IFT209 – Programmation système

Université de Sherbrooke

Examen périodique

Enseignant: Michael Blondin
Date: mercredi 22 février 2023
Durée: 110 min.

Directives:

- Vous devez répondre aux questions dans le **cahier de réponses**, pas sur ce questionnaire;
- **Une feuille (recto verso)** de notes au format $8\frac{1}{2}'' \times 11''$ est permise, et les fiches en **annexe**;
- **Aucun matériel additionnel** (notes de cours, 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 **6 questions** sur **8 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 de cette architecture est présenté en **annexe**.

Question 1: systèmes de numération

Effectuez les conversions des nombres *non signés* suivants, en laissant une trace de votre démarche. Si une base intermédiaire est requise, ne passez pas par la base 10.

(a) 69 de la base 10 vers la base 2 — 2 pts

1000101 (car $69 = 64 + 4 + 1$)

(b) 10000111 de la base 2 vers la base 10 — 2 pts

135 (car $128 + 4 + 2 + 1 = 135$)

(c) 3F6A de la base 16 vers la base 8 — 2 pts

3	F	6	A
0011	1111	0110	1010
3	7	5	5	2

Effectuez cette addition directement dans la base indiquée, c.-à-d. sans convertir dans une base intermédiaire. Laissez une trace des retenues.

(d) $2B2C7_{16} + A947_{16}$ — 2 pts

$$\begin{array}{rr} & {\scriptstyle 1\ 1} \\ + & 2B2C7_{16} \\ & A947_{16} \\ \hline & 35C0E_{16} \end{array}$$

Question 2: architecture des ordinateurs

Considérons l'architecture SUPER209 d'une console de jeux vidéos fictive de quatrième génération. Elle possède quatre registres d'usage général de 16 bits: z_0, z_1, z_2, z_3. Sa mémoire est organisée au format grand-boutiste (« *big-endian* »). Son jeu d'instructions est constitué des six instructions ci-dessous, chacune codée sur 16 bits, où i désigne une valeur immédiate de 5 bits signés. Nous écrivons mem$[a]$ afin de référer aux deux octets stockés à l'adresse a de la mémoire principale.

instruction	**effet**	**format du code machine**	
		1er octet	2ème octet
`mov zd, i`	$z_d \leftarrow i$	`_____ 1__` (i, d)	`0000 1111`
`add zd, zn, zm`	$z_d \leftarrow z_n + z_m$	`00000 1__` (d)	`____ 0111` (n, m)
`add zd, zn, i`	$z_d \leftarrow z_n + \text{mem}[\text{PC} + 2i]$	`_____ 1__` (i, d)	`__00 0011` (n)
`bnz zd, i`	additionne $2i$ au compteur d'instructions si $z_d \neq 0$	`_____ 1__` (i, d)	`0000 0001`
`ldz zd, i`	charge mem$[\text{PC} + 2i]$ dans z_d	`_____ 1__` (i, d)	`0000 0000`
`stz zd, i`	stocke z_d dans mem$[\text{PC} + 2i]$	`_____ 0__` (i, d)	`0000 0000`

Rappel: le compteur d'instructions, dénoté PC *(« program counter »), est le registre interne qui contient l'adresse de la ligne de code actuelle à exécuter par l'unité de contrôle.*

(a) Le tableau ci-dessous illustre un programme SUPER209 et son code machine. Remplissez l'instruction manquante ainsi que les deux portions de code machine manquantes: 6 pts

programme	**code machine** (en hexadécimal)
`var:  /* donnée en mémoire */`	`0x0003`
`const:  /* donnée en mémoire */`	`0xFFFF`
`main:  ldz z1, -2`	**`0x????`**
`???  ???????????`	`0x0E0F`
`boucle:  add z1, z1, -3`	`0xED43`
`add z2, z2, z2`	**`0x????`**
`bnz z1, -2`	`0xF501`
`stz z2, -7`	`0xCA00`
`// fin du programme ici`	

Solution:

programme	code machine (en hexadécimal)
`main: ldz z1, -2`	`0xF500`
`mov z2, 1`	`0x0E0F`
`add z2, z2, z2`	`0x06A7`

