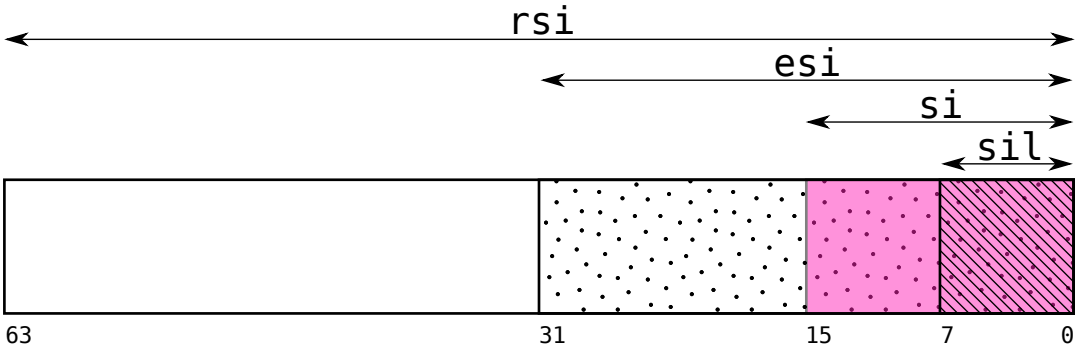

FIG. 2 – Structure interne du registre `rsi`.

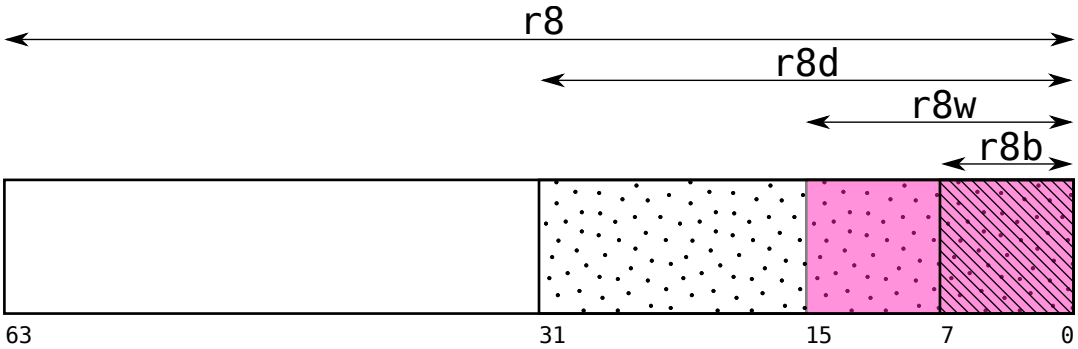

FIG. 3 – Structure interne du registre `r8`.

Ainsi, nous avons à disposition quatorze registres de 64 bits pour stocker les informations en cours de traitement.

1.1.2 Registre des indicateurs et registre pointeur d'instruction

Parmi les registres utilisateur non accessibles directement par leur nom, figurent `rflags` et `rip`. Le premier, appelé registre des indicateurs (*flags*), est décrit dans le TD02. Le second, `rip`, le registre pointeur d'instruction, est examiné dans ce premier TD. Il a, tout comme `rflags`, une taille de 64 bits. Il stocke l'adresse de l'instruction à exécuter *après* celle en cours d'exécution.

Dès lors, parler d'une architecture 64 bits indique également que la mémoire est adressée sur 64 bits. Pour observer la valeur de `rip`, nous utilisons un débogueur (voir la section 4.2). Lors des TD03 et 06, nous voyons comment modifier son contenu autrement que par son incrémentation automatique gérée par le microprocesseur. Cette incrémentation automatique, que nous observons dans la suite de ce TD, génère l'exécution séquentielle des instructions d'un programme.

1.1.3 Autres registres accessibles au programmeur

Il existe d'autres registres utilisateur. Ils ne ne sont cependant pas utilisés lors des labos micro. Certains d'entre eux vous sont présentés plus en détail au cours théorique. Néanmoins, pour des raisons d'exhaustivité, nous les citons ici.

Le coprocesseur mathématique, intégré au processeur [13], contient huit registres de données de 80 bits, `FPR0`-`FPR7`, ainsi que plusieurs registres de contrôle, statut, etc.

