IFT209 – Programmation système

Université de Sherbrooke

Examen final

Enseignant: Michael Blondin
Date: mercredi 14 avril 2021
Durée: 3 heures

Directives:

- Vous devez répondre aux questions dans le **cahier de réponses**, *pas* sur ce questionnaire;
- **Une seule feuille (recto verso)** de notes manuscrites au format 8½" × 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;
- Vous devez donner **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) Un pixel peut être représenté par quatre octets qui spécifient l'intensité de rouge (R), de vert (V), de bleu (B) et d'opacité (A). Considérons deux ordres sous lesquels représenter ces données sur 32 bits: RVBA et ARVB. *3,5 pts*

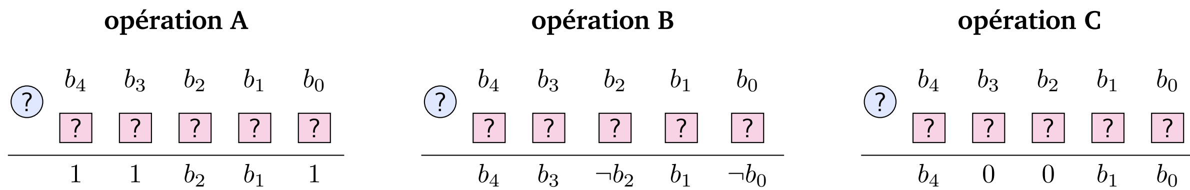
Le registre w_{19} contient un code RVBA qu'on cherche à convertir au format ARVB. Par exemple, si w_{19} débute avec 0xFFA1BC88, alors il doit se terminer avec 0x88FFA1BC. Deux des programmes ci-dessous accomplissent correctement cette conversion. Identifiez-les. Laissez une trace du contenu de w_{19} et w_{20} après l'exécution de chaque ligne de code de chaque programme en débutant avec w_{19} = 0xFFA1BC88 et w_{20} = 0x00000000.

programme A	programme B	programme C
`and w20, w19, 0xFF`	`and w20, w19, 0xFFFF0000`	`bic w20, w19, 0xFF`
`lsl w20, w20, 24`	`eor w19, w19, w20`	`lsr w20, w20, 8`
`lsr w19, w19, 8`	`lsr w20, w20, 8`	`lsl w19, w19, 24`
`orr w19, w20, w19`	`ror w19, w19, 8`	`orr w19, w19, w20`

(b) Écrivez du code qui extrait l'intensité de vert (V) d'un code RVBA stocké dans w_{19}. Par exemple, si w_{19} débute avec 0xFFA1BC88, alors il doit se terminer avec 0x000000A1. *1,5 pts*

Rappel: il est possible de spécifier une valeur hexadécimale avec le préfixe « `0x` », par ex.: « `mov x19, 0xD5` ».

(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



Question 2: chaînes de caractères

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

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

(a) La lettre grecque Ω est représentée par ce codage UTF-8: « `11001110 10101001` ». Donnez le code numérique Unicode associé à cette lettre (en hexadécimal). — 2 pts

(b) Le code numérique Unicode associé à l'émoji 😱 est `0x01F631`. Donnez le codage UTF-8 de cet émoji. — 2 pts

(c) Combien de caractères de cette chaîne de caractères UTF-8 sont représentables en ASCII? Justifiez. — 2 pts

```
01100011 11000011 10110100 01110100 11000011 10101001 00000000
```

(d) Rappelons que le codage ISO 8859-1 (Latin-1) permet de représenter les 256 caractères dont le code numérique Unicode appartient à la plage `0x00` à `0xFF`. Écrivez un sous-programme qui détermine si une chaîne de caractères, spécifiée sous codage UTF-8, serait représentable sous codage ISO 8859-1. Autrement dit, écrivez un sous-programme qui accomplit cette tâche: — 6 pts

> ENTRÉE: adresse d'une chaîne de caractères s sous codage UTF-8 (premier et seul paramètre)
>
> RETOUR: 1 si s serait représentable sous codage ISO 8859-1, 0 sinon

En particulier, votre sous-programme devrait retourner 0 sur les caractères des sous-questions (a) et (b), et retourner 1 sur la chaîne de la sous-question (c).

Rappel: il est possible de spécifier une valeur hexadécimale avec le préfixe « `0x` », par ex.: « `mov x19, 0xD5` ».