Remarque: Afin d'être réaliste, le code de la question effectue réellement une tâche calculatoire sur cet ordinateur fictif. Le programme complet est ci-dessous; il modifie le contenu de `var` par $2^{\texttt{var}}$.

```
var:      /* = 3  */            0x0003
const:    /* = -1 */            0xFFFF
main:     ldz   z1, var         0xF500
          mov   z2, 1           0x0E0F
boucle:   add   z1, z1, const   0xED43
          add   z2, z2, z2      0x06A7
          bnz   z1, boucle      0xF501
          stz   z2, var         0xCA00
// fin: z1 = 0, z2 = 8, var = 8
```

(b) Rappelons qu'une architecture RISC est caractérisée par un jeu d'instructions constitué: 2 pts

- de *peu* d'instructions;
- d'instructions relativement *simples*;
- d'instructions qui *ne combinent pas les accès mémoire* à d'autres types d'opérations;
- d'instructions dont la traduction vers le code machine est de *taille fixe*.

Comme les deux premières propriétés sont subjectives, supposons qu'elles sont satisfaites par SUPER209. Dites si SUPER209 est une architecture RISC. Justifiez brièvement.

Non. L'instruction « `add zd, zn, i` » effectue un accès mémoire *et* une opération arithmétique.

Question 3: mémoire et accès aux données

Rappelons que, dans notre contexte, l'architecture ARMv8 utilise le format petit-boutiste (« little-endian »). Supposons que la mémoire principale contienne ces données:

adresse	contenu
⋮	⋮
$008B_{16}$	FF_{16}
$008C_{16}$	AF_{16}
$008D_{16}$	03_{16}
$008E_{16}$	08_{16}
$008F_{16}$	03_{16}
0090_{16}	$8B_{16}$
0091_{16}	00_{16}
0092_{16}	01_{16}
0093_{16}	50_{16}
⋮	⋮

(a) Quelle est la valeur hexadécimale du *mot* stocké à l'adresse $008E_{16}$? 2 pts

$008B0308_{16}$

(b) L'adresse $008E_{16}$ respecte-t-elle les contraintes d'*alignement* pour l'adressage d'un octet? d'un demi-mot? d'un mot? d'un double mot? Justifiez brièvement. 2 pts

L'adresse $008E_{16} = \cdots 1110_2$ est un multiple de 2, mais pas de 4 ou 8. Elle respecte donc seulement les contraintes pour un octet et un demi-mot.

(c) Supposons que l'adresse numérique associée à l'étiquette « `var:` » soit 0090_{16}, et que les registres soient initialisés à 0. Décrivez l'évolution du contenu de x_{19} et x_{20} après l'exécution de *chacune* de ces instructions: 4 pts

```
instruction1:   adr   x20, var
instruction2:   ldrh  w19, [x20]
instruction3:   ldrb  w20, [x20, -1]
instruction4:   ldr   w19, [x19, x20]

// Rappel: ldr xd (8 octets), ldr wd (4 octets), ldrh wd (2 octets), ldrb wd (1 octet)
```

x_{19}	x_{20}
0x00	**0x0090**
0x008B	0x0090
0x008B	**0x03**
0x008B0308	0x03

Question 4: entiers signés et circuits logiques

(a) Donnez la représentation binaire signée des nombres 23 et 8 sur six bits, puis calculez $23 - 8$ sur six bits. Laissez une trace. Indiquez s'il y a débordement et/ou report. Justifiez. 3 pts

On a $23 = 16 + 4 + 2 + 1 = 010111$ et $8 = 001000$. Calculons -8:

$$001000 \xrightarrow{\text{complément}} 110111 \xrightarrow{+1} 111000.$$

On obtient:

$$\begin{array}{rr} & {\scriptstyle 1\,1} \\ & 010111 \\ + & 111000 \\ \hline & 001111 \end{array}$$

Il y a report car il y a une retenue à la fin, et pas de débordement car 23 et -8 sont de signes opposés.

Alternative: on peut aussi vérifier qu'on obtient bien $23 - 8 = 15 = 001111_2$.

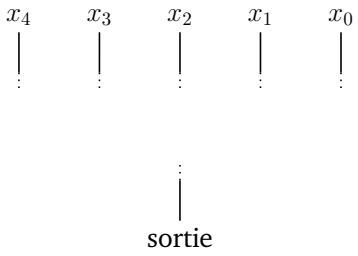
(b) Effectuez le produit de ces deux nombres signés de $n = 3$ bits: 110×011. Utilisez l'algorithme « simple », c.-à-d. celui qui étend les nombres sur $2n$ bits, fait les calculs non signés, puis tronque le résultat sur $2n$ bits. 2 pts

Rappel: vous avez implémenté cet algorithme au labo 2.

