Yves Bailly

Le langage Ada : un peu d'assembleur

Le langage Ada est un langage généraliste de haut niveau. Mais l'une de ses vocations premières était l'informatique embarquée, au plus proche du matériel. Comme tous les langages offrant la possibilité de « toucher à l'électronique », Ada permet d'intégrer directement du code assembleur au sein d'un programme plus vaste.

Naturellement, la manière d'intégrer l'assembleur ainsi que le contenu même de ce code dépendent fortement de la plate-forme visée, aussi bien de l'architecture matérielle que du compilateur utilisé. Ce que nous allons voir ici concerne les machines à base de processeur de la famille 80x86, sous environnement GNU/Linux, en utilisant le compilateur Ada GNAT tel qu'il se trouve intégré à la suite GCC.

Certains s'interrogent peut-être sur la pertinence d'utiliser l'assembleur, surtout de nos jours où le moindre morceau de puce peut être manipulé par une API en langage C, généralement fournie par le fabricant. On pourrait alors se contenter d'interfacer l'API C en Ada, comme nous l'avons vu précédemment (voir *Linux Magazine* 85).

Mais, il est des situations où l'assembleur peut s'avérer indispensable. Il serait (théoriquement) parfaitement possible d'écrire le pilote d'une carte vidéo en Ada, bénéficiant ainsi de toute la puissance et robustesse du langage – mais dans des cas de ce genre, les petites latences induites par une couche d'interface avec les fonctions C sont tout simplement inacceptables, car dégradant les performances. Plus généralement, dans une situation où la vitesse d'exécution est primordiale (imaginez la simulation d'un modèle météorologique ou astronomique), le recours à l'assembleur peut permettre d'optimiser très fortement certaines portions du programme bien plus efficacement qu'un compilateur – nous verrons un exemple dans cet article.

Enfin, imaginez vouloir écrire un système d'exploitation en Ada. Le recours à l'assembleur est incontournable à certains endroits clefs – même un système écrit en C comme Linux n'y échappe pas. Et si vous pensez qu'écrire un système en Ada est une idée farfelue et irréaliste pour autre chose qu'une sonde martienne, jetez un œil au projet *Toy Lovelace* **[1]**.

Intégrer l'assembleur

Commençons par l'intégration d'une instruction assembleur au sein d'une procédure Ada (dans un programme `rien.adb`) :

```
with System.Machine_Code;
use System.Machine_Code;
procedure Rien is
begin
   Asm("nop");
end Rien;
```

L'assembleur est intégré au code Ada au moyen de la procédure `Asm()`, déclarée dans le paquetage `System.Machine_Code`. Le premier paramètre de `Asm()` est une chaîne de caractères contenant les instructions à exécuter. Ici, nous invoquons l'instruction `nop`, synonyme de *no operation*, c'est-à-dire très précisément « ne rien faire ».

L'assembleur étant ce qu'il est, il peut s'avérer fort utile de pouvoir consulter le code généré pour l'ensemble du programme, afin de voir précisément comment les instructions que nous écrivons sont intégrées au reste – surtout quand cela ne fonctionne pas. Pour cela, exécutez simplement :

```
$ gcc -c -S -gnatp rien.adb
```

Ceci va générer un fichier `rien.s` contenant le code assembleur équivalent à l'ensemble du programme. Les paramètres passés à `gcc` sont :

- `-c` pour seulement compiler, sans chercher à effectuer d'édition des liens ;
- `-S` pour générer, justement, le fichier `rien.s` ;
- `-gnatp` désactive (presque) tous les mécanismes de contrôle internes au langage Ada (vérification des intervalles de valeurs, ce genre de choses), ce qui allège le code généré.

Sur notre exemple, on obtient ceci :

```
        .file   "rien.adb"
        .text
.globl _ada_rien
        .type   _ada_rien, @function
_ada_rien:
.LFB3:
        pushl   %ebp
.LCFI0:
        movl    %esp, %ebp
.LCFI1:
#APP
        nop
#NO_APP
        popl    %ebp
        ret
.LFE3:
        .size   _ada_rien, .-_ada_rien
...
```

Plus encore un certain nombre de lignes propres à l'utilisation du *runtime* Ada par Gcc. Ce qui nous intéresse commence à la ligne `_ada_rien` : on retrouve le nom de notre procédure principale, préfixé par `_ada_`. La plupart des instructions ne concernent que la gestion de la pile (elles n'apparaîtraient pas si nous avions passé l'option `-fomit-frame-pointer` à `gcc`).