Les 64 bits de poids faible des huit registres `FPR0`-`FPR7` correspondent aux huit registres de l'extension MMX [14], `MMX0`-`MMX7`.

D'autres extensions SIMD [15] ont vu l'introduction de seize registres de 128 bits [16], nommés `XMM0`-`XMM15`.

2 Premiers programmes en langage d'assemblage

2.1 Premier code source

Voici un premier programme en langage d'assemblage, rédigé pour un processeur x86 64 bits géré par un système d'exploitation GNU/Linux [17] :

```
1 ; mov_imm.asm
2
```

13. https://en.wikipedia.org/wiki/Floating-point_unit#Integrated_FPUs (consulté le 28 janvier 2020).
14. https://en.wikipedia.org/wiki/MMX_(instruction_set) (consulté le 28 janvier 2020).
15. https://en.wikipedia.org/wiki/SIMD (consulté le 28 janvier 2020).
16. https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions#Registers (consulté le 28 janvier 2020).
17. https://fr.wikipedia.org/wiki/Linux_ou_GNU/Linux (consulté le 28 janvier 2020).

```
3  global main
4
5  section .text
6  main:
7
8      nop             ; ne fait rien
9
10     ; rax, rbx, rcx et rdx : idem
11     mov rax, 0x1122334455667788
12     mov eax, -1
13     mov rax, 0x1122334455667788
14     mov ax, 1
15     mov al, -1
16     mov ah, 0xA2
17
18     ; rax alias r0
19     ; ce qui suit n'est pas reconnu par nasm
20     ; r0 -> r7 : pas reconnus par nasm
21 ;    mov r0, 0x1122334455667788
22                     ; mov_imm.asm:21: error: symbol `r0' undefined
23 ;    mov r0d, -1
24                     ; mov_imm.asm:23: error: symbol `r0d' undefined
25 ;    mov r0w, 1
26                     ; mov_imm.asm:25: error: symbol `r0w' undefined
27 ;    mov r0b, -1
28                     ; mov_imm.asm:27: error: symbol `r0b' undefined
29
30     ; rsi et rdi : idem
31     mov rsi, -1
32     mov esi, 12345670q
33     mov si, 1010101010101010b
34     mov sil, 'a'    ; sil reçoit le code ASCII du caractère 'a'
35
36     ; rsp et rbp : comme rsi et rdi, mais utilisation comme registres
37     ;              généraux déconseillée (+ tard)
38
39     ; r8 -> r15 : idem
40     mov r8, 0xFFEEDDCCBBAA9988
41     mov r8d, -1
42     mov r8w, 1
43     mov r8b, -1
44
45     mov r9, 0x11_22_33_44_55_66_77_88
46     mov r10, 0x1122334455667788
```

```
47      mov r11, 0x1122334455667788
48      mov r12, 0x1122334455667788
49      mov r13, 0x1122334455667788
50      mov r14, 0x1122334455667788
51      mov r15, 0x1122334455667788
52
53  ;    mov r16, 0x1122334455667788    ; r16 n'existe pas
54                      ; mov_imm.asm:53: error: symbol `r16' undefined
55
56  ;    mov rip, 0      ; erreur de compilation : rip inaccessible
57                      ; mov_imm.asm:56: error: symbol `rip' undefined
58
59  ;    mov rflags, 0   ; erreur de compilation : rflags inaccessible
60                      ; mov_imm.asm:59: error: symbol `rflags' undefined
61
62      ; fin
63      mov rax, 60
64      mov rdi, 0
65      syscall
66
```

De nombreuses choses sont à dire… pour un programme qui ne fait quasiment rien d'intéressant ! Outre qu'il s'adresse à un processeur x86 64 bits, ce code source se destine à la production d'un exécutable sous GNU/Linux (64 bits), à condition d'utiliser l'assembleur `nasm`[18]. Notez que tous les programmes des TD des MICL sont réalisés pour un système d'exploitation GNU/Linux 64 bits.

Le code source assembleur est un simple fichier texte. N'importe quel éditeur de texte peut convenir pour le produire. Le nom du fichier source n'a pas beaucoup d'importance, mais bien son extension. Celui présenté ici s'appelle `mov_imm.asm`.

