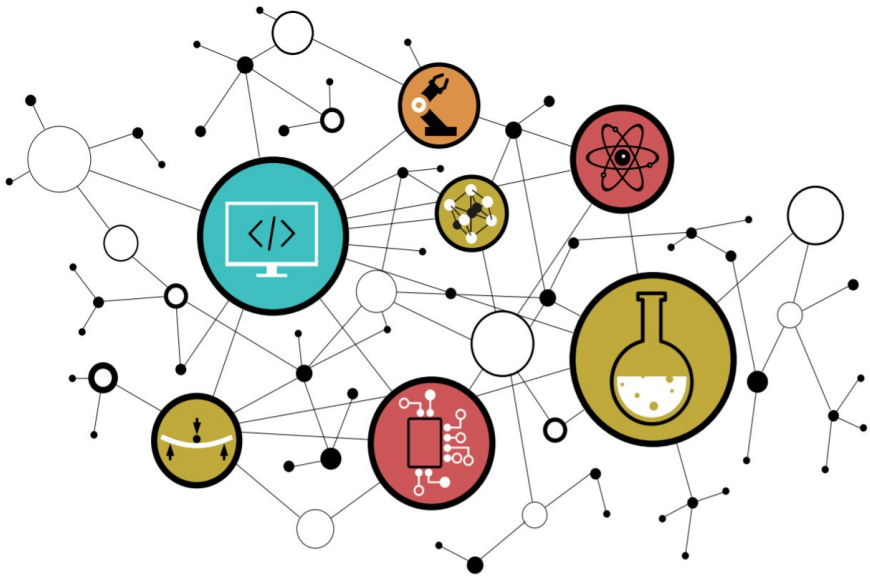
TRAVAUX PRATIQUES

ASSEMBLEUR X86 ET BIBLIOTHÈQUE STATIQUE



Hugo Descoubes – neku – GNU\Linux 5.8 - Ubuntu 20.04 LTS – LibreOffice 6.4.6.2 - 2021

SOMMAIRE

3. ASSEMBLEUR X86 ET BIBLIOTHÈQUE STATIQUE

3.1. Instructions arithmétiques et logiques
3.2. Debugger GDB
3.3. Fonction de conversion entier vers ASCII
3.4. Fonction d'affichage printf
3.5. Suite de Fibonacci
3.6. Bibliothèque statique