Finalement, notre code assembleur se trouve encadré par les lignes #APP et #NO_APP : il est donc facilement repérable – du moins dans un petit programme.

Mais passons à quelque chose de plus intéressant.

Entrées et sorties

L'instruction assembleur div effectue une division entière, en retournant le quotient et le reste. Normalement, pour obtenir ces deux valeurs, il est nécessaire d'effectuer deux opérations. Écrivons une procédure qui invoque div pour les obtenir directement :

```
1  with Ada.Text_IO;
2  with System.Machine_Code;
3  with Interfaces;
4  procedure Div is
5     use ASCII;
6     use Interfaces;
7     use System.Machine_Code;
8     use Ada.Text_IO;
9     procedure Div(dividende: in     Integer;
10                  diviseur : in     Integer;
11                 quotient :    out Integer;
12                 reste    :    out Integer) is
13     quotient_interne: Integer_32 := 0;
14     reste_interne   : Integer_32 := 0;
15   begin
16     Asm("movl $0, %%edx " & LF & HT &
17         "div %%ebx "      & LF & HT &
18         "movl %%eax, %0 " & LF & HT &
19         "movl %%edx, %1 ",
20         Inputs => (Integer_32'Asm_Input("a",
           Integer_32(dividende)),
21                    Integer_32'Asm_Input("b",
                      Integer_32(diviseur))),
22         Outputs => (Integer_32'Asm_Output("=m",
           quotient_interne),
23                     Integer_32'Asm_Output("=m",
                       reste_interne)),
24         Clobber => ("edx"));
25     quotient := Integer(quotient_interne);
26     reste    := Integer(reste_interne);
27   end Div;
28...
29 -- etc.
```

Voilà qui est sensiblement plus compliqué. Voyons cela étape par étape.

Pour commencer, le type Integer_32 introduit lignes 13 et 14 représente un entier signé codé sur 32 bits. Ce type est déclaré dans le paquetage Interfaces. On s'assure ainsi de parfaitement maîtriser le format des données que l'on va échanger avec l'assembleur. En effet, le type standard Integer est simplement défini comme étant « un type entier ». Sa taille (voire sa représentation) peut varier selon la plate-forme, aussi est-il indispensable d'utiliser un type parfaitement délimité quand on manipule l'assembleur, qui supporte assez mal l'approximation dans ce domaine.

Ensuite, vous constatez que la chaîne de caractères contenant l'assembleur est littéralement construite morceau par morceau (lignes 16 à 19), en insérant ce curieux & LF & HT entre deux lignes. Ces deux symboles, déclarés de type Character dans le paquetage ASCII, représentent les caractères ASCII « retour chariot » (*Line Feed*) et « tabulation horizontale » (*Horizontal Tab*). Ils sont introduits afin d'améliorer la présentation du code assembleur généré. Si nous avions écrit simplement ceci :

```
Asm("movl $0, %%edx " &
    "div %%ebx "      &
    "movl %%eax, %0 " &
    "movl %%edx, %1 ",
```

Voici ce que nous aurions obtenu en examinant le code source assembleur obtenu par gcc -S :

```
...
        movl    32(%esp), %ebx
#APP
        movl $0, %edx div %ebx movl %eax, 4(%esp) movl %edx, (%esp)
#NO_APP
        movl    4(%esp), %eax
...
```

Ce qui est, convenez-en, parfaitement illisible – et généralement refusé par l'assembleur GNU. L'ajout de ces deux caractères donne un résultat bien plus sympathique et passe-partout :

```
...
        movl    32(%esp), %ebx
#APP
        movl $0, %edx
        div %ebx
        movl %eax, 4(%esp)
        movl %edx, (%esp)
#NO_APP
        movl    4(%esp), %eax
...
```

Les entrées

Les entrées sont données par le paramètres Inputs de la procédure Asm() (lignes 20 et 21). Ce paramètre attend une valeur, ou un tableau de valeurs, d'un type retourné par l'attribut 'Asm_Input. Cet attribut, tout comme le contenu du paquetage System.Machine_Code, est dépendant de l'implémentation – ici, nous ne traitons que du compilateur GNAT, basé sur GCC. Répétons-le, un compilateur différent utilisera des moyens (types, procédures et attributs) différents pour obtenir un effet équivalent.

