IFT209 – Programmation système

NOTES DE COURS

Michael Blondin

31 mars 2020

Avant-propos

La rédaction de ce document a été entamée à la session d'hiver 2019 comme notes complémentaires du cours IFT209 – Programmation système de l'Université de Sherbrooke, dans le but d'offrir des notes de cours gratuites, faciles d'accès et pouvant évoluer au gré des sessions et des refontes du cours.

La structure et le contenu sont basés sur le plan cadre du cours. Par conséquent, elles suivent une structure souvent similaire aux diaporamas utilisés avant 2019 par les chargés de cours Vincent Ducharme et Mikaël Fortin, eux-mêmes basés en bonne partie sur l'ancien manuel de référence du cours rédigé par le professeur retraité Richard St-Denis: *L'architecture du processeur SPARC et sa programmation en langage d'assemblage* [SD11]. De plus, l'idée d'utiliser les architectures ARMv8 et du NES provient, à ma connaissance, de Mikaël Fortin.

Si vous trouvez des coquilles ou des erreurs dans le document, ou si vous avez des suggestions, n'hésitez pas à me les indiquer sur GitHub (en ajoutant un « *issue* ») ou par courriel à `michael.blondin@usherbrooke.ca`.

Légende

Observation.

Les passages compris dans une région comme celle-ci correspondent à des observations jugées intéressantes mais qui dérogent légèrement du contenu principal.

Remarque.

Les passages compris dans une région comme celle-ci correspondent à des remarques jugées intéressantes mais qui dérogent légèrement du contenu principal.

Les passages compris dans une région colorée sans bordure comme celle-ci correspondent à du contenu qui ne sera *pas* nécessairement présenté en classe, mais qui peut aider à une compréhension plus approfondie.

Les exercices marqués par « ★ » sont considérés plus avancés que les autres.
Les exercices marqués par « ★★ » sont difficiles ou dépassent le cadre du cours.

L'icône « » dans la marge fournit un lien vers le code source associé au passage.

Table des matières

1 Systèmes de numération **1**
- 1.1 Représentation des nombres . . . 1
  - 1.1.1 Système unaire . . . 1
  - 1.1.2 Système de numération positionnelle . . . 2
- 1.2 Changement de base . . . 5
  - 1.2.1 Base b vers base 10 . . . 5
  - 1.2.2 Base 10 vers base b . . . 5
  - 1.2.3 Base b vers base b' . . . 6
- 1.3 Addition . . . 7
- 1.4 Nombres fractionnaires . . . 8
- 1.5 Exercices . . . 10

2 Architecture des ordinateurs **11**
- 2.1 Architecture de von Neumann . . . 11
  - 2.1.1 Mémoire principale . . . 12
  - 2.1.2 Processeur . . . 17
  - 2.1.3 Unités d'entrée/sortie . . . 19
  - 2.1.4 Types d'architectures . . . 20
- 2.2 Organisation . . . 20
  - 2.2.1 Pipeline . . . 20
- 2.3 Exercices . . . 22

3 Programmation en langage d'assemblage **23**
- 3.1 Registres . . . 23
- 3.2 Un premier programme . . . 24
  - 3.2.1 Calcul de $f(n)$ . . . 25
  - 3.2.2 Affichage du contenu d'un registre . . . 26
  - 3.2.3 Lecture d'une valeur dans un registre . . . 26
  - 3.2.4 Vers un premier programme . . . 27
  - 3.2.5 Calcul du temps de vol . . . 28

3.2.6 Programme complet 29
3.3 Détails pratiques 30
3.3.1 Normes de programmation 30
3.3.2 Segments de données 32
3.3.3 Spécificateurs de format 32
3.4 Exercices 34

4 Accès aux données **35**
4.1 Adresses 35
4.2 Adressage 36
4.2.1 Immédiat 36
4.2.2 Direct 37
4.2.3 Par registre 37
4.2.4 Indirect 37
4.2.5 Indirect par registre 37
4.2.6 Indirect par registre indexé 38
4.2.7 Indirect par registre pré/post-incrémenté 38
4.2.8 Relatif 38
4.2.9 Sommaire des modes d'adressage 39
4.3 Particularités de l'architecture ARMv8 39
4.4 Assemblage d'un programme 40
4.5 Exercices 41

5 Circuits logiques **42**
5.1 Arithmétique 42
5.1.1 Addition de deux bits 42
5.1.2 Addition de deux nombres 44
5.2 Décodage du jeu d'instructions 45
5.3 Mémoire 47
5.4 Exercices 48

6 Nombres entiers **49**
6.1 Représentation des entiers signés 49
6.2 Addition 51
6.2.1 Report 51
6.2.2 Débordement 52
6.3 Soustraction 53
6.4 Multiplication 53
6.4.1 Multiplication non signée 53
6.4.2 Multiplication signée 55
6.5 Division 56
6.5.1 Division non signée 56
6.5.2 Division signée 57
6.6 Particularités de l'architecture ARMv8 57
6.6.1 Codes de condition 57
6.6.2 Accès mémoire 58

6.7 Exercices 59

7 Tableaux **60**
7.1 Accès aux éléments 61
7.1.1 Cas unidimensionnel 62
7.1.2 Cas bidimensionnel 63
7.2 Particularités de l'architecture ARMv8 63
7.2.1 Allocation et initialisation 63
7.2.2 Parcours d'un tableau 64
7.3 Autre exemple: tableaux de pointeurs 67
7.4 Exercices 70

8 Programmation structurée **71**
8.1 Structures de contrôle 71
8.1.1 Séquence 71
8.1.2 Sélection 72
8.1.3 Itération 74
8.2 Sous-programmes 76
8.2.1 Paramètres et appel 76
8.2.2 Retour 77
8.2.3 Sauvegarde des registres 78
8.3 Autres particularités de l'architecture ARMv8 79
8.3.1 Distance des adresses 79
8.3.2 Assignation par sélection 79
8.4 Exercices 80

9 Valeurs booléennes et chaînes de bits **81**
9.1 Algèbre de Boole 81
9.2 Représentation des valeurs booléennes 82
9.3 Manipulation de bits 82
9.3.1 Opérateurs logiques 83
9.3.2 Décalages logiques 84
9.3.3 Décalages circulaires 85
9.3.4 Décalages arithmétiques 86
9.4 Masquage 87
9.5 ★ Cryptographie visuelle 88
9.5.1 Format PBM 89
9.5.2 Implémentation 89
9.6 Exercices 91

10 Chaînes de caractères **92**
10.1 ASCII 92
10.2 ISO 8859-1 (Latin-1) 93
10.3 UTF-8 95
10.4 Chaînes de caractères 96
10.5 Exercices 97

11 Sous-programmes et mémoire **99**
11.1 Pile d'exécution 99
11.1.1 Appels de sous-programmes 99
11.1.2 Disposition de la mémoire 99
11.1.3 Fonctionnement de la pile 101
11.1.4 Sauvegarde et restauration 101
11.2 Récursion 102
11.3 Limitations 104
11.4 Exercices 105

12 Nombres en virgule flottante **107**
12.1 Représentation 107
12.2 Précision 108
12.2.1 Erreur d'approximation 109
12.3 Arithmétique 110
12.3.1 Addition 110
12.3.2 Multiplication 111
12.4 Norme IEEE 754 112
12.4.1 Codage des formats 113
12.5 Particularités de l'architecture ARMv8 114
12.5.1 Registres 114
12.5.2 Instructions 115
12.5.3 Exemple de programme 116
12.6 Exercices 118

13 Introduction aux entrées/sorties: NES **119**
13.1 Architecture du NES 119
13.1.1 Organisation de la mémoire 120
13.2 Registres 122
13.3 Jeu d'instructions 122
13.3.1 Valeurs immédiates 122
13.3.2 Modes d'adressage 123
13.3.3 Accès mémoire 123
13.3.4 Arithmétique 123
13.3.5 Logique 124
13.3.6 Comparaisons et branchements 124
13.4 Sorties graphiques 124
13.4.1 Tuiles 124
13.4.2 Affichage de tuiles 125
13.5 Entrées à partir des manettes 126
13.6 Exemple de programme simple 126
13.7 Exercices 130

14 Entrées/sorties **131**
14.1 Attente active 131
14.2 Interruptions 132

14.2.1 Gestionnaires d'interruption 132
14.2.2 Traitement des interruptions 133
14.2.3 Niveaux de priorité 135
14.2.4 Interruptions logicielles 136
14.3 Accès direct à la mémoire 136
14.4 Appels système 137
14.5 Exercices 140

A Solutions des exercices **141**

B Fiches récapitulatives **151**

C Architecture ARMv8: sommaire **157**

D Architecture du NES: sommaire **164**

Bibliographie **167**

Index **168**

1

Systèmes de numération

Les nombres forment l'un des types élémentaires de données de l'ordinateur et ils surgissent dans le développement de la quasi totalité des programmes. Afin de mieux comprendre leur implémentation et leur utilisation, nous explorons différentes façons élémentaires de les représenter, manipuler et convertir. Nous nous limitons pour l'instant aux nombres entiers et fractionnaires non négatifs. Nous couvrirons les entiers négatifs et les nombres réels aux chapitres 6 et 12 respectivement.