Un source en langage d'assemblage est constitué de directives au compilateur, d'instructions, d'identifiants, d'immédiats, de commentaires et d'étiquettes (*labels*). Les immédiats sont des données numériques[19], comme dans l'exemple de ce TD, ou sont liés à une adresse mémoire, comme il est illustré dans certains TD ultérieurs.

La principale instruction de ce code est `mov`[20]. Cette instruction permet de copier une valeur dans un registre ou en mémoire depuis un immédiat, un registre ou un emplacement mémoire.

18. https://www.nasm.us/ (consulté le 28 janvier 2020).
19. En informatique, toute information est représentée par une *suite de bits*. En ce sens, toute donnée est représentée par une (ou un ensemble de) valeur(s) numérique(s), qu'il s'agisse d'une vitesse, d'un mot écrit, d'un mot prononcé, d'une séquence vidéo, etc.
20. https://www.felixcloutier.com/x86/mov (consulté le 28 janvier 2020).

Note L'instruction `nop`[21] ne réalise aucune action. Elle permet de toujours mettre un point d'arrêt (voir la section 4.2) sur la première instruction *effective* du code. Il a en effet été remarqué par le passé à l'école qu'il était parfois impossible de mettre un point d'arrêt sur la toute première instruction d'un source assembleur.

Note L'instruction `syscall`[22] est succinctement présentée dans la section 2.4. Elle est détaillée dans le TD05.

2.2 Langage d'assemblage et dialecte

À chaque microprocesseur correspond un jeu d'instructions en langage binaire. Le langage d'assemblage offre une écriture de ces instructions à l'aide de lettres constituant les *mnémoniques*[23], beaucoup plus accessible à l'être humain que les codes binaires. Comme il existe un jeu d'instructions par microprocesseur, il existe un langage d'assemblage, un ensemble de mnémoniques, par microprocesseur.

Cependant, pour les processeurs x86, deux dialectes du langage d'assemblage coexistent : le dialecte Intel et le dialecte AT&T. Nous travaillons dans le dialecte Intel. Vous n'avez donc pas à vous soucier du dialecte AT&T, ni de ses différences[24] avec le dialecte Intel, si ce n'est que vous y serez confronté lors des laboratoires au moment de l'utilisation du débogueur (section 4.2) et de sa fonction de décompilation.

Le dialecte influence la forme des mnémoniques, l'ordre des opérandes d'une mnémonique et la manière de nommer les registres, par exemple.

Comme indiqué plus haut, le compilateur utilisé est `nasm`. Il travaille dans le dialecte Intel. Cela apparaît lors de l'usage de la mnémonique `mov`. Comme déjà écrit plus avant, celle-ci correspond à l'instruction de copie d'une valeur vers un emplacement donné. Deux opérandes sont attendus par `mov` : la destination et la source, dans cet ordre, séparées par une virgule. Ainsi, dans l'instruction `mov rsi, -1` du code présenté à la section 2.1, la destination est le registre `rsi` et la source est la valeur immédiate -1.

2.3 Immédiat, directive, commentaire et étiquette

Outre le dialecte, le choix de l'assembleur[25] a comme répercussion la manière d'écrire les valeurs immédiates et l'éventail des directives à disposition du programmeur.

Pour ce qui concerne les *immédiats*, l'exemple de code montre comment `nasm` gère les entiers[26] en base 10, 16, 8 et 2 et les caractères.

---

21. https://www.felixcloutier.com/x86/nop (consulté le 28 janvier 2020).
22. https://www.felixcloutier.com/x86/syscall (consulté le 28 janvier 2020).
23. https://en.wikipedia.org/wiki/Assembly_language#Opcode_mnemonics_and_extended_mnemonics (consulté le 28 janvier 2020).
24. http://staffwww.fullcoll.edu/aclifton/courses/cs241/syntax.html (consulté le 28 janvier 2020).
25. Le terme *assembleur* désigne le *logiciel* qui réalise la traduction du langage d'assemblage vers le code machine. Il désigne également, par métonymie, le langage d'assemblage lui-même.
26. https://www.nasm.us/doc/nasmdoc3.html#section-3.4.1 (consulté le 28 janvier 2020).

Mnémoniques et instructions sont directement liées. Rappelons que les instructions d'un programme sont des ordres à destination du processeur. Les directives quant à elles sont des ordres au compilateur. Elles formulent des demandes du programmeur au compilateur. Le processeur ne voit donc pas les directives, mais les instructions qu'il exécute peuvent en être affectées.

