IFT209 – Programmation système

Université de Sherbrooke

Examen final

Enseignant: Michael Blondin
Date: jeudi 27 avril 2023
Durée: 3 heures

Directives:

- Répondez aux questions dans le **cahier de réponses**, pas sur ce questionnaire;
- **Une feuille (recto verso)** de notes au format $8\frac{1}{2}'' \times 11''$ est permise, et les fiches en **annexe**;
- **Aucun matériel additionnel** (notes de cours, examens antérieurs, etc.) n'est permis;
- **Aucun appareil électronique** (calculatrice, téléphone, tablette, ordinateur, etc.) n'est permis;
- Donez **une seule réponse** par sous-question;
- L'examen comporte **5 questions** sur **7 pages** valant un total de **50 points**;
- La correction se base sur la **clarté**, l'**exactitude** et la **concision** de vos réponses, ainsi que sur la **justification** pour les questions qui en requièrent une;
- À moins d'avis contraire, le langage d'assemblage utilisé est celui de l'architecture **ARMv8** tel qu'utilisé en classe; un sommaire est présenté à l'**annexe B**;
- La question 5 utilise le langage d'assemblage du **NES** tel qu'utilisé en classe; un sommaire est présenté à l'**annexe C**.

Question 1: valeurs booléennes et chaînes de bits

(a) Considérons une chaîne de cinq bits $b_4\,b_3\,b_2\,b_1\,b_0$. Chacun des trois schémas ci-dessous représente une opération de masquage. Vous devez trouver des opérateurs et des masques qui mènent à chacun des résultats. Vous devez donc remplacer chaque occurrence de (?) par un opérateur logique parmi $\wedge$, $\vee$ ou $\oplus$, et chaque occurrence de [?] par 0 ou 1. Dans chaque cas, l'opérateur est appliqué bit à bit. 3 pts

opération A

	b_4	b_3	b_2	b_1	b_0
(?)	[?]	[?]	[?]	[?]	[?]
	0	b_3	0	b_1	b_0

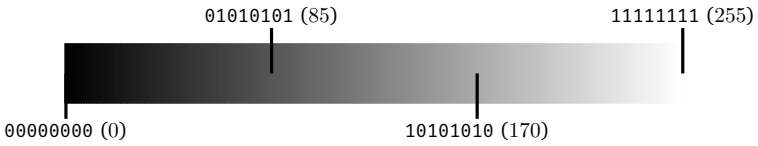
opération B

	b_4	b_3	b_2	b_1	b_0
(?)	[?]	[?]	[?]	[?]	[?]
	1	1	1	1	1

opération C

	b_4	b_3	b_2	b_1	b_0
(?)	[?]	[?]	[?]	[?]	[?]
	b_4	$\neg b_3$	$\neg b_2$	b_1	b_0

(b) Considérons la représentation de pixel, en niveaux de gris, stocké sur n bits, où 0 représente noir et $2^n - 1$ représente blanc. Par exemple, avec $n = 2$, les niveaux possibles sont noir `00`, gris foncé `01`, gris pâle `10` et blanc `11`. Avec $n = 8$, ces 256 niveaux de gris peuvent être représentés: 5 pts



Supposons que $\texttt{w}_{19}$ contienne un pixel au format 2 bits. Nous cherchons à convertir sa représentation au format 8 bits dans $\texttt{w}_{20}$ comme suit:

w_{19}	w_{20}
0 ··· 0 00000000	0 ··· 0 00000000
0 ··· 0 00000001	0 ··· 0 01010101
0 ··· 0 00000010	0 ··· 0 10101010
0 ··· 0 00000011	0 ··· 0 11111111

Complétez les portions trouées de ces deux programmes afin qu'ils accomplissent chacun cette tâche:

programme A

```
lsl     w20, ???, 2
orr     w20, w20, w19
lsl     w19, w20, ???
orr     w20, ???, w20
```

programme B

```
ror     w20, w19, 1
asr     w20, w20, 31
and     w20, w20, 0x55
ror     w19, w19, 2
asr     w19, w19, 31
and     w19, w19, ???
orr     w20, w19, w20
```

Question 2: chaînes de caractères

Rappelons le format du codage UTF-8 tel que présenté dans les notes de cours:

# bits	plage de codes début	plage de codes fin	format binaire des octets 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******`

(a) Le pictogramme « ❤ » et l'émoji « 🤯 » sont des caractères représentés, respectivement, par les codages UTF-8 « `11100010 10011101 10100100` » et « `11110000 10011111 10011001 10001010` ». Donnez le code numérique Unicode associé à chacun de ces deux caractères (en hexadécimal). 2 pts

(b) Le code numérique Unicode de la lettre accentuée « ê » est `0xEA`. Donnez son codage UTF-8. 2 pts

(c) Rappelons que le codage ISO 8859-1 (Latin-1) permet de représenter les caractères dont le code numérique Unicode appartient à la plage `0x00` à `0xFF`. Combien de caractères de cette chaîne de caractères UTF-8 sont représentables en Latin-1? Justifiez. 2 pts

```
11010111 10000110  11000011 10101010  01111010  11100011 10000011 10000001  00000000
```

(d) Rappelons que les lettres accentuées ont un code numérique dans la plage `0xC0` à `0xFF`, et que le sixième bit de poids faible du code numérique vaut 1 si et seulement si la lettre est en minuscule. Par exemple, le code numérique de « ê » est 11101010_2 (`0xEA`) et celui de « Ê » est 11001010_2 (`0xCA`). 6 pts

Écrivez un sous-programme qui accomplit cette tâche:

> ENTRÉE: adresse d'une chaîne de caractères s sous codage UTF-8, dont chaque caractère est d'au plus deux octets (premier et seul paramètre)
>
> RETOUR: quantité de lettres accentuées en minuscule dans s

En particulier, votre sous-programme devrait retourner 2 sur la chaîne de caractères « `Été à Sherbrooke` ». Vous avez accès aux macros `SAVE` et `RESTORE`.

