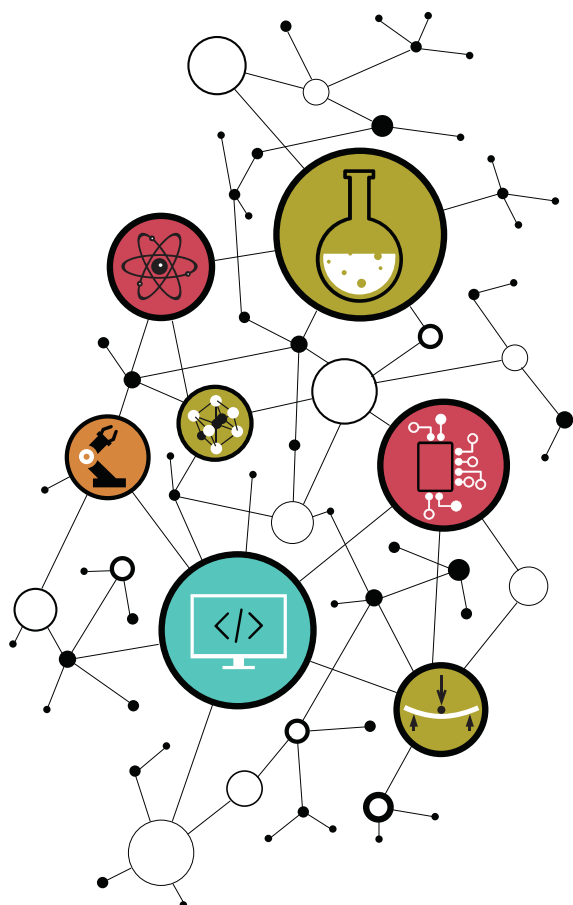


PARTIE 4 ASSEMBLEUR INTEL x86



I. L'assembleur sous architecture x86

I.1. Le langage d'assemblage

L'**assembleur** (*assembly*) ou **langage d'assemblage** est le langage de programmation de plus bas niveau sur la machine. Il est la conversion directe lisible voire éditable par l'homme (texte) du programme exécutable par le CPU de la machine (binaire). Il est de ce fait, le langage le moins universel au monde, car dépendant du jeu d'instructions supporté par le CPU cible. Entre les marchés des ordinateurs et des systèmes embarqués, un grand nombre de fabricants implémentent des modèles d'exécution (RISC ou CISC, Von Neumann ou Harvard, 8-16-32-64-bit, entier voire flottant, VLIW ou superscalaire, vectoriel ou scalaire, ...) sur des technologies différentes (x86, x64, ARM, MIPS, C6000, PIC18, ...).

```
int main (void)
{
    return 0;
}
```

```
main:
    pushl    %ebp
    movl     %esp,%ebp
    movl     $0,%eax
    popl     %ebp
    ret
```

L'assembleur présenté ci-dessus est par exemple de l'assembleur 32-bit IA-32 (Intel Architecture 32-bit) souvent nommé x86 et supporté par des technologies CPU compatibles (IA-32 chez Intel et AMD). Comme tout langage de programmation, un programme assembleur se lit de haut en bas, à l'image du modèle d'exécution séquentiel d'un CPU. Même si l'assembleur n'est pas universel, certaines représentations liées au langage peuvent néanmoins être généralisées :

- **Label** : un label ou étiquette, est une référence symbolique (simple chaîne de caractères) représentant l'adresse mémoire logique (emplacement) de la première information suivant celui-ci. Les labels sont remplacés par les adresses logiques voire physiques à l'édition des liens. Un label se termine par `:` afin de le différencier d'éventuelles directives d'assemblage ou instructions.
- **Instruction** : une instruction est un traitement élémentaire à réaliser par le CPU. Par exemple, charger/*load* ou sauver/*store* une donnée depuis ou vers la mémoire principale, réaliser une opération arithmétique ou logique, déplacer une donnée de registre à registre, ... L'ensemble des instructions exécutables par un CPU est nommé **jeu d'instructions (ISA ou Instruction Set Architecture)**. L'ISA représente ce que sait faire nativement un CPU.
- **Opérandes** : les opérandes, lorsque l'instruction en utilise, sont les données ou les emplacements de données (registres ou adresses en mémoire principale) manipulées par l'instruction. Nous distinguons les opérandes sources, utilisées comme entrées avant l'exécution de l'instruction, de l'opérande de destination pour sauver le résultat. Les méthodes d'accès aux données en assembleur sont nommées des modes d'adressage :
 - **Mode d'adressage registre** : l'opérande est un registre CPU dans lequel est sauvé une donnée. Par exemple, les instructions `push`, `mov` et `pop` ci-dessus.
 - **Mode d'adressage immédiat** : l'opérande est une constante dont la valeur sera sauvée dans le code binaire de l'instruction. Par exemple, l'instruction `mov $0` ci-dessus
 - **Mode d'adressage direct** (accès mémoire) : l'opérande est directement l'adresse de la case mémoire de la donnée
 - **Mode d'adressage indirect** (accès mémoire) : l'opérande est l'adresse de la case mémoire de la donnée stockée indirectement dans un registre CPU.