Dans le source étudié ici, `global` [27] est une directive. Elle impose de rendre public le label `main`. La raison en est donnée plus bas (section 3.3). Une autre directive présente ici se nomme `section` [28]. Suivie de `.text`, elle indique que ce qui suit doit être placé dans la section des instructions du fichier produit par l'assembleur ; donc que ce qui suit constitue les instructions du programme en cours de réalisation. Ceci est lié au modèle mémoire du microprocesseur, qui est développé davantage au cours théorique et lors de TD suivants (aux TD04 et 07, notamment). Notez que d'autres sections existent en général dans les exécutables, telles les sections `.rodata`, `.data` et `.bss`. Elles apparaissent alors également dans le fichier source en langage d'assemblage. On en reparle au TD04.

On trouve dans le source `mov_imm.asm` également des commentaires et une étiquette. Avec `nasm` [29], les premiers commencent par un point-virgule (`;`) et s'achèvent en fin de ligne, les seconds se terminent par deux points (`:`).

2.4 Fin du programme et système d'exploitation

Pour ce qui concerne le contenu du source de la page 4, il reste à évoquer ses toutes dernières lignes. La dernière instruction est `syscall`. Il s'agit d'une instruction qui permet d'invoquer un bout de code du système d'exploitation. Celui-ci, en effet, ne disparaît pas lors de l'exécution du programme. Il reste prêt à offrir ses services, sous la forme d'appels système [30], par exemple. Les détails — enfin, bien davantage que des détails — vous sont livrés lors des cours théoriques microprocesseur et système, et lors du TD05. Sachez provisoirement que l'appel système utilisé ici arrête le programme et lui fait rendre la main au système d'exploitation.

Notez que c'est ici et uniquement ici qu'il est possible d'affirmer que ce source est destiné à GNU/Linux. Le même code source pour Windows est composé des mêmes lignes de code, *à l'exception de celles de fin*, car les appels systèmes, leur numérotation et leurs arguments diffèrent d'un système d'exploitation à l'autre. Par contre, quel que soit le système d'exploitation, c'est l'instruction `syscall`, pour la famille x86 64 bits, qui permet d'invoquer un appel système.

Comme nous travaillons exclusivement sous GNU/Linux, tous les programmes que nous écrivons se terminent par les mêmes trois (ou quatre, en comptant le commentaire) lignes que le source donné ici en exemple.

27. https://www.nasm.us/doc/nasmdoc6.html#section-6.6 (consulté le 28 janvier 2020).
28. https://www.nasm.us/doc/nasmdoc6.html#section-6.3 (consulté le 28 janvier 2020).
29. https://www.nasm.us/doc/nasmdoc3.html#section-3.1 (consulté le 28 janvier 2020).
30. https://en.wikipedia.org/wiki/System_call (consulté le 28 janvier 2020).

2.5 Davantage sur les registres 64 bits et sur l'instruction `mov`

Dans la section 1.1.1, on renseigne la structure des registres généraux. Prenant le registre `rax` (Fig. 1, p. 3), par exemple, on voit qu'on peut accéder à ses 32 bits de poids faible en utilisant le nom `eax` ou encore à ses 8 bits de poids faible à l'aide du nom `al`. Lorsqu'on modifie, à l'aide de `mov`, les parties `ax`, `ah` ou `al`, cela n'a aucune répercussion sur les autres bits de `rax`. Ainsi, par exemple, lorsqu'à la ligne 14 du source `mov_imm.asm` (p. 4) on modifie `ax`, seuls les 16 bits de `ax` sont modifiés. Par contre, si on modifie `eax`, comme à la ligne 12 du même code source, alors les 32 bits de poids élevé de `rax` sont mis à zéro[31] (0) ! Cela vaut pour les 16 registres généraux de 64 bits, pour toutes les instructions qui effectivement modifient leur destination.

