Exemple de compilation d'un programme simple

Ces notes décrivent la manipulation réalisée au cours le 28 avril, lors de laquelle un programme simple rédigé en C a été compilé. L'ordinateur utilisé est doté d'un processeur *Pentium M*, dont l'architecture est une extension à 32 bits de celle vue au cours. Le système d'exploitation installé est *Linux*.

1 Code source

On considère le programme C suivant, placé dans un fichier source `max.c`.

```
int max(int x, int y)
{
  if (x <= y)
    return y;
  else
    return x;
}
```

Ce programme définit une fonction `max`, prenant deux arguments entiers signés `x` et `y`, et retournant une valeur entière égale au maximum de ces deux valeurs.
Note: Dans l'architecture considérée, les entiers sont représentés sur 32 bits, par la technique du complément à deux.

2 Code assembleur

2.1 Compilation

Le programme C `max.c` est compilé en code assembleur grâce à la commande

```
gcc -S -masm=intel max.c
```

Explication de cette commande:

- `gcc` est le nom du compilateur.
- `-S` est une option demandant l'arrêt de la compilation après la génération de code assembleur.

- `-masm=intel` indique que le code assembleur produit doit être exprimé dans le dialecte "intel". Cette précision est utile, car plusieurs langages d'assemblage de syntaxes différentes ont été définis pour l'architecture considérée.

- `max.c` est le nom du fichier source compilé.

Le résultat de la compilation est fourni dans un fichier `max.s` dont le contenu est donné ci-après (des numéros de ligne ont été ajoutés afin de pouvoir faire facilement référence à ce code):

```
 1:         .file   "max.c"
 2:         .intel_syntax
 3:         .text
 4: .globl max
 5:         .type   max, @function
 6: max:
 7:          push    %ebp
 8:          mov     %ebp, %esp
 9:          sub     %esp, 4
10:          mov     %eax, DWORD PTR [%ebp+8]
11:          cmp     %eax, DWORD PTR [%ebp+12]
12:          jg      .L2
13:          mov     %eax, DWORD PTR [%ebp+12]
14:          mov     DWORD PTR [%ebp-4], %eax
15:          jmp     .L1
16: .L2:
17:          mov     %eax, DWORD PTR [%ebp+8]
18:          mov     DWORD PTR [%ebp-4], %eax
19: .L1:
20:          mov     %eax, DWORD PTR [%ebp-4]
21:          leave
22:          ret
23:          .size   max, .-max
24:          .section        .note.GNU-stack,"",@progbits
25:          .ident  "GCC: (GNU) 3.4.2 20041017 (Red Hat 3.4.2-6.fc3)"
```

Comme on le voit, la syntaxe de ce fichier diffère légèrement de celle vue au cours:

- Les noms de registre apparaissant dans les opérandes des instructions sont précédés du symbole "%".

- Les registres EBP, ESP et EAX sont des registres de 32 bits similaires aux registres (de 16 bits) BP, SP et AX de l'achitecture simplifiée vue au cours.
- Le préfixe "`DWORD`" désigne une donnée de 32 bits.
- Le marqueur "`PTR`" précède toujours un adressage indirect, ou indirect indexé.
- Les directives présentes aux lignes 1–5 et 23–25 servent, notamment, à fournir les informations qui permettront de lier ce code à d'autres fonctions en vue de construire un programme exécutable.

