

IFT209 – Programmation système
 Université de Sherbrooke
Examen final

Enseignant: Michael Blondin
 Date: mercredi 27 avril 2022
 Durée: 3 heures

Directives:

- Répondez aux questions dans le **cahier de réponses**, pas sur ce questionnaire;
- **Une seule feuille (recto verso)** de notes au format $8\frac{1}{2}'' \times 11''$ est permise;
- **Aucun matériel additionnel** (notes de cours, fiches récapitulatives, 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 **9 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 A**;
- La question 5 utilise le langage d'assemblage du **NES** tel qu'utilisé en classe; un sommaire est présenté à l'**annexe B**.

Question 1: valeurs booléennes et chaînes de bits

- (a) Considérons le scénario où le registre w_{19} contient les bits $00 \dots 0b_7b_6 \dots b_1b_0$ et où nous cherchons à transformer son contenu vers $00 \dots 0b_0b_7b_6 \dots b_1$. Autrement dit, nous cherchons à effectuer un décalage circulaire d'un bit vers la droite, sur les huit bits de poids faible de x_{19} . 3,5 pts

Deux des programmes ci-dessous accomplissent correctement cette tâche. Identifiez-les. Laissez une trace du contenu de w_{19} et w_{20} après l'exécution de chaque ligne de code de chaque programme. Initialement, nous supposons que $w_{19} = 2F_{16}$ et $w_{20} = 00_{16}$.

programme A	programme B	programme C
<code>ror w19, w19, 1</code>	<code>and w20, w19, 0xFE</code>	<code>lsr w20, w19, 1</code>
<code>asr w20, w19, 24</code>	<code>eor w19, w19, w20</code>	<code>bic w19, w19, 0xFE</code>
<code>orr w19, w19, w20</code>	<code>lsr w20, w20, 1</code>	<code>lsl w19, w19, 7</code>
<code>and w19, w19, 0xFF</code>	<code>orr w19, w20, w19</code>	<code>orr w19, w19, w20</code>

Il s'agit des programmes A et C:					
programme A		programme B		programme C	
w ₁₉	w ₂₀	w ₁₉	w ₂₀	w ₁₉	w ₂₀
10...0	00010111	00...0	00000000	00...0	00101111
		00...0	00101110	00...0	00010111
	11...1	10000000		00...0	00000001
11...1	10010111		00...0	00010111	00...0
00...0	10010111	00...0	00010111	00...0	10000000
		00...0	00010111	00...0	10010111

(b) Ajoutez *une* ligne de code (à l'endroit de votre choix) au programme incorrect afin qu'il devienne correct. 1,5 pts

```
and w20, w19, 0xFE
eor w19, w19, w20
lsr w20, w20, 1
lsl w19, w19, 7 // <--- ici ou avant ligne précédente
orr w19, w20, w19
```

(c) 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 $\textcircled{?}$ par un opérateur logique parmi \wedge , \vee ou \oplus , et chaque occurrence de $\boxed{?}$ par 0 ou 1. Dans chaque cas, l'opérateur est appliqué bit à bit. 3 pts