1.2. Syntaxe assembleur AT&T proposée par GNU AS

L'assembleur x86 est une ISA développée historiquement par Intel pour son CPU 16-bit 8086 produit en 1978. Les générations suivantes de processeurs Intel et AMD sorties dans les années 80 (80286, 80386, ...) sont restées compatibles avec leur prédécesseur. Ce langage d'assemblage s'est donc nommé au fil du temps x86. Il est à noter que les processeurs 64-bit compatibles x64 (IA-64 chez Intel et AMD64 chez AMD) restent également rétrocompatibles x86. Ceci reste également toujours vrai à notre époque (technologies Pentium, Core2, Core-i, etc).

Syntaxe Intel

```
main:
    push    ebp
    mov     ebp, esp
    mov     eax, 0
    pop     ebp
    ret
```

Syntaxe AT&T

```
main:
    pushl   %ebp
    movl    %esp,%ebp
    movl    $0,%eax
    popl    %ebp
    ret
```

Le langage C et sa syntaxe sont maintenant normalisés et standardisés depuis des décennies même si la norme continue d'évoluer (normes ANSI C, C89, C99, C18, ...). En revanche, différentes syntaxes assembleur liées à la chaîne de compilation et aux outils de développement (ouverts ou fermés, libres ou propriétaires) existent sur le marché. Prenons l'exemple ci-dessus d'un même programme assembleur en syntaxe AT&T (généré par AS ou GAS ou GNU AS – étage d'assemblage de GCC et généralisé sur système *Unix-like*) et en syntaxe Intel (généré par ICC – *Intel C++ Compiler*). Nous pouvons constater ci-dessus que les opérandes sources et de destinations ne sont pas placés dans le même sens (destination à droite en syntaxe AT&T). Observons quelques particularités de la syntaxe AT&T :

- **%** : signifie que l'opérande qui suit est un registre CPU (mode d'adressage registre) ;
- **\$** : signifie que l'opérande qui suit est une constante (mode d'adressage immédiat) ;
- suffixe d'instruction : précise la taille en octet des données manipulées : **b**=byte=1octet, **s**=short=2octets, **l**=long=4octets et **q**=quad=8octets. Ce suffixe est facultatif à l'édition ;
- **(%registre)** ou **(\$adresse)** : signifie que l'instruction nécessite de charger ou de sauver une donnée depuis ou vers la mémoire principale (respectivement modes d'adressages indirect et direct). Par exemple, en mode d'adressage indirect, si le pointeur BP (adresse) est sauvé dans EBP (registre), l'opérande avec offset suivante **-4(%ebp)** se lit en pseudo-code ***(BP - 4)**, soit accès à la case mémoire pointée par l'adresse BP, moins 4 octets.

```
.global main

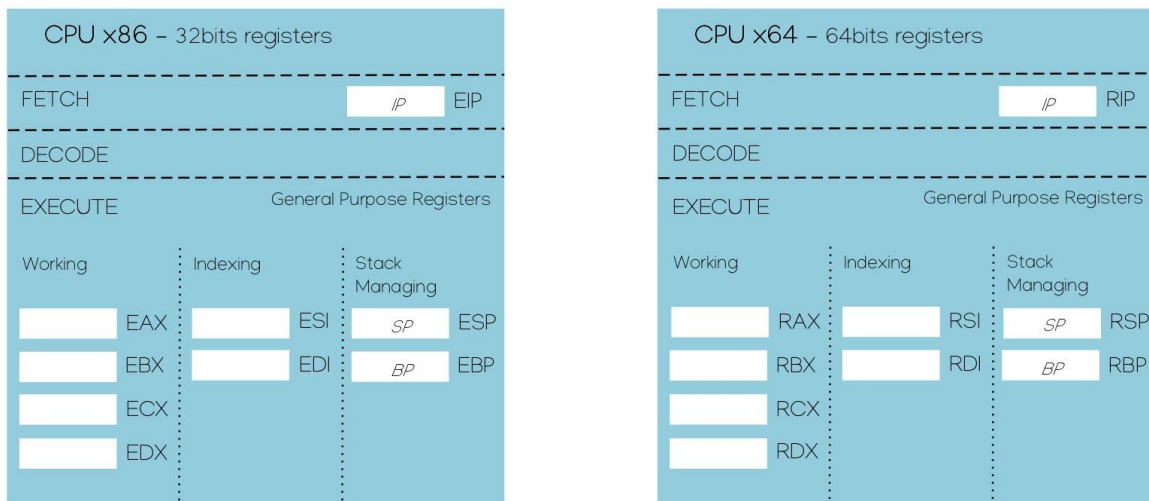
.text
main:
    ...
    ret
```