Le premier paramètre de l'attribut est une chaîne de caractères donnant une *contrainte* (*constraint*) sur la valeur que l'on souhaite passer à l'assembleur, laquelle valeur est donnée en deuxième paramètre. On retrouve là les éléments qui sont communiqués à l'instruction asm() de GCC. Pour reprendre un exemple de la documentation de GNAT, l'instruction suivante dans un code en C :

```
asm("fsinx %1 %0" : "=f" (resultat) : "f" (angle));
```

...se présenterait ainsi en Ada (du moins pour le compilateur GNAT) :

```
Asm("fsinx %1 %0",
    Outputs => Float'Asm_Output ("=f", resultat),
    Inputs  => Float'Asm_Input  ("f",  angle));
```

C'est plus long à écrire, mais cela semble plus lisible.

Naturellement, le type de la valeur donnée en deuxième paramètre doit être le même que le type utilisé en préfixe de l'attribut :Ada est un langage au typage strict. C'est pourquoi nous effectuons ici un transtypage, `dividende` et `diviseur` étant de type `Integer`, alors que nous voulons un type `Integer_32`. On obtient ainsi un minimum de vérifications, donc de robustesse.

Nous ne nous étendrons pas sur le contenu de la chaîne de caractères. Il correspond à la syntaxe propre à GCC (voir le chapitre « *Extensions to the C Language Family/Assembler Instructions with C Expression Operands* » et le suivant dans la documentation de GCC). Disons simplement qu'ici nous demandons à ce que la valeur (transtypée) de `dividende` soit placée dans le registre `EAX`, tandis que la valeur (transtypée) de `diviseur` est placée dans le registre `EBX`. Le compilateur se chargera lui-même de ces actions.

Les sorties

Vous l'aurez sans doute deviné, les sorties sont données par le paramètre `Outputs` de la procédure `Asm()`, en utilisant l'attribut `'Asm_Ouput` (lignes 22 et 23). Le principe général est le même que pour les entrées, sauf que le deuxième paramètre de l'attribut doit être une variable et non une simple valeur. C'est pourquoi il a été nécessaire de déclarer deux variables temporaires, `quotient_interne` et `reste_interne`, du type `Integer_32`.

Pour information, la contrainte `"=m"` indique que l'on désigne une sortie (le signe =) qui sera référencée par une adresse mémoire (la lettre `m`). En pratique, cela signifie que les écritures `%0` et `%1` dans le code assembleur seront remplacées par les adresses effectives des variables `quotient_interne` et `reste_interne`.

Intégrer gentiment

Enfin, notre code assembleur commence par placer la valeur 0 dans le registre `EDX` (ligne 16), car l'instruction `div` (dans notre contexte) suppose que la valeur à diviser est contenue dans la paire `EDX:EAX`. De plus, `EDX` est également utilisé pour contenir le reste de la division. Seulement, le registre `EDX` n'est référencé ni dans les paramètres d'entrées, ni dans les paramètres de sortie. Le compilateur ne peut donc pas « deviner » que le registre `EDX` est utilisé et modifié par notre code assembleur, car il n'analyse pas celui-ci. Cela n'a pas grande conséquence ici, mais cela pourrait être catastrophique dans un programme plus vaste : le compilateur pourrait parfaitement utiliser ce registre `EDX` pour ses besoins propres, notre code perturbant alors le déroulement du programme.

Aussi est-il généralement nécessaire de passer un quatrième paramètre à la procédure `Asm()`, nommé `Clobber` (ligne 24). Celui-ci attend une chaîne de caractères contenant les noms des registres utilisés par le code assembleur, séparés par une virgule. Par exemple, `"eax, ebx, ecx"`. Si nécessaire, le compilateur s'adaptera et le programme pourra se dérouler normalement.