Remarque: pour simplifier la question, vous avez la promesse qu'il n'y aura aucun caractère de trois ou quatre octets.

Question 3: sous-programmes et mémoire

Considérons cet algorithme qui détermine récursivement si un tableau trié t contient un certain élément x:

Entrée : tableau t d'entiers signés de 64 bits, triés en ordre croissant,
entier signé de 64 bits n qui représente le nombre d'éléments du tableau,
entier signé de 64 bits x
Retour : un indice m tel que $t[m] = x$ s'il en existe un, -1 sinon

```
fouille(t, n, x):
    retourner fouille_aux(t, x, 0, n − 1)          // lancer une fouille dichotomique

fouille_aux(t, x, i, j):
    si i > j alors
        retourner −1                                // élément pas dans le tableau
    sinon
        m ← (i + j) ÷ 2
        si x < t[m] alors
            retourner fouille_aux(t, x, i, m − 1)   // l'élément est à gauche?
        sinon si t[m] < x alors
            retourner fouille_aux(t, x, m + 1, j)   // l'élément est à droite?
        sinon
            retourner m                             // t[m] = x, élément trouvé
```

(a) Implémentez l'algorithme en complétant les sous-programmes « `fouille:` » et « `fouille_aux:` ». 5 pts

```
fouille:
    /* à compléter au besoin */
    SAVE
    /* à compléter */
    RESTORE
    /* à compléter au besoin */

fouille_aux:
    /* à compléter au besoin */
    SAVE
    /* à compléter */
    RESTORE
    /* à compléter au besoin */
```

Remarque: ne modifiez pas l'algorithme pour le rendre itératif, il doit demeurer récursif.

(b) Expliquez ce qui se produit si on retire `SAVE` et `RESTORE` de votre code et qu'on l'appelle avec l'entrée $t = [10]$, $n = 1$ et $x = 15$. 2,5 pts

(c) Remplacez `SAVE` et `RESTORE` par votre propre code afin de sauvegarder uniquement le contenu des registres nécessaires, par ex. si vous n'utilisez pas x_{28}, alors il ne devrait pas être sauvegardé. 2,5 pts

Question 4: nombres en virgule flottante

(a) Considérons le système de nombres en virgule flottante où la base est $\beta = 2$, la mantisse possède $n = 4$ bits, et l'exposant varie entre $e_{\text{min}} = -5$ et $e_{\text{max}} = 5$. Effectuez la multiplication suivante: 5 pts

$$(1{,}110 \times 2^{-2}) \cdot (1{,}100 \times 2^{2}).$$

Votre résultat doit être *normalisé* et approximé par *arrondi avec bris d'égalité vers chiffre pair* (l'arrondi vu en classe). Laissez une trace de votre démarche.

(b) Rappelons que la norme IEEE 754 représente un nombre en virgule flottante ainsi en binaire:

format	signe	exposant	mantisse
simple	1 bit	8 bits (biais de 127)	23 bits (+1 bit caché)
double	1 bit	11 bits (biais de 1023)	52 bits (+1 bit caché)

Par exemple, le nombre $1{,}5$ est codé sous précision simple par « `0 01111111 10000000000000000000000` ».

(i) Donnez le codage du nombre $13{,}625$ au format simple précision. 2,5 pts

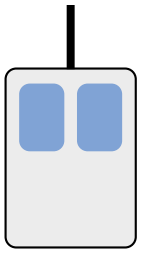
(ii) Vrai ou faux: afin de convertir un nombre en virgule flottante du format simple vers le format double, on doit ajouter 32 zéros à sa gauche. Justifiez. 2,5 pts

Question 5: entrées/sorties

Rappelons les trois types d'interruptions du NES, du plus au moins prioritaire:

0. RESET:	*lancée au démarrage de la console (bouton « POWER » enfoncé) ou lorsque la console est redémarrée (bouton « RESET » appuyé)*
1. NMI:	*lancée lors de l'intervalle de rafraîchissement vertical (VBLANK)*
2. IRQ:	*ne possède pas d'usage particulier, mais peut, par exemple, être lancée par une puce électronique*

(a) Le *NES Mouse209* est un périphérique fictif du NES. Il s'agit d'une souris munie de deux boutons: 7 pts



Le *NES Mouse209* se connecte dans le second port et fonctionne comme suit:

— Pour demander l'état, on envoie 1, puis 0, à l'adresse 4017_{16} liée au port de communication;

— Ensuite, chaque octet $b_7 \cdots b_0$ lu à l'adresse 4017_{16} donne de l'information comme suit:

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
Première lecture:	—	—	—	—	—	—	—	—
Deuxième lecture:	bouton droit appuyé?	bouton gauche appuyé?	—	—	—	—	—	—
Troisième lecture:	haut = 1, bas = 0	déplacement vertical (entier non signé de 7 bits)						
Quatrième lecture:	gauche = 1, droite = 0	déplacement horizontal (entier non signé de 7 bits)						

Complétez le code ci-dessous afin que le curseur se déplace horizontalement selon l'information fournie par la souris. Vous n'avez pas à gérer les débordements hors de l'écran; le curseur peut réapparaître de l'autre côté. Vous n'avez pas à gérer le déplacement vertical.

