1. Systèmes de numération

Système unaire

- Chaque nombre $n \in \mathbb{N}$ se représente par $\overbrace{1 \cdots 11}^{n \text{ fois}}$
- L'addition correspond à la concaténation

Représentation positionnelle

- Généralisation du système décimal à une base $B \in \mathbb{N}_{\geq 2}$
- *Systèmes répandus*: binaire ($B = 2$), octal ($B = 8$), décimal ($B = 10$), hexadécimal ($B = 16$)
- *Chiffres*: éléments de $\{0, 1, \ldots, B-1\}$
- *Chiffres au-delà de 9*: $A = 10, B = 11, \ldots, F = 15, \ldots$
- *Valeur de s en base B*: $s_B = s_{n-1} \cdot B^{n-1} + \ldots + s_1 \cdot B^1 + s_0 \cdot B^0$
- *Exemple*: $8B5_{16} = 8 \cdot 16^2 + 11 \cdot 16^1 + 5 \cdot 16^0$
- Les zéros tout à gauche ne changent rien: $(0 \cdots 0s)_B = s_B$

Conversions

- *B à 10*: $s_0 + B \cdot (s_1 + B \cdot (s_2 + B \cdot (\ldots + B \cdot s_{n-1})))$
- *10 à B*: diviser à répétition par B et concaténer les restes de droite à gauche, par ex. $6_2 = \mathbf{110}$:
  $6 \div 2 = 3$ reste $\mathbf{0}$, $3 \div 2 = 1$ reste $\mathbf{1}$, $1 \div 2 = 0$ reste $\mathbf{1}$
- *B à B^m*: remplacer chaque bloc de taille m par sa valeur en base B^m, par ex. si $B^m = 2^3$: $10110 \to 26$
- *B^m à B*: éclater chaque symbole vers sa représentation de taille m en base B, par ex. si $B^m = 2^3$: $73 \to 111011$

Addition

- Se fait comme en base 10: additionner chiffre à chifre en base B et propager une retenue vers la gauche

Fractions

- *Exemple*: $(11{,}01)_2 = 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 3{,}25$
- *Chiffres non significatifs*: $(0 \cdots 0s{,}t0 \cdots 0)_B = (s{,}t)_B$

2. Architecture des ordinateurs

Architecture et organisation

- *Architecture*: spécification des services des composants
- *Organisation*: description physique des composants

Architecture de von Neumann

- *Mémoire principale*: stocke les programmes et leurs données
- *Processeur*: unité centrale de traitement de l'ordinateur
- *Unités d'entrée/sortie*: contrôlent les périphériques
- *Bus*: systèmes de communication entre les composants

Mémoire principale

- Suite de cellules d'octets identifiées par des *adresses* uniques
- Une adresse peut référer à: 1 *octet* (8 bits), 2 octets (*demi-mot*), 4 octets (*mot*), 8 octets (*double mot*)
- Quantité de mémoire utilisable limitée par taille des adresses
- *Big-endian*: `[00, 58, 40, 0F]` vaut 0058400F
  Little-endian: `[00, 58, 40, 0F]` vaut 0F405800
- *Alignement*: adresser 2^k octets à une adresse non divisible par 2^k — parfois: *interdit*, souvent: *ralentit l'accès*

Processeur

- *Jeu d'instructions* élémentaires, par ex: `add` `x10, x11, x12` (code d'opér. / opérandes)
- *Registres*: cellules de mémoire interne, très rapide d'accès
- *Code machine*: traduction des instructions en suite de bits
- *Compteur d'instruction*: pointe vers prochaine instruction
- *Unité de contrôle*: coordonne l'exécution des instructions
- *Unité arithmétique et logique*: calculs sur $\mathbb{Z}$ et chaines bits
- *Pipeline*: parallélisation des étapes d'exécution
- *RISC*: instructions simples, taille fixe, mémoire–*ou*–autre

3. Programmation en langage d'assemblage: ARMv8

Registres

- *Registres:* x_0–x_{30} (64 bits) ou w_0–w_{30} (32 bits)
- *Arguments:* x_0–x_7
- *Usage libre (sauvegardés par l'appelé):* x_{19}–x_{28}
- *Usage semi-libre (sauvegardés par l'appelant):* x_9–x_{15}

