Architecture, Système, Réseau 2017-2018
{alice.pellet___mary,fabrice.mouhartem}@ens-lyon.fr, florent.de-dinechin@insa-lyon.fr

ASR1, TD4 : un peu d'assembleur dans notre ISA à nous

1 Décision sur l'encodage des constantes

Voici quelques possibilités :

- option 1 : une valeur k sur 3 bits qui code une taille 2^k : tailles 1, 2, 4, 8, 16, 32, 64, 128
- option 2 : une valeur k sur 2 bits qui code 2^{k+2} bits : tailles 4, 8, 16, 32.
- option 3 : une valeur k sur 3 bits qui code $4(k+1)$ bits : tailles 4, 8, 12, 16, 20, 24, 28, 32
- option 4 : un encodage prefix-free, par exemple
  - 0 : 1 bit (total 2 bits)
  - 10 : 8 bits (total 10 bits)
  - 110 : 16 bits (total 19)
  - 1110 : 32 bits (total 36)
  - 1111 : 64 bits (total 68)
- Une proposition intéréssante : on reçoit 4 bits, puis un bit qui dit si on doit recevoir encore des bits. Le paquet suivant peut faire 8 bits, le suivant 24, etc. J'ai l'impression que cette proposition coûte autant qu'un encodage prefix-free de la taille de la constante, et parfois un bit de plus. Elle sera aussi légèrement plus compliquée à implémenter.
- Une solution qui recule pour mieux sauter : on s'assied complètement sur l'orthogonalité du jeu d'instruction. On propose plusieurs versions de chaque instruction impliquant une constante, par exemple 3 instructions de sauts `j8` `j16` `j32` qui font `PC+=d` où `d` est une distance relative encodée sur respectivement 8, 16 et 32 bits. On encodera `j8` sur moins de bits que `j16` et `j32`... plus tard.
  Cette solution est écartée pour des raisons du temps qui serait nécessaire à l'implémenter. Du reste, plus d'opcodes c'est aussi plus de bits...

Exemples de constantes utiles :

valeur	option 1	option 2	option 3	option 4
0,1	000 x	00 000x	000 000x	0 x
3	001 11	00 0011	000 0011	01 0000 0011
65-90 ('A'-'Z' ascii)	011 000xxxxx	01 000xxxxx	010 000xxxxx	10 000xxxxx
Adresses plus petites que 4095	3+16 bits	2+16 bits	3+12 bits	3+16 bits
Valeur max	2^{128} sur 131 bits	2^{32} sur 34 bits	2^{32} sur 35 bits	2^{64} sur 68 bits

J'ai l'impression que c'est l'option 4 qui gagne, surtout si on veut que l'ISA fonctionne pour des ALU de 16 à 64 bits...

En principe il faudrait utiliser des encodages différents selon que la constante est un shift, une adresse, un déplacement dans un saut relatif, un incrément d'addition, un masque...

Et donc la solution adoptée dans la suite est la suivante :

- Encodage *prefix-free* des adresses (`jump`, `call`, `load` et `store`) :

0 + 8 bits	adresse ou déplacement sur 8 bits
10 + 16 bits	
110 + 32 bits	
111 + 64 bits	

- Encodage *prefix-free* des constantes de shift :

0 + 6 bits	constante entre 0 et 63
1	constante 1

  Remarque : la direction du shift peut être encodée dans un bit après l'instruction pour économiser un opcode.
- Encodage *prefix-free* des constantes pour les opérations ALU :

0 + 1 bit	constante 0 ou 1
10 + 8 bits	octet
110 + 32 bits	
111 + 64 bits	

2 Décision sur le nombre d'opérandes

Avant de reprendre la discussions sur les fenêtres de registres, les piles, etc, définissons proprement une référence :

— Toutes les opérations unaires (`not` `neg` `lsl` `lsr` `asr`) existent en version 1 et 2 opérandes. On note par exemple `not1` et `not2`.
— Toutes les opérations binaires (`add` `sub` `and` `or` `xor`) existent en version 2 et 3 opérandes. On note par exemple `add2` et `add3`.
— Le registre destination est en premier.
— On va essayer d'encoder les versions courtes avec des opcodes plus courts dans la suite... En attendant, disons que chaque code opération est codé sur c bits, plus (pour les instructions dont un des opérande peut être une constante ou un registre) un bit qui distingue les deux.
— Constantes encodées comme ci-dessus.
— 2^k registres.

On laisse c et k comme paramètres pour le moment.

Exercice 1: Combien de bits pour les instructions suivantes ? : (pour vérifier que tout est bien défini)

— add3 r3 r2 r1
— add3 r3 r2 1
— add3 r3 r2 65
— add2 r1 1
— let r3 0
— let r3 r1
— let r3 -1
— jump -81

Et pour compter proprement les bits de notre multiplication il faut encore des instructions de saut conditionnel.

3 Instructions de contrôle de flot

Le but est toujours d'être capable d'écrire la multiplication itérative du TD3.

3.1 Sauts relatifs conditionnels

Remarque : vos camarades de l'an dernier se sont assis sur ce paragraphe.

Traditionnellement, on contrôle l'exécution par des *drapeaux*, produits par l'UAL et testables par les instructions de saut relatif.

Les drapeaux de base de toute UAL sont :

— Z, qui vaut 1 si le résultat d'une opération est nul, et 0 sinon,
— N, qui vaut 1 si le résultat (interprété en complément à 2) est strictement négatif, et 0 sinon,
— C (carry) qui attrape le bit qui sort lors des instructions d'addition, de soustraction, et possiblement de décalage.
— V (oVerflow) qui vaut 1 si une opération arithmétique a provoqué un dépassement de capacité en complément à 2.