3. ASSEMBLEUR X86 ET BIBLIOTHÈQUE STATIQUE

```
int main (void)
{

return 0 ;
}
```

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

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, 8-16-32-64bits, entier voire flottant, VLIW ou superscalaire, vectoriel ou scalaire, etc) sur des technologies différentes (x86, x64, ARM, MIPS, C6000, PIC18, etc). L'assembleur présenté ci-dessus est par exemple de l'assembleur 32bits IA-32 (Intel Architecture 32 bits) 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:    instruction    opérandes
```

- Label : un label ou étiquette, est une référence symbolique (simple chaîne de caractères) représentant l'adresse mémoire logique (emplacement) de la première instruction 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 voire d'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, etc. 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.

Syntaxe assembleur AT&T proposée par GNU AS

Syntaxe Intel	Syntaxe AT&T

main:
push ebp
mov ebp,esp
mov eax, 0
pop ebp
ret
	main:
	pushl %ebp
	movl %esp,%ebp
	movl $0,%eax
	popl %ebp
	ret

L'assembleur x86 est une ISA développée historiquement par Intel pour son CPU 16bits 8086 produit en 1978. Les générations suivantes de processeurs Intel et AMD sorties dans les années 80 (80286, 80386, etc) 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 64bits 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, Corei, etc).

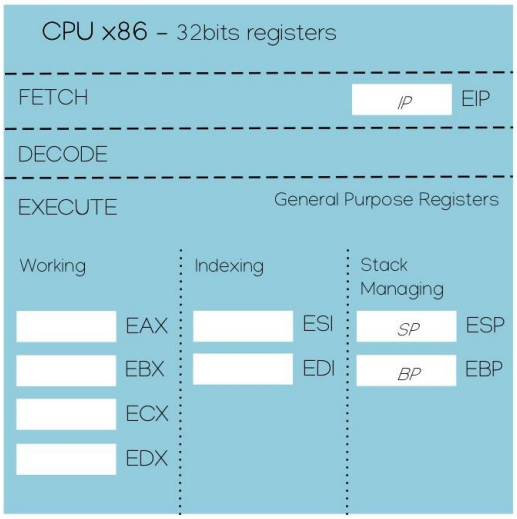
Le langage C et sa syntaxe sont maintenant normalisés et standardisés depuis des décennies même si la norme continue d'évoluer (normes ANSI C, C89, C99, C18, etc). En revanche, différentes syntaxes assembleur liées à la chaîne de compilation et aux outils de développement (ouverts ou fermés, libres ou 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 placées dans le même sens (destination à droite en syntaxe AT&T). Observons quelques particularités de la syntaxe AT&T :

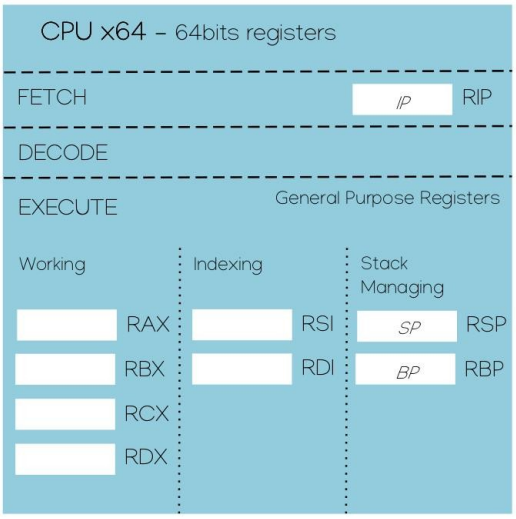
- % : signifie que l'opérande qui suit est un registre CPU (mode d'adressage registre)
- $: signifie que l'opérande que suit est une constante (mode d'adressage immédiat)
- suffixe d'instruction : signifie que l'instruction manipule des opérandes d'une certaine longueur en octets : b=byte=8bits=1octet, s=short=16bits=2octets, l=long=32bits=4octets et q=quad=64bits=8octets. Ce suffixe est facultatif à l'édition.
- (%registre) ou ($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 (lecture ou écriture) à la case mémoire pointée par l'adresse BP – 4o

```
    .global main

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

Un programme assembleur intégrera également certaines directives d'assemblage préfixées par un point (.global, .text, .macro, etc). 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), etc (https://ftp.gnu.org/old-gnu/Manuals/gas-2.9.1/html_chapter/as_7.html).

Registres CPU x86-64 et environnement d'exécution assembleur

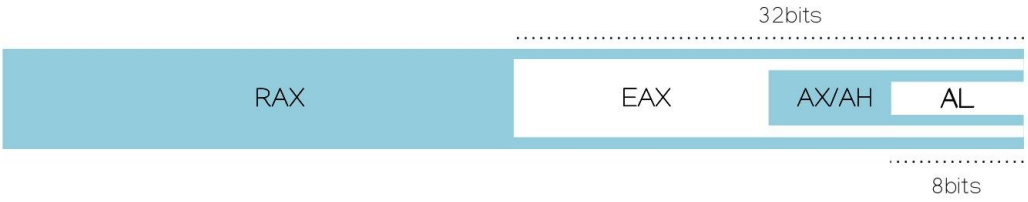




Les schémas ci-dessus présentent les principaux registres d'usages généralistes présents dans les CPU d'architectures compatibles x86 (32bits) et x64 (64bits). Rappelons que ces architectures restent rétrocompatibles avec le CPU 16bits 8086 datant de 1978. Ces registres sont présents dans chaque CPU d'une machine multicœurs (un cœur est l'ensemble CPU, MMU et Caches locaux). Cependant, 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, etc) : registre contenant les différents flags d'états associés aux unités d'exécution arithmétiques et logiques (carry, zero, sign, etc). Utilisé notamment afin d'implémenter des sauts conditionnels (if, else if, while, for, etc).
- 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 courant.
- Registres vectoriels (XMM0 à XMM15 128bits extension ISA SSE, YMM0 à YMM15 256bits extension ISA AVX et ZMM0 à ZMM15 512bits 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).
- Registres de test (TR3 à DR7), registres de debug (DR0 à DR7) et registres d'accès à la table de pagination mémoire (GDTR, LDTR, IDTR)

