

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:

<code>mvn x19, x20</code>	<code>and x19, x20, x21</code>	<code>orr x19, x20, x21</code>	<code>eor x19, x20, x21</code>
$\neg \dots \neg \neg \neg$	$0 \dots 1 \ 0 \ 1$	$0 \dots 1 \ 0 \ 1$	$0 \dots 1 \ 0 \ 1$
$0 \dots 1 \ 0 \ 1$	$\wedge \dots \wedge \wedge \wedge$	$\vee \dots \vee \vee \vee$	$\oplus \dots \oplus \oplus \oplus$
$1 \dots 0 \ 1 \ 0$	$1 \dots 1 \ 0 \ 0$	$1 \dots 1 \ 0 \ 0$	$1 \dots 1 \ 0 \ 0$
	$0 \dots 1 \ 0 \ 0$	$1 \dots 1 \ 0 \ 1$	$1 \dots 0 \ 0 \ 1$

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

Décalages logiques et arithmétiques

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

$11000101 \xrightarrow{3 \text{ bits vers la gauche}} 00101000$ `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`

- 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 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*:

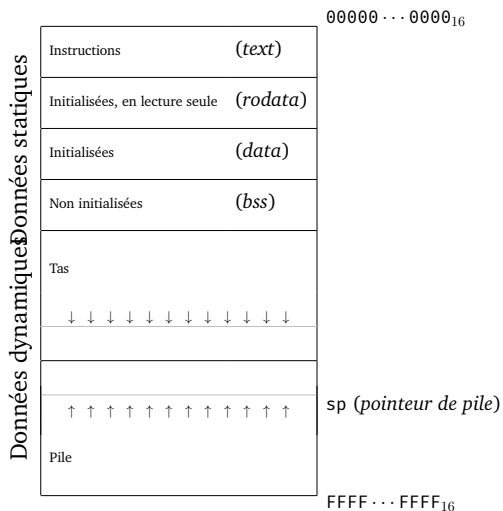
# bits	plage de codes ¹		format binaire des octets			
	début	fin	octet 1	octet 2	octet 3	octet 4
7	000000 ₁₆	00007F ₁₆	0*****	—	—	—
11	000080 ₁₆	0007FF ₁₆	110*****	10*****	—	—
16	000800 ₁₆	00FFFF ₁₆	1110****	10*****	10*****	—
21	010000 ₁₆	10FFFF ₁₆	11110***	10*****	10*****	10*****

- *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

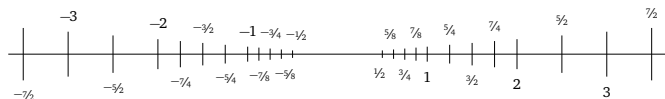
Représentation.

► *Nombre en virgule flottante:*

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

► *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:* $\text{err}(x) := \frac{x - \bar{x}}{x}$ 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...0 = -1,11 × 2¹³⁻¹²⁷

► ±0 (s0...00...0); ±∞ (s1...10...00); NaN (s1...1e0...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** ($\wedge$), **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 $0X00_{16}$ 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 `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	<code>write(flux, chaîne, #octets)</code>	<code>// Afficher chaîne</code> <code>mov x8, 64</code> <code>mov x0, 1</code> <code>adr x1, chaîne</code> <code>mov x2, 10</code> <code>svc 0</code>
63	<code>read(flux, tampon, #octets)</code>	
Flux d'entrée standard = 0		
Flux de sortie standard = 1		