Dans le même source `mov_imm.asm` (p. 4), on a toujours un immédiat comme source et un registre comme destination pour l'instruction `mov`. Comme signalé en fin de la section 2.1, cette instruction offre une plus grande souplesse. La source peut être un immédiat, un registre ou un emplacement mémoire. La destination doit être un registre ou un emplacement mémoire. Trois contraintes viennent cependant s'ajouter :

- la source et la destination ne peuvent être *simultanément* des emplacements mémoire ;
- la source et la destination doivent être de même taille ;
- il est interdit[32] d'utiliser un des registres `ah`, `bh`, `ch` ou `dh` en même temps qu'un des registres de 8 bits n'existant que depuis l'architecture 64 bits, à savoir les registres `sil`, `dil`, `spl`, `bpl` et les registres `r8b` à `r15b`.

2.6 Second code source

Voici un second source qui illustre les propriétés et contraintes décrites dans la section 2.5, à l'exception de celle où interviennent les emplacement mémoire, car la notion de variable n'est abordée que lors du TD04 :

```
1  ; mov_reg.asm
2
3  global main
4
5  section .text
6  main:
7
8      nop             ; ne fait rien
9
10     mov rax, 0x1122334455667788
11
12     mov rbx, rax
```

31. https://stackoverflow.com/q/11177137 (consulté le 28 janvier 2020).
32. https://x86asm.net/articles/x86-64-tour-of-intel-manuals/ (consulté le 28 janvier 2020).

```
13      mov r12, rbx
14
15      mov eax, ebx     ; effet sur rax
16      mov r12d, ebx    ; effet sur r12
17
18      mov r8, -1
19
20      mov bx, r8w
21      mov r12b, r8b
22
23      ; la source et la destination doivent avoir même taille
24  ;    mov rax, esi  ; error: invalid combination of opcode and operands
25  ;    mov r8, ax    ; error: invalid combination of opcode and operands
26  ;    mov eax, r8b  ; error: invalid combination of opcode and operands
27
28      mov ah, al
29      mov al, r8b
30      mov sil, al
31      mov r15b, al
32      mov sil, spl
33
34      ; interdit d'utiliser ah, bh, ch ou dh avec :
35      ;   + sil, dil, spl, bpl ;
36      ;   + r8b -> r15b.
37  ;    mov sil, ah  ; error: cannot use high register in rex instruction
38  ;    mov r8b, ah  ; error: cannot use high register in rex instruction
39
40      ; fin
41      mov rax, 60
42      mov rdi, 0
43      syscall
44
```

3 Chaîne de production d'un exécutable

Comme indiqué précédemment, le source est produit et modifié à l'aide d'un éditeur de texte. La création d'un exécutable n'est pas directement obtenue après l'assemblage (la compilation). Chaque fichier source est assemblé (compilé) séparément. Ce processus donne un fichier objet [33] [34] par fichier source. Un programme appelé *éditeur de liens*

33. https://fr.wikipedia.org/wiki/Fichier_objet (consulté le 28 janvier 2020).
34. Cette dénomination de *fichier objet* n'a rien à voir avec la programmation *orienté objet*.

(*linker*) construit l'exécutable à partir des fichiers objet et des bibliothèques logicielles [35] (*software library*) présentes sur la machine. L'exécutable accède directement au processeur, sans interface telle qu'une machine virtuelle. Il est démarré via le système d'exploitation.

Pour commencer, démarrez un shell [36].

3.1 Édition du code source

Éditez le code `mov_tst.asm` avec l'éditeur de texte [37] (*text editor*) vi [38] :

```
vi mov_tst.asm
```

ou avec nano [39] :

```
nano mov_tst.asm
```

de sorte à produire un source de même structure que ceux des sections 2.1 et 2.6.

3.2 Assemblage

Quand vous avez fini, sauvez-le et invoquez l'assembleur [40] (*assembler*) `nasm` comme suit :

```
nasm -f elf64 -F dwarf mov_tst.asm
```

L'option `-f` [41] sert à choisir le format du fichier objet. Le format `elf` [42] est courant sous GNU/Linux. L'option suivante est facultative, mais elle va bien nous servir dans la suite de ce TD et dans les suivants. `-F` [43] sert à produire des informations de débogage et à spécifier leur format. Le format `DWARF` [44] va bien avec le débogueur (section 4.2) utilisé ensuite.

Si vous listez le contenu du répertoire courant, vous y découvrez un nouveau fichier, nommé `mov_tst.o`. Il s'agit d'un fichier objet. C'est un fichier binaire *non* exécutable. Il est produit par l'assembleur `nasm`.