Un programme assembleur intégrera également certaines directives d'assemblage¹¹ préfixées par un point (**.global**, **.text**, **.macro**, ...). Celles-ci renseignent l'outil chargé de convertir l'assembleur en binaire (également nommé assembleur en français ou assembler en anglais) ainsi que l'éditeur des liens. Ces directives fixent par exemple les sections du firmware où ranger le code et les données statiques (**.section**, **.text**, **.data**, **.rodata**, etc), étendent les portées de labels à l'éditeur des liens (**.global**), définissent des macros (**.macro**, **.endm**), ...

¹¹ https://ftp.gnu.org/old-gnu/Manuals/gas-2.9.1/html_chapter/as_7.html

I.3. Registres CPU x86-64

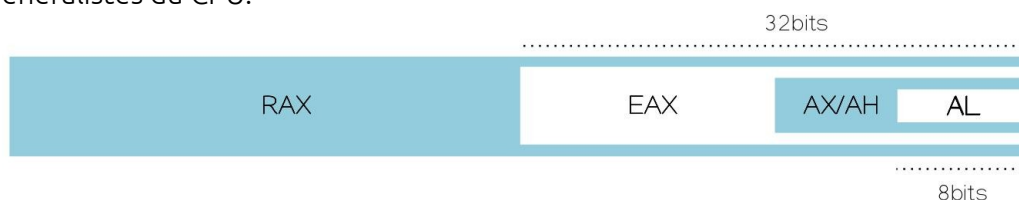
Les schémas ci-dessous présentent les principaux registres d'usages généralistes présents dans les CPU d'architectures compatibles x86 (32-bit) et x64 (64-bit). Rappelons que ces architectures restent rétrocompatibles avec le CPU 16-bit 8086 datant de 1978. Ces registres sont présents dans chaque CPU d'une machine multicœurs (un cœur est l'ensemble CPU, MMU et caches locaux).



D'autres registres aux usages plus spécifiques existent dans chaque CPU. Présentons-les succinctement (non vus en enseignement) :

- **Registre d'état FLAGS** (CF, PF, ZF, ...) : registre contenant les différents *flags* d'états associés aux unités d'exécution arithmétiques et logiques (*carry, zero, sign, ...*). Utilisé notamment afin d'implémenter des sauts conditionnels (*if, else if, while, for, ...*) ;
- **Registres de contrôle** (CR0 à CR7) : registres permettant de contrôler les services matériels du CPU (mode protégé, etc), ainsi que du cache et de la MMU associés au cœur ;
- **Registres vectoriels** (XMM0 à XMM15 128-bit extension ISA SSE, YMM0 à YMM15 256-bit extension ISA AVX et ZMM0 à ZMM15 512-bit extension ISA AVX-512) : registres de travail pour les instructions vectorielles dans un contexte d'optimisation algorithmique ;
- **Registres de segments** (CS, DS, ES, SS, FS, GS) : registres historiquement utilisés lorsque la segmentation logique d'une application nécessitait un support d'adressage physique. La segmentation est maintenant virtualisée et gérée entièrement logiquement par le noyau du système à travers la gestion de la PMMU (*Paged MMU* ou unité de pagination).