opération A	opération B	opération C																																																												
$\textcircled{?}$ <table border="0"> <tr><td>b_4</td><td>b_3</td><td>b_2</td><td>b_1</td><td>b_0</td></tr> <tr><td>$\boxed{?}$</td><td>$\boxed{?}$</td><td>$\boxed{?}$</td><td>$\boxed{?}$</td><td>$\boxed{?}$</td></tr> <tr><td>\hline</td><td></td><td></td><td></td><td></td></tr> <tr><td>$\neg b_4$</td><td>b_3</td><td>b_2</td><td>$\neg b_1$</td><td>b_0</td></tr> </table>	b_4	b_3	b_2	b_1	b_0	$\boxed{?}$	$\boxed{?}$	$\boxed{?}$	$\boxed{?}$	$\boxed{?}$	\hline					$\neg b_4$	b_3	b_2	$\neg b_1$	b_0	$\textcircled{?}$ <table border="0"> <tr><td>b_4</td><td>b_3</td><td>b_2</td><td>b_1</td><td>b_0</td></tr> <tr><td>$\boxed{?}$</td><td>$\boxed{?}$</td><td>$\boxed{?}$</td><td>$\boxed{?}$</td><td>$\boxed{?}$</td></tr> <tr><td>\hline</td><td></td><td></td><td></td><td></td></tr> <tr><td>1</td><td>1</td><td>b_2</td><td>1</td><td>1</td></tr> </table>	b_4	b_3	b_2	b_1	b_0	$\boxed{?}$	$\boxed{?}$	$\boxed{?}$	$\boxed{?}$	$\boxed{?}$	\hline					1	1	b_2	1	1	$\textcircled{?}$ <table border="0"> <tr><td>b_4</td><td>b_3</td><td>b_2</td><td>b_1</td><td>b_0</td></tr> <tr><td>$\boxed{?}$</td><td>$\boxed{?}$</td><td>$\boxed{?}$</td><td>$\boxed{?}$</td><td>$\boxed{?}$</td></tr> <tr><td>\hline</td><td></td><td></td><td></td><td></td></tr> <tr><td>b_4</td><td>b_3</td><td>b_2</td><td>b_1</td><td>b_0</td></tr> </table>	b_4	b_3	b_2	b_1	b_0	$\boxed{?}$	$\boxed{?}$	$\boxed{?}$	$\boxed{?}$	$\boxed{?}$	\hline					b_4	b_3	b_2	b_1	b_0
b_4	b_3	b_2	b_1	b_0																																																										
$\boxed{?}$	$\boxed{?}$	$\boxed{?}$	$\boxed{?}$	$\boxed{?}$																																																										
\hline																																																														
$\neg b_4$	b_3	b_2	$\neg b_1$	b_0																																																										
b_4	b_3	b_2	b_1	b_0																																																										
$\boxed{?}$	$\boxed{?}$	$\boxed{?}$	$\boxed{?}$	$\boxed{?}$																																																										
\hline																																																														
1	1	b_2	1	1																																																										
b_4	b_3	b_2	b_1	b_0																																																										
$\boxed{?}$	$\boxed{?}$	$\boxed{?}$	$\boxed{?}$	$\boxed{?}$																																																										
\hline																																																														
b_4	b_3	b_2	b_1	b_0																																																										

opération A	opération B	opération C																																																												
\oplus <table border="0"> <tr><td>b_4</td><td>b_3</td><td>b_2</td><td>b_1</td><td>b_0</td></tr> <tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>\hline</td><td></td><td></td><td></td><td></td></tr> <tr><td>$\neg b_4$</td><td>b_3</td><td>b_2</td><td>$\neg b_1$</td><td>b_0</td></tr> </table>	b_4	b_3	b_2	b_1	b_0	1	0	0	1	0	\hline					$\neg b_4$	b_3	b_2	$\neg b_1$	b_0	\vee <table border="0"> <tr><td>b_4</td><td>b_3</td><td>b_2</td><td>b_1</td><td>b_0</td></tr> <tr><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td></tr> <tr><td>\hline</td><td></td><td></td><td></td><td></td></tr> <tr><td>1</td><td>1</td><td>b_2</td><td>1</td><td>1</td></tr> </table>	b_4	b_3	b_2	b_1	b_0	1	1	0	1	1	\hline					1	1	b_2	1	1	\wedge <table border="0"> <tr><td>b_4</td><td>b_3</td><td>b_2</td><td>b_1</td><td>b_0</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>\hline</td><td></td><td></td><td></td><td></td></tr> <tr><td>b_4</td><td>b_3</td><td>b_2</td><td>b_1</td><td>b_0</td></tr> </table>	b_4	b_3	b_2	b_1	b_0	1	1	1	1	1	\hline					b_4	b_3	b_2	b_1	b_0
b_4	b_3	b_2	b_1	b_0																																																										
1	0	0	1	0																																																										
\hline																																																														
$\neg b_4$	b_3	b_2	$\neg b_1$	b_0																																																										
b_4	b_3	b_2	b_1	b_0																																																										
1	1	0	1	1																																																										
\hline																																																														
1	1	b_2	1	1																																																										
b_4	b_3	b_2	b_1	b_0																																																										
1	1	1	1	1																																																										
\hline																																																														
b_4	b_3	b_2	b_1	b_0																																																										

Question 2: chaînes de caractères

Le codage fictif *pseudo-UTF-16* code chaque caractère sur deux ou quatre octets selon ce format:

# bits	plage de codes		format binaire des octets			
	début	fin	octet 1	octet 2	octet 3	octet 4
16	00000 ₁₆	0D7FF ₁₆	*****	*****	—	—
16	0E000 ₁₆	0FFFF ₁₆	*****	*****	—	—
20	10000 ₁₆	FFFFF ₁₆	110110**	*****	110111**	*****

Remarque: les codes 0D800₁₆ à 0DFFF₁₆ sont invalides.

