Architecture des ordinateurs

TP5 : Compiler un programme C vers DigMips

C. Alias

Dans ce TP, on va aborder la traduction d'un programme C en langage machine DigMips.

Exercice 0. *Le compilateur* DigCC

Quand on veut écrire un logiciel de taille raisonnable, la programmation en langage machine est rapidement fastidieuse, sinon impossible. Heureusement, il existe un outil qui traduit automatiquement un programme C en code assembleur DigMips. Cet outil est appelé *compilateur*.

Questions.

- Ajoutez le répertoire `/home/tpetu/INF2003L/digcc/bin/` à votre variable `$PATH`.
- **Entrez le programme C** vu dans le TP précédent en remplaçant `printf("x")` par `print_char('x')` et en entrant d'abord la fonction g(), puis la fonction f() puis la fonction main().
- **Compilez votre programme** avec la commande `digcc monprogramme.c`.
- DigCC produit un fichier `monprogramme.asm`. **Assemblez le**, et **testez le** dans DigMips.

Le fichier .asm généré par DigCC comporte trois parties:

Code d'initialisation (lignes 1 – 47). Initialisation de la pile (ligne 1), initialisation du tas (appel à la fonction __init_heap() ligne 16), et enfin appel à la fonction main() (ligne 38).

Programme (lignes 48 – 251). On trouvera les fonctions g() (lignes 48 – 93), f() (lignes 94 – 161), et main() (lignes 162 – 251).

Bibliothèque d'exécution (lignes 242 – 534). Il s'agit d'une couche système rudimentaire:

- `void __init_heap()` (lignes 252 – 295). Initialise le tas. Toujours appelé dans le code d'initialisation.
- `void* malloc(int size)` (lignes 296 – 359). Alloue `size` octets dans le tas, et retourne l'adresse du premier élément (un pointeur, donc).
- `void free(void* data)` (lignes 360 – 402). Libére la zone du tas qui commence à l'adresse `data`.
- `void print_char(char c)` (lignes 403 – 425). Affiche le caractère `c` à l'écran.
- `void print_int(int n)` (lignes 426 – 456). Affiche l'entier `n` à l'écran. On suppose $0 \leq n \leq 19$.
- `void print_string(char* s)` (lignes 427 – 456). Affiche la chaine de caractère `s` (p. ex. "bonjour"). Utiliser avec modération.
- `void print_newline()` (lignes 457 – 509). Passe à la ligne.
- `char input_char()` (lignes 510 – 534). Lit un caractère au clavier.

Il ne s'agit bien sur que du minimum vital. On pourrait ajouter des fonctions arithmétiques (*, /, $\sqrt{}$, sin, cos, tan, etc), des fonctions de dessin, etc.

Exercice 1. *Fonctions, suite et fin*

On considère le programme C suivant:

```
int g(int n, int[2] tab)
{
  return tab[n];
}

int f(int n)
{
  int[2] t;
  int v;

  t[0] = 1;
  t[1] = 2;
  v = g(n,t); //v <-- t[n]

  return v; //t[n] (= t[1] = 2)
}

void main()
{
  int r;
  result = f(1);
  print_int(result);
}
```

Questions.

- Récupérez le fichier `/home/tpetu/INF2003L/diglog/tp/tp5.tgz` et décompactez. **Ouvrez** le fichier main.c. **Compilez le**, assemblez le, et testez le dans DigMips.
- Ouvrez le fichier main_commente.asm, allez à la fonction g. Il y a visiblement des choses supplémentaires sur la pile, et on y accède à l'aide du registre r7: [r7+4], [r7+5]. Le but de cet exercice est d'éclaircir ces mystères...

La dernière fois, nous avons vu comment utiliser une pile pour sauvegarder l'adresse de retour d'une fonction. En réalité, la pile peut faire bien plus... Nos fonctions possèdent des arguments et des variables locales. Même question que précédement: où les stocker? Ici, on pourrait réserver 5 emplacements dans une zone mémoire dédiée pour les arguments n et tab de g, et pour n, t et v de f. Malheureusement, çà ne fonctionne plus si f est récursive, puisqu'il faudrait stocker l'argument n et les variables locales t et v pour chaque exécution (*activation*) de f. Le plus simple est de regroupement tout çà (arguments, variables locales) avec l'adresse de retour sur la pile. Chaque activation de fonction possède ainsi un enregistrement sur la pile, appelé *enregistrement d'activation* qui regroupe les valeurs des arguments, des variables locales, l'adresse de retour et d'autres informations.