Amélioration... ou pas

Généralement, les programmeurs qui utilisent l'assembleur ont le souci constant de l'optimisation. Le code de l'exemple précédent pourrait être rédigé ainsi :

```
Asm("cltd "       & LF & HT &
    "div %%ebx ",
    Inputs => (Integer_32'Asm_Input("a", Integer_32(dividende)),
               Integer_32'Asm_Input("b", Integer_32(diviseur))),
    Outputs => (Integer_32'Asm_Output("=a", quotient_interne),
                Integer_32'Asm_Output("=d", reste_interne)));
```

C'est-à-dire qu'on réduit au minimum les instructions insérées. Le changement essentiel se situe sur les paramètres de sortie : plutôt que d'indiquer une adresse mémoire, on nomme explicitement les registres remplis par l'instruction `div`. Par ailleurs, le paramètre `Clobber` a disparu, devenu inutile (car `EDX` est mentionné dans la deuxième contrainte).

Le tableau suivant compare la partie « intéressante » de l'assembleur généré, à gauche la première version, à droite la version « améliorée » :

```
   movl   $0, 4(%esp)
   movl   $0, (%esp)
   movl   28(%esp), %eax
   movl   32(%esp), %ebx
#APP
   movl $0, %edx
   div %ebx
   movl %eax, 4(%esp)
   movl %edx, (%esp)
#NO_APP
```

```
   movl   $0, 8(%esp)
   movl   $0, 12(%esp)
   movl   28(%esp), %eax
   movl   32(%esp), %ebx
#APP
   cltd
   div %ebx
#NO_APP
   movl   %eax, 8(%esp)
   movl   %edx, 12(%esp)
```

Mis à part l'utilisation de `cltd` pour mettre à 0 le registre `EDX`, ces deux codes sont en fait équivalents. Peut-être peut-on considérer qu'une version est plus « parlante » que l'autre, mais l'autre est moins « dangereuse » que l'une...

Calculs en virgule flottante

Si vous pratiquez le développement d'applications graphiques en 3D, il est probable que vous ayez fréquemment à calculer le sinus et le cosinus d'un angle donné. La plupart du temps, les deux sont d'ailleurs nécessaires dès que l'on manipule quelque chose ressemblant de près ou de loin à un cercle. Or, il existe une instruction assembleur qui permet justement de calculer ces deux valeurs en une seule fois : `fsincos`. Voilà qui semble un bon candidat si on se trouve dans un objectif d'optimisation.

Réalisons donc un petit programme pour vérifier que l'utilisation de l'assembleur apporte effectivement un gain de temps :

```
with Ada.Text_IO;           use Ada.Text_IO;
with Ada.Real_Time;         use Ada.Real_Time ;
with Ada.Numerics;          use Ada.Numerics;
with Interfaces;            use Interfaces;
with System.Machine_Code;   use System.Machine_Code;
with Ada.Numerics.Long_Elementary_Functions;
use Ada.Numerics.Long_Elementary_Functions;
procedure Sin_Cos is
   procedure Sin_Cos(angle: in     Long_Float;
                     sin_a:    out Long_Float;
                     cos_a:    out Long_Float) is
   begin
      sin_a := Sin(angle);
      cos_a := Cos(angle);
   end Sin_Cos;
   pragma Inline(Sin_Cos);
```

Les fonctions sinus et cosinus pour les flottants en double précisions (`Long_Float` en Ada) sont déclarées dans le paquetage `Ada.Numerics.Long_Elementary_Functions`, aux côtés de nombreuses autres fonctions élémentaires. Pour commencer, on crée une procédure classique réalisant ce que nous voulons, par des moyens « normaux ». Remarquez la directive `Inline` qui suit la procédure, elle informe le compilateur d'étendre le code correspondant « en ligne » afin d'économiser un appel de fonction.