Le curseur doit également changer d'apparence. Il est représenté graphiquement par la tuile 129  lorsque le bouton gauche est appuyé, et par la tuile 128  sinon.

```
posX:       .rs 1               ; Position horizontale du curseur
posY:       .rs 1               ; Position verticale du curseur
btnGauche:  .rs 1               ; Bouton gauche appuyé?
btnDroite:  .rs 1               ; Bouton droite appuyé?
                                ;
point_entree:                   ; point_entree() {
    lda     #%00000000          ;   Désactiver temporairement les interruptions NMI
    sta     $2000               ;
                                ;
    ldx     #$FF                ;
    txs                         ;   Initialiser la pile d'exécution
                                ;
    jsr     init                ;   Initialiser les variables
                                ;
    lda     #%10011000          ;   Réactiver les interruptions NMI et
    sta     $2000               ;               choisir les tables de tuiles
    lda     #%00010000          ;
    sta     $2001               ;   Activer les tuiles
                                ;
boucle:                         ;   while (true) { }
    jmp     boucle              ; }
                                ;
gestion:                        ; gestion()
                                ; {
    SAVE                        ;   /* empiler a, x et y */
    jsr     deplacer            ;   deplacer()
    jsr     afficher            ;   afficher()
    RESTORE                     ;   /* dépiler a, x et y */
    rti                         ; }
                                ;
init:                           ; init()
    lda     #100                ; {
    sta     posX                ;   posX = 100
    sta     posY                ;   posY = 100
    lda     #0                  ;
    sta     btnGauche           ;   btnGauche = 0
    sta     btnDroite           ;   btnDroite = 0
    rts                         ; }
                                ;
```

```
;; Déplacer curseur selon       ;
;;       l'état du NES Mouse209 ;
deplacer:                       ; deplacer()
                                ; {
    ;; À COMPLÉTER              ;
                                ; }
                                ;
;; Afficher le curseur          ;
afficher:                       ; afficher()
                                ; {
    ;; À COMPLÉTER              ;   ; Ordre: pos. verticale, identifiant, attributs,
                                ;                                        pos. horizontale
    lda     #$05                ;   Envoi par accès direct à la mémoire (DMA)
    sta     $4014               ;                      de la plage $0500 à $05FF
                                ;
    ;; À COMPLÉTER              ;
                                ; }
                                ;
;; Segment des interruptions ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    .word    À COMPLÉTER        ; NMI
    .word    À COMPLÉTER        ; RESET
    .word    À COMPLÉTER        ; IRQ
```

(b) Rappelons que l'intervalle de rafraîchissement vertical (*VBLANK*) se produit à intervalle régulier. On peut en être averti par une interruption (NMI) ou en inspectant le bit de poids fort à l'adresse 2002_{16}. Considérons cette légère modification du code précédent: 3 pts

```
;; COMME AVANT ICI              ;   ...
                                ;
boucle:                         ;   while (true) {
    jsr     deplacer            ;     deplacer()   // l'appel a été déplacé ici <---
    jmp     boucle              ;   }                                              |
                                ; }                                                |
                                ;                                                  |
gestion:                        ; gestion()                                        |
                                ; {                                                |
    SAVE                        ;   /* empiler a, x et y */   ----------------------
    jsr     afficher            ;   afficher()
    RESTORE                     ;   /* dépiler a, x et y */
    rti                         ; }
                                ;
;; COMME AVANT ICI              ; ...
```

Le déplacement du curseur se fait maintenant trop rapidement par rapport à l'affichage. Expliquez comment ralentir la lecture de l'état de la souris afin qu'elle se fasse à la même cadence que l'affichage.

Annexe A:

Fiches récapitulatives

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`

¬	…	¬	¬	¬
0	…	1	0	1
1	…	**0**	**1**	**0**

`and x19, x20, x21`

0	…	1	0	1
∧	…	∧	∧	∧
1	…	1	0	0
0	…	**1**	**0**	**0**

`orr x19, x20, x21`

0	…	1	0	1
∨	…	∨	∨	∨
1	…	1	0	0
1	…	**1**	**0**	**1**

`eor x19, x20, x21`

0	…	1	0	1
⊕	…	⊕	⊕	⊕
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 **`lsl xd, xn, 3`**

11000101 $\xrightarrow{\text{3 bits vers la droite}}$ 00011000 **`lsr 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:

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 $6^{\text{ème}}$ 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 codés différemment
- *Format général*:

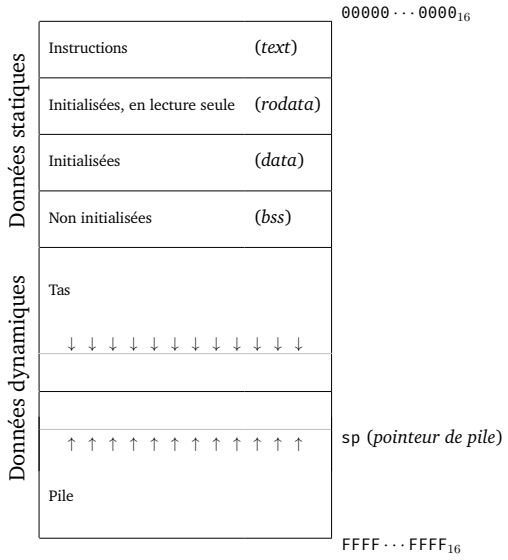
# bits	plage de codes [1] début	plage de codes [1] fin	format binaire des octets octet 1	octet 2	octet 3	octet 4
7	000000_{16}	$00007F_{16}$	0*******	—	—	—
11	000080_{16}	$0007FF_{16}$	110*****	10******	—	—
16	000800_{16}	$00FFFF_{16}$	1110****	10******	10******	—
21	010000_{16}	$10FFFF_{16}$	11110***	10******	10******	10******

- *Exemples*:

car.	code	codage
a	1100001_2	01100001_2
é	$000\ 11101001_2$	$11000011\ 10101001_2$
ケ	$00110000\ 10110001_2$	$11100011\ 10000010\ 10110001_2$
𐍈	$00001\ 00100100\ 00001101_2$	$11110000\ 10010010\ 10010000\ 10001101_2$

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

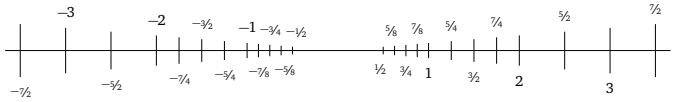
12. Nombres en virgule flottante

Représentation.

