

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
- ▶ Pas concis

Représentation positionnelle

- ▶ Généralisation du système décimal à une base $b \in \mathbb{N}_{\geq 2}$
- ▶ *Systèmes particuliers*: binaire ($b = 2$), octal ($b = 8$), décimal ($b = 10$), hexadécimal ($b = 16$)
- ▶ *Chiffres*: éléments de $\{0, 1, \dots, b-1\}$
- ▶ *Chiffres au-delà de 9*: A = 10, B = 11, ..., F = 15, ...
- ▶ *Valeur de x en base b* : $x_b = x_{n-1} \cdot b^{n-1} + \dots + x_1 \cdot b^1 + x_0 \cdot b^0$
- ▶ *Exemple*: $8B5_{16} = 8 \cdot 16^2 + 11 \cdot 16^1 + 5 \cdot 16^0$
- ▶ Les zéros tout à gauche ne changent rien: $(0 \cdots 0x)_b = x_b$

Conversions

- ▶ b à 10: $x_0 + b \cdot (x_1 + b \cdot (x_2 + b \cdot (\dots + b \cdot x_{n-1})))$
- ▶ 10 à b : diviser à répétition par b et concaténer les restes de droite à gauche, par ex. $6_2 = 110$:
 $6 \div 2 = 3$ reste 0, $3 \div 2 = 1$ reste 1, $1 \div 2 = 0$ reste 1
- ▶ b à b^m : remplacer chaque bloc de taille m par sa valeur en base b^m , par ex. si $b^m = 2^3$: $10110 \rightarrow 26$
- ▶ b^m à b : éclater chaque symbole vers sa représentation de taille m en base b , par ex. si $b^m = 2^3$: $73 \rightarrow 111011$

Addition

- ▶ Se fait comme en base 10: additionner chiffre à chiffre 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*: $(0 \cdots 0x,y0 \cdots 0)_b = (x,y)_b$

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 qui n'est pas un multiple de 2^k — parfois: *interdit*, souvent: *ralentit l'accès*

Processeur

- ▶ *Jeu d'instructions* élémentaires, par ex: $\overbrace{\text{add}}^{\text{code d'opér.}} \overbrace{x10, x11, x12}^{\text{opérandes}}$
- ▶ *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 chaînes de 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} (sous-registres 32 bits)
- ▶ *Usage libre*: x_0-x_7 (arguments) et $x_{19}-x_{28}$ (sauveg. par l'appelé)
- ▶ *Usage semi-libre*: x_9-x_{15} (sauvegardés par l'appelant)

Organisation du code

- ▶ *Ligne*: **étiquette**: opcode opérandes // **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
```

Quelques instructions

<code>mov</code> x_d, v	$x_d \leftarrow v$ où v est regis. ou const.
<code>add</code> x_d, x_n, v	$x_d \leftarrow x_n + v$ où v est regis. ou const.
<code>mul</code> x_d, x_n, x_m	$x_d \leftarrow x_n \cdot x_m$
<code>udiv</code> x_d, x_n, x_m	$x_d \leftarrow x_n \div x_m$

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`

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 via C)

- ▶ *Affichage*: `printf(&format, val1, val2, ...)`
- ▶ *Lecture*: `scanf(&format, &var1, &var2, ...)`
- ▶ *Format nombres*: int32 (%d), uint32 (%u), uint32-hex (%X), 64 bits via préfixe l, par ex. int64 (%ld)

4. Accès aux données

Adresses

- **Numérique**: entier non négatif, souvent en hexadécimal
- **Symbolique**: chaîne représentant une adresse à déterminer

Modes d'adressage

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

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

Accès mémoire sur ARMv8

- **Chargement et stockage**:

# octets	chargement	stockage
1	<code>ldrb wd, a</code>	<code>strb wd, a</code>
2	<code>ldrh wd, a</code>	<code>strh wd, a</code>
4	<code>ldr wd, a</code>	<code>str wd, a</code>
8	<code>ldr xd, a</code>	<code>str xd, a</code>

- **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
    
```