- (a) La syllabe inuktitut « ᐃ » et l'emoji « 😊 » sont des caractères représentés respectivement par les codages pseudo-UTF-16 « 00010100 11000100 » et « 11011000 01111101 11011110 00000111 ». Donnez le code numérique Unicode associé à chacun de ces deux caractères (en hexadécimal). 2 pts

14C4₁₆ (obtenu de 00010100 11000100) et 1F607₁₆ (obtenu de 11011000 01111101 11011110 00000111)

- (b) Le code numérique Unicode du symbole musical « ♪ » est 0x1F631. Donnez son codage pseudo-UTF-16. 2 pts

11011000 01111101 11011110 00110001 (obtenu de 0x010000 ≤ 0x1F631 ≤ 0xFFFFF)

- (c) Combien de caractères de cette chaîne de caractères pseudo-UTF-16 sont représentables en ASCII? Justifiez. 2 pts

00000000 11101001 11011011 11111111 11011100 11110000 00000000 01100011 00000000 00000000

2 car les caractères ASCII ont un code numérique entre 0x00 et 0x7F. En pseudo-UTF-16, il s'agit d'un octet nul, suivi d'un octet dont le bit de gauche est nul:

00000000 11101001 11011011 11111111 11011100 11110000 00000000 01100011 00000000 00000000

- (d) Écrivez un sous-programme qui accomplit la tâche suivante: 6 pts

ENTRÉE: adresse d'une chaîne de caractères *s* sous codage pseudo-UTF-16 (premier et seul paramètre)

RETOUR: nombre de caractères non nuls de *s*

En particulier, votre sous-programme devrait retourner 5 sur la chaîne de caractères « allo😊 », et retourner 3 sur la chaîne de caractères de la sous-question (c). Vous avez accès aux macros SAVE et RESTORE.

Rappel: il est possible de spécifier une valeur hexadécimale avec le préfixe « 0x », par ex.: « mov x19, 0x3F ».

```

taille:
    SAVE
    mov    x19, x0
    mov    x0, 0

taille_boucle:
    ldrb  w20, [x19], 1
    ldrb  w21, [x19], 1
    cbnz  w20, taille_calcul
    cbz   w21, taille_ret

taille_calcul:
    add   x0, x0, 1

    and   w20, 0xFC
    cmp   w20, 0xD8
    b.ne  taille_boucle
    add   x19, x19, 2
    b     taille_boucle

taille_ret:
    RESTORE
    ret

```

Question 3: sous-programmes et mémoire

Considérons cet algorithme qui détermine récursivement si une chaîne de caractères est un palindrome, c.-à-d. si elle se lit de la même façon dans les deux sens:

Entrée : chaîne de caractères ASCII spécifiée par une adresse s ,
entier non signé de 64 bits n qui représente la taille de la chaîne (sans son caractère nul)

Retour : *vrai* si s est un palindrome, *faux* sinon

$\text{pal}(s, n)$:

| retourner $\text{pal_aux}(s, 0, n - 1)$

$\text{pal_aux}(s, i, j)$:

| si $i \geq j$ alors
| retourner *vrai*

| sinon

| retourner $(s[i] = s[j]) \wedge \text{pal_aux}(s, i + 1, j - 1)$

- (a) Implémentez l'algorithme en complétant les sous-programmes « `pal:` » et « `pal_aux:` ». Représentez « *faux* » 5 pts
et « *vrai* » respectivement par 0 et 1.

```

pal:
    /* à compléter au besoin */
    SAVE
    /* à compléter */
    RESTORE
    /* à compléter au besoin */

pal_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.
Remarque: si le côté gauche du « \wedge » est faux, alors vous n'avez pas à évaluer le côté droit.*

```

pal:
    SAVE
    sub    x2, x1, 1
    mov    x1, 0
    bl    pal_aux
    RESTORE
    ret

pal_aux:
    SAVE
    mov    x19, x0
    mov    x0, 1
    cmp    x1, x2
    b.ge   pal_aux_ret

pal_aux_gauche:
    mov    x0, 0
    ldrb   w20, [x19, x1]
    ldrb   w21, [x19, x2]
    cmp    w20, w21
    b.ne   pal_aux_ret

pal_aux_droite:
    mov    x0, x19
    add    x1, x1, 1
    sub    x2, x2, 1
    bl    pal_aux

pal_aux_ret:
    RESTORE
    ret

```

- (b) 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