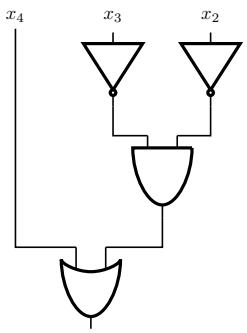
111010 obtenu ainsi:

$$\begin{array}{rrl} & 111110 & \text{(-2)} \\ \times & 000011 & \text{(3)} \\ \hline & 111110 & \\ + & 111110 & \\ \hline & 10111010 & \text{(-6)} \end{array}$$

(c) Complétez ce circuit afin qu'il détermine si un entier signé de 5 bits est inférieur ou égal à 3: 3 pts



Remarque: dans le doute quant à la lisibilité, utilisez le nom des portes plutôt que leur pictogramme.



Explication: le nombre est négatif ($x_4 = 1$), ou positif et inférieur à quatre ($x_3 = x_2 = 0$).

Question 5: programmation en langage d'assemblage

Considérons un logiciel qui permet de fusionner deux vidéos. La *durée* d'une vidéo est son nombre de minutes et de secondes. Par exemple, « `22:15` » correspond à une durée de 22 minutes + 15 secondes. Une durée est *optimale* si elle est comprise entre 30 et 90 secondes (inclusivement).

Nous cherchons à implémenter un programme qui, étant donné la durée de deux vidéos, indique la somme de leur durée, et affiche un message si celle-ci est optimale (et rien sinon). Voici des exemples d'entrées et sorties:

Entrée	`00:13` `00:10`	`00:15` `00:30`	`00:30` `00:42`	`01:15` `01:50`
Sortie	`00:23`	`00:45` `optimale!`	`01:12` `optimale!`	`03:05`

Complétez les blocs A et B du programme ci-dessous afin d'accomplir la tâche décrite. 8 pts

```
.global main

main:
    /*
      Ici, deux lectures sont effectuées à l'aide de scanf et fmtEntree.

      La durée 1 est stockée dans x19:x20.
      La durée 2 est stockée dans x21:x22.
      Ces durées sont supposées valides, donc 0 ≤ x20 < 60 et 0 ≤ x22 < 60.

      Vous n'avez pas à implémenter ces lectures. */

    // Calculer durée 1 + durée 2 dans x23:x24
    /**** CODE À COMPLÉTER (BLOC A) ****/

    adr  x0, fmtSortie
    mov  x1, x23
    mov  x2, x24
    bl   printf

    // Durée optimale?
    /**** CODE À COMPLÉTER (BLOC B) ****/

    adr  x0, msgOptimal
    bl   printf

    // Quitter
fin:
    mov  x0, 0
    bl   exit

.section ".bss"
tempMin:    .skip   8
tempSec:    .skip   8

.section ".rodata"
fmtEntree:  .asciz  "%lu:%lu"
fmtSortie:  .asciz  "%02lu:%02lu\n"
msgOptimal: .asciz  "optimale!\n"
```

Bloc A, solution 1:

```
    add  x25, x20, x22      // total_sec = sec1 + sec2
                            //
    mov  x28, 60            //
    udiv x23, x25, x28      // min = total_sec / 60
    msub x24, x28, x23, x25 // sec = total_sec - (60 * min)
                            //
    add  x23, x23, x19      // min += min1
    add  x23, x23, x21      // min += min2
```

Bloc A, solution 2:

```
    add  x23, x19, x21      // min = min1 + min2
    add  x24, x20, x22      // sec = sec1 + sec2
                            //
    cmp  x24, 60            // if (sec >= 60)
    b.lo afficher           // {
    add  x23, x23, 1        //   min += 1
    sub  x24, x24, 60       //   sec -= 60
afficher:                   // }
```

Bloc B, solution 1:

```
    mov  x28, 60            //
    madd x28, x28, x23, x24 // num_sec = (60 * min) + sec
    cmp  x28, 30            //
    b.lo fin                // if (num_sec < 30 ||
    cmp  x28, 90            //     num_sec > 90)
    b.hi fin                //   goto fin
```