Organisation du code

- *Ligne*: `etiquette:  opcode  operandes  // Commentaire`
- *Étiquette*: nom symbolique d'une ligne de code
- *Exemple*:

```
impair:
        mov     x20, 3          // tmp = 3
        mul     x20, x20, x19   // tmp = tmp * n
        add     x19, x20, 1     // n   = tmp + 1
```

Données statiques

- *Adresse divisible par k*: `.align k`
- *Alloue k octets consécutifs*: `.skip k`
- *1, 2, 4, 8 octets*: `.byte v`, `.hword v`, `.word v`, `.xword v`
- *Chaîne de car.*: `.asciz s`
- *Nb. virg. fottante*: `.single f`, `.double f`

Segments de données

- *Instructions*: `.section ".text"`
- *Données en lecture seule*: `.section ".rodata"`
- *Données initialisées*: `.section ".data"`
- *Données non-initialisées*: `.section ".bss"`

Entrée/sortie (de haut niveau)

- *Affichage*: `printf(&format, val1, val2, ...)`
- *Lecture*: `scanf(&format, &var1, &var2, ...)`
- *Formats nombres*: `int32` (`%d`), `uint32` (`%u`), `uint32-hex` (`%X`), `float` (`%f`); 64 bits via préfixe `l`, par ex. `int64` (`%ld`)
- *Formats caractères*: `char` (`%c`) = 1 octet, `string` (`%s`)

4. Accès aux données

Adresses

- *Numérique*: entier non négatif, souvent en hexadécimal
- *Symbolique*: chaîne qui représente une adresse non connue

Modes d'adressage

- *Mode*: méthode pour récupérer la valeur d'un opérande
- *Récapitulatif des modes*:

Nom	Valeur récupérée	Exemple
immédiat	$i \mapsto i$	`mov x0, 42`
direct	$a \mapsto \text{mem}[a]$	—
par registre	$n \mapsto \text{reg}[n]$	`mov x0, x1`
indirect	$a \mapsto \text{mem}[\text{mem}[a]]$	—
indirect par registre	$n \mapsto \text{mem}[\text{reg}[n]]$	`ldr x0, [x1]`
indir. par reg. indexé	$n, i \mapsto \text{mem}[\text{reg}[n] + i]$	`ldr x0, [x1, i]`
indir. par reg. indexé pré-incrémenté	$\text{reg}[n] \leftarrow \text{reg}[n] + i$, suivi de $n, i \mapsto \text{mem}[\text{reg}[n]]$	`ldr x0, [x1, i]!`
indir. par reg. indexé post-incrémenté	$n, i \mapsto \text{mem}[\text{reg}[n]]$, suivi de $\text{reg}[n] \leftarrow \text{reg}[n] + i$	`ldr x0, [x1], i`
relatif	$i \mapsto \text{mem}[\text{reg}[pc] + i]$	`ldr x0, var`

Accès mémoire sur ARMv8

- *Chargement et stockage*:

# octets	chargement	stockage
1	`ldrb wd, a`	`strb wd, a`
2	`ldrh wd, a`	`strh wd, a`
4	`ldr wd, a`	`str wd, a`
8	`ldr xd, a`	`str xd, a`

- *Autres instructions*:

```
adr r, etiq   // charge adr(etiq) dans reg. r
mov r, s      // charge reg. s    dans reg. r
mov r, i      // charge valeur i  dans reg. r
```

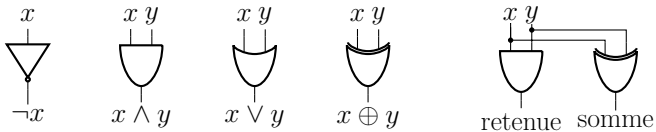
Assemblage

- *Assembleur*: instructions → code machine; la plupart des adresses symboliques → adresses numériques
- *Éditeur de liens*: fichiers objets → fichier exécutable; recalcule certaines adresses; adresses symboliques → numériques

5. Nombres entiers

Circuits logiques

- « Blocs » de base d'un processeur, faits de portes logiques qui permettent d'implémenter le jeu d'instructions:



Représentation des entiers signés

- *Complém. à 2*: $\text{val}(b_{n-1}\cdots b_1 b_0) = -b_{n-1}\cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i \cdot 2^i$
- *Nombres représentables sur n bits*: $[-2^{n-1}, 2^{n-1}-1]$
- *Bit de signe*: bit de gauche = 1 ssi le nombre est négatif
- *Ajout de bits*: répéter bit de signe à gauche: $101 \to 1\cdots101$
- *Changement de signe*: $010 \xrightarrow{\text{complément}} 101 \xrightarrow{+1} 110$

Opérations arithmétiques

- *Addition*: même procédure que pour les entiers non signés
- *Soustraction*: addition/changement de signe: $a-b = a+(-b)$
- *Multiplication et division non signées*: comme en base 10:

$$\begin{array}{rr} & 101 \quad (5) \\ \times & 11 \quad (3) \\ \hline & 101 \\ + & 101 \\ \hline & 1111 \quad (15) \end{array} \qquad \begin{array}{rr} & 10011 \;\big|\; 11 \\ - & 11 \quad\;\; 00110 \\ \hline & 111 \\ - & 11 \\ \hline & 1 \end{array}$$

- *Mult. signée*: étendre opérandes à $2n$ bits et garder $2n$ bits faibles du résultat (s'implémente sans extension explicite)
- *Division signée*: calculer $|a| \div |b|$ et ajuster signe

Codes de condition

- *Codes*: `N` (négatif), `Z` (zéro), `C` (report) (bit de retenue), `V` (débordement) (résultat trop grand/petit)
- *Codes modifiés par*: **`adds`**, **`subs`**, **`negs`**, **`adcs`**, **`cmp`**
- *Comparaison*: codes mis à jour via soustraction bidon
- *Accès aux codes*: via **`b.condition`** `etiq`
- *Accès au report*: **`adc`** `rd, rn, rm`: $\texttt{r}_\texttt{d} \leftarrow \texttt{r}_\texttt{n} + \texttt{r}_\texttt{m} + \texttt{C}$

6. Tableaux

Généralités

- *Tableau*: collection d'éléments identifiés par des indices
- *Éléments*: tous de même taille, contigus en mémoire
- *Indice*: tuplet $\boldsymbol{i}$ de dimension $d \geq 1$
- *Bornes*: $0 \leq \boldsymbol{i} < n_i$ pour chaque dimension i
- *Taille*: $n_0 \cdot n_1 \cdots n_{d-1}$ éléments
- *Exemples tableau 1D et tableau 2D*:

0	123
1	5
2	0
3	255
4	42

$n_0 = 5$
5 éléments

$(0,0)$	2
$(0,1)$	33
$(1,0)$	65535
$(1,1)$	73
$(2,0)$	9000
$(2,1)$	255

$n_0 = 3$, $n_1 = 2$
6 éléments

Calcul d'index

- *Index*: adresse relative à laquelle est stocké un élément
- *Calcul*: si a = adresse du tableau et k = nombre d'octets d'un élément, alors l'adresse de l'élément $\boldsymbol{i}$ =

$$\underbrace{a + \underbrace{i \cdot k}_{\text{index élém. } i}}_{\text{(tableau 1D)}} \qquad \underbrace{a + \underbrace{(i \cdot n_1 + j) \cdot k}_{\text{index élém. } (i,j)}}_{\text{(tableau 2D)}}$$

Allocation/accès mémoire

- *Tableau non initialisé*:

```
.section ".bss"
        .align 2
tab:    .skip  3*2*2      // n0 * n1 * # octets
```

- *Tableau initialisé*:

```
.section ".data"
tab:    .hword  2, 33, 65535, 73, 9000, 255
```

- *Accès*: via **str**/**ldr** (+ variantes) et modes d'adressage

7. Programmation structurée

Séquence

- Composition séquentielle d'instructions
- Une instruction de haut niveau peut nécessiter plusieurs instructions de bas niveau; par ex. `x19 *= 7` devient:

```
mov     x20, 7
mul     x19, x19, x20
```

Sélection

- Exécution conditionnelle d'instructions (**`if`**, **`switch`**, ...)
- *Implémentation*: branchements avant:

```
if (cond(xd, xn)) {
  // code si
}
else {
  // code sinon
}
```

```
si:
        cmp         xd, xn
        b.¬cond     sinon
        // code si
        b           fin
sinon:
        // code sinon
fin:
```