1.1 Représentation des nombres

1.1.1 Système unaire

L'un des plus anciens systèmes de numération, et probablement le plus simple, est le *système unaire*, où chaque nombre entier $n \in \mathbb{N}$ est représenté en répétant n fois un même symbole arbitraire σ. Nous utilisons parfois le système unaire lorsque nous comptons, par exemple, avec les doigts, des traits, des entailles ou bien des bâtonnets. Par exemple, si $\sigma = |$, alors:

$$\begin{array}{ll} 1 \text{ s'écrit:} & | \\ 5 \text{ s'écrit:} & ||||| \\ n \text{ s'écrit:} & \underbrace{||\cdots|}_{n \text{ fois}}. \end{array}$$

Le symbole $\sigma = 1$ est aussi souvent utilisé puisqu'il préserve quelques propriétés de la notation positionnelle que nous verrons plus tard. Ainsi, avec ce choix de symbole:

$$\begin{array}{ll} 1 \text{ s'écrit:} & 1 \\ 5 \text{ s'écrit:} & 11111 \\ n \text{ s'écrit:} & \underbrace{11\cdots1}_{n \text{ fois}}. \end{array}$$

Le nombre 0 ne peut être représenté dans ce système que par l'absence de symbole. Notons que le système unaire est aussi en quelque sorte utilisé dans l'arithmétique de Peano, où chaque nombre est ou bien le symbole spécial 0, ou bien le successeur d'un autre nombre:

$$\begin{aligned} 1 &\stackrel{\text{déf}}{=} \text{succ}(0) \\ 2 &\stackrel{\text{déf}}{=} \text{succ}(\text{succ}(0)) \\ n &\stackrel{\text{déf}}{=} \underbrace{\text{succ}(\text{succ}(\cdots \text{succ}(0)))}_{n \text{ fois}}. \end{aligned}$$

La plupart des opérations sont particulièrement simples à implémenter dans le système unaire. Par exemple, l'addition correspond à la concaténation:

$$3 + 5 = 111 \cdot 11111 = 11111111.$$

L'une des lacunes considérable du système unaire est son manque de concision: la représentation d'un nombre $n \in \mathbb{N}$ requiert n symboles. D'autres systèmes de numération plus concis ont été inventés au fil du temps. Par exemple, la *numération romaine* peut être vue comme une extension du système unaire où l'on contracte certaines répétitions de symboles par d'autres symboles. Ce système abrège notamment IIIII par V, et VV par X. Ainsi, le nombre 16 s'écrit avec seulement trois symboles (XVI) plutôt que seize symboles (IIIIIIIIIIIIIIII). Cette concision a un coût; les opérations arithmétiques sont plus complexes que dans le système unaire de base; par ex. pensez à un algorithme pour l'addition.

1.1.2 Système de numération positionnelle

En comparaison aux système unaire et romain, le système de numération que vous utilisez probablement chaque jour, le *système décimal*, permet à la fois d'être concis et d'implémenter efficacement les opérations arithmétiques.

Ce système fait partie de la famille plus générale des *systèmes de numération positionnelle*. Dans celle-ci, une *base* $b \in \mathbb{N}_{\geq 2}$ est fixée et chaque nombre est constitué de symboles, appelés *chiffres*, parmi $\{0, 1, \ldots, b-1\}$. La position de chaque chiffre correspond à une *puissance* de b. Un *nombre* est une séquence non vide de chiffres de la forme $x_{n-1} \cdots x_1 x_0 \in \{0, 1, \ldots, b-1\}^n$. La valeur d'une telle séquence x est dénotée x_b et est définie par:

$$(x_{n-1} \cdots x_1 x_0)_b \stackrel{\text{déf}}{=} \sum_{i=0}^{n-1} x_i \cdot b^i.$$

Exemple.

Dans le système décimal, c.-à-d. où $b = 10$, nous avons:

$$\begin{aligned} 103_{10} &= 1 \cdot 10^2 + 0 \cdot 10^1 + 3 \cdot 10^0 = 103, \\ 564_{10} &= 5 \cdot 10^2 + 6 \cdot 10^1 + 4 \cdot 10^0 = 564. \end{aligned}$$

Si l'on utilise plutôt la base $b = 7$, nous obtenons:

$$\begin{aligned} 103_7 &= 1 \cdot 7^2 + 0 \cdot 7^1 + 3 \cdot 7^0 = 52, \\ 564_7 &= 5 \cdot 7^2 + 6 \cdot 7^1 + 4 \cdot 7^0 = 291. \end{aligned}$$

Notons que l'ajout du chiffre 0 à gauche d'un nombre ne change pas sa valeur puisque cela ajoute zéro fois une puissance de b, donc zéro. Autrement dit, nous avons $x_b = (0x)_b = (00x)_b = (00 \cdots 0x)_b$, peu importe les valeurs de x et b. Nous disons que de tels zéros sont *non significatifs*.

Le système de numération positionnelle est *exponentiellement* plus concis que le système unaire: tout nombre se représente avec une quantité logarithmique de chiffres. En effet, considérons un nombre x représenté par n chiffres sans zéro significatif. Nous avons $x \geq b^{n-1}$ et ainsi $\log_b(x) + 1 \geq n$. Par conséquent:

Proposition 1. *La plus petite quantité de chiffres permettant de représenter un nombre $x \in \mathbb{N}_{\geq 1}$ en base b est $\lfloor \log_b(x) + 1 \rfloor$.*

Remarquons que le plus grand nombre formé de n chiffres dans le système décimal est $10^n - 1$. Par exemple, pour $n = 3$, le plus grand nombre est $999 = 10^3 - 1$. Cette observation se généralise à une base arbitraire:

Proposition 2. *Soit $b \in \mathbb{N}_{\geq 2}$ une base et soit $n \in \mathbb{N}_{\geq 1}$. Le plus grand nombre pouvant être représenté en base b avec n chiffres est $b^n - 1$.*

Démonstration. Soit $m_{b,n}$ le plus grand nombre représentable en base b avec n chiffres. Nous montrons que $m_{b,n} = b^n - 1$ par induction sur n.

Cas de base. Si $n = 1$, alors le plus grand nombre est la valeur du plus grand chiffre, c'est-à-dire $b - 1$. Nous avons donc bien $m_{b,n} = b - 1 = b^1 - 1$.

Étape d'induction. Supposons que $n > 1$ et $m_{b,n} = b^n - 1$. Par définition de valeur en base b, le plus grand nombre formé de $n + 1$ chiffres est obtenu en concaténant $(b-1)$ à gauche du plus grand nombre formé de n chiffres. Ainsi:

$$\begin{aligned} m_{b,n+1} &= (b-1) \cdot b^n + m_{b,n} \\ &= (b-1) \cdot b^n + (b^n - 1) && \text{(par hypothèse d'induction)} \\ &= b^{n+1} - b^n + b^n - 1 \\ &= b^{n+1} - 1. \end{aligned}$$

□

Systèmes binaire, hexadécimal et octal. Le *système binaire* est celui utilisé dans essentiellement tous les ordinateurs en raison de sa simplicité; il ne possède que deux chiffres: 0 et 1. Il s'agit de l'instance du système de numération positionnelle où $b = 2$. Dans ce système, les chiffres sont appelés *bits*. Ainsi, une séquence de n bits peut représenter un nombre compris entre 0 et $2^n - 1$ inclusivement.

Le *système hexadécimal* est également répandu en informatique, notamment pour représenter des séquences de bits de façon plus succincte, par ex. les adresses en mémoire ou encore les codes de couleur. Il s'agit de l'instance du système de numération positionnelle où $b = 16$. Afin d'éviter toute ambiguïté, les chiffres $10, 11, \ldots, 15$ sont remplacés par $\mathrm{A}, \mathrm{B}, \ldots, \mathrm{F}$ respectivement. Bien qu'il s'agisse de lettres dans l'alphabet latin, nous les considérons comme des chiffres dans ce contexte.

Exemple.

Nous avons $8\mathrm{B}5_{16} = 8 \cdot 16^2 + 11 \cdot 16^1 + 5 \cdot 16^0 = 2229$.

Le *système octal* est aussi occasionnellement utilisé en informatique et est supporté par certains langages de programmation. Il s'agit de l'instance du système de numération positionnelle où $b = 8$.

Décimal	Binaire	Hexadécimal	Octal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17
16	10000	10	20
17	10001	11	21
18	10010	12	22
19	10011	13	23
20	10100	14	24

FIGURE 1.1 – Nombres de 0 à 20 écrits en bases 10, 2, 16 et 8.

Les 21 premiers nombres naturels décrits dans les bases 10, 2, 16 et 8 apparaissent à la figure 1.1.

1.2 Changement de base

Il s'avère parfois pratique de convertir un nombre d'un système de numération positionnelle d'une certaine base vers une autre base. Nous expliquons comment effectuer un tel changement de façon algorithmique.

1.2.1 Base b vers base 10

Soit $x = x_{n-1} \cdots x_1 x_0$ un nombre décrit en base b avec n chiffres. Il est possible de convertir x en base 10 en évaluant simplement la somme:

$$\sum_{i=0}^{n-1} x_i \cdot b^i.$$

Notons que si ce calcul est implémenté de façon naïve, il requiert $1+2+\ldots+n = n(n+1)/2$ multiplications et $n-1$ additions. Il est possible d'obtenir la même valeur avec $n-1$ multiplications et $n-1$ additions en évaluant la somme:

$$x_0 + b \cdot (x_1 + b \cdot (x_2 + b \cdot (\ldots + x_{n-1} \cdot b_{n-1}))).$$

Exemple.

Le nombre binaire 10110 donne la valeur suivante en décimal:

$$\begin{aligned} 0 + 2 \cdot (1 + 2 \cdot (1 + 2 \cdot (0 + 2 \cdot 1))) &= 0 + 2 \cdot (1 + 2 \cdot (1 + 2 \cdot 2)) \\ &= 0 + 2 \cdot (1 + 2 \cdot 5) \\ &= 0 + 2 \cdot 11 \\ &= 22. \end{aligned}$$

1.2.2 Base 10 vers base b

Soit x un nombre décrit en base 10. On peut convertir x vers la base b en divisant itérativement x par b jusqu'à l'obtention de 0. Le *reste* de chaque division entière donne un chiffre du nombre résultant (de droite à gauche). La procédure générale de conversion est décrite à l'algorithme 1.

Exemple.