```

pal:
    stp    x29, x30, [sp, -16]!
    mov    x29, sp
    /* ... */
    ldp    x29, x30, [sp], 16
    /* ... */

pal_aux:
    stp    x29, x30, [sp, -48]!
    mov    x29, sp
    stp    x19, x20, [sp, 16]
    stp    x21, xzr, [sp, 32]
    /* ... */
    ldp    x19, x20, [sp, 16]
    ldp    x21, xzr, [sp, 32]
    ldp    x29, x30, [sp], 48
    /* ... */

```

- (c) Supposons que le pointeur de pile sp contienne l'adresse a . Quelles sont les adresses minimales et maximales que prendront sp lors de l'exécution de votre code complet obtenu en (b) sur la chaîne « lol ». Justifiez. 2,5 pts

Remarque: vous pouvez donner des expressions symboliques comme $\min = a + 5 \cdot 4$ et $\max = a - 42$.

$\max = a$ et $\min = a - 16 - 2 \cdot 48$ car il y a un appel de « pal: » et deux appels de « pal_aux: » qui allouent respectivement 16 et 48 octets (en décrémentant sp).

Question 4: nombres en virgule flottante

- (a) Considérons le système \mathcal{S} de nombres en virgule flottante où la base est $\beta = 2$, la mantisse possède $n = 4$ bits, et l'exposant varie entre $e_{\min} = -99$ et $e_{\max} = 100$. Effectuez l'addition suivante: 5 pts

$$(1,010 \times 2^{90}) + (1,111 \times 2^{92}).$$

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.

$$\begin{aligned} 1,010 \times 2^{90} + 1,111 \times 2^{92} &= 1,010 \times 2^{90} + 111,1 \times 2^{90} \\ &= 1000,110 \times 2^{90} \\ &= 1,000110 \times 2^{93} \\ &\approx 1,001 \times 2^{93} \text{ (arrondi vers le haut, pas de bris d'égalité ici)} \end{aligned}$$

- (b) Donnez deux nombres normalisés x et y du système \mathcal{S} tels que $x + y = x$. Justifiez. 2,5 pts

Par exemple, $x = 2^4$ et $y = 2^0$ car:

$$\begin{aligned} 1,000 \times 2^4 + 1,000 \times 2^0 &= 1,000 \times 2^4 + 0,0001000 \times 2^4 \\ &= 1,0001000 \times 2^4 \\ &\approx 1,000 \times 2^4 \end{aligned}$$

- (c) Le système \mathcal{S} permet de représenter k nombres normalisés. Si l'on augmente e_{\max} à 101, et n à 5, nous obtenons un nouveau système \mathcal{S}' qui permet de représenter k' nombres normalisés. Exprimez k' par rapport à k . Par exemple, si \mathcal{S}' permet d'exprimer 42 fois plus de nombres que \mathcal{S} , alors la réponse est $k' = 42 \cdot k$. 2,5 pts

Remarque: s'il fallait identifier les valeurs numériques de k et k' , une calculatrice aurait été permise.

Sol. 1: L'ajout d'un bit à la mantisse double la quantité de nombres pour les exposants existants. Le nouvel exposant permet de former $2 \cdot 2^4$ nombres (signe \times bits après virgule). On a donc $k' = 2k + 2 \cdot 2^4$.

Sol. 2: $k' = 2 \cdot 2^4 \cdot (101 - (-99) + 1) = 2 \cdot (2 \cdot 2^3 \cdot (100 - (-99) + 1)) + 2 \cdot 2^4 = 2k + 2 \cdot 2^4$.

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*

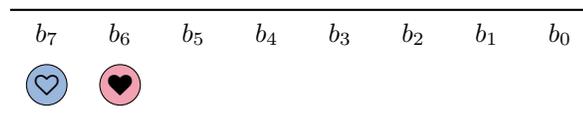
(a) Le *Power209* est un périphérique fictif du NES qui possède deux boutons:

6 pts



Le *Power209* se connecte dans le premier port. Son fonctionnement diffère de celui d'une manette standard:

- Il lance une interruption de type «IRQ» lorsqu'au moins un bouton est appuyé ou relâché;
- L'octet $b_7 \dots b_0$ lu à l'adresse 4016_{16} donne simultanément l'état des deux boutons selon ce format:



- 1 correspond à «appuyé», et 0 correspond à «non appuyé».

Complétez le code de la page suivante afin que le personnage *Tux* , représenté graphiquement par la tuile 42, descende verticalement d'un pixel lorsque  est appuyé; monte verticalement d'un pixel lorsque  est appuyé; et ne bouge pas lorsque les deux boutons sont appuyés.