Bloc B, solution 2:

```
    cmp  x23, 1             //
    b.hi fin                //
    b.eq cas1               // if (min == 0)
cas0:                       // {
    cmp  x24, 30            //   if (sec < 30) {
    b.lo fin                //     goto fin
    b    optimal            // }
cas1:                       // else if (min == 1) {
    cmp  x24, 30            //   if (sec > 30)
    b.hi fin                //     goto fin
optimal:                    // }
```

Question 6: tableaux

Considérons une matrice **B** de m lignes et n colonnes, dont les éléments sont des entiers non signés de 32 bits. Nous appelons la première et la dernière colonne de **B** ses *colonnes extrémales*.

Par exemple, voici un telle matrice où les colonnes extrémales sont colorées:

10	0	0	0	50
20	0	0	0	60
30	0	0	0	70
40	0	0	0	80

Attention: il ne s'agit que d'un exemple; répondez aux questions par rapport à m et n, pas 4 et 5.

Supposons que **B** soit stockée en mémoire (ligne à ligne) dans un tableau bidimensionnel qui débute à l'adresse a.

(a) Sur combien d'*octets* sont stockées les colonnes extrémales de **B**? 1 pt

$2 \cdot 4 \cdot m$ octets

(b) Quelle est l'*adresse* du dernier élément de la première colonne de **B** (comme celui hachuré ci-dessus)? 2 pts

$a + 4(m-1)n$

(c) Complétez le code suivant afin d'intervertir le contenu des colonnes extrémales de **B**. 7 pts

```
// N'implémentez pas la lecture. On suppose que
//  x19 = a  (adresse du tableau 2D qui contient B)
//  x20 = m  (m > 0)
//  x21 = n  (n > 0)

/************************
  CODE ARMv8 À COMPLÉTER
************************/

// Le contenu des première et dernière colonnes de B doit
// maintenant être inversé. N'implémentez pas l'affichage.
```

Par exemple, le tableau ci-dessus doit devenir comme suit:

50	0	0	0	10
60	0	0	0	20
70	0	0	0	30
80	0	0	0	40

```
    mov   x28, 4              //
    sub   x22, x21, 1         //
    mul   x22, x28, x22       // bout = 4 * (n - 1)
    mul   x23, x28, x21       // saut = 4 * n
boucle:                       //
    cbz   x20, fin            // while (m != 0)
                              // {
    ldr   w24, [x19]          //
    ldr   w25, [x19, x22]     //
    str   w25, [x19]          //
    str   w24, [x19, x22]     //   intervertir *a et *(a + bout)
                              //
    add   x19, x19, x23       //   a += saut
    sub   x20, x20, 1         //   m -= 1
    b     boucle              // }
fin:                          //
```

Annexe:

Fiches récapitulatives

1. Systèmes de numération

Système unaire

- Chaque nombre $n \in \mathbb{N}$ se représente par $\overbrace{1 \cdots 11}^{n \text{ fois}}$
- L'addition correspond à la concaténation
- Pas concis

Représentation positionnelle

- Généralisation du système décimal à une base $b \in \mathbb{N}_{\geq 2}$
- *Systèmes particuliers*: binaire ($b = 2$), octal ($b = 8$), décimal ($b = 10$), hexadécimal ($b = 16$)
- *Chiffres*: éléments de $\{0, 1, \ldots, b-1\}$
- *Chiffres au-delà de 9*: $\mathrm{A} = 10, \mathrm{B} = 11, \ldots, \mathrm{F} = 15, \ldots$
- *Valeur de x en base b*: $x_b = x_{n-1} \cdot b^{n-1} + \ldots + x_1 \cdot b^1 + x_0 \cdot b^0$
- *Exemple*: $8\mathrm{B}5_{16} = 8 \cdot 16^2 + 11 \cdot 16^1 + 5 \cdot 16^0$
- Les zéros tout à gauche ne changent rien: $(0 \cdots 0x)_b = x_b$