Question 3: sous-programmes et mémoire

Considérons cet algorithme qui calcule le maximum d'un tableau en le scindant récursivement:

Entrée : tableau d'entiers signés de 64 bits spécifié par une adresse t et un nombre d'éléments n
Retour : plus grande valeur du tableau

max(t, n):
- **si** $n = 1$ **alors**
  - **retourner** *premier élément du tableau* // un seul élément, donc forcément le max.
- **sinon**
  - $k \leftarrow n \div 2$ // $\div$ = division entière
  - $u \leftarrow$ adresse de l'élément à l'indice k du tableau
  - $a \leftarrow$ max(t, k) // a = max. des k premiers éléments
  - $b \leftarrow$ max($u, n - k$) // b = max. des $n - k$ derniers éléments
  - **si** $a \geq b$ **alors retourner** a
  - **sinon retourner** b

(a) Implémentez l'algorithme en complétant le sous-programme « max: ». 5 pts

```
max:
    /* 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) 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) La convention d'appel demande à ce que l'appelé préserve les registres x_{19} à x_{28}, mais pas les registres x_9 à x_{15}. Considérons une implémentation de la sous-question (a) qui effectue ses calculs dans x_0, x_1 et x_{19} à x_{23}. Peut-on retirer les macros SAVE et RESTORE en utilisant les registres x_9 à x_{13} plutôt que x_{19} à x_{23}? Justifiez. 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 $n = 5$ bits, et l'exposant varie entre $e_{\text{min}} = -3$ et $e_{\text{max}} = 3$. Effectuez l'addition suivante: 5 pts

$$(1{,}1101 \times 2^{-2}) + (1{,}1100 \times 2^{-1}).$$

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)

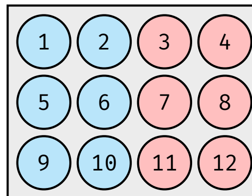
Par exemple, le nombre $1{,}5$ est codé sous précision simple par « `0 01111111 10000000000000000000000` ».

(i) Donnez le codage du nombre $-22{,}75$ au format simple précision. *2,5 pts*

(ii) Vrai ou faux: le format double précision permet de représenter exactement deux fois plus de nombres que le format simple précision. Justifiez. *2,5 pts*

Question 5: entrées/sorties

(a) Le *Power Pad* est un périphérique du NES qui se place au sol, par ex. comme tapis de danse ou d'exercice: *5 pts*



Le *Power Pad* se connecte dans le second port de manette et possède douze boutons (1 à 12). Son fonctionnement s'apparente à celui d'une manette standard:

— Pour demander l'état du *Power Pad*, 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 l'état de deux ou d'un boutons comme suit:

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
Première lecture:				4	2			
Deuxième lecture:				3	1			
Troisième lecture:				12	5			
Quatrième lecture:				8	9			
Cinquième lecture:					6			
Sixième lecture:					10			
Septième lecture:					11			
Huitième lecture:					7			

— Comme pour une manette standard: `1` correspond à « appuyé », `0` correspond à « non appuyé », et il n'est pas obligatoire d'effectuer les huit lectures.

Complétez ce code afin que le sous-programme « `generique_fin:` » soit appelé lorsque les boutons 9 et 4 (et possiblement d'autres) sont appuyés simultanément. Vous devez utiliser le mécanisme d'*attente active*.

```
grand_ecart:
    ; à compléter
    rts
```

(b) Considérons une solution au devoir 5 qui permet de déplacer correctement Mario à l'écran: 5 pts

```
    ;  Variables ici
main:                           ; main() {
    lda     #%00000000          ;   Désactiver interruptions NMI
    sta     $2000               ;
                                ;
    jsr     initialisation      ;   Initialiser pile, variables, palettes, arrière-plan, etc.
                                ;
    lda     #%10010000          ;
    sta     $2000               ;   Réactiver interruptions NMI
                                ;
    lda     #%00011000          ;
    sta     $2001               ;   Activer les tuiles et l'arrière-plan
boucle:                         ;
    jmp     boucle              ; }
                                ;
initialisation:                 ; initialisation()
                                ; {
    ; Code d'initialisation ici ;   Beaucoup de code ici (utilise a, x, y et des variables)
    rts                         ; }
                                ;
update:                         ; update()
    lda     #$02                ; {
    sta     $4014               ;
                                ;
    jsr     lire_manette        ;   Lire et stocker l'état des boutons
    jsr     deplacer_mario      ;   Déplacer Mario selon boutons appuyés
    jsr     update_mario        ;   Mettre tuiles à jour à partir de $0200
                                ;
    rti                         ; }

