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 $8\frac{1}{2}'' \times 11''$ 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;
- Vous devez donner **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) 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 $\texttt{w}_{19}$ contient un code `RVBA` qu'on cherche à convertir au format `ARVB`. Par exemple, si $\texttt{w}_{19}$ 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 $\texttt{w}_{19}$ et $\texttt{w}_{20}$ après l'exécution de chaque ligne de code de chaque programme en débutant avec $\texttt{w}_{19} = \texttt{0xFFA1BC88}$ et $\texttt{w}_{20} = \texttt{0x00000000}$.

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`

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}
0xFFA1BC88	0x00000088	0xFFA1BC88	0xFFA10000	0xFFA1BC88	0xFFA1BC00
	0x88000000	0x0000BC88			0x00FFA1BC
0x00FFA1BC			0x00FFA100	0x88000000	
0x88FFA1BC		0x880000BC		0x88FFA1BC	

(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` ».

```
lsr    w19, w19, 16        // Solution 1
and    w19, w19, 0xFF

and    w19, w19, 0xFF0000 // Solution 2
lsr    w19, w19, 16

lsr    w19, w19, 16        // Solution 3
bic    w19, w19, 0xFF00

and    w19, w19, 0xFFFFFF // Solution 4
lsr    w19, w19, 16

lsl    w19, w19, 8         // Solution 5
lsr    w19, w19, 24

ror    w19, w19, 24        // Solution 6
lsr    w19, w19, 24
```

(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
	[?]	[?]	[?]	[?]	[?]
	1	1	b_2	b_1	1

opération B

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

opération C

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

opération A

$\vee$	b_4	b_3	b_2	b_1	b_0
	1	1	0	0	1
	1	1	b_2	b_1	1

opération B

$\oplus$	b_4	b_3	b_2	b_1	b_0
	0	0	1	0	1
	b_4	b_3	$\neg b_2$	b_1	$\neg b_0$

opération C

$\wedge$	b_4	b_3	b_2	b_1	b_0
	1	0	0	1	1
	b_4	0	0	b_1	b_0

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

$3A9_{16}$ (obtenu de `01110 101001`)

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

`11110000 10011111 10011000 10110001` (obtenu de `0x010000` ≤ `0x01F631` ≤ `0x10FFFF`)

(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
```

3 car un caractère est représentable en ASCII ssi son bit de gauche est `0`.

(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` ».

```
latin:                          // bool latin(char* s)
  SAVE                          // {
  mov   x19, x0                 //
  mov   x0, 1                   //   bool ok = true
latin_boucle:                   //
  ldrb  w20, [x19], 1           //
  cbz   w20, latin_ret          //   for (; *s != 0; s++) {
                                //
  tbz   w20, 7, latin_boucle    //     if (*s & 10000000₂ != 00000000₂) {  // 0x00 à 0x7F?
  and   w20, w20, 0xFC          //       if (*s & 11111100₂ != 11000000₂)  // 0x80 à 0xFF?
  cmp   w20, 0xC0               //         ok = false; break
  b.ne  latin_faux              //       else
  add   x19, x19, 1             //         s++                             // sauter 2e octet
                                //     }
  b     latin_boucle            //   }
latin_faux:                     //
  mov   x0, 0                   //
latin_ret:                      //
  RESTORE                       //   return ok
  ret                           // }
```

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**
  - $k \leftarrow n \div 2$ `// ÷ = 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.

```
max:                                    // long max(long* t, unsigned long n)
      SAVE                              // {
      mov     x19, x0                   //
      mov     x20, x1                   //
      ldr     x0, [x19]                 //   r = t[0]
      cmp     x20, 1                    //
      b.eq    max_ret                   //   if (n != 1) {
                                        //
      lsr     x21, x20, 1               //     k = n / 2
      mov     x0, x19                   //
      mov     x1, x21                   //
      bl      max                       //
      mov     x22, x0                   //     a = max(t, k)
                                        //
      add     x0, x19, x21, lsl 3       //     u = t + 8*k
      sub     x1, x20, x21              //
      bl      max                       //     b = max(u, n - k)
                                        //     r = b
      cmp     x22, x0                   //
      b.lt    max_ret                   //     if (a >= b)
      mov     x0, x22                   //       r = a
max_ret:                                //   }
      RESTORE                           //   return r
      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

```
stp     x29, x30, [sp, -48]!
mov     x29, sp
stp     x19, x20, [sp, 16]
stp     x21, x22, [sp, 32]
// ...
ldp     x19, x20, [sp, 16]
ldp     x21, x22, [sp, 32]
ldp     x29, x30, [sp], 48
```

(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

Solution 1: Non, utiliser d'autres registres ne change pas le fait que le contenu sera écrasé par les appels récursifs comme ni l'appelé ni l'appelant les sauvegardent/restaurent.

Solution 2: Non, le contenu de x_{30} ne sera pas sauvegardé lors des appels récursifs et on perdra donc la suite d'adresses de retour.

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.

$$\begin{aligned} 1{,}1101 \times 2^{-2} + 1{,}1100 \times 2^{-1} &= 0{,}11101 \times 2^{-1} + 1{,}11000 \times 2^{-1} \\ &= 10{,}10101 \times 2^{-1} \\ &= 1{,}010101 \times 2^{0} \\ &\approx 1{,}0101 \times 2^{0} \text{ (arrondi vers le bas, pas de bris d'égalité ici)} \end{aligned}$$

(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

Comme $-22{,}75 = -10110{,}11 \times 2^0 = -1{,}011011 \times 2^4$ et $4 + 127 = 128 + 2 + 1$, on obtient:

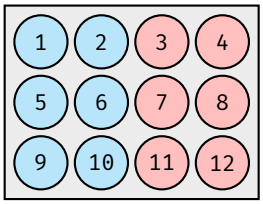
`1 10000011 01101100000000000000000`

(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

Faux. Même avec les mêmes exposants, chaque ajout de bit de mantisse double la quantité de nombres représentables, il y a donc au moins 2^{29} fois plus de nombres.

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
```

```
grand_ecart:
    lda     #1
    sta     $4017
    lda     #0
    sta     $4017

    lda     $4017
    and     #%00010000
    cmp     #%00010000
    bne     grand_ecart

    lda     $4017
    lda     $4017
    lda     $4017
    and     #%00001000
    cmp     #%00001000
    bne     grand_ecart

    jsr     generique_fin
    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.

La sous-routine « `update:` » est le gestionnaire d'interruption NMI appelé à chaque intervalle *VBLANK* qui se produit fréquemment. Ainsi, si on active les interruptions NMI dès le départ, on va rapidement interrompre l'initialisation, puis modifier et afficher le personnage. À ce moment, les composantes graphiques seront partielles en mémoire. De plus, les trois registres et les variables seront corrompus, avant de reprendre l'initialisation, comme le processeur ne s'occupe pas de les restaurer.