Le tableau donné en annexe montre l'état de la pile au cours de l'exécution du programme.

Appel de fonction. Comme on l'a vu, il se fait en 2 temps.

- *Dans la fonction appelante*, on empile les arguments, l'adresse de retour, et on saute vers la fonction.
- *Dans la fonction appelée*, on complète l'enregistrement d'activation, on construisant les variables locales, notamment. Cette partie est réalisée par un morceau de code placé en début de fonction et appelé **prélude**.

Retour de fonction. Il se réalise à nouveau en 2 temps, de façon complétement symétrique.

- *Dans la fonction appelée*, on désalloue les variables locales, on récupère l'adresse de retour, et on désalloue le reste de l'enregistrement d'activation (en dépilant). On saute à la fonction appelante. Cette partie est réalisée par un morceau de code placé en fin de fonction et appelé **postlude**. Enfin, le résultat de la fonction (return) est passé par le registre r2.
- *Dans la fonction appelante*, on récupére la valeur de retour dans r2, et on enchaine sur le reste du calcul.

Questions.

- Dans la fonction g(), répérez le prélude et le postlude.
- Le prélude fait pointer le registre r7 sur le premier octet de la première variable locale. On peut ainsi accéder aux variables locales avec [r7] pour la première, [r7-1] pour la deuxième, etc. Dans f(), **a quoi correspond** [r7]? [r7-1]? [r7+4]?
- Le r7 de la fonction appelante doit être sauvegardé par le prélude, et restauré par le postlude. Sur le tableau, **réperez l'endroit** où r7 est sauvegardé dans la pile.

Exercice 2. *Traduction du contrôle*

On spécifie comment traduire une construction syntaxique (for, while, etc) en assembleur à l'aide d'un *schéma de traduction*. Voilà par exemple le schéma de traduction du if:

```
TRADUIRE(if C BLOC1 else BLOC2, adresses) =
    temp_bool = TRADUIRE(C, adresses) //convention: true = 1, false = 0
    ldi temp_0,0
    ble temp_bool,temp_0,else
    TRADUIRE(BLOC1, adresses)
    j end_if
    else:
    TRADUIRE(BLOC2, adresses)
    endif:
```

Le texte en `verbatim` correspond à du code DIGMIPS généré.
Le tableau `adresses` contient les emplacements mémoire (en pile) des arguments et des variables locales de la fonction courante. Il est nécessaire pour lire et écrire les variables. TRADUIRE(C,adresse) produit du code DIGMIPS qui évaluer la condition C, et place 0 dans un registre si elle est fausse, et 1 si elle est vraie. Elle retourne le registre résultat, qui est placé, ici, dans temp_bool.

Questions.

- En vous inspirant de cet exemple, donnez le schéma de traduction du while.
- Donnez le schéma de traduction du for

Voilà. On pourrait aussi donner un schéma de traduction pour chaque catégorie syntaxique du programme (expression, condition, controle, fonction, etc). Il reste encore plusieurs étapes délicates avant la production de code DIGMIPS, mais c'est une autre histoire qui sera abordée plus tard.

Contexte	Pile main()	f()	g()	Action

main()
{ int r;
  f(1)
  { int[2] t; int v;
  t[0]=1; t[1]=2
  g(n,t)
  {
  return tab[n];
  }
	r=0
	r=0
	r=0
	r=0
		n=1 | lo(rf) | hi(rf) | r7=255 | t=&t[0] | v=? | t[0]=? | t[1]=?
		n=1 | lo(rf) | hi(rf) | r7=255 | t=&t[0] | v=? | **t[0]=1** | **t[1]=2**
		n=1 | lo(rf) | hi(rf) | r7=255 | t=&t[0] | v=? | t[0]=1 | t[1]=2
			n=1 | tab=&t[0] | lo(rg) | hi(rg) | r7=250	prélude(main)
				prélude(f)
				prélude(g)
				r2 = 2
				postlude(g)

rg: val = ... return val; }	r=0	n=1 | lo(rf) | hi(rf) | r7=255 | t=&t[0] | v=? | t[0]=1 | t[1]=2		val = r2 (=2) r2 = 2 postlude(f)
rf: r = ... }	**r=2**			postlude(main)