Le nombre 22 vaut 10110 en binaire, puisque:

$$\begin{aligned} 22 \div 2 &= 11 \text{ reste } 0, \\ 11 \div 2 &= 5 \text{ reste } 1, \\ 5 \div 2 &= 2 \text{ reste } 1, \\ 2 \div 2 &= 1 \text{ reste } 0, \\ 1 \div 2 &= 0 \text{ reste } 1. \end{aligned}$$

Algorithme 1 : Conversion d'un nombre décimal vers une base b.

Entrées : base $b \in \mathbb{N}_{\geq 2}$ et un nombre x décrit en base 10
Sorties : x décrit en base b

$y \leftarrow \varepsilon$ `// ε dénote la séquence vide`
faire
  $y \leftarrow (x \bmod b) \cdot y$ `// · dénote la concaténation`
  $x \leftarrow x \div b$
tant que $x \neq 0$
retourner y

1.2.3 Base b vers base b'

Une approche simple afin de convertir un nombre d'une base b vers une autre base b' consiste à passer de la base b vers la base 10, puis de la base 10 vers la base b', en suivant les deux algorithmes décrits plus tôt. Lorsque l'une des bases est une *puissance* de l'autre, il existe des approches plus simples. Cela s'avère pratique notamment avec les systèmes binaire, hexadécimal et octal.

Base b vers base b^m. Soit x un nombre décrit en base b et soit $m \in \mathbb{N}_{>1}$. Afin de convertir x en base b^m, nous procédons en deux étapes:

— les chiffres de x sont regroupés en blocs de taille m, de la droite vers la gauche, en ajoutant des 0 non significatifs tout à gauche au besoin;

— chaque bloc est remplacé par le chiffre correspondant en base b^m.

Exemple.

Cherchons à convertir le nombre binaire 1010110001 en hexadécimal. Remarquons que le système hexadécimal possède la base $16 = 2^4$. Ainsi,

nous avons $b = 2$ et $m = 4$, ce qui mène à:

$$1010110001 \to 0010\ 1011\ 0001 \to \text{2B1}.$$

Base b^m vers base b. Soit $m \in \mathbb{N}_{>1}$ et soit x un nombre décrit en base b^m. Afin de convertir x en base b, nous *éclatons* chaque chiffre de x vers sa représentation de taille m en base b, puis nous effaçons les 0 non significatifs à gauche.

Exemple.

Le nombre hexadécimal 2B1 est converti en nombre binaire ainsi:

$$\text{2B1} \to 0010\ 1011\ 0001 \to 1010110001.$$

Base b^m vers base b^k. Afin de convertir un nombre de la base b^m vers la base b^k, où $m \neq k$, on peut passer par la base intermédiaire b, plutôt que 10. En effet, il suffit de convertir de la base b^m vers la base b, puis de la base b vers la base b^k à l'aide des deux procédures décrites plus tôt.

1.3 Addition

Voyons comment additionner deux nombres décrits dans une base commune b. Nous couvrirons la soustraction, la multiplication et la division plus tard au chapitre 6. Soient x et y deux nombres de n chiffres décrits en base b. La somme de x et y s'effectue de façon analogue à l'addition en base 10; c.-à-d. itérativement, de droite à gauche, en faisant la somme des chiffres en base b et en propageant une retenue.

Exemple.

L'addition $\text{D40C}_{16} + \text{6FA5}_{16} = \text{143B1}_{16}$ se calcule selon ces cinq étapes:

$$\begin{array}{r} {\scriptstyle 1} \\ \text{D40C} \\ +\ \text{6FA5} \\ \hline 1 \end{array} \quad \begin{array}{r} {\scriptstyle 1} \\ \text{D40C} \\ +\ \text{6FA5} \\ \hline \text{B1} \end{array} \quad \begin{array}{r} {\scriptstyle 1\ 1} \\ \text{D40C} \\ +\ \text{6FA5} \\ \hline \text{3B1} \end{array} \quad \begin{array}{r} {\scriptstyle 1\,1\ 1} \\ \text{D40C} \\ +\ \text{6FA5} \\ \hline \text{43B1} \end{array} \quad \begin{array}{r} {\scriptstyle 1\,1\,1} \\ \text{D40C} \\ +\ \text{6FA5} \\ \hline \text{143B1} \end{array}$$

Il est possible d'additionner deux nombres dont le nombre de chiffres diffère. En effet, il suffit simplement d'ajouter des zéros non significatifs au nombre le plus court, ou, de façon équivalente, d'aligner les deux nombres à droite et d'interpréter les chiffres manquants comme des zéros.

La procédure générale d'addition est décrite à l'algorithme 2. Remarquons que l'algorithme d'addition utilise lui-même l'addition, ce qui paraît circulaire.

En fait, comme la retenue ne peut jamais excéder 1, nous avons $0 \leq r + x_i + y_i < 2b$. Il suffit donc de connaître une quantité finie de tables d'addition pour implémenter l'addition générale.

Algorithme 2 : Addition de deux nombres dans une base commune.

Entrées : deux nombres x et y décrits en base $b \in \mathbb{N}_{\geq 2}$
Sorties : $x + y$ décrit en base b

$m \leftarrow |x|$; $n \leftarrow |y|$ `// nombre de chiffres de` x `et` y
$z \leftarrow \varepsilon$; $r \leftarrow 0$ `// séq. vide et retenue nulle`
pour i ***de*** 0 ***à*** $\max(m, n) - 1$
  si $i \geq m$ **alors** $x_i \leftarrow 0$
  si $i \geq n$ **alors** $y_i \leftarrow 0$
  $s \leftarrow r + x_i + y_i$
  $z \leftarrow (s \bmod b) \cdot z$
  si $s \geq b$ **alors** $r \leftarrow 1$ `// retenue créée?`
  sinon $r \leftarrow 0$
si $r = 1$ **alors** $z \leftarrow 1 \cdot z$ `// ajouter la dernière retenue`
retourner z

1.4 Nombres fractionnaires

Le système de numération positionnelle peut être étendu naturellement afin de représenter les fractions. Soit b une base. Un *nombre fractionnaire* dans la base b est constitué de deux séquences x et y de n et k chiffres respectivement, et se dénote x,y. La séquence x est appelée la *partie entière* et la séquence y la *partie fractionnaire*. La valeur du nombre x,y dans la base b est définie par:

$$(x,y)_b \overset{\text{déf}}{=} \sum_{i=0}^{n-1} x_i \cdot b^i + \sum_{i=1}^{k} x_i \cdot b^{-i}.$$

Exemple.

Nous avons:

$$\begin{aligned}(110,101)_2 &= 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} \\ &= 4 + 2 + \frac{1}{2} + \frac{1}{8} \\ &= 6 + \frac{5}{8} \\ &= 6{,}625.\end{aligned}$$

En base 10, cette notation correspond au système décimal habituel. Notons que l'ajout de zéros à droite de la partie fractionnaire d'un nombre ne change pas sa valeur. Il s'agit donc de *zéros non significatifs*. Remarquons également qu'un nombre fractionnaire, pouvant être représenté dans une certaine base, n'est pas nécessairement représentable dans une autre base. Par exemple, le nombre décimal $0{,}1$ ne peut pas être représenté (de façon finie) en base 2. Ainsi, les méthodes de conversion décrites à la section 1.2 ne s'appliquent pas nécessairement. L'addition se fait telle que décrite pour les entiers à la section 1.3, en alignant les deux nombres à leur virgule.

1.5 Exercices

1.1) Expliquez comment effectuer la soustraction et la multiplication dans le système unaire.

1.2) Convertissez les nombres suivants:

- — 68301 de la base 9 vers la base 10,
- — 103678 de la base 10 vers la base 16,
- — 26654 de la base 7 vers la base 14,
- — 111011010 de la base 2 vers la base 16,
- — 865723 de la base 9 vers la base 3,
- — ABCDEF de la base 16 vers la base 8.

(*tiré du devoir 1 de l'hiver 2019*)

1.3) Quel est la plus petite quantité de bits nécessaire afin de représenter le nombre 2FEDCB_{16} en binaire? (*tiré de l'examen périodique de l'hiver 2019*)

1.4) Quel est le plus grand multiple de 5 pouvant être représenté par un nombre de 9 chiffres en base 4? (*tiré du devoir 1 de l'hiver 2019*)

1.5) Quel est le plus grand nombre inférieur à 1 pouvant être représenté par un nombre fractionnaire binaire possédant jusqu'à 8 bits avant la virgule et 6 bits après la virgule? (*tiré du devoir 1 de l'hiver 2019*)

1.6) Dites pourquoi il est impossible d'obtenir une retenue excédant 1 lors de l'addition de deux nombres dans une base commune.

1.7) ★ Montrez que $0{,}1$ ne peut pas être représenté (de façon finie) en base 2.

2

Architecture des ordinateurs

Dans ce chapitre, nous faisons un survol du fonctionnement de l'ordinateur moderne. Celui-ci dépend de deux aspects: son architecture et son organisation.

L'*architecture* réfère aux services fournis par les composants de l'ordinateur, comme le processeur et la mémoire principale:

— types de données et leur représentation;

— modes d'adressage et accès aux données;

— jeu d'instructions;

— mécanismes d'entrée/sortie.

L'*organisation* réfère quant à elle à la description physique des composants et de leurs connexions:

— organisation interne du processeur;

— séquencement des instructions;

— gestions des conflits de ressources;

— interface entre processeur, mémoire et périphériques;

— organisation hiérarchique de la mémoire.

Autrement dit, l'architecture est la *spécification* de l'ordinateur, alors que l'organisation est une *implémentation* de cette spécification. Il peut exister plusieurs implémentations d'une même architecture. Par exemple, les fabricants Intel et AMD développent tous deux des processeurs x86-64. Nous mettrons l'emphase sur l'*architecture* des ordinateurs.

