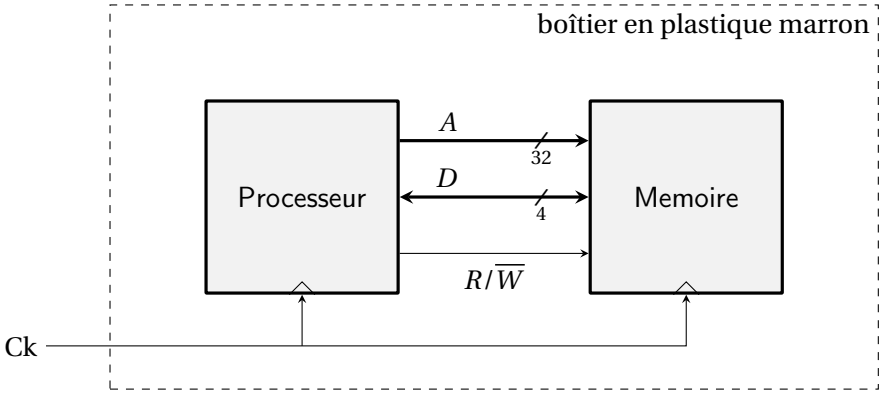
Architecture, Système, Réseau 2019-2020
julien.braine@ens-lyon.fr,mateusz.skomra@ens-lyon.fr, florent.de-dinechin@insa-lyon.fr

Instruction Set Architecture 2019, version 1

1 Généralités

N'hésitez pas à faire remonter des imprécisions, incohérences ou erreurs : cela ne vous fera pas perdre des points, au contraire !



On parle d'un processeur 4/32 bits : les données sont sur 4 bits et les adresses sont sur 32 bits.

Dans tout ce DM on appelle *mioche* (pour MI-Octet HExadécimal) un paquet de 4 bits. La mémoire est adressée par mioche avec des adresses de 32 bits : la taille du bus d'adresse est de 32 bits, la taille du bus de données est de 4 bits.

L'architecture est petitboutiste (little endian) : lorsqu'une valeur de 16 ou 32 bits est encodée en mémoire à l'adresse a, le mioche de poids faible est placé à l'adresse a et les mioches de poids plus élevés sont placés aux adresses $a+1$, $a+2$, etc.

Toutes les instructions sont encodées en un mioche ou deux, éventuellement suivis par d'autres mioches pour les arguments de l'instruction.

Les registres du processeur font tous 32 bits. Il y a 16 registres de travail appelés `R0` à `RF` car on les compte en hexa. `R0` vaut toujours 0, et on a le droit de l'écrire `0` ; `R1` vaut toujours 1 et on a le droit de l'écrire `1` ; Pour les instructions qui ont R0 ou R1 comme destination, sauf avis contraire seul le drapeau est éventuellement modifé.

Il y a aussi des registres spéciaux (implicites dans les instructions) : un PC, un PCRA (PC return address) pour les appels de procédures, et un drapeau d'un bit, appelé ϕ comme флажок parce que F et D sont des chiffres hexa qui sèmeraient la confusion.

Sauf avis contraire, toute l'arithmétique est en complément à 2 sur 32 bits.

L'assemblage commence à l'adresse 1024, qui est celle à laquelle notre processeur démarre. Les adresses précédentes sont utilisées pour les vecteurs d'appels systèmes, ce que vous comprendrez peut-être un bientôt.

L'écran occupe la fin de la mémoire (voir l'adresse exacte dans le simulateur).

2 Syntaxe de l'assembleur

Exemple de programme (non testé) qui calcule la somme des nombres de 1 à n, n etant la valeur de R2 à l'entrée du programme :

```
  copy r3 r0 ;
boucle:             ; ceci est une déclaration d'étiquette
  add r3, r2
  sub r2, r1        ; pourrait s'écrire sub r2 1
  isequal r2, r0    ; pourrait s'écrire isequal r2 0
  jf boucle        ; pourrait s'écrire bnz -9
; le résultat est dans r3
```

Exemple de programme (non testé) qui calcule la somme de deux vecteurs pointés par RA et RB et écrit le résultat dans RC, RF étant la taile du vecteur

```
boucle:
  isgt 1 rf
  jt fin
  mr8 r2 ra
  mru8 r3 rb
  add r2 r3
  mwu8 r2 rc
  sub rf 1
  j boucle
fin:
```

Le code assembleur est composé de mots séparés par des espaces, des tabulations ou des passages à la ligne. Un mot peut être :

- un mnémonique
- une constante (écrite en décimal, ou en hexadécimal précédé de `0x`). L'assembleur devra vérifier l'absence de débordement de capacité.
- une déclaration d'étiquette (en étranger *label*, voir ci-dessous)
- une étiquette

Dans tout ceci, seuls les étiquettes sont sensibles à la casse.

