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

Instruction Set Architecture 2017 - Version 2

Les changements par rapport à la version 1 sont en rouge.

1 Syntaxe de l'assembleur

Les registres généralistes sont notés r0 à r7. Ils sont tous parfaitement identiques, sauf r7 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.

Les instructions ALU viennent en version 2 et 3 opérandes, la destination venant toujours en premier. Par exemple,

`add2 r0 r1` réalise $r_0 \leftarrow r_0 + r_1$.
`add3 r0 r1 r2` réalise $r_0 \leftarrow r_1 + r_2$.

Le suffixe `i` signifie que le dernier opérande est une constante immédiate, par exemple :

`add2 r0 1` réalise $r_0 \leftarrow r_0 + 1$.

L'assemblage commence à l'adresse 0, qui est celle à laquelle notre processeur démarre.

Sucre syntaxique offert par l'assembleur :

— On peut utiliser des labels pour les sauts.
— Le mot-clé `.const n xxxx` réserve n bits de mémoire, initialisés à la constante xxxx, qui est priée de tenir sur n bits.
— Les constante hexadécimales sont préfixées par `0x`, par exemple `0xff`
— Le commentaire est introduit par un point-virgule ;

Exemple de programme :

```
  let r0 17     ; l'assembleur va calculer combien de bits il faut pour 17
boucle:
  sub2i r0 1    ; encodé en 9 bits, et ceci est un commentaire
  jumpif nz boucle  ; encodé en 16 bits, signifie  jump -25
```

2 Les instructions et leur encodage

La table 1 décrit l'opcode qui commence chaque instruction.

Remarques en vrac :

— le not logique est implémenté par xor -1
— la direction du shift est encodée dans un bit après l'instruction pour économiser un opcode. On aurait pu définir deux opcodes comme pour `uread`/`sread` mais c'est plus rigolo de lire `shift left r1 1`.

2.1 Les instructions de branchement

Soit a l'adresse du premier bit suivant l'instruction `jump` ou `call` (i.e. la valeur du PC lorsqu'il a fini de lire l'instruction et ses opérandes). Soit d la valeur de déplacement (encodée dans une constante de type *addr*, et signée).

L'instruction `jump` réalise `pc` $\leftarrow a + c$. L'instruction `jumpif` aussi, mais seulement si la condition est vraie.

La condition est encodée sur trois bits selon la table 3.

La différence entre `sgt` et `gt` s'observe par exemple sur la comparaison entre `r0` et -1.

L'instruction `call` copie `a` dans r_7, puis réalise `pc` $\leftarrow$ `d` (c'est un peu bizarre de sauter à des adresses négatives mais du coup *addr* est toujours signé).

L'instruction `return` copie `r7` dans `pc`.

TABLE 1 – Liste des instructions.

Les opérandes d'une instruction la suivent en mémoire. Ils sont encodés comme suit :

- *reg* ∈ {`r0`, `r1`, ..., `r7`} et est encodé par le numéro du registre en binaire.
- *const*, *shiftval* et *addr* sont définis par la table 2. La dernière colonne de la table 1 précise si une constante est étendue avec son signe (s) ou des zéros (z).
- *cond* est défini par la table 3.
- *ctr* est défini par la table 4.
- *dir* peut être le mnemnonique `left`, encodé par 0, ou le mnemonique `right`, encodé par 1.

opcode	mnemonic	operands	description	ext.	MàJ flags
0000	`add2`	*reg reg*	addition		zcvn
0001	`add2i`	*reg const*	add immediate constant	z	zcvn
0010	`sub2`	*reg reg*	subtraction		zcvn
0011	`sub2i`	*reg const*	subtract immediate constant	z	zcvn
0100	`cmp`	*reg reg*	comparison		zcvn
0101	`cmpi`	*reg const*	comparison with immediate constant	s	zcvn
0110	`let`	*reg reg*	register copy		
0111	`leti`	*reg const*	fill register with constant	s	
1000	`shift`	*dir reg shiftval*	logical shift		zcn
10010 10011	`readze` `readse`	*ctr size reg* *ctr size reg*	read *size* memory bits (zero-extended) to *reg* read *size* memory bits (sign-extended) to *reg*		
1010	`jump`	*addr*	relative jump		
1011	`jumpif`	*cond addr*	conditional relative jump		
110000	`or2`	*reg reg*	logical bitwise or		zcn
110001	`or2i`	*reg const*	logical bitwise or	z	zcn
110010	`and2`	*reg reg*	logical bitwise and		zcn
110011	`and2i`	*reg const*	logical bitwise and	z	zcn
110100	`write`	*ctr size reg*	write the lower *size* bits of *reg* to mem		
110101	`call`	*addr*	sub-routine call	s	
110110	`setctr`	*ctr reg*	set one of the four counters to the content of *reg*		
110111	`getctr`	*ctr reg*	copy the current value of a counter to *reg*		
1110000	`push`	*reg*	push value of register on stack		
1110001	`return`		return from subroutine		
1110010	`add3`	*reg reg reg*			zcvn
1110011	`add3i`	*reg reg const*		z	zcvn
1110100	`sub3`	*reg reg reg*			zcvn
1110101	`sub3i`	*reg reg const*		z	zcvn
1110110	`and3`	*reg reg reg*			zcn
1110111	`and3i`	*reg reg const*		z	zcn
1111000	`or3`	*reg reg reg*			zcn
1111001	`or3i`	*reg reg const*		z	zcn
1111010	`xor3`	*reg reg reg*			zcn
1111011	`xor3i`	*reg reg const*		z	zcn
1111100	`asr3`	*reg reg shiftval*			zcn
1111101			reserved		
1111110			reserved		
1111111			reserved		