; Reste du code ici (comme au devoir 5)

; Table d'interruptions
    .bank    1
    .org     $FFFA
    .word    update             ; NMI
    .word    main               ; RESET
    .word    0                  ; IRQ
```

Remarquons que les interruptions NMI sont désactivées avant l'initialisation, puis activées. Modifions légèrement le code de « `main:` » afin que tout soit activé dès le départ:

```
main:                           ; main() {
    lda     #%10010000          ;
    sta     $2000               ;   Activer interruptions NMI
                                ;
    lda     #%00011000          ;
    sta     $2001               ;   Activer les tuiles et l'arrière-plan
                                ;
    jsr     initialisation      ;   Initialiser pile, variables, palettes, arrière-plan, etc.
boucle:                         ;
    jmp     boucle              ; }
```

En lançant le jeu, on obtient maintenant plusieurs incohérences visuelles: certaines tuiles incorrectes, mauvaises couleurs, etc. Expliquez pourquoi la modification crée ce problème.

Annexe A:

Sommaire de l’architecture ARMv8

Registres.

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

Registres	Nom	Utilisation
$\texttt{x}_0 - \texttt{x}_7$	—	registres d'arguments et de retour de sous-programmes
$\texttt{x}_8$	`xr`	registre pour retourner l'adresse d'une structure
$\texttt{x}_9 - \texttt{x}_{15}$	—	registres temporaires sauvegardés par l'appelant
$\texttt{x}_{16} - \texttt{x}_{17}$	$\texttt{ip}_0 - \texttt{ip}_1$	registres temporaires intra-procéduraux
$\texttt{x}_{18}$	`pr`	registre temporaire pouvant être réservé par le système
$\texttt{x}_{19} - \texttt{x}_{28}$	—	registres temporaires sauvegardés par l'appelé
$\texttt{x}_{29}$	`fp`	pointeur vers l'ancien sommet de pile (*frame pointer*)
$\texttt{x}_{30}$	`lr`	registre d'adresse de retour (*link register*)
$\texttt{x}_{zr}$	`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 $\texttt{r}_\texttt{d}$ et $\texttt{r}_\texttt{m}$	`cmp x19, x21`
	`cmp rd, i`	compare $\texttt{r}_\texttt{d}$ et i	`cmp x19, 42`
	`cmp rd, rm, decal j`	compare $\texttt{r}_\texttt{d}$ et $\texttt{r}_\texttt{m}$ *decal* j	`cmp x19, x21, lsl 1`
`add`	`add rd, rn, rm`	$\texttt{r}_\texttt{d} \leftarrow \texttt{r}_\texttt{n} + \texttt{r}_\texttt{m}$	`add x19, x20, x21`
	`add rd, rn, i`	$\texttt{r}_\texttt{d} \leftarrow \texttt{r}_\texttt{n} + i$	`add x19, x20, 42`
	`add rd, rn, rm, decal j`	$\texttt{r}_\texttt{d} \leftarrow \texttt{r}_\texttt{n} + (\texttt{r}_\texttt{m}\ \textit{decal}\ j)$	`add x19, x20, x21, lsl 1`
`adc`	`adc rd, rn, rm`	$\texttt{r}_\texttt{d} \leftarrow \texttt{r}_\texttt{n} + \texttt{r}_\texttt{m} + \texttt{C}$	`adc x19, x20, x21`
`sub`	`sub rd, rn, rm`	$\texttt{r}_\texttt{d} \leftarrow \texttt{r}_\texttt{n} - \texttt{r}_\texttt{m}$	`sub x19, x20, x21`
	`sub rd, rn, i`	$\texttt{r}_\texttt{d} \leftarrow \texttt{r}_\texttt{n} - i$	`sub x19, x20, 42`