Voyons maintenant la version assembleur :

```
procedure Sin_Cos_Asm(angle: in     Long_Float;
                      sin_a:    out Long_Float;
                      cos_a:    out Long_Float) is
begin
   Asm("fsincos",
     Inputs  => Long_Float'Asm_Input("0", angle),
     Outputs => (Long_Float'Asm_Output("=t", cos_a),
                 Long_Float'Asm_Output("=u", sin_a))
      );
end Sin_Cos_Asm;
pragma Inline(Sin_Cos_Asm);
```

Non, pas d'erreur : le code assembleur se résume au seul appel de `fsincos`. En fait, tout le travail pénible est effectué par les déclaration des entrées/sorties. L'instruction opère sur la valeur sur la pile du co-processeur (`ST0`) en la dépilant, puis empile le sinus *puis* le cosinus – on récupère donc les résultats dans `ST1` et `ST0`, respectivement. Ces résultats sont obtenus par les contraintes `"u"` et `"t"` (respectivement), comme toujours précédées sur signe `"="` signalant qu'il s'agit de valeurs de retour.

Mais comme `fsincos` attend également une valeur en `ST0`, lequel registre est également utilisé pour le résultat, on passe la contrainte `"0"` pour la déclaration de la valeur d'entrée, signalant ainsi qu'elle doit être placée dans le même emplacement que la première valeur de retour.

De fait, toutes les manipulations nécessaires de la pile du co-processeur sont automatiquement assurées par le compilateur, ce que l'on peut vérifier en jetant un œil à l'assembleur généré :

```
$ gcc -c -S -gnatp sin_cos.adb && cat sin_cos.s
...
sin_cos__sin_cos_asm.394:
.LFB5:
        pushl   %ebp
.LCFI12:
        movl    %esp, %ebp
.LCFI13:
        subl    $24, %esp
.LCFI14:
        movl    8(%ebp), %edx
        movl    12(%ebp), %eax
        movl    %eax, -24(%ebp)
        movl    16(%ebp), %eax
        movl    %eax, -20(%ebp)
        fldl    -24(%ebp)
#APP
        fsincos
#NO_APP
        fstpl   -8(%ebp)
        fstpl   -16(%ebp)
        fldl    -16(%ebp)
        fstpl   (%edx)
        fldl    -8(%ebp)
        fstpl   8(%edx)
        movl    %edx, %eax
        leave
        ret     $4
...
```

Naturellement, il convient tout de même de vérifier que l'on ne s'est pas trompé quelque part, par exemple en intervertissant les résultats.

Vous aurez sans doute remarqué que nous avons utilisé directement le type `Long_Float`. Cela fonctionne, car, dans notre situation précise (compilateur GNAT, machine 80x86), ce type correspond exactement aux nombres à virgule flottante sur 64 bits du processeur. Si toutefois vous vous placez sur une architecture différente, avec un compilateur différent, il peut s'avérer nécessaire d'effectuer un transtypage comme nous l'avons fait précédemment. Pour information, le paquetage Interfaces qui accompagne le compilateur GNAT comporte les types `IEEE_Float_32` et `IEEE_Float_64` pour les nombres à virgule flottante sur 32 et 64 bits respectant la norme IEEE.

Côté performances, le calcul de dix millions de sinus/cosinus par la méthode classique prend environ 5.86 secondes sur la machine de votre serviteur, tandis qu'en utilisant l'assembleur cela prend... moins d'une seconde.

Passer des adresses mémoires

Pour finir, voyons comment nous pouvons passer l'adresse de blocs de données depuis l'Ada vers l'assembleur. L'exemple consiste à reproduire une fonctionnalité de la procédure `Put_Line()`, permettant d'afficher une chaîne de caractères. L'affichage sera en fait réalisé par la fonction système `write()`. Enfin, nous pourrons saluer le monde comme il se doit.

Une adresse mémoire est représentée en Ada par le type `Address`, déclaré dans le paquetage `System`, l'adresse d'un objet étant fournie par l'attribut `'Address`. Contrairement au langage C, il ne s'agit pas (forcément) d'un type équivalent à un entier. C'est généralement le cas, en particulier sur plate-formes x86, mais pas forcément. Pour un maximum de portabilité, il est donc nécessaire de convertir une valeur de type `Address` en une valeur d'un type entier reconnaissable directement par l'assembleur.