- *Nombre en virgule flottante:*

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

- *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:*
  (a) arrondir (égalité: dernier chiffre pair): $1{,}9565 \to 1{,}956$
  (b) troncation: $1{,}5416 \to 1{,}541$
- *Erreur relative*: $\text{err}(x) := \frac{x-\overline{x}}{x}$ où $\overline{x}$ 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:* $1000011011100\cdots0 = -1{,}11 \times 2^{13-127}$
- ± 0 ($s0\cdots00\cdots0$); $\pm\infty$ ($s1\cdots10\cdots00$); NaN ($s1\cdots1\,e\,0\cdots01$)

ARMv8.

- *Registres*: $\mathtt{d}_n$ (64 bits) et $\mathtt{s}_n$ (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), `$` (hexadécimal), `%` (binaire)
- *Accès mémoire:* `lda`, `ldx`, `ldy` (chargement d'octet); `sta`, `stx`, `sty` (stockage d'octet); `txa`, `tax`, `tya`, etc. (copie)
- *Arithmétique:* `adc` (addition avec report); `sbc` (soustraction avec emprunt); `inc`, `inx`, `iny`, `dec` (inc/décrémentation)
- *Logique:* `asl` (<< 1), `lsr` (>> 1), `and` ($\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 $0\text{X}00_{16}$ à $0\text{X}\text{FF}_{16}$ 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	`write(flux, chaine, #octets)`
63	`read(flux, tampon, #octets)`

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

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

Annexe B:

Sommaire de l’architecture ARMv8

Registres.

- Chaque registre $\mathtt{x}_n$ possède 64 bits: $b_{63}b_{62}\cdots b_1b_0$
- Notation: $\mathtt{x}_n\langle i\rangle \coloneqq b_i$, $\mathtt{x}_n\langle i,j\rangle \coloneqq b_ib_{i-1}\cdots b_j$, $\mathtt{r}_n$ réfère au registre $\mathtt{x}_n$ ou $\mathtt{w}_n$
- Chaque sous-registre $\mathtt{w}_n$ possède 32 bits et correspond à $\mathtt{x}_n\langle 31,0\rangle$
- Le compteur d'instruction `pc` n'est pas accessible
- Conventions:

Registres	Nom	Utilisation
$\mathtt{x}_0-\mathtt{x}_7$	—	registres d'arguments et de retour de sous-programmes
$\mathtt{x}_8$	`xr`	registre pour retourner l'adresse d'une structure
$\mathtt{x}_9-\mathtt{x}_{15}$	—	registres temporaires sauvegardés par l'appelant
$\mathtt{x}_{16}-\mathtt{x}_{17}$	$\mathtt{ip}_0-\mathtt{ip}_1$	registres temporaires intra-procéduraux
$\mathtt{x}_{18}$	`pr`	registre temporaire pouvant être réservé par le système
$\mathtt{x}_{19}-\mathtt{x}_{28}$	—	registres temporaires sauvegardés par l'appelé
$\mathtt{x}_{29}$	`fp`	pointeur vers l'ancien sommet de pile (*frame pointer*)
$\mathtt{x}_{30}$	`lr`	registre d'adresse de retour (*link register*)
$\mathtt{x}_{zr}$	`sp`	registre contenant la valeur 0, ou pointeur de pile (*stack pointer*)

Arithmétique (entiers).

- Les codes de condition sont modifiés par `cmp`, `adds`, `adcs`, `subs`, `sbcs` et `negs`
- À cette différence près, `adds`, `adcs`, `subs`, `sbcs` et `negs` se comportent respectivement comme `add`, `adc`, `sub`, `sbc` et `neg`
- Instructions, où i est une valeur immédiate de 12 bits et j est une valeur immédiate de 6 bits:

Code d'op.	Syntaxe	Effet	Exemple
`cmp`	`cmp rd, rm`	compare $\mathtt{r}_d$ et $\mathtt{r}_m$	`cmp x19, x21`
	`cmp rd, i`	compare $\mathtt{r}_d$ et i	`cmp x19, 42`
	`cmp rd, rm, decal j`	compare $\mathtt{r}_d$ et $\mathtt{r}_m$ *decal* j	`cmp x19, x21, lsl 1`
`add`	`add rd, rn, rm`	$\mathtt{r}_d \leftarrow \mathtt{r}_n + \mathtt{r}_m$	`add x19, x20, x21`
	`add rd, rn, i`	$\mathtt{r}_d \leftarrow \mathtt{r}_n + i$	`add x19, x20, 42`
	`add rd, rn, rm, decal j`	$\mathtt{r}_d \leftarrow \mathtt{r}_n + (\mathtt{r}_m$ *decal* $j)$	`add x19, x20, x21, lsl 1`
`adc`	`adc rd, rn, rm`	$\mathtt{r}_d \leftarrow \mathtt{r}_n + \mathtt{r}_m + \mathtt{C}$	`adc x19, x20, x21`
`sub`	`sub rd, rn, rm`	$\mathtt{r}_d \leftarrow \mathtt{r}_n - \mathtt{r}_m$	`sub x19, x20, x21`
	`sub rd, rn, i`	$\mathtt{r}_d \leftarrow \mathtt{r}_n - i$	`sub x19, x20, 42`
	`sub rd, rn, rm, decal j`	$\mathtt{r}_d \leftarrow \mathtt{r}_n - (\mathtt{r}_m$ *decal* $j)$	`sub x19, x20, x21, lsl 1`
`sbc`	`sbc rd, rn, rm`	$\mathtt{r}_d \leftarrow \mathtt{r}_n - \mathtt{r}_m - 1 + \mathtt{C}$	`sbc x19, x20, x21`
`neg`	`neg rd, rm`	$\mathtt{r}_d \leftarrow -\mathtt{r}_m$	`neg x19, x21`
	`neg rd, rm, decal j`	$\mathtt{r}_d \leftarrow -(\mathtt{r}_m$ *decal* $j)$	`neg x19, x21, lsl 1`
`mul`	`mul rd, rn, rm`	$\mathtt{r}_d \leftarrow \mathtt{r}_n \cdot \mathtt{r}_m$	`mul x19, x20, x21`
`udiv`	`udiv rd, rn, rm`	$\mathtt{r}_d \leftarrow \mathtt{r}_n \div \mathtt{r}_m$ (non signé)	`udiv x19, x20, x21`
`sdiv`	`sdiv rd, rn, rm`	$\mathtt{r}_d \leftarrow \mathtt{r}_n \div \mathtt{r}_m$ (signé)	`sdiv x19, x20, x21`
`madd`	`madd rd, rn, rm, ra`	$\mathtt{r}_d \leftarrow \mathtt{r}_a + (\mathtt{r}_n \cdot \mathtt{r}_m)$	`madd x19, x20, x21, x22`
`msub`	`msub rd, rn, rm, ra`	$\mathtt{r}_d \leftarrow \mathtt{r}_a - (\mathtt{r}_n \cdot \mathtt{r}_m)$	`msub x19, x20, x21, x22`

