Projet de compilation

November 17, 2016

1 Description de la machine PICO

La machine PICO exécute des programmes en assembleur PICODE. PICO a un espace mémoire sur 15 bits (32 ko). Elle possède 16 registres R_i ($i \in [0:15]$) de 32 bits. La mémoire est gérée en petit indien (octet de poids faible dans les adresses faibles). Les entiers sont codés en complément à deux.

Une instruction est composée d'une suite d'octets dont seul le premier est obligatoire.

OPSZ Sur un octet.

OP Le code opération sur 6 bits (bits 7 à 2).

SZ La taille sur la quelle porte l'opération sur 2 bits (bits 1 à 0) avec 00:non utilisée, 01:1 octet, 10:2 octets, 11:4 octets.

N L'adresse de l'instruction suivante sur 2 octets.

D L'opérande destination sur 1 ou 5 octets (voir le paragraphe Effective Adresse).

S_0 La première opérande source sur 1 ou 5 octets (voir le paragraphe Effective Adresse).

S_i La $i^{ième}$ opérande source sur 1 ou 5 octets (voir le paragraphe Effective Adresse).

bits du 1^{er} octet					mode	information
7	6	5	4	3-0		
0	0	0	d	0	immédiat	la donnée est sur 1, 2 ou 4 octets suivant le paramètre SZ de l'instruction.
0	0	1	d	0	direct mémoire	Les 2 octets suivants donnent l'adresse.
0	1	0	d	0	indirect mémoire	Les 2 octets suivants donnent l'adresse.
1	0	0	d	#	direct registre	# est le numéro du registre, pas d'octet supplémentaire.
1	0	1	d	#	indirect registre	# est le numéro du registre, pas d'octet supplémentaire.
1	1	0	d	#	indirect registre post-incrémenté	# est le numéro du registre, pas d'octet supplémentaire.
1	1	1	d	#	indirect registre pré-incrémenté	# est le numéro du registre, pas d'octet supplémentaire.

d=0: ce n'est pas la dernière "effective addresse" de l'instruction.
d=1: c'est la dernière "effective addresse" de l'instruction.

Table 1: Types et codages des "Effective Adresses" de l'assembleur PICODE.

fonction	OP	SZ	N	D	S_0	S_1	S_i	S_{N-1}	N2
$D = \sum_0^{N-1} S_i$	1	M	M	M	M	M	*	*	
$D = S_0 - \sum_1^{N-1} S_i$	2	M	M	M	M	M	*	*	
$D = \prod_0^{N-1} S_i$	3	M	M	M	M	M	*	*	
$D = S_0$	4	M	M	M	M				
branchement	5	0	M						
branch. à N2 si $D = S_0$	6	M	M	M	M				M
branch. à N2 si $D < S_0$	7	M	M	M	M				M
branch. à N2 si $D \leq S_0$	8	M	M	M	M				M
branch. à N2 & empile N	9	0	M						M
dépile x & branch. à x	10	0							
input into D	11	M	M	M					
output from D	12	M	M	M					
stop	13	0							

M: champ obligatoire; *: champ facultatif; ni M ni *: champ interdit

Table 2: Instructions et codages des instruction de l'assembleur PICODE.

S_{N-1} La dernière opérande source sur 1 ou 5 octets (voir le paragraphe Effective Adresse).

N2 Une deuxième adresse de branchement sur 2 octets.

Le format et les instructions supportées sont donnés sur la table 2.

Une **Effective Adresse** commence par un octet. Suivant la valeur de cet octet, il y a 0, 1, 2 ou 4 octets supplémentaires. La table 1 spécifie leurs différents types ainsi que leurs codages.

2 Description de la VM PICO

L'exécutable vmpico est un simulateur de la machine pico. Il a un argument optionnel (défaut picode) qui est un fichier binaire d'au maximum 2^{15} octets de PICODE.
Il lit ce fichier et lance l'exécution du PICODE en commençant à l'adresse 0. Si le PICODE fait des lectures, elles sont faites sur le le flux standard d'entrée, les écritures sont faites sur le le flux standard de sortie. D'autres options sont disponibles et peuvent être affichées avec l'option -h.

3 Sujet

Réalisez l'assembleur aspico de PICODE pour la machine PICO. aspico lit un fichier assembleur passé en argument et génére un fichier PICODE exécutable par la vmpico. Ses options sont:

-t affiche sur le flux standard de sortie la table des symboles puis se termine.

-d affiche sur le flux standard de sortie un dump humainement lisible du PICODE généré.

-o file le fichier de PICODE généré. Si cette option est absente le fichier est par défaut picode.