- *Conditions multiples*: obtenues via plusieurs sélections

Itération

- Exécution répétée d'instructions (**`while`**, **`do while`**, **`for`**, ...)
- *Implémentation*: branchements arrière, et parfois avant:

```
while (cond(xd, xn)) {
  // code
}
```

```
boucle:
        cmp         xd, xn
        b.¬cond     fin
        // code
        b           boucle
fin:
```

Sous-programmes

- Permettent de modulariser le code en sous-routines
- *Paramètres*: par valeur ou adresse dans x_0 à x_7 (en ordre)
- *Appel*: **`bl`** `sprog` assigne $x_{30} \leftarrow pc + 4$ et branche à `sprog:`
- *Retour*: **`ret`** branche vers l'adresse de retour x_{30}
- *Sauvegarde*: l'appelé doit rétablir les registres x_{19} à x_{30}

8. Valeurs booléennes et bits

Valeurs booléennes

- *Correspond à* un bit: $1 = vrai$, $0 = faux$
- *Représentation*: sur un octet, puisque bits non adressables

Opérateurs logiques

- *Opérations*: $\neg$, $\wedge$, $\vee$, $\oplus$ « bit à bit » étendues aux chaînes:

`mvn x19, x20`

$\neg$	…	$\neg$	$\neg$	$\neg$
0	…	1	0	1
1	…	**0**	**1**	**0**

`and x19, x20, x21`

0	…	1	0	1
$\wedge$	…	$\wedge$	$\wedge$	$\wedge$
1	…	1	0	0
0	…	**1**	**0**	**0**

`orr x19, x20, x21`

0	…	1	0	1
$\vee$	…	$\vee$	$\vee$	$\vee$
1	…	1	0	0
1	…	**1**	**0**	**1**

`eor x19, x20, x21`

0	…	1	0	1
$\oplus$	…	$\oplus$	$\oplus$	$\oplus$
1	…	1	0	0
1	…	**0**	**0**	**1**

- *Échange de valeurs*: se fait sans registre temporaire avec **`eor`**

Décalages logiques et arithmétiques

- Décale les bits de j positions vers la gauche/droite:

11000101 $\xrightarrow{\text{3 bits vers la gauche}}$ 00101000 **`lsr xd, xn, 3`**

11000101 $\xrightarrow{\text{3 bits vers la droite}}$ 00011000 **`lsl xd, xn, 3`**

- Bit de signe copié lors d'un décalage arithmétique à droite:

11000101 $\xrightarrow{\text{3 bits vers la droite}}$ 11111000 **`asr xd, xn, 3`**

- *Multiplication/division*: par 2^k correspond à un décalage de k bits vers la gauche/droite

Décalages circulaires

- Comme un décalage logique, mais les bits « perdus » sont réinsérés de l'autre côté:

11000101 $\xrightarrow{\text{3 bits vers la gauche}}$ 00101110 n'existe pas sur ARMv8

11000101 $\xrightarrow{\text{3 bits vers la droite}}$ 10111000 **`ror xd, xn, 3`**

Masquage

- Permet d'isoler certains bits à manipuler:

$r \wedge m$	Sélection	Met à 0 les bits de r non spécifiés par m
$r \vee m$	Activation	Met à 1 les bits de r spécifiés par m
$r \wedge \neg m$	Désactivation	Met à 0 les bits de r spécifiés par m
$r \oplus m$	Basculement	Inverse les bits de r spécifiés par m

9. Chaînes de caractères

Généralités

- *Caractère*: symbole représenté par une chaîne de bits
- *Chaîne de caractères*: séquence finie de caractères, normalement terminée par un caractère nul

ASCII

- Représente 128 caractères codés sur 7 bits
- Lettre minuscule mise en majuscule en assignant le $6^{\text{ème}}$ bit de poids faible à 0, par ex. $a = 1100001_2$ et $A = 1000001_2$

ISO 8859-1 (Latin-1)

- Représente 256 caractères codés sur 8 bits
- Caractères 0 à 127: ASCII
- Caractères 128 à 255: lettres accentuées et autres caractères

