julien.braine@ens-lyon.fr,mateusz.skomra@ens-lyon.fr, florent.de-dinechin@insa-lyon.fr

ASR1, TD2 : 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 long DM avec plusieurs rendus intermédiaires pour implémenter un assembleur pour cet ISA, un simulateur d'un processeur implémentant cet ISA, et du logiciel de base pour ce processeur. Le tout faisant ressortir en principe quelques bugs de conception de l'ISA ;
- 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. Vérifiez que vous comprenez cette phrase en l'expliquant à votre voisin(e).

Contexte et prétexte

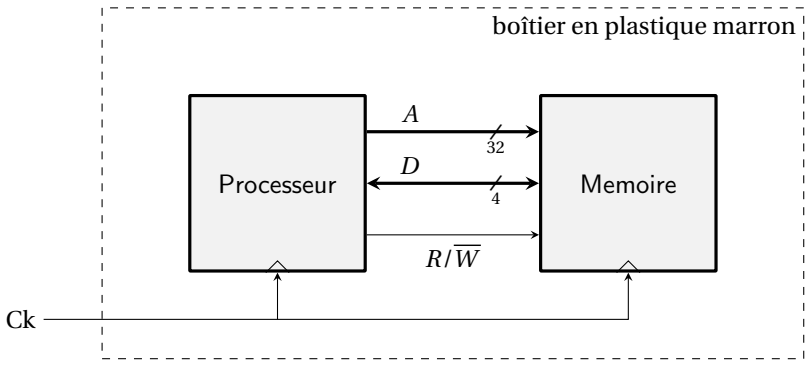
Il y a deux ans, vos petits camarades ont conçu un processeur totalement révolutionnaire dans lequel instructions comme données peuvent s'encoder au bit près, dans le but de minimiser la quantité d'information échangée entre processeur et mémoire. Ils en ont conclu que 1/ sans surprise, la majorité des échanges, dans une machine de von Neumann, c'est du code, et 2/ leur jeu d'instruction universel arrivait à une moyenne de 12 bits/instruction (opérandes compris).

Cette année il s'agit de faire mieux (en terme de taille de code). Nous allons pour cela revenir à une granularité d'encodage de 4 bits, parce que nous avons observé que l'encodage des tailles de toute chose avait un coût. Remarque : c'est moins révolutionnaire car les ISA x86 (IA32, EMT64) ont une granularité de 8 bits. Cela donne espoir qu'on saura implémenter notre ISA.

Si vous n'avez rien compris aux deux paragraphes précédents, ce n'est pas grave à ce stade.

Copions le processeur Saturn des calculettes HP

La dernière fois que j'ai programmé en binaire en assemblant du code à la main, c'était pour une architecture qui ressemblait à ceci (à part que le bus d'adresse faisait 20 bits seulement) :



P s'appelle un processeur 4/32 bits : les données sont sur 4 bits et les adresses sont sur 32 bits.

Nous allons tenter de concevoir un processeur moderne dans cette interface d'un autre âge.

Pour commencer, un petit vote : comment va-t-on appeler un paquet de 4 bits? Wikipedia suggère nibble (occasionally nybble or nyble to match the spelling of byte), half-byte, tetrade, semi-octet, quadbit, quartet, hex digit. Vous pouvez inventer un terme français.

Dans la suite j'utilise quartet.

Dans toute la suite, chaque fois qu'on se trouve devant un compromis/dilemme entre 2 solutions, la bonne méthodologie est d'implémenter les deux, d'écrire les deux compilateurs, de compiler Linux puis TuxKart, et de regarder la taille du code et sa vitesse dans les deux options. Nous on va faire tout cela de tête, forcément ce sera un peu approximatif.

1 Considérations générales

Sur 4 bits, on peut coder combien d'instructions différentes?

Est-ce assez?

Prenons 1/4 d'heure pour essayer de définir un processeur universel dont toutes les instructions sont sur 4 bits (données comprises, sinon ce n'est pas drôle). Vous devrez définir les registres internes du processeur.

La conclusion devrait être : laissons tomber, ce sera pénible à programmer (relire "Un bon jeu d'instruction c'est ..." p. 71 du poly).

On va donc se permettre des instructions de 1, 2, 3, 4 quartets... et plus si nécessaire. Essayez de reformuler la description du cycle de von Neumann dans ce contexte.

Dans toute la suite, on essaiera de distinguer des instructions "indispensables" (sans lesquelles le processeur n'est pas une machine universelle) et des instructions "de confort" (qui vont faciliter l'écriture de programmes compacts et/ou performants).

On essaiera aussi de se poser la question de la fréquence de chaque instruction dans un programme typique.