35. https://fr.wikipedia.org/wiki/Biblioth%C3%A8que_logicielle (consulté le 28 janvier 2020).
36. https://fr.wikipedia.org/wiki/Shell_Unix (consulté le 28 janvier 2020).
37. https://fr.wikipedia.org/wiki/%C3%89diteur_de_texte (consulté le 28 janvier 2020).
38. https://openclassrooms.com/fr/courses/43538-reprenez-le-controle-a-laide-de-linux/42693-vim-lediteur-de-texte-du-programmeur (consulté le 28 janvier 2020).
39. https://openclassrooms.com/fr/courses/43538-reprenez-le-controle-a-laide-de-linux/39267-nano-lediteur-de-texte-du-debutant (consulté le 28 janvier 2020).
40. https://fr.wikipedia.org/wiki/Programme_assembleur (consulté le 28 janvier 2020).
41. https://www.nasm.us/doc/nasmdoc2.html#section-2.1.2 (consulté le 28 janvier 2020).
42. https://cirosantilli.com/elf-hello-world (consulté le 28 janvier 2020).
43. https://www.nasm.us/doc/nasmdoc2.html#section-2.1.12 (consulté le 28 janvier 2020).
44. http://dwarfstd.org/ (consulté le 28 janvier 2020).

3.3 Édition des liens

La production d'un fichier exécutable à partir du source `mov_tst.asm` requière le fichier objet `mov_tst.o` et un nouveau programme appelé éditeur de lien[45] (*linker*). Nous utilisons `ld`[46] comme suit :

```
ld -o mov_tst -e main mov_tst.o
```

Les options[47] servent à spécifier le nom de l'exécutable et le point d'entrée du programme, respectivement. Le point d'entrée spécifie l'adresse de la première instruction à exécuter quand le programme démarre. On a identifié celle-ci dans le source à l'aide de l'étiquette `main`[48] qui a été rendue publique grâce à la directive `global main`.

3.4 Exécution

Vous avez désormais un fichier exécutable nommé `mov_tst`.
Pour l'exécuter, tapez :

```
./mov_tst
```

Si le contenu de `mov_tst.asm` est semblable à celui de `mov_imm.asm` ou `mov_reg.asm`, rien de visible ne se passe. C'est normal car ces codes sources ne réalisent aucune interaction avec les périphériques. Le programme s'exécute puis se termine et l'interpréteur de commande reprend la main.

4 Erreurs et débogage

4.1 Erreurs

En toute généralité, des erreurs peuvent se produire lors de l'assemblage, lors de l'édition des liens ou lors de l'exécution. Dans les deux premiers cas, lisez les messages d'erreur de l'assembleur ou de l'éditeur de liens et apportez les corrections nécessaires à votre code source. Dans le troisième cas, replongez-vous dans votre code (section 3.1) ou recourrez aux services d'un débogueur (section 4.2).

4.2 Débogage

Comme ici le programme doit s'exécuter sans manifestation externe, nous allons utiliser un débogueur[49] (*debugger*) pour voir ce qui se passe dans son intimité.
Démarrez le débogueur par la commande :

45. https://en.wikipedia.org/wiki/Linker_(computing) (consulté le 28 janvier 2020).
46. https://sourceware.org/binutils/docs-2.29/ld/Overview.html#Overview (consulté le 28 janvier 2020).
47. https://sourceware.org/binutils/docs-2.29/ld/Options.html#Options (consulté le 28 janvier 2020).
48. Remarquez que rien n'empêche de donner un autre nom que `main` au point d'entrée du programme.
49. https://fr.wikipedia.org/wiki/D%C3%A9bogueur (consulté le 28 janvier 2020).