Accès mémoire.

- `ldrsw`, `ldrsh` et `ldrsb` se comportent respectivement comme `ldr` (4 octets), `ldrh` et `ldrb` à l'exception du fait qu'ils effectuent un chargement dans $\mathtt{x_d}$ où les bits excédentaires sont le bit de signe de la donnée chargée, plutôt que des zéros
- Instructions, où a est une adresse et $\mathrm{mem}_b[a]$ réfère aux b octets à l'adresse a de la mémoire principale:

Code d'op.	Syntaxe	Effet	Exemple
`mov`	`mov rd, rm`	$\mathtt{r_d} \leftarrow \mathtt{r_m}$	`mov x19, x21`
	`mov rd, i`	$\mathtt{r_d} \leftarrow i$	`mov x19, 42`
`ldr`	`ldr xd, a`	charge 8 octets: $\mathtt{x_d}\langle 63,0\rangle \leftarrow \mathrm{mem}_8[a]$	`ldr x19, [x20]`
	`ldr wd, a`	charge 4 octets: $\mathtt{x_d}\langle 31,0\rangle \leftarrow \mathrm{mem}_4[a]$; $\mathtt{x_d}\langle 63,32\rangle \leftarrow 0$	`ldr w19, [x20]`
`ldrh`	`ldrh wd, a`	charge 2 octets: $\mathtt{x_d}\langle 15,0\rangle \leftarrow \mathrm{mem}_2[a]$; $\mathtt{x_d}\langle 63,16\rangle \leftarrow 0$	`ldrh w19, [x20]`
`ldrb`	`ldrb wd, a`	charge 1 octet: $\mathtt{x_d}\langle 7,0\rangle \leftarrow \mathrm{mem}_1[a]$; $\mathtt{x_d}\langle 63,8\rangle \leftarrow 0$	`ldrb w19, [x20]`
`str`	`str xd, a`	stocke 8 octets: $\mathrm{mem}_8[a] \leftarrow \mathtt{x_d}\langle 63,0\rangle$	`str x19, [x20]`
	`str wd, a`	stocke 4 octets: $\mathrm{mem}_4[a] \leftarrow \mathtt{x_d}\langle 31,0\rangle$	`str w19, [x20]`
`strh`	`strh wd, a`	stocke 2 octets: $\mathrm{mem}_2[a] \leftarrow \mathtt{x_d}\langle 15,0\rangle$	`str w19, [x20]`
`strb`	`strb wd, a`	stocke 1 octet: $\mathrm{mem}_1[a] \leftarrow \mathtt{x_d}\langle 7,0\rangle$	`strb w19, [x20]`
`ldp`	`ldp xd, xn, a`	charge 16 octets: $\mathtt{x_d}\langle 63,0\rangle \leftarrow \mathrm{mem}_8[a]$, $\mathtt{x_n}\langle 63,0\rangle \leftarrow \mathrm{mem}_8[a+8]$	`ldp x19, x20, [sp]`
`stp`	`stp xd, xn, a`	stocke 16 octets: $\mathrm{mem}_8[a] \leftarrow \mathtt{x_d}\langle 63,0\rangle$, $\mathrm{mem}_8[a+8] \leftarrow \mathtt{x_n}\langle 63,0\rangle$	`stp x19, x20, [sp]`

Conditions de branchement.

- Codes de condition: `N` (négatif), `Z` (zéro), `C` (report), `V` (débordement)
- `C` indique aussi l'*absence* d'emprunt lors d'une soustraction
- Conditions de branchement:

Entiers non signés

Code	Signification	Codes de condition
`eq`	$=$	$\mathtt{Z}$
`ne`	$\neq$	$\neg\mathtt{Z}$
`hs`	$\geq$	$\mathtt{C}$
`hi`	$>$	$\mathtt{C} \wedge \neg\mathtt{Z}$
`ls`	$\leq$	$\neg\mathtt{C} \vee \mathtt{Z}$
`lo`	$<$	$\neg\mathtt{C}$

Entiers signés

Code	Signification	Codes de condition
`eq`	$=$	$\mathtt{Z}$
`ne`	$\neq$	$\neg\mathtt{Z}$
`ge`	$\geq$	$\mathtt{N} = \mathtt{V}$
`gt`	$>$	$\neg\mathtt{Z} \wedge (\mathtt{N} = \mathtt{V})$
`le`	$\leq$	$\mathtt{Z} \vee (\mathtt{N} \neq \mathtt{V})$
`lt`	$<$	$\mathtt{N} \neq \mathtt{V}$
`vs`	débordement	$\mathtt{V}$
`vc`	pas de débordement	$\neg\mathtt{V}$
`mi`	négatif	$\mathtt{N}$
`pl`	non négatif	$\neg\mathtt{N}$

Branchement.