Ses caractéristiques générales sont: 1) il doit tourner sur toute machine Unix après régénération; 2) le programme vérifie que les instructions sont valides; 3) le programme vérifie qu'il n'y a pas de conflits.

4 Syntaxe de l'assembleur

La syntaxe de l'assembleur est composée de constantes (nommée <cst> par la suite), de définitions de labels, de primitives et d'instructions.

4.1 Commentaire

Le caractère '#' marque le début d'un commentaire, il va jusqu'au bout de la ligne.

4.2 Constante

Constantes numériques terminales Ce sont des caractères, des nombres décimals et hexadécimals et elles sont définies comme en C. Voici des exemples: 7, 0xF4, 'A', -12, +122, -'c', +0xa7.

Constantes label Il sont définis comme un identifiant suivi d'un ':' (sans espace). L'identifiant (sans le ':') devient une constante. Voici des exemples de définitions
de labels correctes: gnu:, bee:, g1n1n1:
de labels incorrectes: gnu :, 3bee:, g$:

Constantes (<cst> dans la suite) Ce sont des expressions arithmétiques éventuellement parenthésées formées avec les opérateurs +, -, *, << et des constantes numériques et/ou des labels.
Voici des exemples de constantes: bee+12, (2*(gnu+-1)*3), a-+40, a - +40

4.3 Primitive

Les primitives sont des mots clé commençant par le caractère '.'.

.org <cst> Indique que l'instruction ou la primitive suivante se trouvent à l'adresse <cst>.

.long <cst> Réserve 4 octets et y met la valeur <cst>.

.word <cst> Réserve 2 octets et y met la valeur <cst>.

.byte <cst> Réserve 1 octets et y met la valeur <cst>.

.string "str" Réserve autant d'octets plus 1 que la chaîne de caractères et y met la chaîne suivie d'un $\emptyset$.

Il y a des restrictions pour l'utilisation de la primitive ".org <cst>"

- Son <cst> ne doit pas contenir de label.
- Si dans un fichier, on a une primitive ".org <cst1>" et un peu plus loin une primitive ".org <cst2>" alors <cst2> doit être plus grand que <cst1>.
- Le <cst> d'une primitive ".org" ne doit pas positionner sur une partie de la mêmoire déjà initialisée.

4.4 Instruction avec opérandes

Le format général d'une instruction avec opérandes est le suivant.

OPSZ une instruction commence par un mnémotechnique.

DEST le mnémotechnique est suivi de l'opérande destination,

, $\mathbf{SRC}_i$ puis des opérandes sources (chaque opérande source est précédée d'une virgule),

AIS puis de l'adresse de l'instruction suivante. Celle-ci est facultative. Son absence indique que l'exécution continue en séquence.

Les mnémotechnique des instructions (OPSZ) sont avec sz indiquant la taille (b pour 1, w pour 2, ou l pour 4): add_{sz}, sub_{sz}, mul_{sz}, mov_{sz}, if_{sz}, in_{sz}, out_{sz}.

Le format des opérandes sources et destinations (DEST, SRC_i) est: $<cst> pour de l'immédiat, <cst> pour du direct memoire, [<cst>] pour de l'indirect mémoire, %i pour du direct registre sur R_i, [%i] pour de l'indirect registre sur R_i, ++[%i] pour de l'indirect registre pré-incrémenté sur R_i, [%i]++ pour de l'indirect registre post-incrémenté sur R_i.

Le format de l'adresse de l'instruction suivante (AIS) est: @<cst>.

Les formats de l'instruction "if_{sz}" sont
  if_{sz} OP1 <oc> OP2 <cst-true>
ou
  if_{sz} OP1 <oc> OP2 <cst-true> else <cst-false>
OP1 et OP2 suivent le format standard des opérandes, <oc> est un des opérateurs de comparaison suivants: '==', '!=', '<', '>', '<=', '>=', '=<', '=>'. <cst-true> est l'adresse de branchement si la condition est vrai. <cst-false> est l'adresse de branchement si la condition est fausse. L'instruction "if_{sz}" ne peut pas avoir d'AIS.

4.5 Instruction sans opérandes

jmp <cst-bra> Branche à l'adresse <cst-bra>.

call <cst-bra> Branche à l'adresse <cst-bra> et empile l'adresse de l'instruction suivante.

call <cst-bra> AIS Branche à l'adresse <cst-bra> et empile l'adresse AIS.

ret Dépile une adresse et s'y branche.

stop Arrête l'exécution.