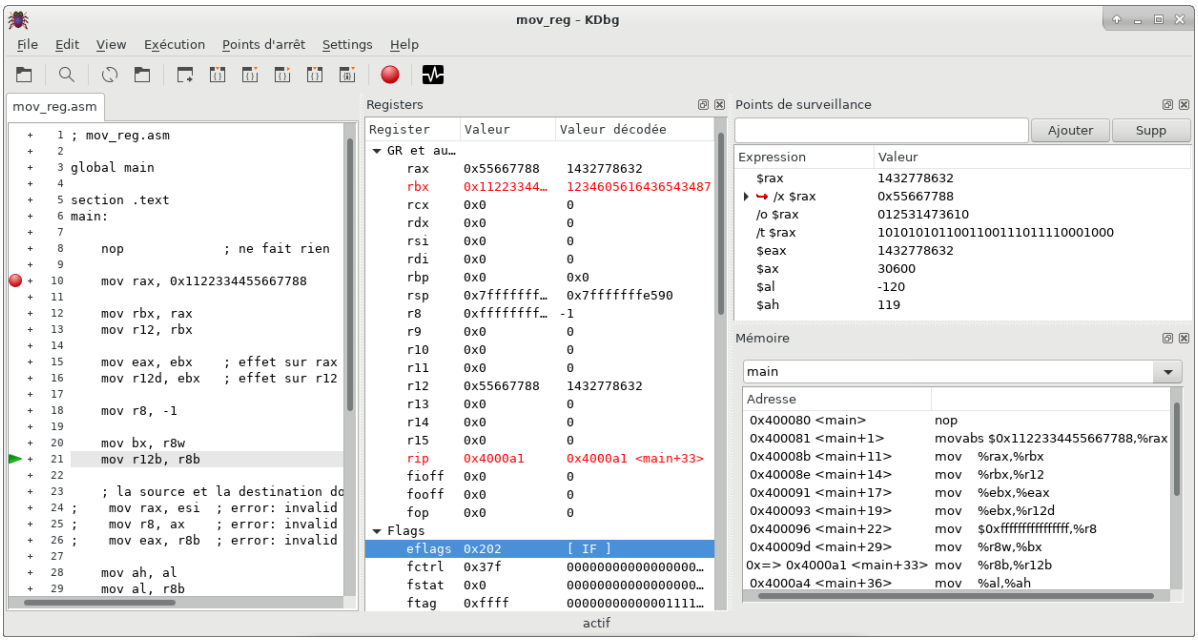

FIG. 4 – KDbg en action.

```
kdbg mov_tst &
```

KDbg[50] est un programme à interface graphique pour GDB[51].

Une des capacités d'un débogueur est de pouvoir exécuter pas à pas, c'est-à-dire instruction après instruction, le programme qui lui a été fourni en entrée. Dans notre cas, on donne `mov_tst` à KDbg. Pendant la pause qui intervient entre l'exécution des instructions du programme, on peut inspecter le contenu des registres, dont `rip`.

L'esperluette `&` en fin de commande provoque l'exécution de KDbg en arrière-plan[52]. Elle n'est pas obligatoire.

4.2.1 Problèmes avec KDbg

À l'ESI. Si KDbg ne démarre pas quand vous le demandez à `linux1`, voici quelques vérifications à faire :

— avez-vous bien démarré VcXsrv[53] ?

— avez-vous bien configuré PuTTY[54] pour l'échange d'informations graphiques (*Enable X11 forwarding*[55]) ?

Davantage de détails se trouvent dans le tutoriel de connexion renseigné à la section 5.1.

50. http://www.kdbg.org/ (consulté le 28 janvier 2020).
51. https://www.gnu.org/software/gdb/ (consulté le 28 janvier 2020).
52. https://openclassrooms.com/courses/reprenez-le-controle-a-l-aide-de-linux/executer-des-programmes-en-arriere-plan#/id/r-2282037 (consulté le 28 janvier 2020).
53. https://sourceforge.net/projects/vcxsrv/files/vcxsrv/ (consulté le 28 janvier 2020).
54. https://www.chiark.greenend.org.uk/~sgtatham/putty/ (consulté le 14 février 2020).
55. http://ocean.stanford.edu/courses/ESS141/PuTTY/ (consulté le 14 février 2020).

Absence de code source. Si lors d'une session de débogage avec `KDbg` aucun source ne s'affiche dans le portion de fenêtre correspondante, voici diverse pistes à vérifier pour corriger le problème :

— lors de l'assemblage, avez-vous requis les informations de débogage à l'aide de l'option `-F dwarf` ?

— dans votre source, avez-vous respecté la convention de nommage [56] `section .text` de la section où se trouvent les instructions du programme ?

```
1 section .text   ; ok
2
3 section .texte  ; erreur : .text et non .texte
4
5 section.text    ; erreur : il faut une espace entre « n » et « . »
6
7 section .code   ; erreur : .text et non .code
8
```

Impossibilité de placer un point d'arrêt. Si lors d'une session de débogage avec `KDbg`, il manque des icônes [57] et il vous est impossible de placer un point d'arrêt sur quelle que ligne du code source que ce soit, la seule solution trouvée à ce jour est d'utiliser une alternative à `KDbg`, telle `GNU DDD` [58], `nemiver` [59] ou `Insight` [60]...