On a en général une instruction `cmp op1 op2` pour *compare* qui fait la soustraction op1-op2 et ne conserve pas le résultat. Par contre elle met à jour les drapeaux. Le but est de pouvoir écrire par exemple :

```
cmp r1 1
jump ifgt +12
```

ici `gt` signifie *greater than* (strictement) et donc on va exécuter le saut si r1>1.

Exercice 2: Pourquoi n'a-t-on pas l'équivalent de V pour le binaire non signé ? :

Exprimez les conditions suivantes en fonction des drapeaux : résultat positif ou nul, négatif ou nul, strictement positif, strictement négatif, dépassement de capacité, pas de dépassement de capacité, etc.

Combien de bits faut-il pour coder un ensemble minimal de conditions (ne pas oublier le "sans condition") ? Définissez précisément le champ condition, et les mnémoniques `ifxx` correspondant.

3.2 Sauts absolus, call/ret,...

Cette question peut être gardée pour la fin...

On va implémenter le call plutôt par un mécanisme de *branch and link*. Discutez comment il se joue sur notre interface sérielle.

4 Retour à la multiplication itérative

Codez en assembleur le programme suivant, et comptez ses bits.

```
; entrée: deux entiers A et B
C=0
while A!=0
  if (A&1 == 1) then C=C+B endif
  B = B << 1
  A = A>>1
return C
```

Dans le processeur 16 bits de l'an dernier, vos camarades l'ont implémenté en 8 instructions, donc 128 bits. Java (machine 8 bits à pile) l'implémente en une trentaine d'octets si on compile par `javac` puis désassemble par `javap -p`, donc deux fois plus. En écrivant le bytecode à la main on arriverait peut-être aussi à 128 bits. En tout cas il faut faire mieux.

Défense de revenir à des fenêtres de registres avant d'avoir torché la section suivante.

5 Instructions d'accès mémoire

Ici il faut d'abord raffiner un peu notre mémoire sérielle.

Elle doit au moins avoir deux autres compteurs : un pointeur de pile (on verra plus tard pourquoi) et un compteur d'adresse générique.

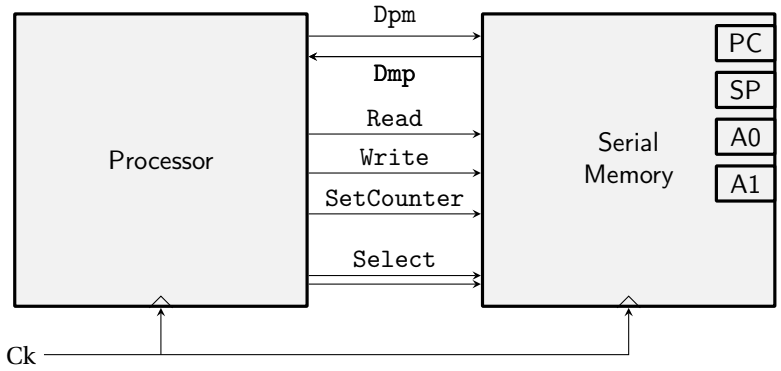
On peut arrondir à 4,et avoir deux compteurs d'adresse génériques.

Une question qui se pose est la suivante : est-ce que le processeur doit également conserver une copie de ces pointeurs? Je pense que oui mais je ne suis pas sûr.

Les signaux de contrôle pourraient par exemple consister en

— un signal Read
— un signal Write
— un signal Select sur deux bits qui sélectionne quel compteur est utilisé
— un signal SetCounter qui permet d'initialiser un des 4 compteurs

Tous ces signaux auront peu d'activité. On arrive au dessin suivant :



Proposez des instruction de lecture et écriture mémoire. On aura des instructions qui initialisent les compteurs. Toutefois les lectures et écritures de données consécutives n'auront pas besoin de retransmettre l'adresse. Par contre il faut spécifier dans l'instruction combien de bits on lit/écrit.

Écrivez une boucle qui fait une copie de R0 octets de l'adresse R1 vers l'adresse R2.

5.1 Encodage des opcodes

Faites un tableau complet des instructions, en précisant :

— si l'instruction nécessite un opcode ou deux (avec ou sans registre de destination explicite)
— si l'instruction a besoin d'un bit I "le dernier opérande est une constante immédiate" (immédiate veut ici dire : encodée dans le programme lui-même, pas à côté)
— si la constante est étendue par des zéros (z) ou son bit de signe (s)
— quel est l'encodage de la constante

Finalement, comptez les instructions et proposez-en un encodage huffmanien en vous inspirant de l'annexe. Combien de bits gagnez-vous sur la multiplication et sur le memcopy par rapport à un encodage naif (sur 5 bits) de chaque opcode ?

6 Si on est arrivés là

... on peut reparler de fenêtres de registres.

7 Annexe : fréquence des instructions dans le rendu de DM 2016/2017

J'ai instrumenté le simulateur de Leia, le lauréat de l'an dernier. Voici les statistiques d'une part sur le code source, d'autre part sur toute l'exécution de la démo.

Instr	dans le code		dans la trace de la démo	(en %)
wmem	34		113 466 979	2.45
add	243		623 444 712	13.47
sub	139		280 577 828	6.06
snif	151		903 264 021	19.52
and	46		270 142 463	5.84
xor	50	dont 20 pour faire ri=0	43 355 541	0.94
or	9		44 179 965	0.95
lsl	41		320 961 115	6.96
lsr	29		294 705 580	6.37
asr	4		97 844	0.002
call	136		103 238 243	2.23
jump + return	150 + 46		594 557 581	12.84
letl	50		342 583 905	7.40
leth	18		316 220 522	6.83
???	56		3 575	0.0007
rmem + copy	174		376 428 822	8.13