Pour des soucis de rétrocompatibilité, toute nouvelle architecture compatible x64 doit rester compatible avec les générations antérieures x86. Cette rétrocompatibilité remonte jusqu'au 8086. Par exemple, les registres 16-bit et 8-bit de ce processeur sont toujours supportés à notre époque. Prenons l'exemple du registre à usage général A (*Accumulator*), déjà présent sur Intel 4004 en 1971 (premier microprocesseur), ainsi que ses déclinaisons 8-16-32-64-bit imbriquées les unes dans les autres sur architectures x86-64 (respectivement AL 8-bit, AX 16-bit, EAX 32-bit et RAX 64-bit). L'exemple donné ci-dessous sur le registre 64bits RAX peut être étendu à tous les registres de travail généralistes du CPU.



II. Instructions arithmétiques et logiques

```

--- crt0.s ---

.global _start
.text
_start:
    push    %ebp
    mov     %esp, %ebp
    call    main
    mov     $1, %eax
    int     $0x80

```

```

--- logic_arithmetic.s ---

.global main
.text
main:
    xor     %eax, %eax
    mov     $9, %ebx
    add     %ebx, %eax
    sub     $2, %eax
    ret

```

Ouvrez un terminal et placez-vous dans le répertoire de travail `disco/asm/`. Assemblez les fichiers `logic_arithmetic.s` et `disco/toolchain/build/startup/crt0.s` (fichier de startup minimal). Puis réalisez l'édition des liens en utilisant le linker script minimal utilisé dans le premier chapitre de TP.

```

as --32 -g ../toolchain/build/startup/crt0.s -o obj/crt0.o
as --32 -g logic_arithmetic.s -o obj/logic_arithmetic.o
ld -melf_i386 -T ../toolchain/build/script/linker_script_minimal.ld obj/crt0.o
obj/logic_arithmetic.o -o bin/logic_arithmetic

```

Affichez et analysez le programme assembleur `logic_arithmetic.s`.

Quel traitement implémente l'instruction `XOR` ? Comment aurait-on pu écrire autrement ce même traitement ?

Des deux instructions implémentant un même traitement proposées à la question précédente (`XOR` vs `MOV $0`), laquelle offre une implémentation plus optimisée et pourquoi ?

Analyser le firmware pour vous aider.

```
objdump -S bin/logic_arithmetic
```

Quel est le contenu du registre `eax` à la fin de l'exécution de la fonction `main` ?

III. GNU Debugger

Nous allons maintenant découvrir quelques outils complémentaires pouvant vous aider à l'analyse et au développement d'un script assembleur. Nous allons découvrir quelques une des principales commandes de GDB, le débogueur ou *debugger* du projet GNU¹².

Commandes (raccourcis)	Descriptions (nom complet)
<code>l</code>	(<i>list</i>) Affiche le listing avec numéros de lignes du programme C
<code>la a</code>	(<i>layout assembly</i>) Affiche le listing assembleur du binaire désassemblé
<code>b label</code>	(<i>break</i>) place un point d'arrêt (<i>breakpoint</i>) sur un label connu
<code>b file.c:line_nb</code>	(<i>break</i>) place un point d'arrêt sur une ligne spécifique du programme C
<code>i b</code>	(<i>info break</i>) Liste tous les point d'arrêts du programme
<code>r</code>	(<i>run</i>) exécute le programme jusqu'au premier point d'arrêt
<code>c</code>	(<i>continue</i>) reprend l'exécution jusqu'au prochain point d'arrêt (ou la fin)
<code>s</code>	(<i>step</i>) Exécute l'instruction ASM suivante (pas à pas)
<code>ni</code>	(<i>next instruction</i>) Exécute l'instruction C suivante (pas à pas)
<code>p variable</code>	(<i>print</i>) Affiche le contenu d'une variable
<code>i reg</code>	(<i>info</i>) Affiche le contenu de tous les registres de l'architecture
<code>i reg names</code>	(<i>info</i>) Affiche le contenu des registres demandés
<code>x/NFb 0xaddress</code>	Examine N octets de la mémoire au format F spécifié (d ou x)
<code>q</code>	(<i>quit</i>) Quitter la session de debug courante
<code>more ...</code>	http://www.gdbtutorial.com/gdb_commands

Assemblez jusqu'à l'édition des liens incluse le fichier `logic_arithmetic.s`. Vous aurez peut-être remarqué l'option `-g` passée à GCC, qui lui demande d'intégrer des directives de *debug* au sein du code compilé.

```
as --32 -g logic_arithmetic.s -o obj/logic_arithmetic.o

ld -melf_i386 -T ../toolchain/build/script/linker_script_minimal.ld obj/crt0.o
obj/logic_arithmetic.o -o bin/logic_arithmetic
```