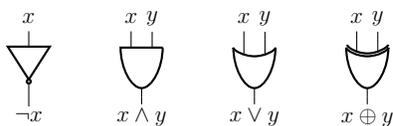
Assemblage

- **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. Circuits logiques

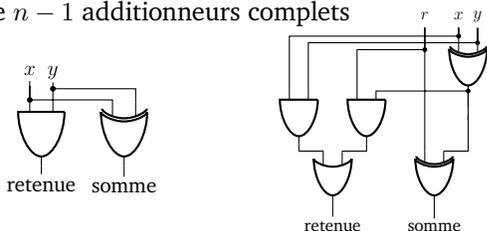
Circuits

- « Blocs » de base constitués de portes logiques qui permettent d'implémenter l'ordinateur:



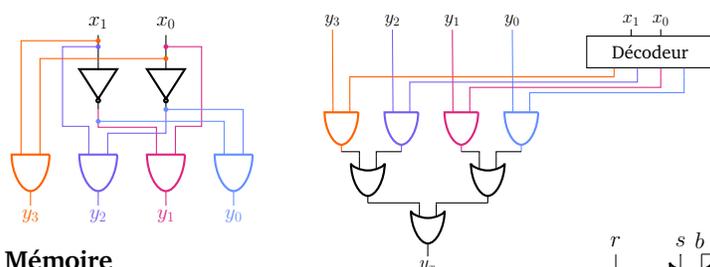
Arithmétique

- **Demi-additionneur**: somme de deux bits
- **Additionneur complet**: somme de deux bits et d'une retenue
- **Addition**: somme sur n bits avec un demi-additionneur et une cascade de $n - 1$ additionneurs complets



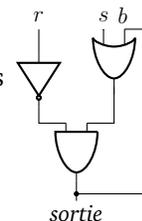
Décodage

- **Décodeur**: sur entrée x , sortie: $y_x = 1$ et $y_j = 0$ pour $j \neq x$
- **Multiplexeur**: sur entrée x , sélectionne le bit y_x
- **Instructions**: décodables/exécutables à l'aide de tels circuits



Mémoire

- **Circuits séquentiels**: peuvent mémoriser des bits
- **Verrou**: stocke un bit b , remise à 0 avec r , et mise à 1 avec s



6. Nombres entiers

Représentation des entiers signés

- **Compl. à 2**: $\text{val}(x_{n-1} \dots x_1 x_0) = -x_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} x_i \cdot 2^i$

bits	000	001	010	011	100	101	110	111
valeur	0	1	2	3	-4	-3	-2	-1

- **Représentables sur n bits**: $[-2^{n-1}, 2^{n-1} - 1]$
- **Bit de signe**: négatif ssi bit de gauche = 1
- **Ajout de bits**: répéter bit de signe à gauche: `101` → `1...101`
- **Changement de signe**: `010` $\xrightarrow{\text{complément}}$ `101` $\xrightarrow{+1}$ `110`

Opérations arithmétiques

- **Addition**: comme les entiers non signés
- **Soustraction**: addition/changement de signe: $a - b = a + (-b)$
- **Report**: lors d'une retenue sur la somme des bits de poids fort
- **Débordement**: lorsque le résultat ne peut pas être représenté

- **Multiplication et division non signées**: comme en base 10:

$$\begin{array}{r}
 \times \quad 101 \quad (5) \\
 \quad \quad 11 \quad (3) \\
 \hline
 \quad 101 \\
 \quad 110 \\
 \hline
 1111 \quad (15)
 \end{array}
 \qquad
 \begin{array}{r}
 10011 \quad | \quad 11 \\
 - \quad 11 \quad \quad 00110 \\
 \hline
 \quad 111 \\
 - \quad 11 \\
 \hline
 \quad \quad 1
 \end{array}$$

- **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), V (débordement)
- **Codes modifiés par**: `cmp`, `adds`, `subs`, `negs`, `adcs`, `sbcs`
- **Comparaison**: codes mis à jour via soustraction bidon
- **Accès aux codes**: avec `b.condition` etiq
- **Accès au report**: « `adc rd, rn, rm` » $\equiv r_d \leftarrow r_n + r_m + C$