```

posX:      .rs 1      ; Position horizontale de Tux
posY:      .rs 1      ; Position verticale de Tux
;
main:      ; main() {
    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:    ;
    jmp    boucle     ; }
;
init:      ; init()
    lda    #100       ; {
    sta    posX       ; posX = 100
    sta    posY       ; posY = 100
    rts              ; }
;
;; Déplacer Tux selon l'état
;; du Power209
deplacer:  ; deplacer()
;
; À COMPLÉTER
; }
;
;
;; Afficher Tux
afficher:  ; afficher()
;
; À COMPLÉTER
; ; Ordre: pos. verticale, identifiant, attributs,
; pos. horizontale
; Envoi par accès direct à la mémoire (DMA)
; de la plage $0300 à $03FF
;
; }
;
;; Segment des interruptions ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
.word    À COMPLÉTER ; NMI
.word    À COMPLÉTER ; RESET
.word    À COMPLÉTER ; IRQ

```

```

;; Déplacer Tux selon l'état ;
;; du Power209 ;
deplacer: ; deplacer()
; {
    lda    $4016 ; a = état du Power209
    and    #%11000000 ; a &= 0xC0
    cmp    #%10000000 ;
    bne    deplacer_suite ; if (a == 0x80) {
    inc    posY ; posY++
    jmp    deplacer_ret ; }
deplacer_suite: ; else if (a == 0x40) {
    cmp    #%01000000 ;
    bne    deplacer_ret ;
    dec    posY ; posY--
deplacer_ret: ; }
    rti ; }
;; Afficher Tux ;
afficher: ; afficher()
; {
    lda    posY ;
    sta    $0300 ; mem[0x0300] = posY
    ;
    lda    #42 ;
    sta    $0301 ; mem[0x0301] = 42
    ;
    lda    %00000000 ;
    sta    $0302 ; mem[0x0302] = 0
    ;
    lda    posX ;
    sta    $0303 ; mem[0x0303] = posX
    ;
    lda    #03 ; Envoi par accès direct à la mémoire (DMA)
    sta    $4014 ; de la plage 0x0300 à 0x03FF
    ;
    rti ; }
;; Segment des interruptions ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
.word    afficher ; NMI
.word    main ; RESET
.word    deplacer ; IRQ

```

- (b) Si l'intervalle de rafraîchissement vertical (*VBLANK*) se produit lors de l'exécution de « *deplacer* », est-ce que l'exécution de « *deplacer* » va se compléter comme attendu? Justifiez. 4 pts

Non, pas nécessairement. Comme les interruptions NMI sont de plus haute priorité que les interruptions IRQ. Le gestionnaire « *afficher* » mettra « *deplacer* » en attente. Comme les registres ne sont pas automatiquement sauvegardés et restaurés par le processeur, le contenu de *a* sera détruit.

Annexe A:

Sommaire de l'architecture ARMv8

Registres.

- ▶ Chaque registre x_n possède 64 bits: $b_{63}b_{62} \dots b_1b_0$
- ▶ Notation: $x_n\langle i \rangle := b_i$, $x_n\langle i, j \rangle := b_i b_{i-1} \dots b_j$, r_n réfère au registre x_n ou w_n
- ▶ Chaque sous-registre w_n possède 32 bits et correspond à $x_n\langle 31, 0 \rangle$
- ▶ Le compteur d'instruction pc n'est pas accessible
- ▶ Conventions:

Registres	Nom	Utilisation
$x_0 - x_7$	—	registres d'arguments et de retour de sous-programmes
x_8	xr	registre pour retourner l'adresse d'une structure
$x_9 - x_{15}$	—	registres temporaires sauvegardés par l'appelant
$x_{16} - x_{17}$	ip ₀ - ip ₁	registres temporaires intra-procéduraux
x_{18}	pr	registre temporaire pouvant être réservé par le système
$x_{19} - x_{28}$	—	registres temporaires sauvegardés par l'appelé
x_{29}	fp	pointeur vers l'ancien sommet de pile (<i>frame pointer</i>)
x_{30}	lr	registre d'adresse de retour (<i>link register</i>)
x_{31}	sp	registre contenant la valeur 0, ou pointeur de pile (<i>stack pointer</i>)