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 16bits et 8 bits 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 8-16-32-64bits imbriquées les unes dans les autres sur architectures x86-64 (respectivement AL/AH 8bits, AX 16bits, EAX 32bits et RAX 64bits). L'exemple donné ci-dessous sur le registre 64bits RAX peut être étendu aux registres de travail généralistes du CPU.



3.1. 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
```

- Se placer dans le répertoire de travail *disco/asm*. Assembler les fichiers *logic_arithmetic.s* et *disco/toolchain/build/startup/crt0.s* (fichier de startup minimal). En continuité du chapitre sur la compilation et l'édition des liens, utiliser le script *linker* minimal et réaliser l'édition des liens du projet. Analyser le programme assembleur *logic_arithmetic.s*

```
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
```

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

- Des deux instructions implémentant un même traitement proposées à la question précédente (XOR vs MOV $0), laquelle offre un 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 ?

3.2. Debugger GDB

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 GNU (http://www.gdbtutorial.com/).

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

- Assembler jusqu'à l'édition des liens incluse le fichier *logic_arithmetic.s*. Ne pas oublier d'ajouter l'option de debug *-g* à l'assemblage. Lancer ensuite une session de debug avec GDB

```
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

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 élémentaires mais très puissants d'analyse. A force de persévérance, aucun *bug* ne peut se cacher indéfiniment !

- Suivre et comprendre la séquence de commandes GDB proposée ci-dessous

```
(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
```

3.3. 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
```

- Modifier le fichier *logic_arithmetic.s* afin d'appeler la fonction *itoa* (integer to string conversion). Assembler les fichiers *logic_arithmetic.s*, *itoa.s* et *disco/toolchain/build/startup/crt0.s* (fichier de startup minimal) puis réaliser l'édition des liens du projet. Analyser le projet assembleur

```
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* ? Préciser son contenu au démarrage du programme. Proposer une écriture équivalente en langage C. *Constater 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), vérifier la valeur contenu dans EAX après conversion par la fonction *itoa* et avant la fin de la fonction *main* (instruction RET). Vérifier 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
```

3.4. 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
```

- Modifier le fichier *logic_arithmetic.s* afin d'appeler la fonction *printf*. Assembler les fichiers *logic_arithmetic.s, itoa.s, printf.s* et *disco/toolchain/build/startup/crt0.s* (fichier de startup minimal) puis réaliser l'édition des liens du projet. Exécuter le binaire de sortie et analyser 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 3 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 32bits x86 permettant de donner la main au noyau Linux en lui passant des arguments par registres. Il s'agit d'une interruption logicielle (interrompre l'exécution synchrone d'un programme). Un appel système en assembleur 64bits x64 est implémenté par l'instruction *syscall*. 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, etc) sur laquelle est portée le Système d'exploitation (Distributions GNU\Linux, Mac OS X, Windows, etc). Observons les opérandes en x86 sous Linux (http://www.lxhp.in-berlin.de/lhpsysc0.html) :

- EAX : Fixe la fonction noyau à exécuter et donc la nature de l'appel système. Fonction *write* dans notre cas
- EBX : Modes 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