7. Tableaux

Généralités

- **Tableau**: collection d'éléments identifiés par des indices
- **Éléments**: tous de même taille, contigus en mémoire
- **Indice**: d -uplet i où $d \geq 1$ est la dimension
- **Bornes**: $0 \leq i_j < n_j$ pour chaque dimension j
- **Taille**: $n_0 \cdot n_1 \cdot \dots \cdot n_{d-1}$ éléments
- **Types**: le type des éléments est implicite
- **Exemples de tableau 1D et tableau 2D**:

0	01010101	(0,0)	2
1	11110000	(0,1)	33
2	01101101	(1,0)	65535
3	11111111	(1,1)	73
4	11110101	(2,0)	9000
		(2,1)	255

$n_0 = 5$
 5 éléments

$n_0 = 3, n_1 = 2$
 6 éléments

Calcul d'adresse

- **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 d'un élément correspond à:

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

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

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 // six demi-mots
```

- **Accès**: avec **str**/**ldr** (ou variantes) + modes d'adressage

8. 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 avec 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
- Registres partagés par programme et sous-programmes
- **Arguments**: passés par valeur ou adresse dans x_0 - 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}

9. Valeurs booléennes et chaînes de 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	and x19, x20, x21	orr x19, x20, x21	eor x19, x20, x21
\neg 0 ... 1	\wedge 0 ... 1	\vee 0 ... 1	\oplus 0 ... 1
1 ... 0	1 ... 0	1 ... 1	1 ... 0
0 ... 1	0 ... 0	0 ... 0	0 ... 1
1 ... 0	0 ... 0	1 ... 1	1 ... 0

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

Décalages logiques et arithmétiques

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

$11000101 \xrightarrow{3 \text{ bits vers la gauche}} 00101000$ **lsl** xd, xn, 3
 $11000101 \xrightarrow{3 \text{ bits vers la droite}} 00011000$ **lsr** xd, xn, 3

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

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

- **Multipliation/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 ré-insérés de l'autre côté:

$11000101 \xrightarrow{3 \text{ bits vers la gauche}} 00101110$ n'existe pas sur ARMv8
 $11000101 \xrightarrow{3 \text{ bits vers la droite}} 10111000$ **ror** xd, xn, 3

Masquage

- Permet d'isoler certains bits à manipuler:

sélection	$r \wedge m$	met à 0 les bits de r non spécifiés par m
activation	$r \vee m$	met à 1 les bits de r spécifiés par m
désactivation	$r \wedge \neg m$	met à 0 les bits de r spécifiés par m
basculement	$r \oplus m$	inverse les bits de r spécifiés par m

10. Chaînes de caractères

Généralités

- ▶ *Caractère*: symbole représenté par une chaîne de bits
- ▶ *Chaîne de caractères*: suite 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^{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 codés différemment
- ▶ *Format général*:

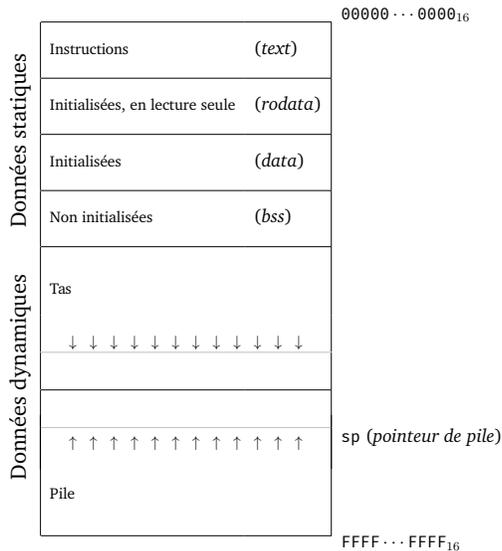
# bits	plage de codes		format binaire des octets			
	Début	Fin	Octet 1	Octet 2	Octet 3	Octet 4
7	000000 ₁₆	00007F ₁₆	0*****	—	—	—