2.1 Architecture de von Neumann

Le premier ordinateur d'usage général, l'ENIAC (*Electronic Numerical Integrator and Calculator*), a été conçu vers la fin des années 1940 par J. Presper Eckert et John Mauchly de l'Université de Pennsylvanie [PH17]. Cet ordinateur était utilisé pour calculer des tables de tir d'artillerie. L'ENIAC était programmable à

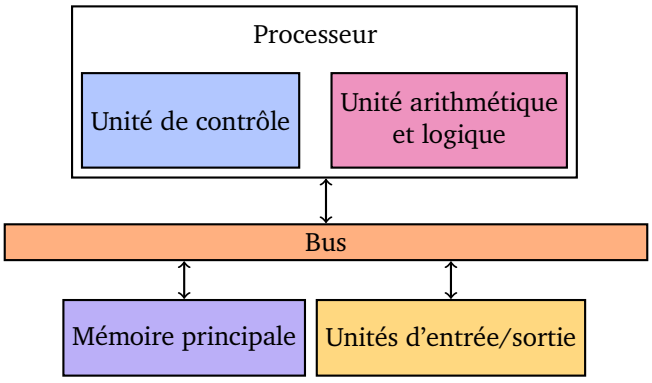


FIGURE 2.1 – Architecture de von Neumann. (Figure reproduite à partir de [SD11, Fig. 2.1])

l'aide de câbles et d'interrupteurs, alors que les données étaient entrées à l'aide de cartes perforées.

En 1944, Eckert et Mauchly cherchent déjà à simplifier l'entrée fastidieuse des programmes dans l'ENIAC. John von Neumann, qui se joint entre temps au groupe, écrit un mémo en 1945 sur une proposition d'ordinateur à programme enregistré, où programmes *et* données sont stockés dans la *même* mémoire. Un tel ordinateur, l'EDVAC (*Electronic Discrete Variable Automatic Computer*), sera construit quelques anneés plus tard [PH17].

Le modèle général de l'EDVAC, connu sous le nom d'*architecture de von Neumann*, est celui de la plupart des ordinateurs modernes. Dans cette architecture, un ordinateur est composé:

— d'un processeur constitué de registres, d'une unité de contrôle, et d'une unité arithmétique et logique;

— d'une mémoire principale qui stocke données et programmes;

— d'unités d'entrée/sortie.

Ces différents composants sont connectés par des systèmes de communication appelés *bus*. Le *bus interne* relie le processeur et la mémoire principale, alors que les *bus externes* relient l'ordinateur aux unités d'entrée/sortie. Ces composants sont illustrés à la figure 2.1.

2.1.1 Mémoire principale

La mémoire principale, qui correspond à la mémoire vive en pratique, stocke les programmes et leurs données. Elle peut être vue abstraitement comme une séquence $c_0, c_1, \ldots, c_{n-1}$ de n cellules. Chacune de ces cellules contient une donnée provenant d'un ensemble $\mathbb{D}$. L'index i de chaque cellule c_i est son *adresse* qui permet de l'identifier uniquement. Dans la plupart des architectures modernes, les celulles contiennent 8 bits; autrement dit, $\mathbb{D} = \{0,1\}^8$. Une telle

séquence de 8 bits se nomme un *octet*. Les bits d'un octet n'ont à priori aucune signification, leur interpétation est faite par un langage de programmation ou par la personne qui écrit un programme. Par exemple, l'octet `01000001` peut autant représenter le nombre 65 que le caractère `A`. Lors du chargement d'un programme, son code est stocké dans les cellules de la mémoire principale aux côtés de ses données (variables, constantes, etc.) On pourrait donc en théorie imaginer un programme qui manipule son propre code, par ex. un virus. La figure 2.2 illustre un contenu possible d'une mémoire principale.

0	`00000000`
1	`01011000`
2	`01000000`
3	`00001111`
4	`00011000`
5	`01010101`
6	`11110000`
7	`00001111`
⋮	⋮
$n-2$	`11111111`
$n-1$	`01100001`

FIGURE 2.2 – Exemple de contenu d'une mémoire principale. Chaque celulle est représentée par une case rectangulaire dont l'adresse apparaît à sa gauche.

Remarque.

Plusieurs outils, comme les débogueurs, affichent les adresses sous leur valeur *hexadécimale* plutôt que décimale, souvent avec le préfixe « `0x` ». Par exemple, l'adresse `0x0000555548ee` affichée par un tel outil correspond à l'adresse $0000555548\text{EE}_{16} = 1431652590$.

Granularité de l'accès mémoire. Bien qu'une adresse réfère typiquement à un octet, les architectures modernes permettent aussi d'interpréter une adresse comme faisant référence à plusieurs octets, souvent 2, 4 ou 8 octets. Par exemple, sous l'architecture ARMv8, ces unités se nomment:

	nombre de bits	***nombre d'octets***
octet	8 bits	1 octet
demi-mot	16 bits	2 octets
mot	32 bits	4 octets
double mot	64 bits	8 octets

Cette terminologie et le nombre d'octets adressables diffèrent d'une architecture à l'autre. À moins d'avis contraire, nous utiliserons la terminologie du tableau ci-dessus. Le contenu de l'octet, du demi-mot, du mot ou du double mot situé à l'adresse i couvre respectivement les adresses suivantes:

	adresses
octet	i
demi-mot	$i, i+1$
mot	$i, i+1, i+2, i+3$
double mot	$i, i+1, i+2, i+3, \ldots, i+7$

Exemple.

Considérons la mémoire principale illustrée à la figure 2.2. Le contenu de l'octet, du demi-mot, du mot et du double-mot situé à l'adresse 0 est respectivement:

```
00000000,
00000000 01011000,
00000000 01011000 01000000 00001111,
00000000 01011000 01000000 00001111 00011000 01010101 11110000 00001111,
```

ou, de façon équivalente, en hexadécimal:

```
00,
00 58,
00 58 40 0F,
00 58 40 0F 18 55 F0 0F.
```

Ordre des octets. L'interpétation des valeurs situées en mémoire dépend de l'ordre dans lequel les octets sont organisés sur l'architecture en question. Considérons une séquence d'octets $x_0 x_1 \cdots x_{n-1}$ obtenue à partir d'une adresse. Dans le format dit « *big-endian* », aussi appelé *gros-boutiste*, les octets sont organisés de gauche à droite, c.-à-d. de x_0 vers x_{n-1}. À l'inverse, dans le format dit « *little-endian* », aussi appelé *petit-boutiste*, les octets sont organisés de droite à gauche, c.-à-d. de x_{n-1} vers x_0.

Exemple.

Considérons le mot x situé à l'adresse 2 de la mémoire principale illustrée à la figure 2.2. Nous avons $x =$ `0058400F`. Ainsi, la valeur hexadécimale du mot x dans le format « big-endian » est $0058400F_{16}$, alors que dans le format « little-endian » il s'agit plutôt de $0F405800_{16}$.

Observons que la valeur se renverse au niveau des *octets* et non des chiffres, ainsi la valeur dans le format « little-endian » n'est *pas* $F0048500_{16}$.

L'architecture x86-64 utilise le format « little-endian », alors que ARMv8 supporte les deux formats. Nous utiliserons le format « little-endian » lors de la programmation sur l'architecture ARMv8. La plupart du temps, le format utilisé n'est pas perceptible. Il peut néanmoins être observé, par exemple, lorsqu'un programme lit une valeur octet par octet.

Exemple.

Le code C suivant[a] produit une sortie différente selon le format de l'architecture de la machine sur laquelle il est exécuté:

```c
#include <stdint.h>
#include <stdio.h>

union Mot
{
  uint32_t valeur;
  uint8_t  octets[4];
};

int main()
{
  union Mot mot;

  scanf("%X",   &mot.valeur);    // Sur entrée A1B2C3D4,
  printf("%02X", mot.octets[0]); // affiche:
  printf("%02X", mot.octets[1]); //   A1B2C3D4 si big-endian
  printf("%02X", mot.octets[2]); //   D4C3B2A1 si little-endian
  printf("%02X", mot.octets[3]);
}
```

a. Exemple basé sur un exemple de diaporamas de Vincent Ducharme.

Alignement en mémoire. Certaines architectures imposent des contraintes d'*alignement en mémoire*. Ces contraintes limitent les adresses auxquelles il est possible d'adresser plus d'un octet: il est seulement possible d'adresser 2^k octets aux adresses qui sont des multiples de 2^k. Par exemple, sous ces contraintes, on peut seulement adresser un mot aux adresses: $\{0, 4, 8, 12, \ldots\}$.

Remarquons qu'une adresse i est un multiple de 2^k si et seulement si les k bits de poids faible de sa représentation binaire sont égaux à zéro. Ainsi, une adresse sous notation hexadécimale respecte la contrainte d'alignement pour un *double mot* si elle se termine par 0 ou 8; pour un *mot* si elle se termine par 0, 4, 8 ou C; et pour un *demi-mot* si elle se termine par 0, 2, 4, 6, 8, A, C ou E.

Exemple.

L'adresse $0\text{A}5\text{C}_{16} = 101001011100_2$ satisfait les contraintes d'alignement pour l'adressage d'un mot ou d'un demi-mot, mais *pas* d'un double mot.

Sur certaines architectures sans contrainte d'alignement, l'adressage de 2^k octets à une adresse non alignée *ralentit* l'accès mémoire. Par exemple, considérons l'adressage du mot situé à l'adresse 3 tel qu'illustré à la figure 2.3. La valeur du mot est obtenue en deux accès: d'abord à l'adresse 0, puis à l'adresse 4. Les octets qui apparaissent dans la zone hachurée sont ensuite assemblés.

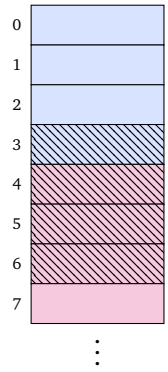


FIGURE 2.3 – Adressage d'un mot situé à l'adresse 3 (zone hachurée).