Arithmétique (entiers).

- ▶ Les codes de condition sont modifiés par **cmp**, **adds**, **adcs**, **subs**, **sbc** et **negs**
- ▶ À cette différence près, **adds**, **adcs**, **subs**, **sbc** 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 r_d et r_m	cmp x19, x21
	cmp rd, i	compare r_d et i	cmp x19, 42
	cmp rd, rm, decal j	compare r_d et r_m <i>decal j</i>	cmp x19, x21, lsl 1
add	add rd, rn, rm	$r_d \leftarrow r_n + r_m$	add x19, x20, x21
	add rd, rn, i	$r_d \leftarrow r_n + i$	add x19, x20, 42
	add rd, rn, rm, decal j	$r_d \leftarrow r_n + (r_m \text{ decal j})$	add x19, x20, x21, lsl 1
adc	adc rd, rn, rm	$r_d \leftarrow r_n + r_m + C$	adc x19, x20, x21
sub	sub rd, rn, rm	$r_d \leftarrow r_n - r_m$	sub x19, x20, x21
	sub rd, rn, i	$r_d \leftarrow r_n - i$	sub x19, x20, 42
	sub rd, rn, rm, decal j	$r_d \leftarrow r_n - (r_m \text{ decal j})$	sub x19, x20, x21, lsl 1
sbc	sbc rd, rn, rm	$r_d \leftarrow r_n - r_m - 1 + C$	sbc x19, x20, x21
neg	neg rd, rm	$r_d \leftarrow -r_m$	neg x19, x21
	neg rd, rm, decal j	$r_d \leftarrow -(r_m \text{ decal j})$	neg x19, x21, lsl 1
mul	mul rd, rn, rm	$r_d \leftarrow r_n \cdot r_m$	mul x19, x20, x21
udiv	udiv rd, rn, rm	$r_d \leftarrow r_n \div r_m$ (non signé)	udiv x19, x20, x21
sdiv	sdiv rd, rn, rm	$r_d \leftarrow r_n \div r_m$ (signé)	sdiv x19, x20, x21
madd	madd rd, rn, rm, ra	$r_d \leftarrow r_a + (r_n \cdot r_m)$	madd x19, x20, x21, x22
msub	msub rd, rn, rm, ra	$r_d \leftarrow r_a - (r_n \cdot r_m)$	msub x19, x20, x21, x22

Accès mémoire.

- **ldrsb**, **ldrsh** et **ldrsb** se comportent respectivement comme **ldr** (4 octets), **ldrh** et **ldrb** à l'exception du fait qu'ils effectuent un chargement dans 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 $mem_b[a]$ réfère aux b octets à l'adresse a de la mémoire principale:

Code d'op.	Syntaxe	Effet	Exemple
mov	mov rd, rm	$r_d \leftarrow r_m$	mov x19, x21
	mov rd, i	$r_d \leftarrow i$	mov x19, 42
ldr	ldr xd, a	charge 8 octets: $x_d \langle 63, 0 \rangle \leftarrow mem_8[a]$	ldr x19, [x20]
	ldr wd, a	charge 4 octets: $x_d \langle 31, 0 \rangle \leftarrow mem_4[a]$; $x_d \langle 63, 32 \rangle \leftarrow 0$	ldr w19, [x20]
ldrh	ldrh wd, a	charge 2 octets: $x_d \langle 15, 0 \rangle \leftarrow mem_2[a]$; $x_d \langle 63, 16 \rangle \leftarrow 0$	ldrh w19, [x20]
ldrb	ldrb wd, a	charge 1 octet: $x_d \langle 7, 0 \rangle \leftarrow mem_1[a]$; $x_d \langle 63, 8 \rangle \leftarrow 0$	ldrb w19, [x20]
str	str xd, a	stocke 8 octets: $mem_8[a] \leftarrow x_d \langle 63, 0 \rangle$	str x19, [x20]
	str wd, a	stocke 4 octets: $mem_4[a] \leftarrow x_d \langle 31, 0 \rangle$	str w19, [x20]
strh	strh wd, a	stocke 2 octets: $mem_2[a] \leftarrow x_d \langle 15, 0 \rangle$	str w19, [x20]
strb	strb wd, a	stocke 1 octet: $mem_1[a] \leftarrow x_d \langle 7, 0 \rangle$	strb w19, [x20]
ldp	ldp xd, xn, a	charge 16 octets: $x_d \langle 63, 0 \rangle \leftarrow mem_8[a]$, $x_n \langle 63, 0 \rangle \leftarrow mem_8[a + 8]$	ldp x19, x20, [sp]
stp	stp xd, xn, a	stocke 16 octets: $mem_8[a] \leftarrow x_d \langle 63, 0 \rangle$, $mem_8[a + 8] \leftarrow x_n \langle 63, 0 \rangle$	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	=	Z
ne	≠	¬Z
hs	≥	C
hi	>	C ∧ ¬Z
ls	≤	¬C ∨ Z
lo	<	¬C

Entiers signés

Code	Signification	Codes de condition
eq	=	Z
ne	≠	¬Z
ge	≥	N = V
gt	>	¬Z ∧ (N = V)
le	≤	Z ∨ (N ≠ V)
lt	<	N ≠ V
vs	débordement	V
vc	pas de débordement	¬V
mi	négatif	N
