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

ASR1, TD3 : Notre ISA à nous

Présentation générale

Ce projet va se dérouler sur plusieurs séances de TD/TP et deux devoirs à la maison :
— Une séance de TP consacrée à la conception d'un jeu d'instruction ;
— Un travail de synthèse des TDMen, pour vous imposer à tous le même jeu d'instruction.
— Une seconde séance de TP consacrée à écrire en assembleur quelques programmes simples.
— Un premier rendu de DM qui pour implémenter un assembleur et un simulateur d'un processeur implémentant ce jeu d'instruction [1] ;
— Un second rendu de DM consistant à écrire du logiciel de base pour ce processeur [2] ;
— Une séance de TD pour construire le plan de masse de ce processeur ;
— Une séance de TD pour définir l'automate de commande ;
— Au moins deux séances de TP pour décrire le processeur lui-même (le hardware) en VHDL.
Il y aura au milieu de tout cela des TP normaux (de mise en application du cours).

Pour les DM comme pour les TP finaux, vous ne partirez pas de rien : on vous donnera des squelettes de code à compléter.

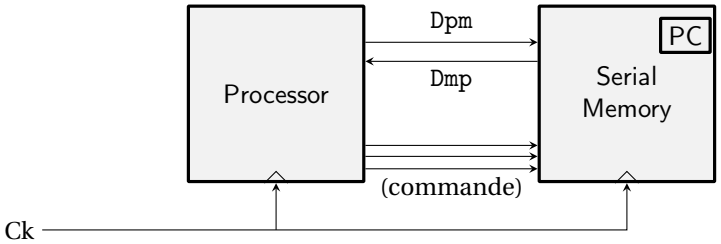
Dans ce premier TD, il s'agit de définir un jeu d'instruction par ses mnémoniques et son encodage.

Contexte et prétexte

Bill Dally, architecte en chef chez nVidia (donc quelqu'un que vous respectez) montrait en 2010 ces quelques statistiques sous le titre *Fetching operands costs more than computing on them* :
— Calculer une multiplication-addition fusionnée en virgule flottante 64 bits coûte 20 pJ.
— Déplacer les données correspondantes 1mm sur la puce coûte 26 pJ.
— Les déplacer à l'autre bout de la puce coûte 1 nJ (1 nJ = 1000 pJ).
— Les lire ou les écrire en DRAM coûte 16 nJ.
Depuis 2010 cet écart entre *coût du calcul* et *coût des communications* a plutôt empiré.

L'objet de ce TD est donc de concevoir un jeu d'instruction **minimisant le nombre de bits échangés entre processeur et mémoire** pour exécuter un programme qui résout un problème donné.

Dans ce but, l'interface entre le processeur et la mémoire va consister d'un seul signal [3] dans chaque sens, que nous appellerons `Dpm` et `Dmp` (Donnée du Processeur à la Mémoire, et vice versa). Plus des signaux de commande/contrôle à déterminer.



Naturellement, les instructions et données devront transiter **en série** sur ces deux fils. Ainsi, on espère pouvoir n'envoyer que le minimum d'information nécessaire à l'exécution de chaque instruction (et non systématiquement des paquets de 32 ou 64 bits comme dans une machine de von Neumann classique).

Une conséquence est que les instructions peuvent être de taille arbitraire (au bit près). Leur encodage devra être le plus compact possible pour les instructions les plus courantes, et pourra être moins compact pour les instructions plus rares.

Par rapport à la problématique de la construction d'un jeu d'instructions RISC classique vu en cours (comment encoder les instructions les plus puissantes possibles dans un mot de 32 bits), on a donc remplacé la contrainte des 32 bits par un degré de liberté. Car nous aimons la liberté.

1. Ce qui devrait faire ressortir quelques bugs de conception
2. Voir note précédente
3. Rappel : "signal", cela veut dire "fil".

Fonctionnement d'une mémoire sérielle

Le bloc Serial Memory est construit autour d'une mémoire adressable (de 1 bit de large). Ce bloc contient aussi un ou plusieurs *compteurs d'adresse* qui lui permettent de sortir des bits consécutifs de la mémoire à chaque cycle d'horloge. La construction d'un tel compteur sera décrite dans la suite du cours. Le PC en est un (figure).

Ces compteurs internes peuvent être initialisés par le processeur, en passant par `Dpm`.