Conversions

- *b à 10*: $x_0 + b \cdot (x_1 + b \cdot (x_2 + b \cdot (\ldots + b \cdot x_{n-1})))$
- *10 à b*: diviser à répétition par b et concaténer les restes de droite à gauche, par ex. $6_2 = 110$:
  $6 \div 2 = 3$ reste 0, $3 \div 2 = 1$ reste 1, $1 \div 2 = 0$ reste 1
- *b à b^m*: remplacer chaque bloc de taille m par sa valeur en base b^m, par ex. si $b^m = 2^3$: $10110 \to 26$
- *b^m à b*: éclater chaque symbole vers sa représentation de taille m en base b, par ex. si $b^m = 2^3$: $73 \to 111011$

Addition

- Comme en base 10: additionner chiffre à chiffre en base b et propager une retenue vers la gauche

Fractions

- *Exemple*: $(11{,}01)_2 = 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 3{,}25$
- *Chiffres non significatifs*: $(0 \cdots 0x{,}y0 \cdots 0)_b = (x{,}y)_b$

2. Architecture des ordinateurs

Architecture et organisation

- *Architecture*: spécification des services des composants
- *Organisation*: description physique des composants

Architecture de von Neumann

- *Mémoire principale*: stocke les programmes et leurs données
- *Processeur*: unité centrale de traitement de l'ordinateur
- *Unités d'entrée/sortie*: contrôlent les périphériques
- *Bus*: systèmes de communication entre les composants

Mémoire principale

- Suite de cellules d'octets identifiées par des *adresses* uniques
- Une adresse peut référer à: 1 *octet* (8 bits), 2 octets (*demi-mot*), 4 octets (*mot*), 8 octets (*double mot*)
- Quantité de mémoire utilisable limitée par taille des adresses
- *Big-endian*: `[00, 58, 40, 0F]` vaut `0058400F`
  Little-endian: `[00, 58, 40, 0F]` vaut `0F405800`
- *Alignement*: adresser 2^k octets à une adresse qui n'est pas un multiple de 2^k — parfois: *interdit*, souvent: *ralentit l'accès*

Processeur

- *Jeu d'instructions* élémentaires, par ex: `add` (code d'opér.) `x10, x11, x12` (opérandes)
- *Registres*: cellules de mémoire interne, très rapide d'accès
- *Code machine*: traduction des instructions en suite de bits
- *Compteur d'instruction*: pointe vers prochaine instruction
- *Unité de contrôle*: coordonne l'exécution des instructions
- *Unité arithmétique et logique*: calculs sur $\mathbb{Z}$ et chaînes de bits
- *Pipeline*: parallélisation des étapes d'exécution
- *RISC*: instructions simples, taille fixe, mémoire–*ou*–autre

3. Programmation en langage d'assemblage: ARMv8

Registres

- *Registres:* x_0–x_{30} (64 bits) ou w_0–w_{30} (sous-registres 32 bits)
- *Usage libre:* x_0–x_7 (arguments) et x_{19}–x_{28} (sauveg. par l'appelé)
- *Usage semi-libre:* x_9–x_{15} (sauvegardés par l'appelant)

Organisation du code

- *Ligne*: `etiquette:  opcode  operandes  // Commentaire`
- *Étiquette*: nom symbolique d'une ligne de code
- *Exemple*:

```
impair:
        mov     x20, 3          // tmp = 3
        mul     x20, x20, x19   // tmp = tmp * n
        add     x19, x20, 1     // n   = tmp + 1
```

Quelques instructions

`mov xd, v`	$x_d \leftarrow v$	où v est regis. ou const.
`add xd, xn, v`	$x_d \leftarrow x_n + v$	où v est regis. ou const.
`mul xd, xn, xm`	$x_d \leftarrow x_n \cdot x_m$	
`udiv xd, xn, xm`	$x_d \leftarrow x_n \div x_m$	

Données statiques

- *Adresse divisible par k*: `.align k`
- *Alloue k octets consécutifs*: `.skip k`
- *1, 2, 4, 8 octets*: `.byte v`, `.hword v`, `.word v`, `.xword v`
- *Chaîne de car.*: `.asciz s`

Segments de données

- *Instructions*: `.section ".text"`
- *Données en lecture seule*: `.section ".rodata"`
- *Données initialisées*: `.section ".data"`
- *Données non-initialisées*: `.section ".bss"`

Entrée/sortie (de haut niveau via C)

- *Affichage*: `printf(&format, val1, val2, ...)`
- *Lecture*: `scanf(&format, &var1, &var2, ...)`
- *Format nombres*: `int32` (`%d`), `uint32` (`%u`), `uint32-hex` (`%X`), 64 bits via préfixe `l`, par ex. `int64` (`%ld`)