¹² <http://www.gdbtutorial.com/>

Lancez maintenant l'exécutable au travers du *debugger* GDB.

```
gdb ./bin/logic_arithmetic
(gdb) ... wait for gdb commands !
```

Une console de *debug* vient maintenant de s'ouvrir. GDB est un programme capable de prendre le contrôle sur d'autres programmes. Nous pouvons placer des points d'arrêts, puis analyser pas à pas l'exécution d'un programme (*dump* mémoire, trace d'exécution, contenu de registres CPU, etc). Un *debugger* s'utilise durant les phases de développement ou de déverminage d'un programme. Il propose des outils d'analyse élémentaires mais très puissants. À force de persévérance, aucun bug ne peut se cacher indéfiniment !

Suivez la séquence de commandes GDB proposée ci-dessous, et interprétez-la à l'aide du tableau de la page précédente.

```
(gdb) la a
(gdb) b _start
(gdb) r
(gdb) b main
(gdb) c
(gdb) s
(gdb) i reg eax
(gdb) s
(gdb) i reg eax ebx
(gdb) s
(gdb) i reg eax ebx
(gdb) s
(gdb) i reg eax ebx
(gdb) s
(gdb) s
... until end of program
(gdb) s
```

IV. Fonction de conversion entier vers ASCII

```

--- logic_arithmetic.s ---

.global main

.text
main:
    xor    %eax, %eax
    mov    $9, %ebx
    add    %ebx, %eax
    sub    $2, %eax
    call   itoa
    ret

```

```

--- itoa.s ---

.global itoa
.global tab

.data
tab:
    .zero    1
    .string  "\n"

.text
itoa:
    add     $48, %eax
    mov     %al, tab
    ret

```

Modifiez le fichier `logic_arithmetic.s` afin d'appeler la fonction `itoa` (*integer to string conversion*). Assemblez `logic_arithmetic.s`, `itoa.s` et `disco/toolchain/build/startup/crt0.s` (fichier de startup minimal) puis réalisez l'édition des liens du projet.

```

as --32 -g logic_arithmetic.s -o obj/logic_arithmetic.o

as --32 -g itoa.s -o obj/itoa.o

ld -melf_i386 -T ../toolchain/build/script/linker_script_minimal.ld obj/crt0.o
obj/logic_arithmetic.o obj/itoa.o -o bin/logic_arithmetic

```

Quelle taille fait le tableau d'adresse `tab` ? Précisez son contenu au démarrage du programme et proposez une écriture équivalente en langage C. Constatez qu'un label peut pointer sur du code (`_start`, `main` ou `itoa`) ou des données statiques (`tab`).

Que représente la constante décimale 48 et en quoi cela assure une conversion entier vers ASCII (uniquement pour un chiffre compris entre 0 et 9) ?

En utilisant GDB (mode pas à pas), vérifiez la valeur contenu dans `EAX` après conversion par la fonction `itoa` et avant la fin de la fonction `main` (instruction `RET`). Vérifiez également le contenu du tableau `tab` avant et après exécution de la fonction `itoa`.

```

(gdb) la a
(gdb) b main
(gdb) r
(gdb) x/3xb 0x<tab_address>
(gdb) s
... Go to and execute itoa function
(gdb) x/3xb (char*)&tab
(gdb) s

```


V. Fonction d'affichage printf

```
--- logic_arithmetic.s ---
.global main

.text
main:
    xor    %eax, %eax
    mov    $9,    %ebx
    add    %ebx, %eax
    sub    $2,    %eax
    call   itoa
    call   printf
    ret
```

```
--- printf.s ---
.global printf

.text
printf:
    mov    $3,    %edx
    mov    $tab, %ecx
    mov    $1,    %ebx
    mov    $4,    %eax
    int    $0x80
    ret
```