4.6 Exemples

Voici 2 exemples de PGCD:

```
1  ### PGCD 1
2  .org 0
3  jmp main
4  a: .long 0
5  b: .long 0
6  main:
7    inl a
8    inl a+4
9  loop:
10   ifl a==b then end
11   ifl a<b  then AinfB
12            else BinfA
13 BinfA:
14   subl a,a,b @loop
15 AinfB:
16   subl b,b,a @loop
17 end:
18   outl a
19   stop
```

```
1  ### PGCD 2
2  .org 0
3  jmp 1024
4  pgcd: # a/b/ret=%1/[%2]/%0
5    ifl %1==[%2] then pgcd_end
6    ifl %1>[%2] then AsupB
7                else BsupA
8  BsupA:
9    subl [%2],[%2],%1 @pgcd
10 AsupB:
11   subl %1,%1,[%2] @pgcd
12 pgcd_end:
13   movl %0,%1
14   ret
15
16 .org 1024
17   inl %1 # a
18   inl 1024-4 # b
19   movw %2,$1024-4
20   call pgcd
21   outl %0
22   stop
```

5 Informations pratiques

Ce projet est à effectuer en binômes. Dès qu'un binôme se sera constitué, il enverra un mail à guillaume.burel@ensiie.fr pour vérification.

Le code rendu comportera un Makefile, et devra pouvoir être compilé avec la commande `make`. **Tout projet ne compilant pas se verra attribuer un 0** : mieux vaut rendre un code incomplet mais qui compile, qu'un code ne compilant pas. Votre code devra être **abondamment commenté et documenté**. Vous prendrez soin d'organiser votre code de façon modulaire.

Une partie de la note sera obtenue en testant votre programme de façon automatique. **Vous prendrez soin de bien respecter les con-**

signes, notamment en ce qui concerne les entrées/sorties et les paramètres de l'exécutable final.

L'analyseur syntaxique demandé à la question 3 sera impérativement obtenu avec les outils lex/yacc[1] ou ocamllex/ocamlyacc[2]. Ceci implique donc que votre projet sera écrit au choix en C ou en OCaml.

Des fichiers d'entrée pour tester votre code seront disponibles depuis l'adresse : http://www.ensiie.fr/~guillaume.burel/compilation/ . Vous attacherez un soin particulier à ce que ces exemples fonctionnent.

Il vous est demandé de ne pas utiliser GitHub (en tout cas pas la partie publique) pour héberger votre code. **Si plusieurs binômes ont des projets dont les sources sont trop similaires, *tous* se verront attribuer la note 0.**

Votre projet est à envoyer sur forme d'une archive `tar.gz` **avant le 2 janvier 2017 à 18h** sur le serveur de projet `exam.ensiie.fr` dans le dépôt `ICO_projet_2016`. **Tout projet rendu en retard se verra attribuer la note 0.** Vous n'oublierez pas d'inclure dans votre dépôt un rapport (PDF) précisant vos choix, les problèmes techniques qui se posent et les solutions trouvées.

6 Questions

1. Proposer une grammaire non ambiguë pour l'assembleur. Vous détaillerez cette grammaire dans votre rapport, et elle devra correspondre à celle utilisée en yacc ou ocamlyacc question 3.

2. Définir des types de données correspondant à la syntaxe abstraite de l'assembleur. En particulier on définira entre autres un type `ass_prog`.

3. À l'aide de lex/yacc, ou ocamllex/ocamlyacc, écrire un analyseur lexical et syntaxique qui lit et qui retourne l'arbre de syntaxe abstraite associé (qui retourne donc une valeur de type `ass_prog`).

   Yacc ou ocamlyacc ne doit pas indiquer de conflit. Si c'est le cas, il vous faut modifier la grammaire que vous avez définie question 1.

4. Écrire une fonction d'analyse sémantique qui vérifie la sémantique du programme assembleur (déclaration des constantes labels utilisées, bonne formation des .org, etc.).

5. Écrire le back-end qui produit le fichier PICODE.

6. Lier le tout pour produire l'exécutable attendu.

7. Si le temps le permet, ajouter, par ordre de priorité, les fonctionnalités de macro-assembleur:

   (a) une macro pour faire des alternatives (si COND alors BLOC [sinon BLOC] fsi) de façon légère comme dans un langage de haut niveau.

   (b) une macro pour faire des boucles "tant que" (tq COND faire BLOC fait) de façon légère comme dans un langage de haut niveau.

   (c) dans les macros précédentes, les conditions peuvent être des expressions booléennes "(COND1 ou non(COND2 et COND3)) et COND4" avec le ET prioritaire sur le OU.

   (d) une macro pour faire des boucles "pour" (pour i de d à f pas de p faire BLOC fait) de façon légère comme dans un langage de haut niveau.

[1] Vous pouvez utiliser les alternatives libres flex/bison. Documentation disponible à la page http://dinosaur.compilertools.net/

[2] Documentation disponible à la page http://caml.inria.fr/pub/docs/manual-ocaml-4.00/manual026.html