- Instructions de branchement, où j est une valeur immédiate de 6 bits:

Code d'op.	Syntaxe	Effet	Exemple
`b.`	`b.cond etiq`	branche à `etiq:` si *cond*	`b.eq main100`
`b`	`b etiq`	branche à `etiq:`	`b main100`
`cbz`	`cbz rd, etiq`	branche à `etiq:` si $\mathtt{r_d} = 0$	`cbz x19 main100`
`cbnz`	`cbnz rd, etiq`	branche à `etiq:` si $\mathtt{r_d} \neq 0$	`cbnz x19 main100`
`tbz`	`tbz rd, j, etiq`	branche à `etiq:` si $\mathtt{r_d}\langle j\rangle = 0$	`tbz x19, 1, main100`
`tbnz`	`tbnz rd, j, etiq`	branche à `etiq:` si $\mathtt{r_d}\langle j\rangle \neq 0$	`tbnz x19, 1, main100`
`bl`	`bl etiq`	branche à `etiq:` et $\mathtt{x_{30}} \leftarrow \mathtt{pc} + 4$	`bl printf`
`blr`	`blr xd`	branche à $\mathtt{x_d}$ et $\mathtt{x_{30}} \leftarrow \mathtt{pc} + 4$	`blr x20`
`br`	`br xd`	branche à $\mathtt{x_d}$	`br x20`
`ret`	`ret`	branche à $\mathtt{x_{30}}$ (retour de sous-prog.)	`ret`

Adressage.

- Modes d'adressages, où k est une valeur immédiate de 7 bits:

Nom	Syntaxe	Adresse	Effet	Exemple
adresse d'une étiquette	`adr xd, etiq`	—	$x_d \leftarrow$ adresse de `etiq:`	`adr x19, main100`
indirect par registre	`[xd]`	x_d	—	`[x20]`
indirect par registre indexé	`[xd, xn]`	$x_d + x_n$	—	`[x20, x21]`
	`[xd, k]`	$x_d + k$	—	`[x20, 1]`
	`[xd, xn, decal k]`	$x_d + (x_n \; decal \; k)$	—	`[x20, x21, lsl 1]`
ind. par reg. indexé pré-inc.	`[xd, k]!`	$x_d + k$	$x_d \leftarrow x_d + k$ avant calcul	`[x20, 1]!`
ind. par reg. indexé post-inc.	`[xd], k`	x_d	$x_d \leftarrow x_d + k$ après calcul	`[x20], 1`
relatif	`etiq`	adresse de `etiq`	—	`main100`

Autres instructions.

Code d'op.	Syntaxe	Effet	Exemple
`csel`	`csel rd, rn, rm, cond`	si *cond*: $r_d \leftarrow r_n$, sinon: $r_d \leftarrow r_m$	`csel x19, x20, x21, eq`

Logique et manipulation de bits.

- Les instructions `lsl`, `lsr`, `asr` et `ror` possèdent également une variante de 32 bits utilisant les registres w_d, w_n et w_m (dans ce cas, les 32 bits de poids fort sont mis à 0)
- Instructions, où i est une valeur immédiate de 12 bits et j est une valeur immédiate de 6 bits:

Code d'op.	Syntaxe	Effet	Exemple
`mvn`	`mvn rd, rn`	$r_d \leftarrow \neg r_n$	`mvn x19, x20`
`and`	`and rd, rn, rm`	$r_d \leftarrow r_n \wedge r_m$	`and x19, x20, x21`
	`and rd, rn, i`	$r_d \leftarrow r_n \wedge i$	`and x19, x20, 4`
	`and rd, rn, rm, decal j`	$r_d \leftarrow r_n \wedge (r_m \; decal \; j)$	`and x19, x20, x21, lsl 1`
`orr`	`orr rd, rn, rm`	$r_d \leftarrow r_n \vee r_m$	`orr x19, x20, x21`
	`orr rd, rn, i`	$r_d \leftarrow r_n \vee i$	`orr x19, x20, 4`
	`orr rd, rn, rm, decal j`	$r_d \leftarrow r_n \vee (r_m \; decal \; j)$	`orr x19, x20, x21, lsl 1`
`eor`	`eor rd, rn, rm`	$r_d \leftarrow r_n \oplus r_m$	`eor x19, x20, x21`
	`eor rd, rn, i`	$r_d \leftarrow r_n \oplus i$	`eor x19, x20, 4`
	`eor rd, rn, rm, decal j`	$r_d \leftarrow r_n \oplus (r_m \; decal \; j)$	`eor x19, x20, x21, lsl 1`
`bic`	`bic rd, rn, rm`	$r_d \leftarrow r_n \wedge \neg r_m$	`bic x19, x20, x21`
	`bic rd, rn, i`	$r_d \leftarrow r_n \wedge \neg i$	`bic x19, x20, 4`
	`bic rd, rn, rm, decal j`	$r_d \leftarrow r_n \wedge \neg(r_m \; decal \; j)$	`bic x19, x20, x21, lsl 1`
`lsl`	`lsl xd, xn, j`	décalage de j bits vers la gauche: $x_d\langle 63, j\rangle \leftarrow x_n\langle 63-j, 0\rangle$; $x_d\langle j-1, 0\rangle \leftarrow 0$	`lsl x19, x20, 1`
`lsr`	`lsr xd, xn, j`	décalage de j bits vers la droite: $x_d\langle 63-j, 0\rangle \leftarrow x_n\langle 63, j\rangle$; $x_d\langle 63, 64-j\rangle \leftarrow 0$	`lsr x19, x20, 1`
`asr`	`asr xd, xn, j`	décalage arithmétique de j bits vers la droite: $x_d\langle 63-j, 0\rangle \leftarrow x_n\langle 63, j\rangle$; $x_d\langle 63, 64-j\rangle \leftarrow x_n\langle 63\rangle$	`asr x19, x20, 1`
`ror`	`ror xd, xn, j`	décalage circulaire de j bits vers la droite: $x_d \leftarrow x_n\langle j-1, 0\rangle \; x_n\langle 63, j\rangle$	`ror x19, xn, 1`

