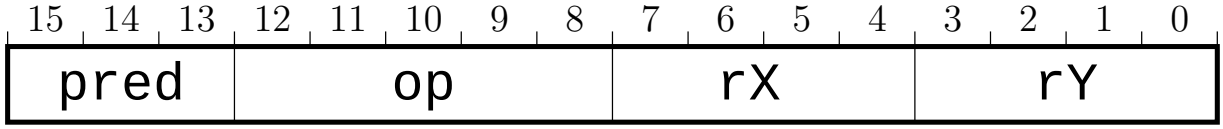
Jeu d'instructions du processeur VLIW 32 bits

13 décembre 2007

On ne décrit ici que le codage binaire des instructions. Vous trouverez des exemples de syntaxe du langage assembleur dans le fichier `asm/deco.asm`.

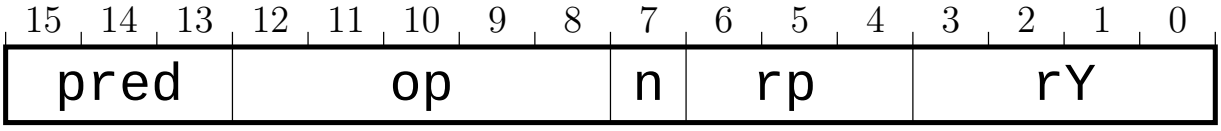
1 Les instructions générales

FIG. 1 – instruction de 16 bits pour les instructions générales à deux adresses



Opcode	Codage	Action
not	00000	rX ← NOT rY
and	00001	rX ← rX AND rY
or	00010	rX ← rX OR rY
xor	00011	rX ← rX XOR rY
lsl	00100	rX ← rX ≪ rY
lsr	00101	rX ← rX ≫ rY
asl	00110	rX ← rX ≪ rY signé
asr	00111	rX ← rX ≫ rY signé
add	01000	rX ← rX + rY
sub	01001	rX ← rX − rY
addc	01010	rX ← rX + rY de quelle retenue s'agit-il ?
subc	01011	rX ← rX − rY de quelle retenue s'agit-il ?
inc	01100	rX ← rY + 1
dec	01101	rX ← rY − 1
ld	01110	rX ← [rY] (voie 1 seulement)
str	01111	[rY] ← rX (voie 1 seulement)
ja	10000	PC ← rY
sra	10001	rX ← PC + 2
mov	10010	rX ← rY
unused	10011	←

FIG. 2 – instruction de 16 bits pour les instructions de comparaison



2 Les instructions de comparaison

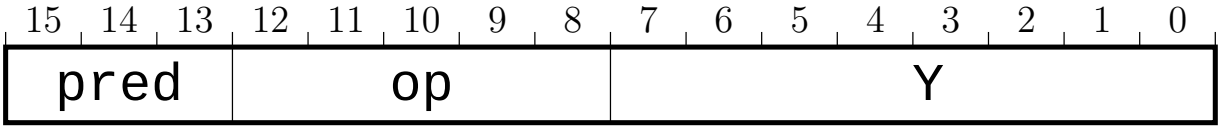
Mnémonique	op n	Action
gez	10100 0	rp ← 1 si rY ≥ 0 0 sinon
lsz	10100 1	rp ← 1 si rY < 0 0 sinon
lez	10101 0	rp ← 1 si rY ≤ 0 0 sinon
gsz	10101 1	rp ← 1 si rY > 0 0 sinon
eqz	10110 0	rp ← 1 si rY = 0 0 sinon
nez	10110 1	rp ← 1 si rY ≠ 0 0 sinon

Lorsque le bit `n` est à 1, on inverse le résultat : `ls` (less, strictly) devient `ge` (greater or equal), etc.

`rp` peut aller de 1 à 7 (il y a 7 registres de prédicat). La valeur du prédicat `000` est toujours vraie. Si on écrit dedans c'est comme pisser dans un violon. En fait, le `nop` est traduit par une comparaison qui écrit dans p0.

