TD1 – **Programmes : du code source à l'exécution**

{nollinge,Emmanuel.Godard,esposito}@cmi.univ-mrs.fr

6 octobre 2004

☞ *L'objet de ce TD est de s'initier au cycle de vie des programmes, depuis leur compilation à partir du code source, jusqu'à leur environnement d'exécution et leurs interactions avec le système d'exploitation. Ce TD mêle des aspects d'architecture des ordinateurs à des aspects de compilation. Ce TD est aussi une initiation à notre architecture de référence et aux systèmes POSIX.*

Introduction

Le but d'un système d'exploitation est avant tout de fournir des services qui permettent de faire fonctionner des programmes en leur proposant une vue abstraite des ressources.

Dans ce TD nous allons suivre le cheminement depuis le programme source C `toto.c` ci-dessous jusqu'à l'exécution du programme `toto` qu'il engendre :

```
1   #include <stdio.h>
2
3   long int factorielle(int n)
4   {
5      if (n<2)
6        return 1;
7      else
8        return n*factorielle(n-1);
9   }
10
11  int main(int argc, char *argv[])
12  {
13    int i;
14
15    printf("Hello World!\n\n");
16    for(i=0;i<10;i++)
17      printf("%i! = %li\n",i,factorielle(i));
18    return 0;
19  }
```

La compilation d'un programme aussi simple s'effectue habituellement en une seule commande :

```
% cc -o toto toto.c
```

En réalité, cette opération se découpe en trois phase distinctes :

compilation. le code source C est transformé en code assembleur (production de `toto.s`)

```
% cc -c -S toto.c
```

assemblage. le code assembleur est transformé en un fichier objet (production de `toto.o`)

```
% as -o toto.o toto.s
```

édition de liens. le fichier objet est lié aux fonctions de bibliothèque et à l'environnement d'exécution C (production de l'exécutable `toto`)

```
% ld -o toto crt0.o toto.o libc.a
```

1 Le processeur Ant-32

☞ *Lire dans le document sur l'architecture de référence uniquement les sections 1, 2.1 et regarder rapidement les familles d'instructions 2.5.1 à 2.5.5.*

Un certain nombre d'instructions usuelles semblent faire défaut à ce processeur. Nous allons montrer comment les mettre en oeuvre. On notera dans la suite `ze` le registre `r0`.

Q1. Proposer un codage pour les instructions :

- `mov des, src1` : recopie du contenu de `r[src1]` dans `r[des]` ;
  ```
  add des,src1,ze
  ```
- `j const32` : saut à l'adresse `const32` ;
  ```
  lcl src1,const16low
  lch src1,const16high
  jez ze,ze,src1
  ```
- `j src1` : saut à l'adresse `r[src1]` ;
  ```
  jez ze,ze,src1
  ```
- `b const32` : branchement à l'adresse `const32` ;
  ```
  lcl src1,const16low
  lch src1,const16high
  bez ze,ze,src1
  ```
- `b src1` : branchement à l'adresse `r[src1]`.
  ```
  bez ze,ze,src1
  ```

Q2. Gestion de pile. En supposant que le registre `sp` soit affecté à la pile, proposer un codage pour les instructions d'empilement et de dépilement. Attention la pile est inversée. Plus sp est grand moins grande est la pile.

- `push src1` : empile `r[src1]` ;
  ```
  subi  sp,sp, 4   -- décrémente sp pour qu'il pointe à un mot de décalage
  st4  src1,sp,0  -- charge r[src1] dans sp
  ```
- `pop des` : dépile dans `r[des]`.
  ```
  ld4  des,sp,0 -- charge r[sp] dans r[des]
  addi  sp,sp, 4  -- incrémente sp de 4 octets
  ```

Q3. Gestion d'appel de fonction. En supposant que le registre `ra` soit affecté à l'adresse de retour dans un appel de fonction, le registre `fp` à l'adresse du bloc d'information, et le registre `g0` au stockage de la valeur de retour, proposer un codage pour les instructions :