Registres (nombres en virgule flottante).

- Possède 32 registres double précision (64 bits) de la forme d_n
- Chaque registre d_n possède un sous-registre simple précision (32 bits) s_n
- v_n réfère au registre d_n ou s_n
- Conventions:

Registres	Utilisation
$d_0 - d_7$	registres d'arguments et de retour de sous-programmes
$d_8 - d_{15}$	registres sauvegardés par l'appelé
$d_{16} - d_{31}$	registres sauvegardés par l'appelant

Manipulation et arithmétique (nombres en virgule flottante).

- Les conditions de branchement sont les mêmes que pour les entiers et sont déterminées à partir de codes de condition mis à jour par `fcmp`

Code d'op.	Syntaxe	Effet	Exemple
`ldr`	`ldr dn, a`	charge un nombre en virgule flottante double précision de l'adresse a vers d_n (8 octets)	`ldr d8, [x19]`
	`ldr sn, a`	charge un nombre en virgule flottante simple précision de l'adresse a vers s_n (4 octets)	`ldr s8, [x19]`
`str`	`str dn, a`	stocke un nombre en virgule flottante double précision de d_n vers l'adresse a (8 octets)	`str d8, [x19]`
	`str sn, a`	stocke un nombre en virgule flottante simple précision de s_n vers l'adresse a (4 octets)	`str s8, [x19]`
`fmov`	`fmov vd, vm`	$v_d \leftarrow v_m$	`fmov d8, d9`
	`fmov vd, i`	$v_d \leftarrow i$	`fmov d8, 1.5`
`fcmp`	`fcmp vd, vm`	compare v_d et v_m	`fcmp d8, d9`
	`fcmp vd, i`	compare v_d et i	`fcmp d8, 0.0`
`fadd`	`fadd vd, vn, vm`	$v_d \leftarrow v_n + v_m$	`fadd d8, d9, d10`
`fsub`	`fsub vd, vn, vm`	$v_d \leftarrow v_n - v_m$	`fsub d8, d9, d10`
`fmul`	`fmul vd, vn, vm`	$v_d \leftarrow v_n \cdot v_m$	`fmul d8, d9, d10`
`fdiv`	`fdiv vd, vn, vm`	$v_d \leftarrow v_n / v_m$	`fdiv d8, d9, d10`
`fsqrt`	`fsqrt vd, vn`	$v_d \leftarrow \sqrt{v_n}$	`fsqrt d8, d9`
`fabs`	`fabs vd, vn`	$v_d \leftarrow \lvert v_n \rvert$	`fabs d8, d9`

`ucvtf`	`ucvtf vd, rn`	convertit l'entier non signé dans r_n vers un nombre en virgule flottante dans v_d (selon le mode d'approximation configuré dans le registre de contrôle FPCR)	`ucvtf d8, x19`
			`ucvtf d8, w19`
			`ucvtf s8, x19`
			`ucvtf s8, w19`

`scvtf`	`scvtf vd, rn`	convertit l'entier signé dans r_n vers un nombre en virgule flottante dans v_d (selon le mode d'approximation configuré dans le registre de contrôle FPCR)	`scvtf d8, x19`
			`scvtf d8, w19`
			`scvtf s8, x19`
			`scvtf s8, w19`

`fcvt`	`fcvt vd, vn`	convertit le nombre en virgule flottante dans v_n vers un nombre en virgule flottante d'une autre précision dans v_d	`fcvt d8, s9`

Appels système.

- x_8: code numérique du service
- x_0 à x_5: arguments
- `svc 0`: appel du service

Données statiques.

Segments de données

Pseudo-instruction	Contenu
`.section ".text"`	instructions
`.section ".rodata"`	données en lecture seule
`.section ".data"`	données initialisées
`.section ".bss"`	données non-initialisées

Données

`.align`	k	donnée suivante stockée à une adresse divisible par k
`.skip`	k	réserve k octets
`.ascii`	s	chaîne de caractères initialisée à s
`.asciz`	s	chaîne de caractères initialisée à s suivi du carac. nul
`.byte`	v	octet initialisé à v
`.hword`	v	demi-mot initialisé à v
`.word`	v	mot initialisé à v
`.xword`	v	double mot initialisé à v
`.single`	f	nombre en virg. flottante simple précision initialisé à f
`.double`	f	nombre en virg. flottante double précision initialisé à f

Entrées/sorties (haut niveau).

- Affichage: `printf(&format, val1, val2, ...)`
- Lecture: `scanf(&format, &var1, &var2, ...)`
- Spécificateurs de format:

Famille	Format	Type
Nombres sur 64 bits	`%ld`	entier décimal signé
	`%lu`	entier décimal non signé
	`%lX`	entier hexadécimal non signé
	`%lf`	nombre en virgule flottante
Nombres sur 32 bits	`%d`	entier décimal signé
	`%u`	entier décimal non signé
	`%X`	entier hexadécimal non signé
	`%f`	nombre en virgule flottante
Nombres sur 16 bits	`%hd`	entier décimal signé
	`%hu`	entier décimal non signé
	`%hX`	entier hexadécimal non signé
Caractères	`%c`	caractère (1 octet)
	`%s`	chaîne de caractères

Annexe C:

Sommaire de l’architecture du NES

Registres.

