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 **5 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 $\texttt{w}_{19}$ contient les bits $00\cdots0b_7b_6\cdots b_1b_0$ et où nous cherchons à transformer son contenu vers $00\cdots0b_0b_7b_6\cdots b_1$. Autrement dit, nous cherchons à effectuer un décalage circulaire d'un bit vers la droite, sur les huit bits de poids faible de $\texttt{x}_{19}$. 3,5 pts

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

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

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

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

opération B

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

opération C

	b_4	b_3	b_2	b_1	b_0
(?)	[?]	[?]	[?]	[?]	[?]
	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	$\texttt{00000}_{16}$	$\texttt{0D7FF}_{16}$	`********`	`********`	—	—
16	$\texttt{0E000}_{16}$	$\texttt{0FFFF}_{16}$	`********`	`********`	—	—
20	$\texttt{10000}_{16}$	$\texttt{FFFFF}_{16}$	`110110**`	`********`	`110111**`	`********`

Remarque: les codes $\texttt{0D800}_{16}$ *à* $\texttt{0DFFF}_{16}$ *sont invalides.*

(a) La syllabe inuktitut « ᓄ » et l'émoji « 😇 » 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

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

(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
```

(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` *».*

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

`pal`(s, n)**:**
  retourner `pal_aux`$(s, 0, n-1)$

`pal_aux`(s, i, j)**:**
  si $i \geq j$ **alors**
    retourner *vrai*
  sinon
    retourner $(s[i] = s[j]) \wedge$ `pal_aux`$(s, i+1, j-1)$

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

```
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.

(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

(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 $= a - 42$.

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 $e_{\text{min}} = -99$ et $e_{\text{max}} = 100$. 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.

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

(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

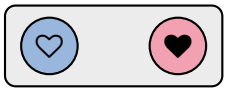
Remarque: s'il fallait identifier les valeurs numériques de k et k', une calculatrice aurait été permise.

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 $b_7 \cdots b_0$ lu à l'adresse 4016_{16} donne simultanément l'état des deux boutons selon ce format:

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
♡	♥						

— `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
    lda     #$03                ;   Envoi par accès direct à la mémoire (DMA)
    sta     $4014               ;                       de la plage $0300 à $03FF
                                ;
    rts                         ; }
                                ;
;; Segment des interruptions ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    .word    À COMPLÉTER        ; NMI
    .word    À COMPLÉTER        ; RESET
    .word    À COMPLÉTER        ; 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

Annexe A:

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 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 $0100_{16} + \mathtt{s}$)

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