- `call const32` : saute à l'adresse `const32` en plaçant l'adresse de retour dans le registre `ra` ;
  ```
  mov ra,pc  -- on met la valeur de l'adresse courante dans ra
  j const32  -- on saute
  ```
- `entry const32` : empile `fp` et `ra`, met la valeur de `sp` dans `fp`, alloue `const32` octets dans la pile pour les variables locales ;
  ```
  push fp
  push ra
  ```

```
mov fp,sp
lcl tmp,const32low
lch tmp,const32high
sub sp,sp,tmp
```

– `return src1` : met la valeur `r[src1]` dans `g0`, met la valeur de `fp` dans `sp`, dépile `ra` et incrémente-le de 4, dépile `fp`, saute en `ra` ;

```
mov g0, src1
mov sp, fp
pop ra
addi ra,ra,4
pop fp
j ra
```

– `return const32` : idem en utilisant `const32` comme valeur de retour au lieu de `r[src1]`.

```
lcl g0, const32low
lch g0, const32high
mov g0, src1
mov sp, fp
pop ra
addi ra,ra,4
pop fp
j ra
```

On appelle une fonction avec `call` après avoir empilé les arguments. La fonction débute par `entry` et termine avec `return`. Dessiner l'état typique de la pile juste après l'exécution de `entry`.

Adresse	Contenu	Commentaires
...	...	Valeurs précédemment sur la pile
fp+8+4xN ⋮ ⋮ fp+8	arg_N arg_0	paramètres de la fonction
fp+4	fp	valeur sauvegardée de fp
fp	ra	valeur sauvegardée de ra
fp-4 ⋮ fp-const32	...	variables locales

2 Compilation

La compilation consiste à transformer le source C en un source assembleur. On supposera que l'assembleur suit nos conventions d'utilisation des registres et connaît `ze`, `ra`, `sp`, `fp`, ainsi que les registres généraux non utilisés `g0` à `g55` qui sont disponibles pour le calcul et quelques registres spéciaux `u0` à `u3` qui sont réservés pour la mise en oeuvre des macro-instructions `mov`, `j`, `b`,`push`, `pop`, `entry`, `call` et `return`.

Un fichier source assembleur est constitué d'une suite d'instructions dans laquelle s'insèrent :

- des définitions de labels qui désignent des endroits spécifiques du fichier (ex. `factorielle:`) ;
- des identificateurs de sections (`.text` et `.data`) qui séparent les parties du programme qui sont du code et des données. Une déclaration de section est valable jusqu'à la prochaine déclaration ;
- des déclarations de données brutes (ex. `.ascii "titi\000"`, `.word 0x12345678`, `.byte 65`) ;
- des déclarations d'alignement (ex. `.align 4`) qui demande l'alignement en mémoire de ce qui suit par rapport à un certain nombre d'octets (ex. alignement sur une adresse multiple de 4) ;
- des déclarations d'exportation de symboles (ex. `.global main`).

Q1. Proposer un fragment de code assembleur correspondant au corps de la fonction `factorielle`.

Q2. Proposer un fragment de code assembleur correspondant à l'en-tête de la fonction `factorielle`.

```
1
2   .text
3   factorielle:
4           entry 0              # on n'alloue rien
5           ld4 g1, fp, 8        # on copie la valeur de n dans g1 ..
6                                # .. en reculant de deux niveaux depuis l'ancienne pile
7           gts g2,2,g1          # g2 vaut 1 si 2>g1 et 0 sinon
8           jnz r0,g2,$basefact  # si g2 vaut 1 alors on saute à basefact
9           push g1              # on garde la valeur de g1 sous la main
10          sub g1,g1,1          # on soustrait 1 à g1
11          push g1              # on empile g1 qui vaut n-1
12          call $factorielle    # on appelle factorielle
13          pop g1               # on dépile l'ancienne valeur de g1, c-à-d n
14          mul g1,g0,g1         # on multiplie g1 et g0 (qui contien le résultat de factorielle (
15          return g1
16  basefact:
17          return 1
18
```

Q3. Proposer un fragment de code assembleur correspondant au corps de la fonction `main`.

Q4. Proposer un fragment de code assembleur correspondant à l'en-tête de la fonction `main`.