Le paquetage `System.Storage_Elements` contient justement une fonction `To_Integer()` effectuant cela, retournant une valeur de type `Integer_Address` (déclaré dans `System.Storage_Elements`). Comme pour le type standard `Integer`, la taille de ce type n'est pas réellement connue, car dépendante de la plate-forme, mais, au moins, nous savons qu'il s'agit d'un entier que nous pourrons transtyper vers, par exemple, `Integer_32`.

Le programme commence donc par définir une fonction `To_Int_32()` recevant une adresse mémoire et retournant un `Integer_32` :

```
with System.Machine_Code;
with Interfaces;
with System.Storage_Elements;
procedure Put_Line is
   use ASCII;
   use Interfaces;
   use System.Machine_Code;
   function To_Int_32(addr: in System.Address)
      return Unsigned_32 is
   begin
      return Unsigned_32(System.Storage_
Elements.To_Integer(addr));
   end To_Int_32;
```

Remarquez que contrairement à ce que nous faisons d'habitude, le paquetage `Ada.Text_IO` n'est pas référencé. Voici notre version de `Put_Line()` :

```
   procedure Put_Line(s: in String) is
      new_s: constant String := s & LF;
   begin
      Asm("movl  $1,%%ebx " & LF & HT &
          "movl  $4,%%eax " & LF & HT &
          "int   $0x80 ",
          Inputs => (Unsigned_32'Asm_Input("d",
new_s'Length),
                     Unsigned_32'Asm_Input("c",
To_Int_32(new_s(1)'Address))),
          Clobber => ("eax, ebx")
         );
   end Put_Line;
   pragma Inline(Put_Line);
```

Il n'y a en fait rien de bien particulier par rapport à ce que nous avons déjà vu. La procédure commence par créer une nouvelle chaîne à partir de la première, en lui ajoutant le caractère `LF` (pour le retour à la ligne). Les entrées du code assembleur consistent, dans l'ordre :

- à placer la longueur de la chaîne dans le registre `EDX` ;
- à placer l'adresse du premier caractère de la chaîne dans le registre `ECX`.

C'est là qu'intervient notre fonction `To_Int_32`. La seule petite subtilité consiste à prendre l'adresse du *premier caractère* de la chaîne (par `s(1)'Address`), plutôt que l'adresse de la chaîne elle-même (par `s'Address`). En effet, en Ada, une chaîne est un tableau de caractères, comme en C, mais contrairement au C, un tableau n'est pas qu'une adresse mémoire. Il est donc possible qu'en mémoire les données d'un tableau soient précédées d'informations « administratives », comme la taille du tableau, ses bornes, etc. Prendre directement l'adresse de la chaîne pourrait donc aboutir à l'affichage d'octets indésirables. On évite cela en prenant explicitement l'adresse du premier caractère. Naturellement, si la chaîne est vide, invoquer `s(1)` provoquera une exception : un code véritable devrait donc s'assurer qu'il y a au moins un caractère à adresser.

Enfin, on place dans `EBX` l'identifiant du descripteur de fichier voulu (ici 1, correspondant à la sortie standard `stdout`), puis dans `EAX` l'identifiant de la fonction système voulue (ici 4, la fonction `write()`). La dernière instruction déclenche l'interruption 0x80, utilisée sous Linux pour invoquer les fonctions système du noyau.

On termine le programme par le message consacré :

```
begin
   Put_Line("Bonjour, Monde !");
end Put_Line;
```

Résultat :

```
$ gnatmake put_line.adb && ./put_line
Bonjour, Monde !
$
```

Conclusion

Voilà pour cette petite présentation de l'utilisation de l'assembleur en Ada, du moins dans le cadre du compilateur GNAT utilisé sur une architecture 80x86. Ceci montre bien qu'Ada est un langage véritablement généraliste pouvant être utilisé dans toutes les situations, même les plus exigeantes.

Yves Bailly,

http://www.kafka-fr.net

RÉFÉRENCES

- **[1]** Toy Lovelace : **http://ipnnarval.in2p3.fr/~xavier/**
- **[2]** Codes sources de l'article : **http://www.kafka-fr.net/articles/presentation-ada05_17-sources.tar.bz2**