2.2 Structure du code produit

2.2.1 Point d'entrée

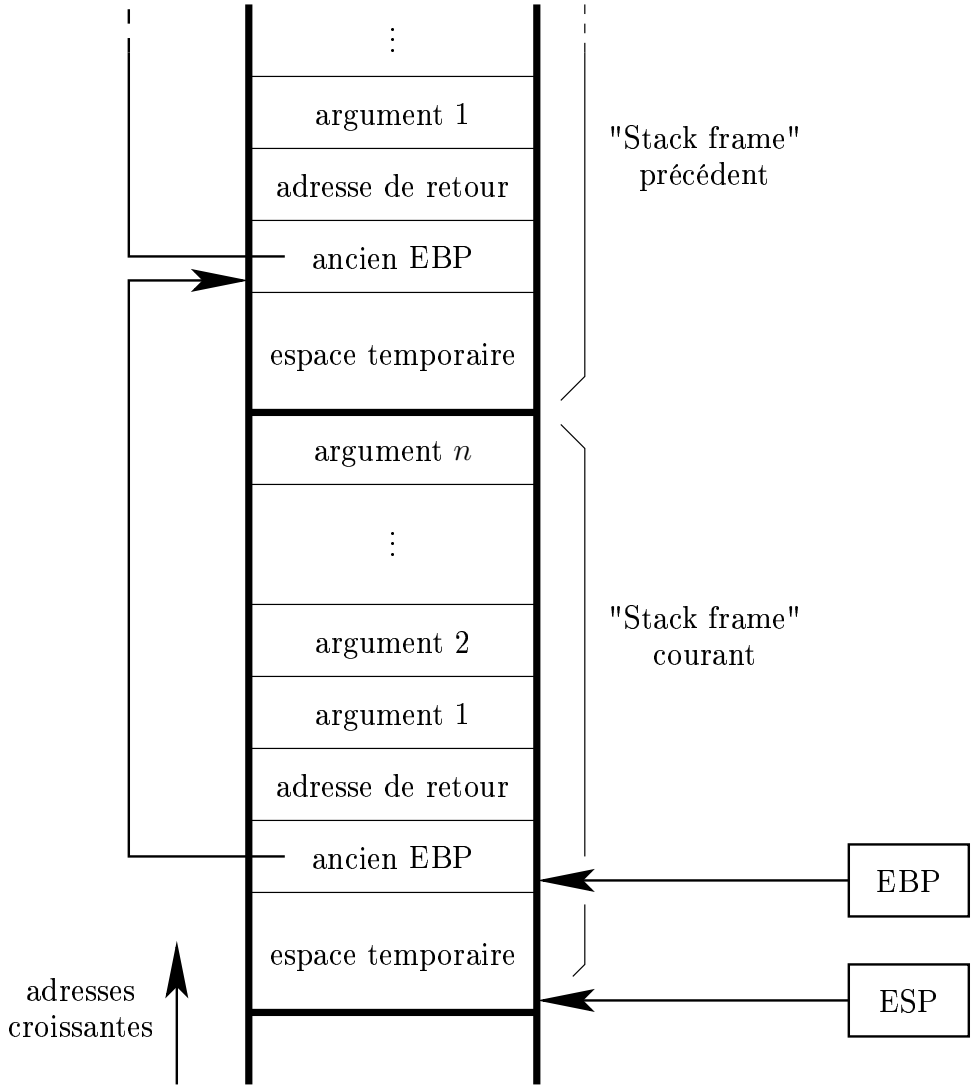
La ligne 6 définit le symbole `max` correspondant au point d'entrée de la fonction (c'est-à-dire, l'*offset* dans le segment de code de sa première instruction).

2.2.2 Construction d'un *stack frame*

Les lignes 7–9 visent à achever la construction d'un *stack frame* (commencée par le programme qui a invoqué la fonction). Un *stack frame* est une structure de données placée sur la pile, contenant les informations relatives à une invocation de fonction en cours:

- l'*adresse de retour* de la fonction (c'est-à-dire, l'*offset* dans le segment de code de l'instruction qui suivra la fin de l'exécution de la fonction).
- la valeur des *arguments* éventuels de la fonction.
- la valeur des *variables locales* définies dans la fonction.
- des données permettant de restaurer l'état initial de la pile après l'exécution de la fonction.

Dans l'architecture considérée, un *stack frame* possède la structure suivante:



Principes:

- Le *stack frame* situé au sommet de la pile correspond à l'invocation non terminée la plus récente ("*stack frame* courant").
- Le registre EBP fournit un point d'accès pratique vers les données composant le *stack frame* courant.
- Avant d'invoquer une fonction, le programme appelant place sur la pile la valeur des arguments de cette fonction, en ordre inverse de leur définition.
- L'instruction d'invocation de fonction ("CALL") empile l'adresse de retour (32 bits dans l'architecture considérée).

- La première instruction d'une fonction (dans notre exemple, la ligne 7) est chargée d'empiler la valeur courante du registre EBP. Cette donnée permettra de restaurer l'état de la pile à la fin de l'invocation courante.
- La deuxième instruction (ligne 8) met à jour EBP afin de le faire pointer vers le nouveau *stack frame* construit.
- Des cellules mémoire supplémentaires peuvent être réservées afin de constituer un espace de stockage temporaire utilisable par la fonction (par exemple, pour y gérer des variables locales). Dans notre exemple, on crée à la ligne 9 un espace temporaire de quatre octets.

2.2.3 Cœur de la fonction

Les lignes 10 et 11 lisent 32 bits de données aux adresses EBP$+8$ et EBP$+12$ et les comparent. En observant la structure de *stack frame* présentée dans la section précédente, on voit que ces deux valeurs correspondent respectivement aux arguments `x` et `y` de la fonction `max`.

Ensuite, à la ligne 12, si la comparaison des deux valeurs en arithmétique signée conclut à $\texttt{x} > \texttt{y}$, on effectue un saut vers le label `.L2`. Les instructions présentes à cette adresse (lignes 17 et 18) recopient alors la valeur de `x` à l'adresse EBP-4, qui correspond à l'espace temporaire de 32 bits alloué lors de la création du *stack frame*

En revanche, si la comparaison effectuée à la ligne 12 conclut à $\texttt{x} \leq \texttt{y}$, alors les instructions présentes au lignes 13 et 14 recopient dans l'espace temporaire la valeur de `y`. Quelle que soit l'issue de la comparaison, on constate donc que l'exécution atteint la ligne 20 avec, dans l'espace temporaire alloué, la valeur maximale de l'ensemble $\{\texttt{x}, \texttt{y}\}$.