Taille de la mémoire. Les unités de mesure de la mémoire portent ces noms:

1 kilo-octet (Ko) $= 1000^1$ octets,	1 kibioctet (Kio) $= 1024^1$ octets,
1 mégaoctet (Mo) $= 1000^2$ octets,	1 mébioctet (Mio) $= 1024^2$ octets,
1 gigaoctet (Go) $= 1000^3$ octets,	1 gibioctet (Gio) $= 1024^3$ octets,
1 teraoctet (To) $= 1000^4$ octets,	1 tébioctet (Tio) $= 1024^4$ octets,
1 pétaoctet (Po) $= 1000^5$ octets,	1 pébioctet (Pio) $= 1024^5$ octets,
1 exaoctet (Eo) $= 1000^6$ octets,	1 exbioctet (Eio) $= 1024^6$ octets.

Remarque.

Dans le langage courant, on utilise souvent {kilo, méga, giga, tera, péta, exa}octets pour référer aux puissances de 1024, bien qu'elles réfèrent techniquement aux puissances de 1000.

Notons qu'une architecture, dont les adresses sont représentées sur k bits, peut accéder jusqu'à 2^k cellules de mémoire. Ainsi, une architecture 32 bits peut accéder à un maximum de:

$$2^{32} = 4 \cdot 2^{30} = 4 \cdot (2^{10})^3 = 4 \cdot 1024^3 = 4 \text{ gibioctets.}$$

En comparaison, une architecture 64 bits peut théoriquement accéder à plus d'un milliard de fois plus d'octets qu'une architecture 32 bits:

$$2^{64} = 16 \cdot 2^{60} = 16 \cdot (2^{10})^6 = 16 \cdot 1024^6 = 16 \text{ exbioctets.}$$

2.1.2 Processeur

Le *processeur* est l'unité centrale de traitement de l'ordinateur, il s'agit donc en quelque sorte du « cerveau » de l'ordinateur. Chaque processeur est associé à un *jeu d'instructions*: un ensemble fini d'instructions machines formant les opérations élémentaires pouvant être exécutées par l'ordinateur. L'*unité de contrôle* du processeur coordonne l'exécution des instructions machines. Son *unité arithmétique et logique* performe les opérations arithmétique et logique nécessaires à l'exécution des instructions. Le processeur peut communiquer avec la mémoire principale via certaines instructions, et possède également une petite quantité de mémoire interne connue sous le nom de *registres*.

Registres. Le processeur possède ses propres cellules de mémoire: les *registres*. Ceux-ci servent à stocker les opérandes et le résultat des instructions machines.

L'accès aux registres ne dépend pas du bus interne et est donc beaucoup plus rapide que l'accès à la mémoire principale. Toutefois, le processeur compte très peu de registres; par ex. 4 registres sur l'architecture NMOS 6502 et 31 registres sur l'architecture ARMv8. Le nombre de bits pouvant être stockés dans un registre varie d'une architecture à l'autre; par exemple, 8 bits sur l'architecture NMOS 6502 et jusqu'à 64 bits sur les architectures ARMv8, RISC-V et x86-64.

Jeu d'instructions. Le *jeu d'instructions* d'une architecture décrit les instructions élémentaires de l'ordinateur. Chaque instruction possède cette forme:

Code d'opération	opérande 1, opérande 2, …, opérande k

Le nombre d'opérandes varie d'une instruction à l'autre, et dans certain cas il peut simplement n'y avoir aucun opérande.

Exemple.

L'instruction « `add x, y, z` » des architectures ARMv8 et RISC-V additionne le contenu des registres `y` et `z`, et socke leur somme dans le registre `x`. Elle possède donc trois opérandes.

L'instruction `nop` n'effectue rien et ne possède aucune opérande.

Un jeu d'instructions possède normalement plusieurs instructions de différents types: arithmétique, logique, contrôle, accès mémoire, etc. Nous verrons plusieurs telles instructions aux chapitres subséquents.

Chaque instruction se traduit en *code machine*: une suite de bits interprétable par le processeur. Selon l'architecture, le nombre de bits requis afin de représenter une instruction en code machine peut varier selon l'instruction.

Exemple.

Imaginons une architecture (fictive) qui possède huit registres, nommés $\texttt{x}_0$, …, $\texttt{x}_7$, et ces quatre instructions:

```
foo a, b, c
bar a, b
baz a, b, c
qux
```

L'architecture pourrait représenter le code d'opération des instructions par 00, 01, 10 et 11 respectivement; et représenter chaque opérande par les trois bits qui correspondent au numéro du registre. Ainsi, « `baz x2, x5, x7` » se traduirait vers « `10 010 101 111` » en code machine; alors que « `qux` » se traduirait simplement vers « `11` ».

Alternativement, l'architecture pourrait utiliser un code machine à taille *fixe*, par ex. en utilisant toujours $2+3\cdot 3 = 11$ bits et en ajoutant des zéros au besoin. Ainsi, « `baz x2, x5, x7` » se traduirait encore vers « `01 010 101 111` »; alors que « `qux` » se traduirait maintenant vers « `11 000 000 000` ». Sous ce codage à taille fixe, plusieurs codes seraient invalides, par ex. « `11 111 111 11` ».

Sur les architecture ARMv8 et RISC-V, *toutes* les instructions sont de *taille fixe*. Chaque instruction est traduite vers une suite de 32 bits dans un format de bits bien précis.

Exemple.

Sur l'architecture ARMv8, « `add x10, x11, x12` » se traduit vers le code machine $\texttt{B0C016A}_{16}$, ou plus précisément en binaire:

$$\texttt{1 0 0 01011 00 0 }\underbrace{\texttt{01100}}_{\texttt{x}_{12}}\texttt{ 000000 }\underbrace{\texttt{01011}}_{\texttt{x}_{11}}\ \underbrace{\texttt{01010}}_{\texttt{x}_{10}}\texttt{.}$$

Quinze bits indiquent les opérandes de l'instructions; le tout premier bit indique si l'addition doit se faire en arithmétique 64 bits ou non; la séquence « `0 0 01011` » est tout simplement fixée, et les autres bits représentent des arguments facultatifs que nous verrons plus tard [ARM18, *ADD (shifted register)*].

Unité de contrôle. L'*unité de contrôle* coordonne le fonctionnement du processeur. Considérons le programme suivant stocké dans la mémoire principale:

i	`add x10, x11, x12`
$i+1$	`add x10, x10, x13`

Le processeur possède un registre spécial nommé *compteur d'instruction*; aussi appelé « *program counter* » (PC) en anglais. Ce registre pointe initialement en mémoire à la première ligne du programme, ici l'adresse i. L'unité de contrôle exécute la ligne indiquée par le compteur d'instruction, puis incrémente ce compteur. Lors de l'exécution de l'instruction « `add x10, x11, x12` », l'unité de contrôle indique à l'unité arithmétique et logique d'additionner les registres $\texttt{x}_{11}$ et $\texttt{x}_{12}$, puis s'occupe de stocker le résultat dans $\texttt{x}_{10}$. L'unité de contrôle incrémente ensuite le compteur d'instruction à $i+1$, demande à l'unité arithmétique et logique de performer l'addition de $\texttt{x}_{10}$ et $\texttt{x}_{13}$, et stocke la somme dans $\texttt{x}_{10}$. Après l'exécution de ces deux instructions, le registre $\texttt{x}_{10}$ contient la somme du contenu des registres $\texttt{x}_{11}$, $\texttt{x}_{12}$ et $\texttt{x}_{13}$.

En général, l'unité arithmétique et logique n'est pas le seul composant avec lequel l'unité de contrôle interagit. Par exemple, l'unité de contrôle doit interagir avec la mémoire principale lors d'une instruction de lecture ou d'écriture en mémoire. De plus, comme nous le verrons à la section 2.2.1, l'unité de contrôle fonctionne souvent en plusieurs étapes qui peuvent être parallélisées.

Remarque.

Le compteur d'instruction n'est pas toujours accessible. Par exemple, il l'est sous le nom `pc` sur l'architecture RISC-V, mais ne l'est pas sur l'architecture ARMv8.

Unité arithmétique et logique. L'*unité arithmétique et logique (UAL)* est le composant du processeur en charge d'effectuer les calculs sur les:

— *nombres entiers*: addition, soustraction, multiplication, division entière et comparaison;

— *séquences de bits*: ET logique, OU logique, OU exclusif, négation, décalages, décalages circulaires, inversion, etc.

Le processeur peut également posséder d'autres unités de calcul comme une *unité de calcul en virgule flottante* dédiée à l'arithmétique en virgule flottante.

2.1.3 Unités d'entrée/sortie

Les unités d'entrée/sortie contrôlent les périphériques comme le disque dur, le moniteur, la souris, le clavier, les lecteurs, la webcam, etc. L'échange de données entre le processeur et une unité d'entrée/sortie se fait grâce à un protocole de communication via un bus externe. Cette communication est particulièrement lente en comparaison à l'accès aux registres et à la mémoire principale. Nous traiterons des entrées/sorties au chapitre 13.

2.1.4 Types d'architectures

Il existe deux grandes familles d'architectures modernes: *RISC* et *CISC*. Il n'existe pas de définition claire de celles-ci, mais en général les architectures RISC sont caractérisées par un jeu d'instructions constitué:

— de *peu* d'instructions;

— d'instructions relativement *simples*;

— d'instructions qui *ne combinent pas les accès mémoire* à d'autres types d'opérations (comme l'arithmétique);

— d'instructions dont la traduction vers le code machine est de *taille fixe*.

Exemple.

Les architectures ARMv8 et RISC-V sont de type RISC, l'architecture x86-64 est de type CISC, et on peut difficilement classifier NMOS 6502.

2.2 Organisation

2.2.1 Pipeline

L'exécution d'une instruction par le processeur se fait souvent en plusieurs étapes. Par exemple, le processeur:

1. récupère l'instruction à partir de la mémoire principale;
2. décode le code d'opération et les opérandes de l'instruction;
3. charge les opérandes nécessaires à partir des registres ou de la mémoire;
4. exécute l'instruction;
5. stocke le résultat de l'instruction.