Modifiez le fichier `logic_arithmetic.s` afin d'appeler la fonction `printf`. Assemblez les fichiers `logic_arithmetic.s`, `itoa.s`, `printf.s` et `disco/toolchain/build/startup/crt0.s` (fichier de startup minimal) puis réalisez l'édition des liens du projet. Exécutez le binaire de sortie et analysez le projet assembleur en s'aidant de GDB.

```
as --32 -g logic_arithmetic.s -o obj/logic_arithmetic.o
as --32 -g printf.s -o obj/printf.o
ld -melf_i386 -T ../toolchain/build/script/linker_script_minimal.ld obj/crt0.o
obj/logic_arithmetic.o obj/itoa.o obj/printf.o -o bin/logic_arithmetic
./bin/logic_arithmetic
```

La fonction standard du langage C `printf` réalise trois traitements distincts. Une première étape optionnelle de conversion de valeurs typées aux formats entiers ou flottants vers une chaîne de caractères (`%d`, `%u`, `%lu`, `%llu`, `%f`, `%lf`, etc). La seconde étape consiste à construire une chaîne de caractères à transmettre vers la sortie standard du système `stdout` (shell courant ou autre sortie en mode texte). La dernière étape consiste à transmettre la chaîne de caractères construite au noyau du système chargé de l'envoyer vers le flux standard de sortie `stdout`. Nous allons nous intéresser à cette dernière étape durant cet exercice, en envoyant la chaîne de caractères précédemment construite par la fonction `itoa`.

L'instruction `int 0x80` implémente un appel système 32-bit x86¹³ permettant de donner la main au noyau Linux en lui passant des arguments par registres. Il s'agit d'une interruption logicielle (interrompt l'exécution synchrone d'un programme). Les registres utilisés en x86 (`EAX`, `EBX`, `ECX` et `EDX`) sont documentés et seront toujours les mêmes pour un appel système sur noyau Linux. Ces registres dépendent donc de la technologie de noyau (Linux, Hurd, XNU, Minix, NT, ...) sur laquelle est portée le système d'exploitation (distribution GNU/Linux, Mac OS X, Windows, ...). Observons les opérandes en x86 sous Linux :

- `EAX` : fixe la fonction noyau à exécuter et donc la nature de l'appel système. Fonction `write` dans notre cas
- `EBX` : Mode d'accès au fichier
- `ECX` : Pointeur vers la chaîne de caractères à transmettre, soit `tab` l'adresse du tableau statique `tab[3] = "?\n"` dans notre cas
- `EDX` : Taille en octet de la chaîne de caractères à transmettre, soit 3 dans notre cas

¹³ En assembleur 64-bit x64, l'appel système est implémenté par l'instruction `syscall` (et non `int 0x80`).

VI. Suite de Fibonacci

```
.global main

.macro
CONVERT_TO_ASCII_AND_PRINT
    mov    %eax,tmp_eax
    mov    %edx,tmp_edx
    call   itoa
    call   printf
    mov    tmp_eax,%eax
    mov    tmp_edx,%edx
.endm

.comm tmp_eax, 4
.comm tmp_edx, 4

.text
main:
    xor    %eax, %eax
    CONVERT_TO_ASCII_AND_PRINT
    mov    $1, %eax
    CONVERT_TO_ASCII_AND_PRINT
    xor    %esi, %esi
    xor    %edx, %edx
.L0:
    mov    %eax, %edi
    add    %esi, %eax
    mov    %edi, %esi
    CONVERT_TO_ASCII_AND_PRINT
    add    $1, %edx
    cmp    $5, %edx
    jb     .L0
    ret
```

Assemblez `fibonacci.s`, `itoa.s`, `printf.s` et `disco/toolchain/build/startup/crt0.s` (fichier de *startup* minimal) puis réalisez l'édition des liens du projet. Exécutez le binaire de sortie et analysez le projet assembleur en vous aidant de GDB.

```
as --32 -g fibonacci.s -o obj/fibonacci.o
```

```
ld -melf_i386 -T ../toolchain/build/script/linker_script_minimal.ld obj/crt0.o
obj/fibonacci.o obj/itoa.o obj/printf.o -o bin/fibonacci
```

```
./bin/fibonacci
```

Pourquoi être passé par une sauvegarde puis restitution de contexte vers deux variables non initialisées (`.comm`) `tmp_eax` et `tmp_edx` avant d'appeler les fonctions `itoa` et `printf` ? Constatez que l'utilisation d'une macro allège la lecture du programme

En vous aidant de la documentation technique (*datasheet*) constructeur Intel présente dans `tp/doc/architectures-software-developer-manuals-vol2.pdf` à la page 474/1284, expliquez le fonctionnement de l'instruction `JB` (*Jump if Below*).