Pour les instructions arithmétiques à deux registres, la destination est en général à gauche. Autrement dit on peut les lire en ajoutant un "to" juste avant les deux opérandes, par exemple `copy R3 R0` se lit "copy to R3 R0"; `add R3 R7` se lit "add to R3 R7".

Pour les instructions mémoires, l'adresse est toujours le second argument.

Sucre syntaxique offert par l'assembleur

- Une déclaration d'étiquette est un mot qui se termine par un `:`, par exemple `boucle:`
- On peut utiliser des étiquettes en places des constantes pour les sauts (`j`, `jt`, `jf` et `call`).
- Les mot-clé `.const4 X`, `.const8 XX` , `.const16 XXXX` et `.const32 XXXXXXXX` réservent respectivement 1, 2, 4 et 8 mioches de mémoire, initialisée à la constante `X...X`, qui est priée de tenir sur ce nombre de bits.
- Le point-virgule commence un commentaire qui se termine à la fin de la ligne.
- vous avez le droit d'écrire r0 0 et r1 1.

3 Les instructions et leur encodage

X désigne un chiffre hexadécimal mais vous pouvez écrire les constantes en décimal ;
x et y sont aussi des chiffres hexa, mais ils désignent des numéros de registres ;
La colonne "P" indique la priorité/l'ordre de compréhension/implémentation.

hexa	opcode	args	description	P
0XX	j	XX ou label	jump : PC+=XX. XX=000 est une boucle infinie (cf remarques)	1
1XX	jt	XX ou label	jump if true (i.e si $\phi = 1$)	1
2XX	jf	XX ou label	jump if false (i.e si $\phi = 0$)	1
3xy	add	Rx Ry	addition, la retenue sortante va dans ϕ. Si y=0, au lieu d'ajouter 0, on ajoute 1 si $\phi = 1$.	1
4xy	sub	Rx Ry	subtraction (which bit do we want to go to ϕ?)	1
5xy	copy	Rx Ry	si Rx=R0, copie Ry dans le PC ; si Rx=R1 copie le PC dans Ry	1
6Xy	shiftdn	Rx	shift de Ry ; $d \in \{\mathtt{l},\mathtt{r}\}$, $n \in \{1,2,3,4,8,12,16,24\}$, encodés par le mioche X, voir les remarques. Si $n = 1$ le bit sorti va dans ϕ.	1
7			*reserved for future extension*, on ne sait jamais	1
8xy	isequal	Rx Ry	ϕ=1 si et seulement si Rx=Ry	1
9xy	isgt	Rx Ry	ϕ=1 si et seulement si Rx>Ry (complément à 2)	1
A0xy ... A7xy	mr1 ... mr8	Rx Ry	*memory read* à partir de l'adresse Ry de n mioches, recopiés dans Rx petit-boutistement avec extension de signe sur 32 bits. Le second mioche de l'instruction encode $n-1$. Si y=0, la destination est le PC.	2
A8xy ... AFxy	mru1 ... mru8	Rx Ry	comme mr1...mr7, mais avec mise à jour de Ry *(update)*, qui est augmenté de la valeur de n. Ici le second mioche de l'instruction encode $n-7$.	2
BXxy	mw1...8 mwu1...8,	Rx Ry	*memory write*, le mioche X est interprété comme dans mr. Les mioches de poids faible de Rx sont écrits en mémoire, petit-boutistement. Si x=0, c'est PCRA qui est écrit en mémoire	2
C00 ... C7F	syscall	n	*system call* : PCRA=(adresse suivant cette instruction). Ensuite, le second mioche (0..7) et le troisième sont concaténés, ce qui donne un entier n entre 0 et 127. Cet entier est multiplié par 8, ce qui donne une adresse alignée sur 32 bits. Le contenu de cette adresse est lu et placé dans le PC. En principe il faudrait aussi changer de mode mais on va zapper cela pour le DM.	3
C8xy ... CFxy	push	Rx Ry	Si mru est un "pop", cette instruction est le push correspondant. Si y=0, plutot qu'un 0 c'est le PCRA qui est écrit en mémoire	4
DX...X	call	XXXXXXXX	PCRA=(adresse suivant cette instruction), puis PC=XXXXXXXX.	
E	ret		PC=PCRA	3
F0	not	Rx Rd	logical bitwise negation	1
F1	and	Rx Rd	logical bitwise AND	1
F2	or	Rx Rd	logical bitwise OR	1
F3	xor	Rx Rd	logical bitwise XOR	1
F4yXX	ld2	Ry XX	chargement d'une constante 8 bits, avec extension de signe	2
F5yXXXX	ld4	Ry XXXX	chargement d'une constante 16 bits, avec extension de signe	2
F6yX...X	ld8	Ry XXXXXXXX	chargement d'une constante 32 bits, ou une étiquette (remplacée par son adresse à l'assemblage)	2
F7 ... FF			"reserved for future extension"	5