pl	non négatif	¬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 <i>cond</i>	b.eq main100
b	b etiq	branche à etiq :	b main100
cbz	cbz rd, etiq	branche à etiq : si $r_d = 0$	cbz x19 main100
cbnz	cbnz rd, etiq	branche à etiq : si $r_d \neq 0$	cbnz x19 main100
tbz	tbz rd, j, etiq	branche à etiq : si $r_d \langle j \rangle = 0$	tbz x19, 1, main100
tbnz	tbnz rd, j, etiq	branche à etiq : si $r_d \langle j \rangle \neq 0$	tbnz x19, 1, main100
bl	bl etiq	branche à etiq : et $x_{30} \leftarrow pc + 4$	bl printf
blr	blr xd	branche à x_d et $x_{30} \leftarrow pc + 4$	blr x20
br	br xd	branche à x_d	br x20
ret	ret	branche à 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]	$x_d + (x_n \text{ decal } k)$	—	[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 <i>cond</i> : $r_d \leftarrow r_n$, 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	$r_d \leftarrow r_n \wedge (r_m \text{ decal } j)$	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	$r_d \leftarrow r_n \vee (r_m \text{ decal } j)$	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	$r_d \leftarrow r_n \oplus (r_m \text{ decal } j)$	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	$r_d \leftarrow r_n \wedge \neg (r_m \text{ decal } j)$	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 d_n, a	charge un nombre en virgule flottante double précision de l'adresse a vers d_n (8 octets)	ldr $d8, [x19]$
	ldr s_n, a	charge un nombre en virgule flottante simple précision de l'adresse a vers s_n (4 octets)	ldr $s8, [x19]$
str	str d_n, a	stocke un nombre en virgule flottante double précision de d_n vers l'adresse a (8 octets)	str $d8, [x19]$
	str s_n, a	stocke un nombre en virgule flottante simple précision de s_n vers l'adresse a (4 octets)	str $s8, [x19]$
fmov	fmov v_d, v_m	$v_d \leftarrow v_m$	fmov $d8, d9$
	fmov v_d, i	$v_d \leftarrow i$	fmov $d8, 1.5$
fcmp	fcmp v_d, v_m	compare v_d et v_m	fcmp $d8, d9$
	fcmp v_d, i	compare v_d et i	fcmp $d8, 0.0$
fadd	fadd v_d, v_n, v_m	$v_d \leftarrow v_n + v_m$	fadd $d8, d9, d10$
fsub	fsub v_d, v_n, v_m	$v_d \leftarrow v_n - v_m$	fsub $d8, d9, d10$
fmul	fmul v_d, v_n, v_m	$v_d \leftarrow v_n \cdot v_m$	fmul $d8, d9, d10$
fdiv	fdiv v_d, v_n, v_m	$v_d \leftarrow v_n / v_m$	fdiv $d8, d9, d10$
fsqrt	fsqrt v_d, v_n	$v_d \leftarrow \sqrt{v_n}$	fsqrt $d8, d9$
fabs	fabs v_d, v_n	$v_d \leftarrow v_n $	fabs $d8, d9$
ucvtf	ucvtf v_d, r_n	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 v_d, r_n	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 v_d, v_n	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		Données	
Pseudo-instruction	Contenu		
<code>.section ".text"</code>	instructions	<code>.align</code> k	donnée suivante stockée à une adresse divisible par k
<code>.section ".rodata"</code>	données en lecture seule	<code>.skip</code> k	réserve k octets
<code>.section ".data"</code>	données initialisées	<code>.ascii</code> s	chaîne de caractères initialisée à s
<code>.section ".bss"</code>	données non-initialisées	<code>.asciz</code> s	chaîne de caractères initialisée à s suivi du carac. nul
		<code>.byte</code> v	octet initialisé à v
		<code>.hword</code> v	demi-mot initialisé à v
		<code>.word</code> v	mot initialisé à v
		<code>.xword</code> v	double mot initialisé à v
		<code>.single</code> f	nombre en virg. flottante simple précision initialisé à f
		<code>.double</code> 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	<code>%ld</code>	entier décimal signé