Ce type d'exécution est illustré à la figure 2.4.

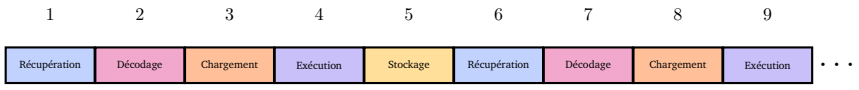


FIGURE 2.4 – Exécution séquentielle d'instructions en cinq étapes.

Afin d'augmenter la vitesse d'exécution de plusieurs instructions, les processeurs utilisent souvent un *pipeline* tel qu'illustré à la figure 2.5. Cela permet de paralléliser l'exécution des différentes étapes d'exécution, à la manière d'une chaîne de production.

L'utilisation d'un pipeline peut créer plusieurs problèmes qui doivent être résolus par le processeur. Par exemple, si une instruction x écrit une valeur en

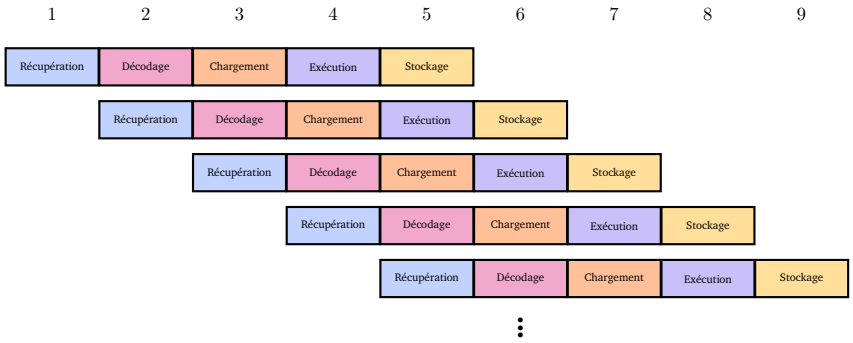


FIGURE 2.5 – Pipeline à cinq étages. L'exécution de n instructions prend $n + 4$ cycles, plutôt que les $5n$ cycles d'une exécution purement séquentielle.

mémoire à l'adresse i, et qu'une instruction subséquente y lit la valeur à la même adresse i, alors la valeur lue par l'instruction y risque d'être erronée puisqu'elle n'aura pas encore été mise à jour par x. Dans ce cas, le processeur peut, par exemple, décider de retarder l'exécution de y, ce qui mitige les avantages du pipeline.

Les instructions qui effectuent des sauts peuvent aussi causer des problèmes. Par exemple, supposons que la première instruction x de la figure 2.5 effectue un saut au $4^{\text{ème}}$ cycle vers une instruction z située plus loin en mémoire. Au $5^{\text{ème}}$ cycle, la seconde instruction y est exécutée. Cela ne devrait pas être le cas; c'est plutôt l'instruction z qui devrait être exécutée après x. Le processeur doit donc corriger la situation. Pour pallier à ce problème, les processeurs utilisent différentes heuristiques de prédiction de branchement afin de « deviner » si un saut sera effectué ou non.

2.3 Exercices

2.1) Considérons le contenu suivant de la mémoire principale:

adresse	contenu
0	01_{16}
1	$A4_{16}$
2	BC_{16}
3	48_{16}
4	$5F_{16}$
5	11_{16}
6	FF_{16}
7	43_{16}
⋮	⋮

Quelle est la valeur binaire du mot stocké à l'adresse 2 selon le format « little-endian » et « big-endian »? Le mot stocké à l'adresse 2 est-il à une adresse alignée? Répondez aux mêmes questions pour l'adresse 4.

(*basé sur une question de l'examen périodique de l'hiver 2019*)

2.2) À combien de tébioctets peut accéder une architecture 64 bits? À combien d'exbioctets pourrait accéder une architecture 128 bits?

2.3) Reconsidérons l'architecture fictive introduite en exemple à la section 2.1.2 avec le jeu d'instructions:

```
foo a, b, c
bar a, b
baz a, b, c
qux
```

Traduisez ces instructions en code machine (d'abord sous le codage à taille variable, puis à taille fixe):

- `foo x6, x1, x3`
- `bar x5, x4`
- `baz x0, x7, x0`

Traduisez le code machine « `01011101110000010111` » (sous codage à taille variable) vers le programme qu'il représente.

2.4) Imaginons une architecture qui offre une instruction « `addm x y a` » qui additionne le contenu du registre y et la valeur de l'octet situé à l'adresse a de la mémoire principale, et stocke le résultat dans le registre x. Pourrions-nous qualifier cette architecture de RISC?

3

Programmation en langage d'assemblage : ARMv8

Maintenant que nous comprenons mieux le fonctionnement interne de l'ordinateur, nous introduisons la programmmation en langage d'assemblage, donc la programmation de (très) bas niveau à l'aide du jeu d'instructions d'un processeur. Nous utiliserons ARMv8-A, que nous dénotons simplement ARMv8. Elle est une architecture 64 bits de type RISC annoncée en 2011, parue en 2013, et développée par la société britannique ARM. L'architecture ARMv8 est rétrocompatible avec l'architecture 32 bits ARMv7. Ces deux architectures se retrouvent dans la plupart des téléphones intelligents et des tablettes numériques, ainsi que dans plusieurs systèmes embarqués. Nous faisons un premier suvol du jeu d'instruction à l'aide d'un programme simple que nous écrirons progressivement.

3.1 Registres

L'architecture ARMv8 possède ces 32 registres de 64 bits [ARM15, Sect. 9.1.1]:

registres	*nom*	*utilisation*
$x_0 - x_7$	—	registres d'arguments et de retour de sous-programmes
x_8	`xr`	registre pour retourner l'adresse d'une structure
$x_9 - x_{15}$	—	registres temporaires sauvegardés par l'appelant
$x_{16} - x_{17}$	$ip_0 - ip_1$	registres temporaires intra-procéduraux
x_{18}	`pr`	registre temporaire pouvant être réservé par le système
$x_{19} - x_{28}$	—	registres temporaires sauvegardés par l'appelé
x_{29}	`fp`	pointeur vers l'ancien sommet de pile (*frame pointer*)
x_{30}	`lr`	registre d'adresse de retour (*link register*)
x_{zr}	`sp`	registre contenant la valeur 0, ou le pointeur de pile (*stack pointer*)

Nous utiliserons principalement les registres du tableau ci-dessus qui apparaissent sur des lignes non colorées, donc: $x_0 - x_7$ et $x_{19} - x_{28}$. Nous n'utiliserons *jamais* les registres $x_{16} - x_{18}$.

Pour tout indice $k \in \{0, 1, \ldots, 30, \texttt{zr}\}$, il existe une version 32 bits du registre $\texttt{x}_k$ nommée $\texttt{w}_k$. Le sous-registre $\texttt{w}_k$ correspond aux 32 bits de poids faible de $\texttt{x}_k$. Autrement dit, si $\texttt{x}_k = b_{63} \cdots b_1 b_0$, alors $\texttt{w}_k = b_{31} \cdots b_1 b_0$.

3.2 Un premier programme

Afin d'explorer le jeu d'instructions de l'architecture ARMv8, ainsi que le fonctionnement d'un programme en langage d'assemblage, nous allons écrire un court programme basé sur un problème algorithmique et mathématique simple.

Algorithme 3 : Calcul de la séquence de Collatz.

Entrées : $n \in \mathbb{N}$
Sorties : —

tant que $n \neq 1$
  si n *est pair* **alors**
    $n \leftarrow n \div 2$
  sinon
    $n \leftarrow 3n + 1$

Considérons l'algorithme 3. Il reçoit un nombre $n \in \mathbb{N}$ en entrée et applique la fonction $f : \mathbb{N} \to \mathbb{N}$ suivante à répétition sur n jusqu'à l'atteinte de 1:

$$f(n) \overset{\text{déf}}{=} \begin{cases} n \div 2 & \text{si } n \text{ est pair,} \\ 3n + 1 & \text{sinon.} \end{cases}$$

La *conjecture de Collatz* affirme que cet algorithme se termine sur *toute* entrée; autrement dit, que la séquence s_n définie par $f(n), f(f(n)), f(f(f(n))), \ldots$ atteint éventuellement 1 *peu importe* la valeur n de départ. Par exemple, la séquence s_3 atteint 1 en sept étapes:

$$s_3 = 10, 5, 16, 8, 4, 2, 1, \ldots,$$

et la séquence s_{14} atteint 1 en dix-sept étapes:

$$s_{14} = 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, \ldots.$$

Le nombre d'étapes afin d'atteindre 1 à partir de n est son *temps de vol*, que nous dénotons t_n. Par exemple, $t_3 = 7$ et $t_{14} = 17$. Écrivons un programme qui lit une entrée $n \in \mathbb{N}$ et affiche son temps de vol t_n. Autrement dit, nous cherchons à calculer le nombre d'itérations de la boucle **tant que** de l'algorithme 3.

Plutôt que d'écrire le programme complet directement, nous présenterons d'abord des segments de code qui accomplissent des sous-tâches, puis nous unifierons et raffinerons ces segments de code afin d'obtenir un programme.

Remarque.

À ce jour, on ne sait pas si l'algorithme 3 termine toujours! En particulier, il n'existe donc pas d'analyseur statique de code qui pourrait détecter s'il boucle parfois à l'infini. Le célèbre mathématicien Paul Erdős a affirmé que les mathématiques ne sont pas encore prêtes pour de tels problèmes!