3 Les instructions à opérande immédiate

FIG. 3 – instruction de 16 bits pour les quatres instructions de chargement



Opcode	Codage	Action
ldi0	11100	(bits 7 à 0) r0 ← Y (voie 0 seulement)
ldi1	11100	(bits 15 à 8) r0 ← Y (voie 1 seulement)
ldi2	11101	(bits 23 à 16) r0 ← Y (voie 0 seulement)
ldi3	11101	(bits 31 à 24) r0 ← Y (voie 1 seulement)
jrf	11110	PC ← PC + Y
jrb	11111	PC ← PC − Y

Ces instructions ne sont pas tout-à fait identiques :

- Si on a `ldi1` et `ldi0` dans une instruction, alors on met les 16 bits de poids forts à 0 et les 16 bits de poids faibles sont définis par les constantes.
- Si on a juste `ldi0` en voie 0, alors on met les 24 bits de poids forts à 0.
- `ldi2` et `ldi3` ne touchent pas les bits de poids faible.
- Une instruction `ldi` seule met à 0 les bits de poids plus fort et laisse les bits de poids plus faibles

– Pour `ldi3/ldi0` disons qu'on verra ce qui s'implémente bien. Ce cas est sans doute anecdotique.

Remarque : toutes nos adresses tiennent sur 16 bits.

4 Ce qu'on a changé et pourquoi

On a ajouté un `mov`. J'invite les sceptiques à essayer d'écrire un programme sans. Ce sera du Perec. J'ai ajouté aussi le `not` après avoir tenu 15 lignes sans.

Au début je voulais laisser le nop comme `mov r0 r0` mais en VLIW c'est faux : si l'autre voie touche aussi `r0` on a une situation indéterminée.

Les sauts relatifs sont désormais à valeur immédiate, sinon il fallait systématiquement deux instructions, et brûler un registre.

Votre `push` a été renommé en `sra` pour *set return address* parce que `push`, c'est sur une pile.

Les comparaisons utilisent désormais le 4ème bit du champ instruction. On économise ainsi des op-codes, et c'est facile à implémenter. Elles ont été renommées aussi mais bon.

5 Situations indéterminées

Les situations indéterminées sont :

– Si deux opérandes destinations (registre ou registre de prédicat) sont identiques, par exemple
`and r0 r1 / or r0 r1`
Exception : `ldi1 / ldi0` et `ldi3/ldi2` sont autorisés comme décrit ci-dessus.

– Si les deux demi-instructions sont des sauts.

Dans le simulateur, la voie 0 (la voie de droite) est exécutée avant la voie 1, et les situations indéterminées se résolvent ainsi. Essayez de construire votre processeur pour qu'il fasse de même. Deux instructions prédiquées dont une seule s'exécute doivent faire ce qui est attendu.

Un objet du DM sera d'ajouter à l'assembleur un warning pour cette situation.

6 Syntaxe de l'assembleur

Vous trouverez dans `asm/deco.asm` un exemple de programme assembleur qui donne les grandes lignes de la syntaxe. Les deux demi-instructions sont séparées par `/`.

En début de demi-instruction, `?p3` se lit "si `p3` alors" et indique que la demi-instruction est prédiquée par `p3`.

On a deux pseudo-instructions utiles :

`ldl16 0x1234` est un raccourci pour `ldi1 0x12 / ldi0 0x34`.

L'utilité est flagrante si la constante est exprimée en décimal.

De même, `ldl32` permet de charger une constante 32 bits. Cette pseudo-instruction va se traduire en deux lignes (4 instructions).

Il y a aussi la pseudo-instruction `nop`.

On peut définir un label en début de ligne. C'est un identifiant suivi de " :". Ensuite, on peut utiliser un tel label derrière un `jrf`, un `jrb`, un `ldl16` ou un `ldl32` (exemple ci-dessous).

Enfin, vous pouvez demander à l'assembleur de produire une constante 32 bits arbitraire en la faisant précéder d'un *underscore* (_). Par exemple

```
pi: _31415926 ; m'étonnerait que cela serve
```

Plus loin dans votre programme vous chargerez une telle constante par exemple par :

```
ldl32 pi / nop ; l'adresse de la constante dans r0
ldr r1 r0 / nop ; accès mémoire
```

Bien sûr, le `;` définit un commentaire qui va jusqu'à la fin de la ligne.