UTF-8

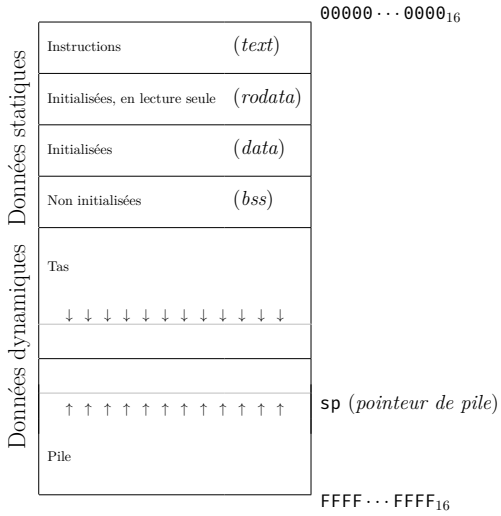
- Représente $> 1\ 000\ 000$ caractères sur 1 à 4 octets
- Caractères 0 à 127: ASCII
- Caractères 128 à 255: ISO 8859-1 mais encodés différemment
- *Format général*:

# bits	Plage de codes		Format binaire des octets			
	Début	Fin	Octet 1	Octet 2	Octet 3	Octet 4
7	000000_{16}	$00007F_{16}$	0*******	—	—	—
11	000080_{16}	$0007FF_{16}$	110*****	10******	—	—
16	000800_{16}	$00FFFF_{16}$	1110****	10******	10******	—
21	010000_{16}	$10FFFF_{16}$	11110***	10******	10******	10******

- *Exemples*:

Car.	Code	Codage
a	1100001_2	01100001_2
é	$000\ 11101001_2$	$11000011\ 10101001_2$
ケ	$00110000\ 10110001_2$	$11100011\ 10000010\ 10110001_2$
𐐷	$00001\ 00100100\ 00001101_2$	$11110000\ 10010010\ 10010000\ 10001101_2$

Disposition de la mémoire.



Tas.

- Contient les données allouées dynamiquement: structures de données, objets, etc.

Pile d'exécution.

- Stocke les données temporaires lors d'appel de sous-prog.
- Données empilées à l'appel et dépilées au retour
- *Pointeur de pile*: `sp` contient l'adresse du sommet de la pile
- *Empiler*: décrémenter `sp` + stocker avec `stp xd, xn, a`
- *Dépiler*: incrémenter `sp` + charger avec `ldp xd, xn, a`

Récursion.

- *Implémentée par*: appels de sous-prog. + usage de la pile
- *Récursion trop profonde*: erreur car la pile est bornée
- *Solution (partielle)*: empiler le moins de données possibles

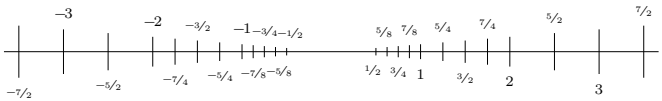
11. Nombres en virgule flottante

Représentation.

- *Nombre en virgule flottante:*

$$\overbrace{\pm}^{\text{signe}} \underbrace{d_0,d_1d_2\cdots d_{n-1}}_{\text{mantisse en base } \beta} \times \underbrace{\beta^{\overbrace{e}^{\text{exposant}}}}_{\text{base}}$$

- *Normalisé*: si $d_0 \neq 0$
- Représente différents ordres de grandeur:



Arithmétique.

- *Addition:* (1) mettre exposants en commun; (2) additionner mantisses; (3) normaliser; (4) arrondir
- *Multiplication:* (1) additionner exposants; (2) multiplier mantisses; (3) normaliser; (4) arrondir

Précision.

- *Approximations de nombres réels:*

(a) arrondir (égalité: dernier chiffre pair): 1,9565 → 1,956

(b) troncation: 1,5416 → 1,541

- *Erreur relative*: $\mathrm{err}(x) \overset{\text{def}}{=} \frac{x-\overline{x}}{x}$ où $\overline{x}$ est l'approximation
- *Borne pour mode (a)*: $|\mathrm{err}(x)| \leq \underbrace{(\beta/2)\cdot\beta^{-n}}_{\varepsilon \text{ machine}}$

Norme IEEE 754.

-

format	signe	exposant	mantisse
simple	1 bit	8 bits	23 bits (+1 bit caché)
double	1 bit	11 bits	52 bits (+1 bit caché)

- *Repr. avec biais:* $1000011010\cdots011 = -1{,}11 \times 2^{13-127}$
- ± 0 ($s0\cdots00\cdots0$); $\pm\infty$ ($s1\cdots10\cdots00$); NaN ($s1\cdots1\,e\,0\cdots01$)

ARMv8.

