ASR1 2016
Florent de Dinechin, Nicolas Louvet, Antoine Plet

Processeur ASR1/2016

Instruction Set Architecture

résultat du vote russe, version 2 (après TD4)

Dans le but de faire avancer la Science, nous avons décidé de garder tous les choix un peu originaux faits lors du TD3 :

- Machine à 3 opérandes, et 16 registres de 16 bits.
- Les opérations de calcul à trois opérandes ne peuvent utiliser que les 8 premiers registres comme destination. Par contre les copies (registre à registre, ou mémoire à registre) peuvent utiliser tous les registres comme destination.
- C'est une machine sans drapeaux, ceux-ci étant remplacés par une instruction de saut conditionnel de la prochaine instruction qui a un encodage presqu'identique aux instructions de calcul [1].

La mémoire est adressée par mot de 16 bits, pas par octet.

1 Syntaxe de l'assembleur

Les registres généralistes sont notés r0 à r15. Ils sont tous parfaitement identiques, sauf r15 qui reçoit l'adresse de retour en cas de `call`. Les instructions commencent toutes par un mnémonique, suivi des opérandes, le tout séparé par des espaces. La destination vient en premier pour les instructions de calcul. L'adresse mémoire vient toujours en second pour les instructions d'accès mémoire.

Sucre syntaxique offert par l'assembleur :

- L'assemblage commence à l'adresse 0, qui est celle à laquelle notre processeur démarre.
- On peut utiliser des labels pour les sauts.
- Le mot-clé `.word xxxx` réserve juste une case mémoire, initialisée à la constante 16 bits xxxx.
- Les constante hexadécimales sont préfixées par `0x`, par exemple `letl r15 0xff`
- Le mot cle `.align16` saute des cases mémoires jusqu'à arriver à une adresse multiple de 16. L'utilité paraîtra dans la suite...

Exemple d'instructions :

```
  letl r0 17
boucle:
  wmem r13 [r0]
  rmem r13 [r2]
  add r0 r10 r11
  snif r0 eq 3
  jump boucle  ; signifie  jump -3, et ceci est un commentaire
  xor r0 r0 -1
```

1. Comme tout a déjà été inventé, je vous invite à aller wikipedier les micro-contrôleurs de la famille PIC, qui ont quelquechose qui ressemble et pourrait même être mieux.

2 Classes d'instructions et leur encodage

Le code instruction est essentiellement encodé dans les bits 12 à 15, selon la table 1. L'encodage précis de chaque classe d'instruction mentionnée dans la table 1 est donné par la table 2. Certaines des entrées de ces tables sont précisées dans la suite.

TABLE 1 – Les 16 (et quelques) instructions

15	14	13	12	mnemonic	class	description	ext(i)
0	0	0	0	`wmem`	wmem	write to memory	
0	0	0	1	`add`	ALU	addition	$z(i)$
0	0	1	0	`sub`	ALU	subtraction	$z(i)$
0	0	1	1	`snif`	snif	skip next if	$s(i)$
0	1	0	0	`and`	ALU	logical bitwise and	$s(i)$
0	1	0	1	`or`	ALU	logical bitwise or	$s(i)$
0	1	1	0	`xor`	ALU	logical bitwise xor	$s(i)$
0	1	1	1	`lsl`	ALU	logical shift left	$z(i)$
1	0	0	0	`lsr`	ALU	logical shift right	$z(i)$
1	0	0	1	`asr`	ALU	arithmetic shift right	$z(i)$
1	0	1	0	`call`	call	sub-routine call	
1	0	1	1	`jump` `return`	jump	relative jump if `offset` $\neq 1$ return from call if `offset` $= 1$	
1	1	0	0	`letl`	letl	8-bit constant to Rd, sign-extended	
1	1	0	1	`leth`	leth	8-bit constant to high half of Rd (see Section 2.3)	
1	1	1	0			"reserved for future extensions"	
1	1	1	1	`rmem` `copy`	rmem	read from memory if i=0 register-to-register copy if i=1	

TABLE 2 – Encodage par classe d'instructions.