5 Au travail !

Maintenant, c'est vraiment à vous de jouer.

5.1 À l'école

Pour jouir des facilités graphiques sous `linux1` via une machine qui tourne sous Windows, quelques manipulations préalables sont nécessaires. Elles sont décrites dans un document mis à votre disposition sur poESI [61].

56. https://linux-audit.com/elf-binaries-on-linux-understanding-and-analysis/#text (consulté le 28 janvier 2020).
57. https://github.com/j6t/kdbg/issues/5 (consulté le 28 janvier 2020).
58. https://www.gnu.org/software/ddd/ (consulté le 28 janvier 2020).
59. https://wiki.gnome.org/Apps/Nemiver (consulté le 28 janvier 2020).
60. https://sourceware.org/insight/ (consulté le 28 janvier 2020).
61. https://poesi.esi-bru.be/mod/resource/view.php?id=646 (consulté le 30 janvier 2020).

5.1.1 Produire

Produisez l'exécutable `mov_reg` avec informations de débogage à partir du fichier source `mov_reg.asm` fourni [62] en suivant les étapes décrites à la section 3 et la sous-section 4.2.

5.1.2 Découvrir

Dans la mesure où `KDbg` est muni d'une interface graphique, nous ne détaillons pas ici sa prise en main. Explorez les menus, survolez les boutons et attendez les bulles d'information et de description, cliquez sur les boutons gauche ou droit de la souris et voyez ce qui se passe. N'ayez pas peur de casser quelque chose. Dans le pire des cas, vous plantez `KDbg` qu'il suffit de redémarrer.

Faites ce qu'il faut pour arriver à une fenêtre dont le contenu est semblable à celui de la Fig. 4. Actuellement, seuls les affichages des registres, des points de surveillance et de la mémoire nous intéressent.

5.1.3 Observer

Exécutez `mov_reg` pas à pas en tenant bien à l'œil l'évolution des contenus des registres, en particulier celui du pointeur d'instructions. Comment expliquer l'évolution de contenu de ce dernier ? Quels liens existent-ils entre celle-ci et le contenu de la mémoire à partir de l'étiquette `main` ? Quel lien établissez-vous entre cet affichage de la mémoire [63] et celui du code source ? Comment les valeurs entières sont-elles codées ? Notez et retenez bien ces observations : elles vous seront utiles dans la suite des MICL.

5.1.4 Modifier

N'hésitez pas à modifier le code source `mov_reg.asm` en changeant les valeurs entières, les noms des registres ou en décommentant les lignes qui y sont commentées. Tout ce que vous risquez, si vous ne touchez pas aux trois dernières lignes de code, c'est une erreur lors de la chaîne de production de l'exécutable. Et si l'exécutable est bien fabriqué, exécutez-le dans le débogueur.

5.2 À la maison

5.2.1 Installer

Installez chez vous les outils de développement utilisés à l'école. La première chose qu'il vous faut est une distribution GNU/Linux pour architecture x86 64 bits avec interface graphique. De prime abord, `vi`, `ld` et `GDB` y sont présents de base. Utilisez son

62. https://poesi.esi-bru.be/pluginfile.php/5439/mod_folder/content/0/code/mov_reg.asm (consulté le 28 janvier 2020).
63. Notez qu'il est possible de basculer d'*Hexadécimal* à *Instructions* (en dialecte AT&T).

gestionnaire de paquets pour déterminer automatiquement les dépendances nécessaires à l'installation de `nasm`, `KDbg` et éventuellement `nano`.

Réalisez ensuite chez vous les sous-points de la section 5.1.

Notions à retenir

x86 64 bits, langage d'assemblage, registre, registres généraux, registre pointeur d'instruction, instruction, mnémonique, directive, immédiat, label (étiquette), assembleur, éditeur de liens, débogueur, fichier source, fichier objet, fichier exécutable.

Références

[1] Igor Zhirkov. *Low-Level Programming*. Apress, 2017. https://www.apress.com/gp/book/9781484224021.