11	000080 ₁₆	0007FF ₁₆	110*****	10*****	—	—
16	000800 ₁₆	00FFFF ₁₆	1110****	10*****	10*****	—
21	010000 ₁₆	10FFFF ₁₆	11110***	10*****	10*****	10*****

Exemples:

car.	code	codage
a	1100001 ₂	01100001 ₂
é	000 11101001 ₂	11000011 10101001 ₂
ヶ	00110000 10110001 ₂	11100011 10000010 10110001 ₂
𐀀	00001 00100100 00001101 ₂	11110000 10010010 10010000 10001101 ₂

11. Sous-programmes et mémoire

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

12. Nombres en virgule flottante

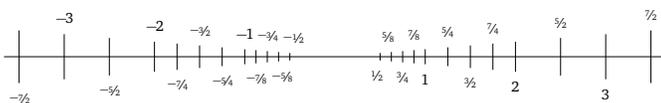
Représentation.

- ▶ *Nombre en virgule flottante*:

$$\underbrace{\pm}_{\text{signe}} \underbrace{d_0, d_1 d_2 \dots d_{n-1}}_{\text{mantisse en base } \beta} \times \underbrace{\beta^e}_{\text{base}}^{\text{exposant}}$$

- ▶ *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*:

- arrondir (égalité: dernier chiffre pair): 1,9565 → 1,956
- troncation: 1,5416 → 1,541

- ▶ *Erreur relative*: $\text{err}(x) \stackrel{\text{def}}{=} \frac{x - \bar{x}}{x}$ où \bar{x} est l'approximation

- ▶ *Borne pour mode (a)*: $|\text{err}(x)| \leq \underbrace{(\beta/2) \cdot \beta^{-n}}_{\varepsilon \text{ machine}}$

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*: $1000011011100 \dots 0 = -1,11 \times 2^{13-127}$

- ▶ ± 0 ($s0 \dots 00 \dots 0$); $\pm \infty$ ($s1 \dots 10 \dots 00$); NaN ($s1 \dots 1e0 \dots 01$)

ARMv8.

- ▶ *Registres*: d_n (64 bits) et s_n (32 bits)
- ▶ *Instructions*: **ldr**, **str**, **fmov**, **fcmp**, **fadd**, **fmul**, **fsqrt**, etc.

13. Introduction aux entrées/sorties : NES

Architecture.

- ▶ *Pas RISC*: possible de manipuler la mémoire directement
- ▶ *Processeurs*: proc. principal + proc. d'images (PPU)
- ▶ *Mémoire principale*: primaire + 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/décrément)ation)
- ▶ *Logique*: **asl** (<< 1), **lsr** (>> 1), **and** (^), **ora** (v), **eor** (⊕)
- ▶ *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 $0X00_{16}$ à $0XFF_{16}$ 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

14. Entrées/sorties

Mécanismes d'entrée/sortie.

- ▶ *Attente active*: interrogation continue d'un registre d'état jusqu'à un événement (ex. *VBLANK*)
- ▶ *Interruption*: signal lancé vers le processeur lors d'un événement (ex. NMI, RESET, IRQ)

Interruptions.

- ▶ *Gestionnaire*: sous-routine qui traite une interruption
- ▶ *Table d'interruptions*: contient l'adresse des gestionnaires
- ▶ *Traitement*: sauvegarder l'état du processeur; appeler le gestionnaire; 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
- ▶ *Non masquable*: top priorité, ne peut pas ignorer (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 $\text{mem}[0x0200, 0x02FF]$ 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
- ▶ *Exemples UNIX + ARMv8*:

code	appel système
64	write(flux, chaîne, #octets)
63	read(flux, tampon, #octets)

Flux d'entrée standard = 0

Flux de sortie standard = 1

```
// Afficher chaîne
mov x8, 64
mov x0, 1
adr x1, chaîne
mov x2, 10
svc 0
```