Remarques :

— Si vous vous sentez trahi par rapport au TD 2 et 3, je tente d'expliquer les raisons de la trahison en section 5.

— Pour les sauts relatifs il faut préciser par rapport à quelle valeur de PC (avant l'instruction ou après l'instruction). Le choix fait ici est que la constante signée (entre -128 et 127) est ajoutée à la valeur du PC au début de l'instruction (i.e. à l'adresse du mioche codant le **b** lui-même). L'intérêt est que si votre PC arrive dans une zone de mémoire non initialisée (que des 0) il y reste piégé...

— Pour les shifts, Le bit 3 de X code la direction (0 :gauche ; 1 : droite). Les bits 0 à 3 codent la distance selon le tableau suivant :

$x_3x_2x_1$	000	001	010	011	100	101	110	111
distance	1	2	3	4	8	12	16	24

— Pour les lectures et écritures mémoires, le second mioche de l'instruction (noté X) est toujours décodé comme suit :

  — Les bits 0 à 2 encodent un entier binaire positif sur 3 bits, donc entre 0 et 7. En y ajoutant 1, on obtient le nombre de mioches à lire n.

  — le bit 3 encode le `u` comme "update" : s'il est à 1, Ra est modifié par l'instruction.

— On peut sauvegarder le PCRA sur une pile pointée par Rx (par exemple) par `push [rx] 0`. Un pop qui fait le retour à cette adresse sera `mr8 [rx] 0`. Dans un monde idéal, on définirait des *pseudoinstructions* plus explicites, par exemple `pushra [rx]` et `poppc [rx]` mais je ne veux pas ajouter trop de bruit au sujet.

— Lors d'un accès mémoire avec update, la mise à jour de Ry a lieu en une seule fois, après que tous les accès mémoire ont été réalisé. Ceci précise ce qui se passe lors de `mru8 r4 r4` par exemple.

— Pour que les `syscall` marchent, il faut qu'au boot, le processeur copie les adresse des routines à appeler au début de la mémoire. Ce sera à faire pour les routines graphiques pour le rendu 2. C'est facilité par la possibilité d'avoir `ld8 ra label` par exemple.

4 Reserved for future extensions

Vous aurez le droit d'ajouter des instructions aux conditions suivantes :

— c'est négocié en amont avec Florent de Dinechin

— cela ne vous empêche pas de supporter tout le jeu d'instruction de base

— vous produisez des programmes en deux versions, avec et sans vos extensions, de telle manière à me convaincre que ces extensions sont généralement utiles

5 Discussion sur les choix fait suite au TD2

mr2, mr4, mr8 versus memory-read-and-shift

Un groupe brillant et motivé est venu avec l'excellente suggestion d'une instruction `read-mem+shift [Ra] Rx` qui ferait en gros
Rx=(Rx≪4)+[Ra] ; Ra=Ra+1.

Cela permet certes de charger des constantes de toutes les tailles, mais

— charger n mioches nécessite la lecture de $3n$ mioches instructions. Par contraste, avec les instructions proposées, charger n mioches toujours la lecture de 4 mioches d'instruction.

— Avec les instructions proposées, le chargement est atomique/ininterruptible, ce qui s'avérera une Bonne Chose.

— ... et un argument d'autorité : la construction en matériel des instructions proposées sera plus intéressante.