	`sub rd, rn, rm, decal j`	$\texttt{r}_\texttt{d} \leftarrow \texttt{r}_\texttt{n} - (\texttt{r}_\texttt{m}\ \textit{decal}\ j)$	`sub x19, x20, x21, lsl 1`
`sbc`	`sbc rd, rn, rm`	$\texttt{r}_\texttt{d} \leftarrow \texttt{r}_\texttt{n} - \texttt{r}_\texttt{m} - 1 + \texttt{C}$	`sbc x19, x20, x21`
`neg`	`neg rd, rm`	$\texttt{r}_\texttt{d} \leftarrow -\texttt{r}_\texttt{m}$	`neg x19, x21`
	`neg rd, rm, decal j`	$\texttt{r}_\texttt{d} \leftarrow -(\texttt{r}_\texttt{m}\ \textit{decal}\ j)$	`neg x19, x21, lsl 1`
`mul`	`mul rd, rn, rm`	$\texttt{r}_\texttt{d} \leftarrow \texttt{r}_\texttt{n} \cdot \texttt{r}_\texttt{m}$	`mul x19, x20, x21`
`udiv`	`udiv rd, rn, rm`	$\texttt{r}_\texttt{d} \leftarrow \texttt{r}_\texttt{n} \div \texttt{r}_\texttt{m}$ (non signé)	`udiv x19, x20, x21`
`sdiv`	`sdiv rd, rn, rm`	$\texttt{r}_\texttt{d} \leftarrow \texttt{r}_\texttt{n} \div \texttt{r}_\texttt{m}$ (signé)	`sdiv x19, x20, x21`
`madd`	`madd rd, rn, rm, ra`	$\texttt{r}_\texttt{d} \leftarrow \texttt{r}_\texttt{a} + (\texttt{r}_\texttt{n} \cdot \texttt{r}_\texttt{m})$	`madd x19, x20, x21, x22`
`msub`	`msub rd, rn, rm, ra`	$\texttt{r}_\texttt{d} \leftarrow \texttt{r}_\texttt{a} - (\texttt{r}_\texttt{n} \cdot \texttt{r}_\texttt{m})$	`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 $\mathtt{x}_\mathtt{d}$ où les bits excédentaires sont le bit de signe de la donnée chargée, plutôt que des zéros
- Instructions, où a est une adresse et $\text{mem}_b[a]$ réfère aux b octets à l'adresse a de la mémoire principale:

Code d'op.	Syntaxe	Effet	Exemple
`mov`	`mov rd, rm`	$\mathtt{r}_\mathtt{d} \leftarrow \mathtt{r}_\mathtt{m}$	`mov x19, x21`
	`mov rd, i`	$\mathtt{r}_\mathtt{d} \leftarrow i$	`mov x19, 42`
`ldr`	`ldr xd, a`	charge 8 octets: $\mathtt{x}_\mathtt{d}\langle 63, 0\rangle \leftarrow \text{mem}_8[a]$	`ldr x19, [x20]`
	`ldr wd, a`	charge 4 octets: $\mathtt{x}_\mathtt{d}\langle 31, 0\rangle \leftarrow \text{mem}_4[a]$; $\mathtt{x}_\mathtt{d}\langle 63, 32\rangle \leftarrow 0$	`ldr w19, [x20]`
`ldrh`	`ldrh wd, a`	charge 2 octets: $\mathtt{x}_\mathtt{d}\langle 15, 0\rangle \leftarrow \text{mem}_2[a]$; $\mathtt{x}_\mathtt{d}\langle 63, 16\rangle \leftarrow 0$	`ldrh w19, [x20]`
`ldrb`	`ldrb wd, a`	charge 1 octet: $\mathtt{x}_\mathtt{d}\langle 7, 0\rangle \leftarrow \text{mem}_1[a]$; $\mathtt{x}_\mathtt{d}\langle 63, 8\rangle \leftarrow 0$	`ldrb w19, [x20]`
`str`	`str xd, a`	stocke 8 octets: $\text{mem}_8[a] \leftarrow \mathtt{x}_\mathtt{d}\langle 63, 0\rangle$	`str x19, [x20]`
	`str wd, a`	stocke 4 octets: $\text{mem}_4[a] \leftarrow \mathtt{x}_\mathtt{d}\langle 31, 0\rangle$	`str w19, [x20]`
`strh`	`strh wd, a`	stocke 2 octets: $\text{mem}_2[a] \leftarrow \mathtt{x}_\mathtt{d}\langle 15, 0\rangle$	`str w19, [x20]`
`strb`	`strb wd, a`	stocke 1 octet: $\text{mem}_1[a] \leftarrow \mathtt{x}_\mathtt{d}\langle 7, 0\rangle$	`strb w19, [x20]`
`ldp`	`ldp xd, xn, a`	charge 16 octets: $\mathtt{x}_\mathtt{d}\langle 63, 0\rangle \leftarrow \text{mem}_8[a]$, $\mathtt{x}_\mathtt{n}\langle 63, 0\rangle \leftarrow \text{mem}_8[a+8]$	`ldp x19, x20, [sp]`