1.1 Instructions arithmétiques

Quelles sont les opérations logiques usuelles? Quelles sont à votre avis celles qui sont indispensables? De confort?

Quelles sont les opérations arithmétiques usuelles sur les entiers? Quelles sont à votre avis celles qui sont indispensables? De confort?

Même question sur les flottants.

(On ne prend pas de décision à ce stade, on liste les options.)

On arrive à plus de 16 ou moins de 16 instructions?

1.2 Combien de registres? Combien d'opérandes?

J'impose arbitrairement un "coeur RISC" : les opérations de calcul calculent sur des registres internes du processeur, et il y a des instructions d'accès mémoire pour copier la mémoire en registre et inversement.

Précisez bien combien chaque opération a d'entrée et de sortie.

Rappelez (pour l'addition) ce que sont une instruction à 3 opérandes, à 2 opérandes, à 1 opérande.

Etudiez l'encodage d'une instruction d'addition entière sous plusieurs hypothèses de nombres de registres, et de nombre d'opérandes de l'instruction.

Des études ARM et RISC-V qui comparent un jeu 32 bits 3 opérandes à un jeu 16 bits 2 opérandes montrent une réduction de la taille totale du code de 30% (ce serait 50% si on n'avait jamais besoin des instructions de copie). Cela peut-il vous aider à choisir entre 3 opérandes et 2 opérandes?

On ne prend pas de décision à ce stade.

1.3 Constantes

Dans les programmes il y a des constantes, qui sont de deux catégories :

- Les "petites" constantes, qu'on va essayer d'intégrer directement dans l'instruction. Pour cela deux solutions classiques :
  - R0 vaut toujours 0, R1 vaut toujours 1.
  - certaines instructions arithmétiques existent en version constante, par exemple
    - `ADD Rd <- Ri + Rj` et `ADDCONST Rd <- Ri + j`. Attention, c'est l'encodage du premier quartet qui détermine si le quatrième est interprété comme un numéro de registre ou comme un entier. Donc cela coute un code instruction! C'est pour marquer cela que j'ai pris un mnémonique différent : en vrai on pourrait écrire ADD dans les deux cas, il n'y a pas d'ambiguité.
    - `INC Rd`, qui fait `Rd <- Rd + 1` et tient en un octet. Mais coûte aussi un code instruction.

— Les “grandes” constantes, de 8, 16 ou 32 bits, qu’une instruction va transférer de la mémoire programme vers un registre. Par exemple `LOAD4 Rd xxxx` ou `Rd <- xxxx` (chaque `x` est un chiffre hexadécimal) qui s’encode en 6 quartets dans le programme.

Quelles sont les instructions indispensables ? De confort/performance ?

Discutez quelles instructions arithmétiques et logiques utilisent souvent de petites constantes, et si cela vaut le coup de les ajouter à notre ISA.

Si on veut gérer des constantes de tailles différentes, quelle vous paraît la meilleure méthode pour encoder la taille de la constante ?

2 Instructions d’accès mémoire

A minima, il faut être capable de charger une valeur de registre depuis une adresse quelconque en mémoire.

Essayez d’écrire un programme qui fait la somme de deux vecteurs d’octets (on n’a pas encore de sauts, donc vous ferez comme si sans vous poser de question)

3 Instructions de contrôle de flot

Définissez d’abord des instructions de saut relatif “pas trop loin” pour faire des boucles infinies.

La question est de rendre ces sauts conditionnels.

Il s’agit ici de comparer une solution à base de drapeaux (x86, ARM) et une solution à base d’instruction de saut embarquant un test sur un ou plusieurs registres (RISC-V).

Remarque : les processeurs de l’époque 8 bits avaient volontiers des instructions de type *decrement and jump if non zero* (djnz) pour les boucles `for` simples.

4 Et maintenant on tranche : la multiplication entière

Posez-moi en binaire sur votre feuille la multiplication de 17 par 42.

Le but de la suite est 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
```

... et de faire les choix qui minimisent la taille du programme.

Dans le processeur 16 bits d’il y a 3 ans, vos camarades l’ont implémenté en 8 instructions de 16 bits, donc 128 bits. Il y a 2 ans, ils sont arrivés à le faire tenir en 69 bits. L’an dernier, on n’a pas vraiment fait mieux (mais on peut dire que c’était la faute au sujet).

Et vous ?

5 Conclusion

On recompte les instructions qu’on aimerait garder et on vérifie qu’on arrive à toutes les encoder, et qu’il nous reste un peu de marge (il faut aussi ajouter de quoi gérer une pile d’appel et des interruptions, mais on est un peu jeunes encore)