4. Nombres entiers

Représentation des entiers signés

- *Compl. à 2*: $\text{val}(x_{n-1}\cdots x_1x_0) = -x_{n-1}\cdot 2^{n-1} + \sum_{i=0}^{n-2} x_i \cdot 2^i$

bits	000	001	010	011	100	101	110	111
valeur	0	1	2	3	-4	-3	-2	-1

- *Représentables sur n bits*: $[-2^{n-1}, 2^{n-1}-1]$
- *Bit de signe*: négatif ssi bit de gauche $= 1$
- *Ajout de bits*: répéter bit de signe à gauche: $101 \to 1\cdots101$
- *Changement de signe*: $010 \xrightarrow{\text{complément}} 101 \xrightarrow{+1} 110$

Opérations arithmétiques

- *Addition*: comme les entiers non signés
- *Soustraction*: addition/changement de signe: $a-b = a+(-b)$
- *Report*: lors d'une retenue sur la somme des bits de poids fort
- *Débordement*: lorsque le résultat ne peut pas être représenté
- *Multiplication et division non signées*: comme en base 10:

$$\begin{array}{rr} & 101 \quad (5) \\ \times & 11 \quad (3) \\ & 101 \\ + & 101 \\ & 1111 \quad (15) \end{array} \qquad \begin{array}{rr} & 10011 \;\big|\; 11 \\ - & 11 \quad\;\; 00110 \\ & 111 \\ - & 11 \\ & 1 \end{array}$$

- *Mult. signée*: étendre opérandes à $2n$ bits et garder $2n$ bits faibles du résultat (s'implémente sans extension explicite)
- *Division signée*: calculer $|a| \div |b|$ et ajuster signe

Codes de condition

- *Codes*: `N` (négatif), `Z` (zéro), `C` (report) (bit de retenue), `V` (débordement) (résultat trop grand/petit)
- *Codes modifiés par*: `cmp`, `adds`, `subs`, `negs`, `adcs`, `sbcs`
- *Comparaison*: codes mis à jour via soustraction bidon
- *Accès aux codes*: avec `b.condition etiq`
- *Accès au report*: « `adc rd, rn, rm` » $\equiv$ $r_d \leftarrow r_n + r_m + C$

5. Accès aux données

Adresses

- *Numérique*: entier non négatif, souvent en hexadécimal
- *Symbolique*: chaîne représentant une adresse à déterminer

Modes d'adressage

- *Mode*: méthode pour récupérer la valeur d'un opérande
- *Récapitulatif des modes*:

Nom	Valeur récupérée	Exemple
immédiat	$i \mapsto i$	`mov x0, 42`
direct	$a \mapsto \text{mem}[a]$	—
par registre	$n \mapsto \text{reg}[n]$	`mov x0, x1`
indirect	$a \mapsto \text{mem}[\text{mem}[a]]$	—
indirect par registre	$n \mapsto \text{mem}[\text{reg}[n]]$	`ldr x0, [x1]`
indir. par reg. indexé	$n, i \mapsto \text{mem}[\text{reg}[n]+i]$	`ldr x0, [x1, i]`
indir. par reg. indexé pré-incrémenté	$\text{reg}[n] \leftarrow \text{reg}[n]+i$, $n, i \mapsto$ suivi de $\text{mem}[\text{reg}[n]]$	`ldr x0, [x1, i]!`
indir. par reg. indexé post-incrémenté	$n, i \mapsto \text{mem}[\text{reg}[n]]$, $\text{reg}[n] \leftarrow$ suivi de $\text{reg}[n]+i$	`ldr x0, [x1], i`
relatif	$i \mapsto \text{mem}[\text{reg}[pc]+i]$	`ldr x0, var`

Accès mémoire sur ARMv8

- *Chargement et stockage*:

# octets	chargement	stockage
1	`ldrb wd, a`	`strb wd, a`
2	`ldrh wd, a`	`strh wd, a`
4	`ldr wd, a`	`str wd, a`
8	`ldr xd, a`	`str xd, a`

- *Autres instructions*:

```
adr r, etiq   // charge adr(etiq) dans reg. r
mov r, s      // charge reg. s    dans reg. r
mov r, i      // charge valeur i  dans reg. r
```