`stp`	`stp xd, xn, a`	stocke 16 octets: $\text{mem}_8[a] \leftarrow \mathtt{x}_\mathtt{d}\langle 63, 0\rangle$, $\text{mem}_8[a+8] \leftarrow \mathtt{x}_\mathtt{n}\langle 63, 0\rangle$	`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 $\mathtt{r}_\mathtt{d} = 0$	`cbz x19 main100`
`cbnz`	`cbnz rd, etiq`	branche à `etiq:` si $\mathtt{r}_\mathtt{d} \neq 0$	`cbnz x19 main100`
`tbz`	`tbz rd, j, etiq`	branche à `etiq:` si $\mathtt{r}_\mathtt{d}\langle j\rangle = 0$	`tbz x19, 1, main100`
`tbnz`	`tbnz rd, j, etiq`	branche à `etiq:` si $\mathtt{r}_\mathtt{d}\langle j\rangle \neq 0$	`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 $\mathtt{x}_\mathtt{d}$ et $\mathtt{x}_{30} \leftarrow \mathtt{pc} + 4$	`blr x20`
`br`	`br xd`	branche à $\mathtt{x}_\mathtt{d}$	`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`	—	$\mathtt{x_d} \leftarrow$ adresse de `etiq:`	`adr x19, main100`
indirect par registre	`[xd]`	$\mathtt{x_d}$	—	`[x20]`
indirect par registre indexé	`[xd, xn]`	$\mathtt{x_d} + \mathtt{x_n}$	—	`[x20, x21]`
	`[xd, k]`	$\mathtt{x_d} + k$	—	`[x20, 1]`
	`[xd, xn, decal k]`	$\mathtt{x_d} + (\mathtt{x_n}\ decal\ k)$	—	`[x20, x21, lsl 1]`
ind. par reg. indexé pré-inc.	`[xd, k]!`	$\mathtt{x_d} + k$	$\mathtt{x_d} \leftarrow \mathtt{x_d} + k$ avant calcul	`[x20, 1]!`
ind. par reg. indexé post-inc.	`[xd], k`	$\mathtt{x_d}$	$\mathtt{x_d} \leftarrow \mathtt{x_d} + k$ 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*: $\mathtt{r_d} \leftarrow \mathtt{r_n}$, sinon: $\mathtt{r_d} \leftarrow \mathtt{r_m}$	`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 $\mathtt{w_d}$, $\mathtt{w_n}$ et $\mathtt{w_m}$ (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`	$\mathtt{r_d} \leftarrow \neg\mathtt{r_n}$	`mvn x19, x20`
`and`	`and rd, rn, rm`	$\mathtt{r_d} \leftarrow \mathtt{r_n} \land \mathtt{r_m}$	`and x19, x20, x21`
	`and rd, rn, i`	$\mathtt{r_d} \leftarrow \mathtt{r_n} \land i$	`and x19, x20, 4`
	`and rd, rn, rm, decal j`	$\mathtt{r_d} \leftarrow \mathtt{r_n} \land (\mathtt{r_m}\ decal\ j)$	`and x19, x20, x21, lsl 1`
`orr`	`orr rd, rn, rm`	$\mathtt{r_d} \leftarrow \mathtt{r_n} \lor \mathtt{r_m}$	`orr x19, x20, x21`
	`orr rd, rn, i`	$\mathtt{r_d} \leftarrow \mathtt{r_n} \lor i$	`orr x19, x20, 4`
	`orr rd, rn, rm, decal j`	$\mathtt{r_d} \leftarrow \mathtt{r_n} \lor (\mathtt{r_m}\ decal\ j)$	`orr x19, x20, x21, lsl 1`
`eor`	`eor rd, rn, rm`	$\mathtt{r_d} \leftarrow \mathtt{r_n} \oplus \mathtt{r_m}$	`eor x19, x20, x21`
	`eor rd, rn, i`	$\mathtt{r_d} \leftarrow \mathtt{r_n} \oplus i$	`eor x19, x20, 4`
	`eor rd, rn, rm, decal j`	$\mathtt{r_d} \leftarrow \mathtt{r_n} \oplus (\mathtt{r_m}\ decal\ j)$	`eor x19, x20, x21, lsl 1`
`bic`	`bic rd, rn, rm`	$\mathtt{r_d} \leftarrow \mathtt{r_n} \land \neg\mathtt{r_m}$	`bic x19, x20, x21`
	`bic rd, rn, i`	$\mathtt{r_d} \leftarrow \mathtt{r_n} \land \neg i$	`bic x19, x20, 4`
	`bic rd, rn, rm, decal j`	$\mathtt{r_d} \leftarrow \mathtt{r_n} \land \neg(\mathtt{r_m}\ decal\ j)$	`bic x19, x20, x21, lsl 1`