L'instruction présente à la ligne 20 recopie enfin la contenu de la zone temporaire dans EAX qui, par convention, récupère la valeur retournée par la fonction.

2.2.4 Terminaison

L'instruction "LEAVE" présente à la ligne 21 effectue l'opération réciproque des lignes 7 et 8, c'est-à-dire, restaure l'état de la pile à sa valeur existante au début de l'invocation de la fonction.

Finalement, le retour vers le code appelant a lieu à la ligne 22.

2.3 Discussion

Le code examiné ici résulte d'une traduction directe par le compilateur des instructions du programme source. Il présente plusieurs défauts entraînant une certaine inefficacité:

- L'allocation d'un espace temporaire afin d'y placer la valeur de retour est inutile. En effet, il suffit de placer directement cette valeur de retour dans EAX dès qu'elle est connue.
- L'instruction présente à la ligne 17 est redondante: Elle place dans le registre EAX une valeur égale à celle qui s'y trouve déjà.

2.4 Compilation avec optimisation

Le compilateur utilisé est cependant capable d'effectuer, à la demande, des opérations d'*optimisation* visant à améliorer l'efficacité du code produit. Compilons à nouveau le fichier source `max.c`, cette fois à l'aide de la commande suivante:

```
gcc -S -masm=intel -O3 max.c
```

Explication: L'option `-O3` spécifie un niveau d'optimisation maximal du code généré.

Le résultat de la compilation (annoté par des numéros de ligne) est donné ci-après:

```
 1:         .file   "max.c"
 2:         .intel_syntax
 3:         .text
 4:         .p2align 2,,3
 5: .globl max
 6:         .type   max, @function
 7: max:
 8:         push    %ebp
 9:         mov     %ebp, %esp
10:         mov     %eax, DWORD PTR [%ebp+12]
11:         cmp     DWORD PTR [%ebp+8], %eax
12:         jle     .L1
13:         mov     %eax, DWORD PTR [%ebp+8]
14: .L1:
15:         leave
```

```
16:         ret
17:         .size   max, .-max
18:         .section        .note.GNU-stack,"",@progbits
19:         .ident  "GCC: (GNU) 3.4.2 20041017 (Red Hat 3.4.2-6.fc3)"
```

Comme on le voit, les deux faiblesses observées dans le code assembleur non optimisé ont cette fois été correctement détectées et corrigées par le compilateur.

3 Code machine

On est maintenant à même d'*assembler* le code assembleur obtenu à l'issue de la compilation de `max.c`, c'est-à-dire, le traduire en *code machine*. Cela s'effectue grâce à la commande suivante:

```
gcc -c max.s
```

Explication de cette commande:

- `gcc` est le nom du compilateur.
- `-c` est une option demandant l'arrêt de la compilation après la génération de code machine.
- `max.s` est le nom du fichier contenant le code assembleur à traduire.

Le résultat de l'assemblage prend la forme d'un fichier `max.o` qui constitue un *module*, c'est-à-dire un fragment de code machine pouvant être combiné à d'autres modules pour former un programme exécutable. La commande suivante permet d'examiner les instructions machine contenues dans le module:

```
objdump -d -Mintel:386 max.o
```

Explication de cette commande:

- `objdump` est le nom du programme capable de visualiser le contenu d'un module.
- `-d` est une option demandant le désassemblage (c'est-à-dire, la traduction en mnémoniques et en opérandes) du code machine contenu dans le module.

- `-Mintel:386` est une option spécifiant le dialecte utilisé pour exprimer les mnémoniques et les modes d'adressage.
- `max.o` est le nom du fichier contenant le module.

Le résultat produit par cette commande pour le code assembleur optimisé (obtenu à la section 2.4) est donné ci-après:

```
max.o:     file format elf32-i386

Disassembly of section .text:

00000000 <max>:
   0:   55                      push   ebp
   1:   89 e5                   mov    ebp,esp
   3:   8b 45 0c                mov    eax,DWORD PTR [ebp+12]
   6:   39 45 08                cmp    DWORD PTR [ebp+8],eax
   9:   7e 03                   jle    e <max+0xe>
   b:   8b 45 08                mov    eax,DWORD PTR [ebp+8]
   e:   c9                      leave
   f:   c3                      ret
```

Explications:

- La première colonne donne l'*offset* dans le segment de code des instructions (en hexadécimal).
- La deuxième colonne donne le code machine (c'est-à-dire, la valeur des octets présents dans le segment de code). Ce code est directement exécutable par le processeur de l'ordinateur. On voit que chaque instruction est représentée par un à trois octets de code machine.
- La troisième colonne donne une traduction du code machine en instructions assembleur. (On remarque que la syntaxe utilisée diffère légèrement de celle utilisée par le compilateur.)