	<code>%lu</code>	entier décimal non signé
	<code>%lX</code>	entier hexadécimal non signé
	<code>%lf</code>	nombre en virgule flottante
Nombres sur 32 bits	<code>%d</code>	entier décimal signé
	<code>%u</code>	entier décimal non signé
	<code>%X</code>	entier hexadécimal non signé
Nombres sur 16 bits	<code>%f</code>	nombre en virgule flottante
	<code>%hd</code>	entier décimal signé
	<code>%hu</code>	entier décimal non signé
Caractères	<code>%hX</code>	entier hexadécimal non signé
	<code>%c</code>	caractère (1 octet)
	<code>%s</code>	chaîne de caractères

Annexe B:

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 $0100_{16} + s$)

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</i>	<i>i</i>	<code>lda \$D010</code>
indexé par x	<i>i, x</i> <i>eti, x</i>	$i + x$ $eti + x$	<code>lda \$D010, x</code> <code>lda tab, x</code>
indexé par y	<i>i, y</i> <i>eti, y</i>	$i + y$ $eti + y$	<code>lda \$D010, y</code> <code>lda tab, y</code>

Accès mémoire.

- ▶ Instructions, où $mem_1[a]$ dénote l'octet situé à l'adresse *a* de la mémoire principale:

Code d'op.	Syntaxe	Effet	Exemple
<code>lda</code>	<code>lda #i</code> <code>lda adr</code>	$a \leftarrow i$ $a \leftarrow mem_1[adr]$	<code>lda #42</code> <code>lda var</code>
<code>ldx</code>	<code>ldx #i</code> <code>ldx adr</code>	$x \leftarrow i$ $x \leftarrow mem_1[adr]$	<code>ldx #42</code> <code>ldx var</code>
<code>ldy</code>	<code>ldy #i</code> <code>ldy adr</code>	$y \leftarrow i$ $y \leftarrow mem_1[adr]$	<code>ldy #42</code> <code>ldy var</code>
<code>sta</code>	<code>sta adr</code>	$mem_1[adr] \leftarrow a$	<code>sta var</code>
<code>stx</code>	<code>stx adr</code>	$mem_1[adr] \leftarrow x$	<code>stx var</code>
<code>sty</code>	<code>sty adr</code>	$mem_1[adr] \leftarrow y$	<code>sty var</code>
<code>txa</code>	<code>txa</code>	$a \leftarrow x$	<code>txa</code>
<code>tax</code>	<code>tax</code>	$x \leftarrow a$	<code>tax</code>
<code>tya</code>	<code>tya</code>	$a \leftarrow y$	<code>tya</code>
<code>tay</code>	<code>tay</code>	$y \leftarrow a$	<code>tay</code>
<code>txs</code>	<code>txs</code>	$s \leftarrow x$	<code>txs</code>
<code>tsx</code>	<code>tsx</code>	$x \leftarrow s$	<code>tsx</code>
<code>pha</code>	<code>pha</code>	empile a sur la pile	<code>pha</code>
<code>pla</code>	<code>pla</code>	dépile le premier octet de la pile vers a	<code>pla</code>

Arithmétique.

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

Logique.

Code d'op.	Syntaxe	Effet	Exemple
asl	asl adr	décalage logique de $\text{mem}_1[\text{adr}]$ d'un bit à gauche (directement en mémoire)	asl var
lsr	lsr adr	décalage logique de $\text{mem}_1[\text{adr}]$ d'un bit à droite (directement en mémoire)	lsr var
and	and #i	$a \leftarrow a \wedge i$	and #%00100011
	and adr	$a \leftarrow a \wedge \text{mem}_1[\text{adr}]$	and var
ora	ora #i	$a \leftarrow a \vee i$	ora #%00100011
	ora adr	$a \leftarrow a \vee \text{mem}_1[\text{adr}]$	ora var
eor	eor #i	$a \leftarrow a \oplus i$	eor #%00100011
	eor adr	$a \leftarrow a \oplus \text{mem}_1[\text{adr}]$	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 $\text{mem}_1[\text{adr}]$	cmp var
cpx	cpx #i	compare x et i	cpx #0
	cpx adr	compare x et $\text{mem}_1[\text{adr}]$	cpx var
cpy	cpy #i	compare y et i	cpy #0
	cpy adr	compare y et $\text{mem}_1[\text{adr}]$	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