3.2.1 Calcul de $f(n)$

Voyons d'abord comment calculer $f(n)$ à partir d'un nombre $n \in \mathbb{N}$. Nous devons d'abord choisir des registres avec lesquels travailler. Les registres $\texttt{x}_{19}$ – $\texttt{x}_{28}$ sont préférables car la convention nous permet de les utiliser librement. Supposons que $\texttt{x}_{19}$ contienne initialement n. Ce segment de code calcule $f(n)$ et stocke sa valeur dans le registre $\texttt{x}_{19}$:

```
        tbnz    x19, 0, impair  // si x19[0] ≠ 0: aller à impair
pair:
        mov     x20, 2          // x20 ← 2
        udiv    x19, x19, x20   // x19 ← x19 ÷ x20
        b       fin             // aller à fin
impair:
        mov     x20, 3          // x20 ← 3
        mul     x20, x20, x19   // x20 ← x20 * x19
        add     x19, x20, 1     // x19 ← x20 + 1
fin:
```

Décortiquons le code ci-dessus. Afin de calculer f, nous devons d'abord tester si n est pair. Nous pourrions tester si $n \bmod 2 = 0$, mais le jeu d'instruction d'ARMv8 n'offre *pas* d'instruction pour le calcul du modulo. Puisque n est stocké en binaire, nous pouvons tester si n est pair grâce à l'observation suivante: n est pair si et seulement si son bit de poids faible est égal à zéro.

L'instruction « `tbnz xd, i, etiq` » vérifie si le $i^{\text{ème}}$ bit du registre $\texttt{x}_\texttt{d}$ diffère de zéro. Si c'est le cas, elle effectue un saut vers l'étiquette « `etiq:` », sinon elle ne fait rien. Ainsi, la première ligne de code teste si n est impair; si c'est le cas, elle branche à l'étiquette « `impair:` »; sinon le programme passe à la ligne suivante, donc à l'étiquette « `pair:` ». Rappelons que ce traitement est coordonné par l'unité de contrôle du processeur.

Si n est pair, alors le programme calcule $n \div 2$ à l'aide de ces instructions:

`mov xd, v`	assigne la valeur v au registre $\texttt{x}_\texttt{d}$
`udiv xd, xn, xm`	stocke le quotient de $\texttt{x}_\texttt{n}$ divisé par $\texttt{x}_\texttt{m}$ dans $\texttt{x}_\texttt{d}$

L'instruction « `b fin` » effectue un saut vers la toute dernière ligne afin d'indiquer que le calcul de $f(n)$ est complété. Les sauts sont aussi connus sous le nom de *branchement*, d'où la lettre « `b` ».

Si n est impair, le programme calcule $3n + 1$ à l'aide de ces instructions:

`mov xd, v`	assigne la valeur v au registre x_d
`mul xd, xn, xm`	stocke le produit de x_n et x_m dans x_d
`add xd, xn, v`	stocke la somme de x_n et la valeur v dans x_d

3.2.2 Affichage du contenu d'un registre

Maintenant que nous savons calculer $f(n)$, voyons comment afficher sa valeur, c'est-à-dire le contenu du registre x_{19}. L'affichage se fait à l'aide du code suivant:

```
        adr     x0, msgRes      // x0 ← adresse(msgRes)
        mov     x1, x19         // x1 ← x19
        bl      printf          // printf(x0, x1)

.section ".rodata"
msgRes: .asciz    "Résultat: %lu"
```

Comme l'affichage nécessiterait notamment d'exploiter les services du noyau d'un système d'exploitation, nous utilisons plutôt les primitives d'entrée/sortie du langage C. Ainsi, l'affichage se fait grâce à la fonction `printf` de la librairie `stdio`. Le premier paramètre de cette fonction est l'adresse de la mémoire principale où se situe la chaîne de caractères à afficher. Nous stockons donc une chaîne de caractères `msgRes` sous forme de « constante ».

Les constantes sont définies dans le *segment de données en lecture seule* identifié par « `.section ".rodata"` ». Il est possible d'y déclarer une constante, de type `.t` portant le nom `etiq` et contenant la valeur `x`, à l'aide de:

```
  etiq:   .t   x
```

L'identifiant « `.asciz` » indique que nous désirons allouer une *chaîne de caractères* [1]. La chaîne de caractères `"Résultat: %lu"` contient le spécificateur de format `"%lu"` qui spécifie un entier non négatif de 64 bits. Le deuxième argument de `printf` est donc, dans ce cas, un entier non négatif à afficher.

Tel que brièvement mentionné à la section 3.1, les paramètres d'un sous-programme sont passés via les registres x_0–x_7. Ainsi, le code suivant stocke l'adresse de la chaîne `msgRes` et la valeur du registre x_{19} dans les registres x_0 et x_1 respectivement, et appelle la fonction `printf` grâce à l'instruction `bl`:

```
        adr     x0, msgRes
        mov     x1, x19
        bl      printf
```

3.2.3 Lecture d'une valeur dans un registre

Nous savons maintenant calculer et afficher $f(n)$. Toutefois, nous n'avons pour l'instant aucune façon de spécifier la valeur de n. Voyons comment une valeur peut être lue et assignée au registre x_{19}. La lecture se fait grâce au code suivant:

1. Plus précisément: une chaîne de caractères se terminant par le caractère nul. Nous discuterons de ce détail technique au chapitre 10.

```
        adr     x0, fmtEntree   // x0 ← adresse(fmtEntree)
        adr     x1, temp        // x1 ← adresse(temp)
        bl      scanf           // scanf(x0, x1)

        ldr     x19, temp       // x19 ← mem[temp]

.section ".bss"
           .align  8
temp:      .skip   8

.section ".rodata"
fmtEntree: .asciz    "%lu"
```

Comme pour la lecture, nous effectuons l'affichage grâce à la librarie `stdio` du langage C. Plus précisément, nous utilisons sa fonction `scanf`. Le premier paramètre de cette fonction est l'adresse à laquelle la valeur lue doit être stockée. Le second paramètre est l'adresse du format de la valeur à lire.

Nous allouons un double mot nommé `temp:` dans le *segment de données non-initialisées* identifié par « `.section ".bss"` ». Cette « variable » possède 8 octets (64 bits) et son adresse est alignée à un multiple de 8:

```
.section ".bss"
           .align  8
temp:      .skip   8
```

Ainsi, l'exécution de `scanf` lira un entier non négatif de 64 bits et stockera sa valeur dans une « variable » temporaire. Afin de transférer son contenu vers x_{19}, donc de la mémoire principale vers le processeur, nous utilisons ce code:

```
        ldr     x19, temp
```

Cette instruction charge le contenu stocké à l'adresse vers x_{19}:

`ldr xd, etiq`	charge, dans x_d, la valeur stockée à l'adresse de l'étiquette `etiq:` de la mémoire principale

3.2.4 Vers un premier programme

Jusqu'ici, nous avons vu comment lire un entier non négatif n, calculer $f(n)$ et afficher sa valeur. Le code qui accomplit ces trois tâches peut être réuni ainsi:

```
        // Lecture de n
        adr     x0, fmtEntree   // x0 ← adresse(fmtEntree)
        adr     x1, temp        // x1 ← adresse(temp)
        bl      scanf           // scanf(x0, x1)

        ldr     x19, temp       // x19 ← mem[temp]
```

```
        // Calcul de f(n)
        tbnz    x19, 0, impair  // si x19[0] ≠ 0: aller à impair
pair:
        mov     x20, 2          // x20 ← 2
        udiv    x19, x19, x20   // x19 ← x19 ÷ x20
        b       fin             // aller à fin
impair:
        mov     x20, 3          // x20 ← 3
        mul     x20, x20, x19   // x20 ← x20 * x19
        add     x19, x20, 1     // x19 ← x20 + 1
fin:
        // Affichage de f(n)
        adr     x0, msgRes      // x0 ← adresse(msgRes)
        mov     x1, x19         // x1 ← x19
        bl      printf          // printf(x0, x1)

.section ".bss"
          .align  8
temp:     .skip   8

.section ".rodata"
fmtEntree: .asciz    "%lu"
msgRes:    .asciz    "Résultat: %lu"
```

Nous sommes près d'avoir un programme complet. Néanmoins, nous n'avons toujours pas accompli la tâche initiale qui était de calculer le temps de vol t_n.

3.2.5 Calcul du temps de vol

Afin de calculer le temps de vol t_n, nous générons la séquence s_n itérativement, et nous comptons le nombre d'itérations qui mènent à la valeur 1:

```
        mov     x21, 0          // x21 ← 0
boucle:
        cmp     x19, 1          //
        b.eq    finboucle       // si n = 1: aller à finboucle
        add     x21, x21, 1     // sinon: incrémenter x21

        /* calculer f(n) dans x19 ici */

        b       boucle          // aller à boucle
finboucle:
```

Décortiquons ce code. La première instruction initialise un compteur à zéro qui sera incrémenté afin de calculer t_n. Rappelons que x_{19} contient n. L'instruction « `cmp x19, 1` » compare le contenu de x_{19} et la constante 1. L'instruction « `b.eq finboucle` » branche à l'étiquette « `finboucle:` » si la comparaison pré-

cédente résulte en une égalité. Autrement dit, si $x_{19} = 1$, alors nous avons terminé de calculer t_n. Si $x_{19} \neq 1$, alors le compteur x_{21} est incrémenté, x_{19} est remplacé par $f(x_{19})$, et le programme branche vers le début de la boucle.

3.2.6 Programme complet

Nous pouvons finalement réunir tous nos segments de code. Afin d'exécuter notre programme, nous devons lui donner un point d'entrée, ici « `main:` », et quitter avec l'instruction « `bl exit` ». Ici, « `exit` » réfère à une fonction de la librairie standard du langage C. Afin de quitter sans s'appuyer sur cette librairie, il faudrait utiliser les services du noyau d'un système d'exploitation.