`lsl`	`lsl xd, xn, j`	décalage de j bits vers la gauche: $\mathtt{x_d}\langle 63, j\rangle \leftarrow \mathtt{x_n}\langle 63-j, 0\rangle$; $\mathtt{x_d}\langle j-1, 0\rangle \leftarrow 0$	`lsl x19, x20, 1`
`lsr`	`lsr xd, xn, j`	décalage de j bits vers la droite: $\mathtt{x_d}\langle 63-j, 0\rangle \leftarrow \mathtt{x_n}\langle 63, j\rangle$; $\mathtt{x_d}\langle 63, 64-j\rangle \leftarrow 0$	`lsr x19, x20, 1`
`asr`	`asr xd, xn, j`	décalage arithmétique de j bits vers la droite: $\mathtt{x_d}\langle 63-j, 0\rangle \leftarrow \mathtt{x_n}\langle 63, j\rangle$; $\mathtt{x_d}\langle 63, 64-j\rangle \leftarrow \mathtt{x_n}\langle 63\rangle$	`asr x19, x20, 1`
`ror`	`ror xd, xn, j`	décalage circulaire de j bits vers la droite: $\mathtt{x_d} \leftarrow \mathtt{x_n}\langle j-1, 0\rangle\ \mathtt{x_n}\langle 63, j\rangle$	`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, val`$_1$`, val`$_2$`, ...)`
- Lecture: `scanf(&format, &var`$_1$`, &var`$_2$`, ...)`
- 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} + 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` `etiq, x`	$i + x$ $etiq + x$	`lda $D010, x` `lda tab, x`
indexé par y	`i, y` `etiq, y`	$i + y$ $etiq + y$	`lda $D010, y` `lda tab, y`

Accès mémoire.

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

Code d'op.	Syntaxe	Effet	Exemple
`lda`	`lda #i` `lda adr`	a $\leftarrow i$ a $\leftarrow \text{mem}_1[adr]$	`lda #42` `lda var`
`ldx`	`ldx #i` `ldx adr`	x $\leftarrow i$ x $\leftarrow \text{mem}_1[adr]$	`ldx #42` `ldx var`
`ldy`	`ldy #i` `ldy adr`	y $\leftarrow i$ y $\leftarrow \text{mem}_1[adr]$	`ldy #42` `ldy var`
`sta`	`sta adr`	$\text{mem}_1[adr] \leftarrow$ a	`sta var`
`stx`	`stx adr`	$\text{mem}_1[adr] \leftarrow$ x	`stx var`
`sty`	`sty adr`	$\text{mem}_1[adr] \leftarrow$ 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` `adc adr`	$\mathtt{a} \leftarrow \mathtt{a} + i + report$ $\mathtt{a} \leftarrow \mathtt{a} + \text{mem}_1[adr] + report$	`lda #1` `adc var`
`sbc`	`sbc #i` `sbc adr`	$\mathtt{a} \leftarrow \mathtt{a} - i - emprunt$ $\mathtt{a} \leftarrow \mathtt{a} - \text{mem}_1[adr] - emprunt$	`sbc #1` `sbc var`
`clc`	`clc`	$report \leftarrow 0$ (utile avant `adc`)	`clc`
`sec`	`sec`	$emprunt \leftarrow 0$ (utile avant `sbc`)	`sec`
`inx`	`inx`	$\mathtt{x} \leftarrow \mathtt{x} + 1$	`inx`
`iny`	`iny`	$\mathtt{y} \leftarrow \mathtt{y} + 1$	`iny`
`inc`	`inc adr`	$\text{mem}_1[adr] \leftarrow \text{mem}_1[adr] + 1$	`inc var`
`dec`	`dec adr`	$\text{mem}_1[adr] \leftarrow \text{mem}_1[adr] - 1$	`dec var`

Logique.

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

Comparaisons et branchements.

Code d'op.	Syntaxe	Effet	Exemple
`cmp`	`cmp #i` `cmp adr`	compare a et i compare a et $\text{mem}_1[adr]$	`cmp #0` `cmp var`
`cpx`	`cpx #i` `cpx adr`	compare x et i compare x et $\text{mem}_1[adr]$	`cpx #0` `cpx var`
`cpy`	`cpy #i` `cpy adr`	compare y et i compare y et $\text{mem}_1[adr]$	`cpy #0` `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`