TABLE 2 – Encodage des constantes

addr : Encodage *prefix-free* des adresses et déplacements	
0 + 8 bits	adresse ou déplacement sur 8 bits
10 + 16 bits	
110 + 32 bits	
111 + 64 bits	
shiftval : Encodage *prefix-free* des constantes de shift	
0 + 6 bits	constante entre 0 et 63
1	constante 1
const : Encodage *prefix-free* des constantes ALU	
0 + 1 bit	constante 0 ou 1
10 + 8 bits	octet
110 + 32 bits	
111 + 64 bits	
size : Encodage *prefix-free* des tailles mémoire	
00	1 bit
01	4 bits
100	8 bits
101	16 bits
110	32 bits
111	64 bits

TABLE 3 – Condition codes

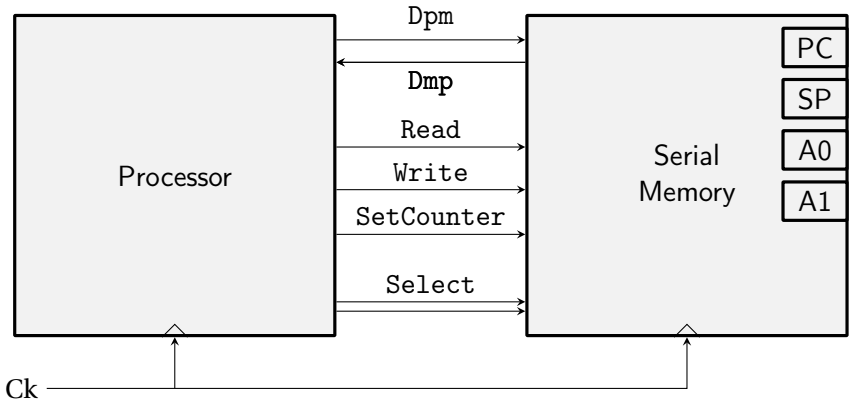
			mnemonic	description (after `cmp` op1 op2)	implem.
0	0	0	`eq`, `z`	equal, op1 = op2	z
0	0	1	`neq`, `nz`	not equal, op1 ≠ op2	$\overline{z}$
0	1	0	`sgt`	signed greater than, op1 > op2, two's complement	$\overline{z} \wedge (n \wedge v) \vee (\overline{n} \wedge \overline{v})$
0	1	1	`slt`	signed smaller than, op1 < op2, two's complement	$(n \wedge \overline{v}) \vee (\overline{n} \wedge v)$
1	0	0	`gt`	op1 > op2, unsigned	$\overline{z} \wedge \overline{c}$
1	0	1	`ge`, `nc`	op1 ≥ op2, unsigned	$\overline{c}$
1	1	0	`lt`, `c`	op1 < op2, unsigned	c
1	1	1	`v`	two's complement overflow	v

TABLE 4 – Counters. Ces deux bits sont transmis sur le signal `Select` de la figure 1.

encoding	mnemonic	description
00	`pc`	program counter
01	`sp`	stack pointer
10	`a0`	generic address counter
11	`a1`	generic address counter

2.2 Les instructions d'accès mémoire

FIGURE 1 – Overview of the processor-memory interface



On a 4 compteurs d'adresses, chacun répliqué dans le processeur et dans la mémoire (Table 4).

Les instructions `readze`, `readse` et `write` lisent ou écrivent le nombre spécifié de bits tout en incrémentant les compteurs correspondant.

On peut émuler une instruction de lecture/écriture mémoire d'un processeur classique en deux instructions : un `setctr` puis un `readze` ou `readse` ou `write`.

Les instructions `push` et `pop` implémentent une pile descendante en mémoire :

— `push` *size reg* réalise :
  sp ← *sp* − *size*
  `setctr` *sp*
  `write` *sp size reg*
  sp ← *sp* − *size*
  `setctr` *sp*

— `pop` *size reg* est un raccourci offert par l'assembleur pour
  `readze` *sp size reg*