- Possède 4 registres d'un octet
- Registre interne: `p` (*registre d'état*), contient des états et codes de conditions dont *report/emprunt* (1 octet)
- Registre interne: `pc` (*compteur d'instruction*), contient l'adresse de la prochaine instruction (2 octets)

Nom	Utilisation principale
a	accumulateur, utilisé comme opérande et valeur de retour des opérations arithmétiques et logiques
x	utilisé comme compteur ou comme index pour l'adressage indexé
y	utilisé comme compteur ou comme index pour l'adressage indexé
s	pointeur de pile (pointe vers $0100_{16} + \mathtt{s}$)

Valeurs immédiates.

- `#`: valeur numérique, sans `#`: adresse
- `$`: valeur hexadécimale
- `%`: valeur binaire
- Exemples:

expression	valeur
`#5`	5_{10}
`#$FF`	FF_{16}
`#%00010011`	00010011_2
`$FF`	adresse FF_{16}

Modes d'adressage.

Nom.	Syntaxe	Adresse	Exemple
absolu	`i`	i	`lda $D010`
indexé par x	`i, x`	$i + x$	`lda $D010, x`
	`etiq, x`	$etiq + x$	`lda tab, x`
indexé par y	`i, y`	$i + y$	`lda $D010, y`
	`etiq, y`	$etiq + y$	`lda tab, y`

Accès mémoire.

- Instructions, où $\mathrm{mem}_1[a]$ dénote l'octet situé à l'adresse a de la mémoire principale:

Code d'op.	Syntaxe	Effet	Exemple
`lda`	`lda #i`	a $\leftarrow i$	`lda #42`
	`lda adr`	a $\leftarrow \mathrm{mem}_1[adr]$	`lda var`
`ldx`	`ldx #i`	x $\leftarrow i$	`ldx #42`
	`ldx adr`	x $\leftarrow \mathrm{mem}_1[adr]$	`ldx var`
`ldy`	`ldy #i`	y $\leftarrow i$	`ldy #42`
	`ldy adr`	y $\leftarrow \mathrm{mem}_1[adr]$	`ldy var`
`sta`	`sta adr`	$\mathrm{mem}_1[adr] \leftarrow$ a	`sta var`
`stx`	`stx adr`	$\mathrm{mem}_1[adr] \leftarrow$ x	`stx var`
`sty`	`sty adr`	$\mathrm{mem}_1[adr] \leftarrow$ y	`sty var`
`txa`	`txa`	a $\leftarrow$ x	`txa`
`tax`	`tax`	x $\leftarrow$ a	`tax`
`tya`	`tya`	a $\leftarrow$ y	`tya`
`tay`	`tay`	y $\leftarrow$ a	`tay`
`txs`	`txs`	s $\leftarrow$ x	`txs`
`tsx`	`tsx`	x $\leftarrow$ s	`tsx`
`pha`	`pha`	empile a sur la pile	`pha`
`pla`	`pla`	dépile le premier octet de la pile vers a	`pla`

Arithmétique.

Code d'op.	Syntaxe	Effet	Exemple
adc	adc #i	a ← a + i + *report*	lda #1
	adc adr	a ← a + mem$_1[adr]$ + *report*	adc var
sbc	sbc #i	a ← a − i − *emprunt*	sbc #1
	sbc adr	a ← a − mem$_1[adr]$ − *emprunt*	sbc var
clc	clc	*report* ← 0 (utile avant adc)	clc
sec	sec	*emprunt* ← 0 (utile avant sbc)	sec
inx	inx	x ← x + 1	inx
iny	iny	y ← y + 1	iny
inc	inc adr	mem$_1[adr]$ ← mem$_1[adr]$ + 1	inc var
dec	dec adr	mem$_1[adr]$ ← mem$_1[adr]$ − 1	dec var

Logique.

Code d'op.	Syntaxe	Effet	Exemple
asl	asl adr	décalage logique de mem$_1[adr]$ d'un bit à gauche (directement en mémoire)	asl var
lsr	lsr adr	décalage logique de mem$_1[adr]$ d'un bit à droite (directement en mémoire)	lsr var
and	and #i	a ← a ∧ i	and #%00100011
	and adr	a ← a ∧ mem$_1[adr]$	and var
ora	ora #i	a ← a ∨ i	ora #%00100011
	ora adr	a ← a ∨ mem$_1[adr]$	ora var
eor	eor #i	a ← a ⊕ i	eor #%00100011
	eor adr	a ← a ⊕ mem$_1[adr]$	eor var

Comparaisons et branchements.

Code d'op.	Syntaxe	Effet	Exemple
cmp	cmp #i	compare a et i	cmp #0
	cmp adr	compare a et mem$_1[adr]$	cmp var
cpx	cpx #i	compare x et i	cpx #0
	cpx adr	compare x et mem$_1[adr]$	cpx var
cpy	cpy #i	compare y et i	cpy #0
	cpy adr	compare y et mem$_1[adr]$	cpy var
beq	beq etiq	branche à etiq: si $=$	beq boucle
bne	bne etiq	branche à etiq: si $\neq$	bne boucle
jmp	jmp etiq	branche à etiq:	jmp boucle
jsr	jsr etiq	branche au sous-programme etiq: et empile l'adresse de retour	jsr func
rts	rts	branche à l'adresse de retour d'un sous-programme	rts
rti	rti	branche à l'adresse de retour d'une interruption	rti