classe	action	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ALU reg	$r_d \leftarrow r_i$ op r_j	opcode				0	d			i				j			
ALU imm4	$r_d \leftarrow r_i$ op ext(i)	opcode				1	d			i				j			
snif	see Section 2.2	0	0	1	1	$c/\overline{r}$	condition			i				j			
letl	$r_d \leftarrow s(b)$	1	1	0	0	d				b							
leth	$r_d[15..8] \leftarrow b$	1	1	0	1	d				b							
call	see Section 2.4	1	0	1	0	c											
jump	see Section 2.4	1	0	1	1	c											
return	see Section 2.4	1	0	1	1	0											1
wmem	`mem`$[r_j] \leftarrow r_i$	0	0	0	0	0	0	0	0	i				j			
rmem	$r_d \leftarrow$ `mem`$[r_j]$	1	1	1	1	d				0	0	0	0	j			
copy	$r_d \leftarrow r_j$	1	1	1	1	d				0	0	0	1	j			

2.1 Instructions arithmétiques et logiques

Ces instructions prennent pour opérande soit deux registres (si le bit 11 vaut 0), soit un registre et une constante immmédiate, c'est-à-dire stockée dans l'instruction elle-même (si le bit 11 vaut 0). Cette constante j tient donc en 4 bits, et le processeur doit l'étendre en une constante de 16 bits avant l'opération. Ceci est précisé par la dernière colonne de la table 1 :

— $z(j)$ signifie l'extension de j par des zéros ; autrement dit j est interprété comme un entier positif.
— $s(j)$ signifie la copie du bit 3 (bit de signe) de j dans les bits 4 à 15 ; autrement dit j est interprété comme un entier signé et est transformé en un entier de 16 bits de même valeur.

2.2 L'instruction `snif`

La syntaxe du code assembleur est `snif op1 <condition> op2` . Cette instruction a pour effet de désactiver l'instruction suivante si la condition est vraie. Les opérandes 1 et 2 sont encodés comme dans les instructions ALU. En particulier le second opérande peut être une constante immédiate, dont le signe sera étendu quelle que soit la condition.

La condition est encodée dans les bits 10 à 8 selon le tableau suivant :

10	9	8	mnemonic	description
0	0	0	`eq`	equal, op1 = op2
0	0	1	`neq`	not equal, op1 ≠ op2
0	1	0	`sgt`	signed greater than, op1 > op2, two's complement
0	1	1	`slt`	signed smaller than, op1 < op2, two's complement
1	0	0	`gt`	op1 > op2, unsigned
1	0	1	`ge`	op1 ≥ op2, unsigned
1	1	0	`lt`	op1 < op2, unsigned
1	1	1	`le`	op1 ≤ op2, unsigned

La différence entre `sgt` et `gt` s'observe sur l'exemple suivant : Si r_0 contient 0, la condition est fausse pour `snif r0 gt -1`, alors qu'elle est vraie pour `snif r0 sgt -1`. En effet, dans les deux cas le -1 est étendu en `ffff` (hexa), qui est interprété comme 65535 par `gt`, et -1 par `sgt`.

2.3 Les instructions `leth` et `letl`

La constante est encodée dans les bits 0 à 7. Pour `letl` le bit 7 de la constante (bit de signe) est répliqué sur les bits 8 à 15 du registre destination. Pour `leth`, la constante est copiée dans les bits 8 à 15 du registre de destination, et les bits 0 à 7 sont inchangés. Ainsi on peut charger une constante arbitraire dans un registre en un `letl` suivi d'un `leth`. Pour les constantes entre -128 et 127, un `letl` suffit.

2.4 Les instructions de branchement

Soit `a` l'adresse de l'instruction, et `c` l'entier formé des bits 0 à 11 du mot d'instruction. L'instruction `call` copie `a` + 1 dans r_{15}, puis réalise pc ← `c` × 16. Ainsi les procédures doivent avoir des adresses multiples de 16.

L'instruction `jump` considère `c` comme un entier signé (donc entre -2048 et 2047) et réalise pc ← `a` + `c` sauf si `c` = 1, auquel cas elle réalise pc ← r_{15} (retour de sous-routine). Dans ce cas on peut utiliser le mnémonique `return`.

2.5 Les instructions d'accès mémoire

L'adresse mémoire est toujours spécifiée par le registre `rj` dont le numéro est dans les bits 0 à 3. L'instruction `rmem rd [rj]` copie dans le registre destination (codé dans les bits 8 à 11) le contenu de la case mémoire dont l'adresse est contenue dans `rj`. L'instruction `wmem ri [rj]` copie le contenu du registre `ri` (codé dans les bits 4 à 7) dans la case mémoire dont l'adresse est contenue dans `rj`. On pourra (plus tard) utiliser les bits inutiles pour `push` et `pop`.