Par exemple, voici un déroulement du cycle de von Neumann sur ce processeur :

1. Le processeur lève un des signaux de commande, appelons-le (temporairement) `Read`.
2. La mémoire (qui a une copie interne du PC) commence à envoyer des bits d'instruction sur Dmp.
3. Au bout d'un certain temps, le processeur sait de quelle instruction il s'agit, et donc combien de bits encodant les opérandes de l'instruction il faut encore recevoir.
4. Lorsque toute l'instruction a été transmise, opérandes compris, le processeur baisse `Read`, puis exécute l'instruction (ce qui peut faire passer plein de choses sur `Dpm` et `Dmp`.
5. Et on recommence à l'étape 1. Remarque : si l'exécution de l'instruction courante n'a pas changé le PC, il pointe déjà vers l'instruction suivante.

Pour le reste, on va organiser le processeur autour d'une ALU classique (16, 32 ou 64 bits) et d'une boîte à registres classique (choix éminemment discutable mais imposé).

1 Reality shouldn't constrain our formalisms

On veut que notre interface série marche lorsque le processeur est à 1cm de sa mémoire. Quelle est la fréquence maximale de l'horloge ?

Et lorsqu'il est à 10cm de la mémoire ?

2 Spécification des opérandes dans le jeu d'instruction

2.1 Constantes

On veut pouvoir spécifier une petite constante sur peu de bits, et une grosse sur plus de bits. Proposez différents encodages des constantes incluant leur taille. Pour chaque instruction, on décidera dans l'ISA si la constante est étendue à gauche par des 0 ou par son bit de signe.

Remarque : on utilisera de telles constantes pour spécifier de valeurs constantes, mais aussi des adresses, des déplacements (saut relatif), ...

On ne doit pas décider à ce stade : un peu plus tard nous écrirons un petit programme et compterons ses bits selon les différents encodages.

2.2 Registres

Discuter du nombre de bits pour encoder chaque registre.

Reprenez la discussion vue en cours entre machine à zéro, un, deux ou trois opérandes.

3 Instructions arithmétiques et logiques

Le but de cette section et de la suivante est d'être capable d'écrire la multiplication itérative suivante :

```
; 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.

3.1 Liste des instructions arithmétiques et logiques

Faites une liste des instructions nécessaires à l'écriture du coeur de boucle du programme précédent.
Complétez et généralisez cette liste.

3.2 Une, deux ou trois opérandes ?

Pour chaque instruction, discutez si elle doit avoir un, deux ou trois opérandes, et si l'un des opérandes peut être une constante, et si cette constante doit être en complément à 2 ou en binaire non signé.

3.3 Encodage des instructions

A combien d'instructions arrivons-nous ?
Savez-vous identifier les instructions les plus courantes ? Sans regarder l'annexe 7 d'abord...

4 Instructions de contrôle de flot

4.1 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 l'instruction 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 et de décalage.
— V (oVerflow) qui vaut 1 si une opération arithmétique a provoqué un dépassement de capacité en complément à 2.

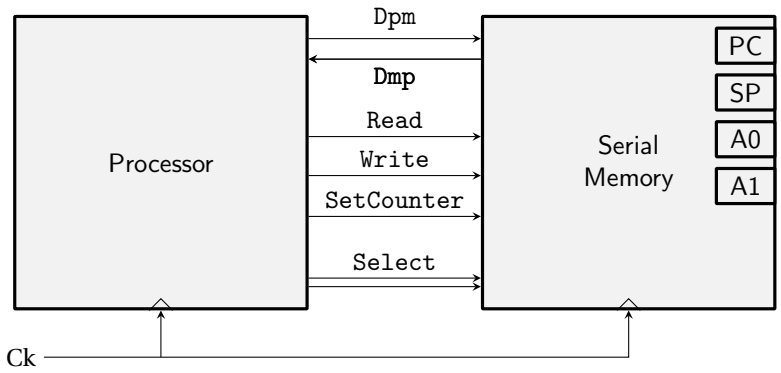
Exprimez les conditions suivantes en fonction des drapeaux : résultat positif ou nul, négatif ou nul, strictement positif, strictement négatif, dépassement de capacité, pas de dépassement de capacité, etc.
Combien de bits faut-il pour coder un ensemble minimal de conditions (ne pas oublier le "sans condition") ? Définissez précisément le champ condition, et les mnémoniques correspondants.

4.2 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.

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.

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

6 Pour la semaine prochaine

Allez vous documenter sur "Huffman coding" sur les internets.

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
rmem	56		3 575	7.73
copy	174		376 428 822	8.13