- *Registres*: `d`$_n$ (64 bits) et `s`$_n$ (32 bits)
- *Instructions*: `ldr`, `str`, `fmov`, `fcmp`, `fadd`, `fmul`, `fsqrt`, etc.

12. Introduction aux entrées/sorties : NES

Architecture.

- *Pas RISC*: possible de manipuler la mémoire directement
- *Processeurs:* proc. principal et proc. d'images (*PPU*)
- *Mémoire principale:* 2Kio de mémoire (générale et pile) + registres d'E/S + programme
- *Mémoire vidéo:* stocke les tuiles et palettes de couleurs

Jeu d'instructions.

- *Registres:* `a` (accumulateur), `x` (index), `y` (index), `s` (pile)
- *Valeurs imm.:* `#` (numérique), `$` (hexadécimal), `%` (binaire)
- *Accès mémoire:* `lda`, `ldx`, `ldy` (chargement d'octet); `sta`, `stx`, `sty` (stockage d'octet); `txa`, `tax`, `tya`, etc. (copie)
- *Arithmétique:* `adc` (addition avec report); `sbc` (soustraction avec emprunt); `inc`, `inx`, `iny`, `dec` (inc/décrémentation)
- *Logique:* `asl` (<< 1), `lsr` (>> 1), `and` ($\wedge$), `ora` ($\vee$), `eor` ($\oplus$)
- *Contrôle:* `cmp`, `cpx`, `cpy` (comparaison); `beq`, `bne` (branch. conditionnel), `jmp` (branch. incond.), `jsr`/`rts` (sous-prog.)

Tuiles.

- *Images:* constituées de tuiles de 8×8 pixels
- *Tuiles:* stockées dans la cartouche, transférées vers le PPU
- *Tuile:* spécifiée par 4 octets (y, i, a, x): position verticale y, numéro de tuile i, attributs a, position horizontale x
- *Attributs:* 8 bits pour réflexions, profondeur et couleurs

Sorties (graphiques).

- L'affichage se fait lors du rafraîchissement vertical
- *Sortie:* stocker tuiles de $\texttt{0X00}_{16}$ à $\texttt{0XFF}_{16}$ en mém. principale
- *Affichage:* transférer au PPU en écrivant `#$0X` à `$4014`

Entrées (manettes).

- *Entrée:* protocole de communication via port de manettes
- *Demande de lecture:* envoyer `#1`, puis `#0`, via `$4016`
- *Lecture:* lire bit de poids faible à `$4016` pour chaque bouton

13. Entrées/sorties

Mécanismes d'entrée/sortie.

- *Attente active:* interrogation constante d'un registre d'état jusqu'à ce qu'un événement se produise (ex. *VBLANK*)
- *Interruption:* signal lancé vers le processeur lorsque'un événement se produit (ex. NES: NMI, RESET, IRQ)

Interruptions.

- *Gestionnaire d'interruption:* sous-routine exécutée afin de traiter une interruption
- *Table d'interruptions:* contient l'adresse des gestionnaires
- *Traitement:* sauvegarder l'état du processeur; appeler le gestionnaire d'interruption; restaurer l'état
- *Priorité:* valeur numérique assignée à une interruption
- *Gestion des priorités:* interruption ignorée si une interruption de priorité $>$ est en cours; gestionnaire en exécution mis en attente si une interruption de priorité $\geq$ est lancée
- *Interruption non masquable:* plus haute priorité; ne peut pas être ignorée (ex. RESET)

Accès direct à la mémoire (DMA).

- *DMA:* permet au processeur d'initier un accès mémoire et de laisser un contrôleur effectuer le transfert de données
- *Sur le NES:* envoi des tuiles `mem[0x0200, 0x02FF]` vers la mémoire de *sprites* via DMA:

```
lda #$02
sta $4014
```

Appels système.

- *Accès E/S:* empêché par le système d'exploitation (sécurité)
- *Appel système:* service offert par le noyau du système d'exploitation; appelé via une interruption logicielle
- *Exemple UNIX + ARMv8:*

code	appel système
64	`write(flux, &chaine, #octets)`
63	`read(flux, &tampon, #octets)`

Flux d'entrée standard $= 0$
Flux de sortie standard $= 1$

```
// Affichage
mov  x8, 64
mov  x0, 1
adr  x1, chaine
mov  x2, 10
svc  0
```