3.5. 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
```

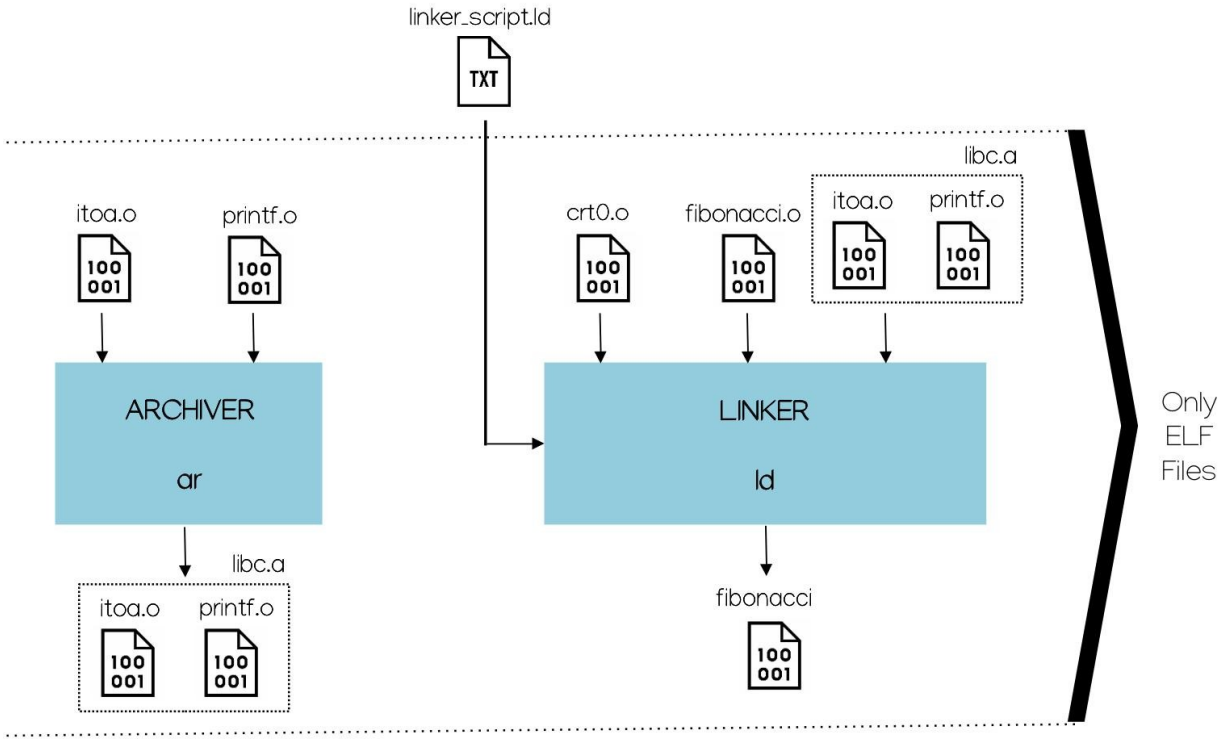
- Assembler les fichiers *fibonacci.s, itoa.s, printf.s* et *disco/toolchain/build/startup/crt0.s* (fichier de startup minimal) puis réaliser l'édition des liens du projet. Exécuter le binaire de sortie et analyser le projet assembleur. S'aider de GDB pour votre analyse

```
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 *iota* et *printf* ? *Constater que l'utilisation d'une macro allège la lecture du programme*

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

3.6. Bibliothèque statique



- Générer une bibliothèque statique (GNU AR ou *archiver*) et la lier manuellement durant l'édition des liens. Une bibliothèque statique est une archive (généralement avec une extension .a comme archive), soit la concaténation de fichiers objets binaires ELF ré-adressables. Une bibliothèque statique n'est pas un fichier ELF, mais en revanche elle regroupe de fichiers ELF. Analyser le processus de compilation et d'édition des liens du projet. Valider l'exécution du binaire de sortie. *Observer avec le service readelf de binutils le contenu de notre modeste bibliothèque statique nommée pour l'occasion libc.a, à l'image de la bibliothèque standard du C libc.so rangée dans /lib dans une arborescence de fichiers Unix-like. Au démarrage du système, la bibliothèque standard libc.so est chargée en mémoire principale puis partagée par tous les programmes en cours d'exécution sur la machine ayant besoin d'exécuter des fonctions standards (dont le processus init).*

```
as --32 -g ../toolchain/build/startup/crt0.s -o obj/crt0.o

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

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

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

ar -rcs lib/libc.a obj/itoa.o obj/printf.o

ld -melf_i386 -T ../toolchain/build/script/linker_script_minimal.ld
obj/crt0.o obj/fibonacci.o lib/libc.a -o bin/fibonacci

readelf -h lib/libc.a
```


notes