Assemblage

- *Assembleur*: instructions $\to$ code machine; la plupart des adresses symboliques $\to$ adresses numériques
- *Éditeur de liens*: fichiers objets $\to$ fichier exécutable; recalcule certaines adresses; adresses symboliques $\to$ numériques

6. Tableaux

Généralités

- *Tableau*: collection d'éléments identifiés par des indices
- *Éléments*: tous de même taille, contigus en mémoire
- *Indice*: d-uplet $\boldsymbol{i}$ où $d \geq 1$ est la dimension
- *Bornes*: $0 \leq \boldsymbol{i}_j < n_j$ pour chaque dimension j
- *Taille*: $n_0 \cdot n_1 \cdots n_{d-1}$ éléments
- *Types*: le type des éléments est implicite
- *Exemples de tableau 1D et tableau 2D*:

0	01010101
1	11110000
2	01101101
3	11111111
4	11110101

$n_0 = 5$
5 éléments

$(0,0)$	2
$(0,1)$	33
$(1,0)$	65535
$(1,1)$	73
$(2,0)$	9000
$(2,1)$	255

$n_0 = 3$, $n_1 = 2$
6 éléments

Calcul d'adresse

- *Index*: adresse relative à laquelle est stocké un élément
- *Calcul*: si a = adresse du tableau et k = nombre d'octets d'un élément, alors l'adresse d'un élément correspond à:

$$\underbrace{a + \underbrace{i \cdot k}}_{\text{index élém. } i \text{ (tableau 1D)}} \qquad \underbrace{a + \underbrace{(i \cdot n_1 + j) \cdot k}}_{\text{index élém. } (i,j) \text{ (tableau 2D)}}$$

Allocation/accès mémoire

- *Tableau non initialisé*:

```
.section ".bss"
        .align 2
tab:    .skip  3*2*2     // n0 * n1 * # octets
```

- *Tableau initialisé*:

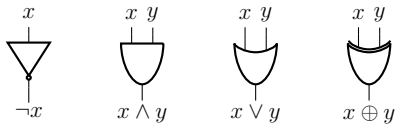
```
.section ".data"
tab:    .hword  2, 33, 65535, 73, 9000, 255 // six demi-mots
```

- *Accès*: avec `str` / `ldr` (ou variantes) + modes d'adressage

7. Circuits logiques

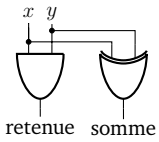
Circuits

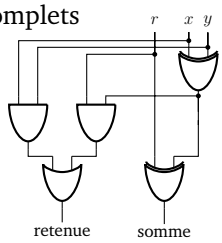
- « Blocs » de base constitués de portes logiques qui permettent d'implémenter l'ordinateur:



Arithmétique

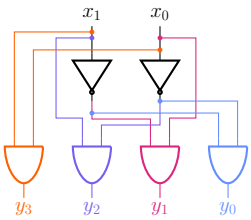
- *Demi-additionneur*: somme de deux bits
- *Additionneur complet*: somme de deux bits et d'une retenue
- *Addition*: somme sur n bits avec un demi-additionneur et une cascade de $n-1$ additionneurs complets

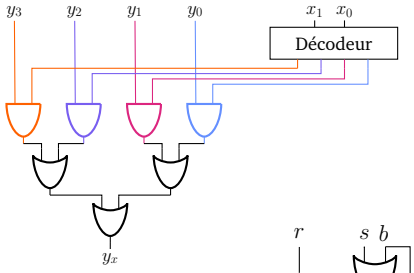




Décodage

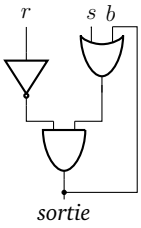
- *Décodeur*: sur entrée x, sortie: $y_x = 1$ et $y_j = 0$ pour $j \neq x$
- *Multiplexeur*: sur entrée x, sélectionne le bit y_x
- *Instructions*: décodables/exécutables à l'aide de tels circuits





Mémoire

- *Circuits séquentiels*: peuvent mémoriser des bits
- *Verrou*: stocke un bit b,
  remise à 0 avec r, et mise à 1 avec s



Annexe:

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`

Les manipulations de bits ne seront couvertes qu'après la relâche.

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`

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