Plutôt que d'afficher x_{19} comme nous l'avions fait auparavant, nous devons maintenant afficher x_{21} qui contient t_n. Nous obtenons ce programme complet:

```
.global main

main:
        // Lecture de n
        adr     x0, fmtEntree   // x0 ← adresse(fmtEntree)
        adr     x1, temp        // x1 ← adresse(temp)
        bl      scanf           // scanf(x0, x1)

        ldr     x19, temp       // x19 ← mem[temp]

        // Calcul du temps de vol
        mov     x21, 0          // x21 ← 0
boucle:
        cmp     x19, 1          //
        b.eq    finboucle       // si n = 1: aller à finboucle
        add     x21, x21, 1     // sinon: incrémenter x21

        // Calcul de f(x19)
        tbnz    x19, 0, impair  // si x19[0] ≠ 0: aller à impair
pair:
        mov     x20, 2          // x20 ← 2
        udiv    x19, x19, x20   // x19 ← x19 ÷ x20
        b       fin             // aller à fin
impair:
        mov     x20, 3          // x20 ← 3
        mul     x20, x20, x19   // x20 ← x20 * x19
        add     x19, x20, 1     // x19 ← x20 + 1
fin:

        b       boucle          // aller à boucle
finboucle:
```

```
        // Affichage du temps de vol
        adr     x0, msgRes      // x0 ← adresse(msgRes)
        mov     x1, x21         // x1 ← x21
        bl      printf          // printf(x0, x1)

        bl      exit            // Quitter le programme

.section ".bss"
          .align  8
temp:     .skip   8

.section ".rodata"
fmtEntree: .asciz    "%lu"
msgRes:    .asciz    "Résultat: %lu"
```

Remarque.

Une adaptation du programme ci-dessus, pour l'architecture x86-64, est disponible sur le répertoire GitHub du cours.

3.3 Détails pratiques

3.3.1 Normes de programmation

Organisation d'une ligne de code. Chaque ligne d'un programme en langage d'assemblage est constituée d'au plus « 4 colonnes »:

```
etiquette:    opcode    opérandes    // Commentaire
```

L'*étiquette* donne un nom symbolique à une ligne du programme, le *code d'opération* (opcode) est le nom symbolique de l'instruction, les *opérandes* sont les paramètres de l'instruction, et le *commentaire* donne des annotations sur le code destinées à l'humain (elles sont ignorées par la machine). Pour faciliter la lecture, il est préférable d'aligner les colonnes et de placer chaque étiquette seule sur sa ligne; comme dans cet extrait de code de la section 3.2:

```
impair:
        mov     x20, 2          // x20 ← 2
        udiv    x19, x19, x20   // x19 ← x19 ÷ x20
        b       fin             // aller à fin
```

Présentation. Comme les registres ne peuvent pas être renommés, il est pratique d'associer implicitement des noms symboliques aux registres et de commenter chaque instruction par un commentaire décrivant l'effet de l'instruction, en pseudocode ou dans un langage de plus haut niveau tel que C. Par exemple:

```
// Usage des registres:
//   x19 -- n
//   x20 -- tmp
        tbnz    x19, 0, impair  // si n pas pair: aller à impair
pair:
        mov     x20, 2          // tmp ← 2
        udiv    x19, x19, x20   // n   ← n ÷ tmp
        b       fin             // aller à fin
impair:
        mov     x20, 3          // tmp ← 3
        mul     x20, x20, x19   // tmp ← tmp * n
        add     x19, x20, 1     // n   ← tmp + 1
fin:
```

ou encore:

```
// Usage des registres:
//   x19 -- n
//   x20 -- tmp
        tbnz    x19, 0, impair  // if (n % 2 == 0) goto impair
pair:
        mov     x20, 2          // tmp = 2
        udiv    x19, x19, x20   // n /= tmp
        b       fin             // goto fin
impair:
        mov     x20, 3          // tmp = 3
        mul     x20, x20, x19   // tmp *= n
        add     x19, x20, 1     // n = tmp + 1
fin:
```

Comme dans les langages de haut niveau, il est recommandé de séparer les blocs de code qui effectuent des tâches distinctes par un saut de ligne afin « d'aérer » le code et d'en faciliter sa lecture. Il est également préférable d'utiliser des noms d'étiquettes significatifs et courts, autant que possible.

Commentaires. Jusqu'ici, nous avons commenté chaque ligne en correspondance une-à-une avec ce qu'elle accomplit précisément. Plus nous deviendrons à l'aise avec le jeu d'instructions, plus il sera préférable de faire surgir les structures de contrôle de haut niveau afin de comprendre ce que le code effectue.

Par exemple, les commentaires ci-dessous évitent l'usage de « `goto` » au profit de « `if { } else { }` » et ignorent le détail technique de l'affectation des constantes 2 et 3:

```
// Usage des registres:
//   x19 -- n
//   x20 -- tmp
        tbnz    x19, 0, impair  // if (n % 2 == 0) {
pair:                           //
        mov     x20, 2          //
        udiv    x19, x19, x20   //   n /= 2
        b       fin             // }
impair:                         // else {
        mov     x20, 3          //
        mul     x20, x20, x19   //   tmp = 3 * n
        add     x19, x20, 1     //   n = tmp + 1
fin:                            // }
```

3.3.2 Segments de données

Il existe quatre types de segments de données pouvant être déclarés à l'aide des directives suivantes:

directive	*contenu*
`.section ".text"`	instructions
`.section ".rodata"`	données en lecture seule
`.section ".data"`	données initialisées
`.section ".bss"`	données non-initialisées

Si aucun segment n'est spécifié, alors `".text"` est supposé par défaut.

Des données statiques et globales peuvent être déclarées et/ou initialisées dans les segments autres que `".text"` grâce aux mots-clés suivants:

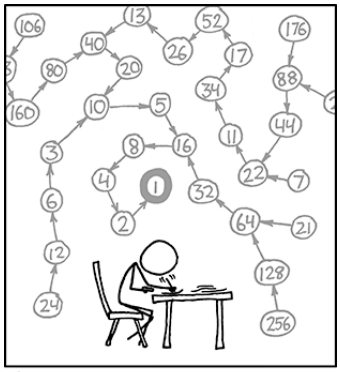
`.align`	k	la donnée suivante est stockée à une adresse divisible par k
`.skip`	k	réserve k octets
`.ascii`	s	chaîne de caractères initialisée à s
`.asciz`	s	chaîne de caractères initialisée à s suivi du caractère nul
`.byte`	v	octet initialisé à v
`.hword`	v	demi-mot initialisé à v
`.word`	v	mot initialisé à v
`.xword`	v	double mot initialisé à v
`.single`	f	nombre en virgule flottante simple précision initialisé à f
`.double`	f	nombre en virgule flottante double précision initialisé à f

3.3.3 Spécificateurs de format

Les (principaux) formats de données des fonctions `printf` et `scanf` de la librairie `stdio` du langage C sont définis comme suit[2]:

2. En général, le nombre de bits de chacun des formats dépend de l'architecture sur laquelle le code est exécuté. Nous faisons ici référence à ARMv8.

famille	*format*	*type*
Nombres sur 32 bits	`%d`	entier décimal
	`%u`	entier décimal non négatif
	`%X`	entier hexadécimal non négatif
	`%f`	nombre en virgule flottante
Nombres sur 64 bits	`%ld`	entier décimal
	`%lu`	entier décimal non négatif
	`%lX`	entier hexadécimal non négatif
	`%lf`	nombre en virgule flottante
Caractères	`%c`	caractère (1 octet)
	`%s`	chaîne de caractères



THE COLLATZ CONJECTURE STATES THAT IF YOU PICK A NUMBER, AND IF IT'S EVEN DIVIDE IT BY TWO AND IF IT'S ODD MULTIPLY IT BY THREE AND ADD ONE, AND YOU REPEAT THIS PROCEDURE LONG ENOUGH, EVENTUALLY YOUR FRIENDS WILL STOP CALLING TO SEE IF YOU WANT TO HANG OUT.

— xkcd

3.4 Exercices

3.1) Écrivez un programme ARMv8 qui lit quatre entiers non négatifs x, y, x', y' de 64 bits, et qui affiche l'aire et le périmètre du rectangle dont deux coins opposés sont situés aux coordonnées (x, y) et (x', y') dans le plan.

(*tiré de [SD11]*)

3.2) Écrivez un programme ARMv8 qui lit deux entiers non négatifs m et n de 64 bits, l'un à la suite de l'autre, et qui affiche toutes les paires (i, j) telles que $0 \leq i < m$ et $0 \leq j < n$.

3.3) Écrivez un programme ARMv8 qui lit un entier non négatif n de 64 bits et qui affiche le $n^{\text{ème}}$ terme de la suite de Fibonacci.

3.4) ★★ Tentez d'adapter vos programmes pour l'architecture x86-64 en vous inspirant de ce programme et/ou en fouillant sur le Web.

4

Accès aux données

Dans ce chapitre, nous explorons les façons d'accéder et de référer aux données de la mémoire principale ainsi que des registres du processeur. En particulier, nous traitons d'adressage et des étapes de vie d'un programme.

4.1 Adresses

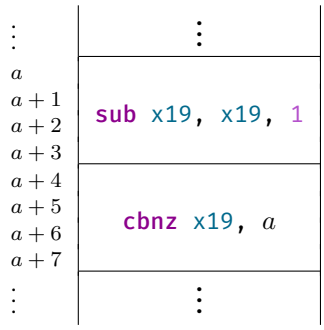
Nous avons indirectement vu deux façons de spécifier une adresse de la mémoire principale. Une *adresse numérique* est une adresse spécifiée par un entier non négatif. Par exemple, l'adresse 26 est numérique et réfère à la cellule c_{26} de la mémoire principale. Les adresses numériques sont souvent écrites en notation hexadécimale, par ex. 0000001A plutôt que 26.

Une *adresse symbolique* est un identifiant unique, par ex. une chaîne de caractères, qui spécifie une cellule mémoire c_a dont nous ne connaissons (à priori) pas l'index a. En langage d'assemblage, les adresses symboliques se manifestent sous forme d'étiquettes.

À titre d'exemple, considérons ce segment de code ARMv8 qui décrémente itérativement le registre x_{19} jusqu'à ce qu'il atteigne zéro:

```
main:
       sub  x19, x19, 1   // x19 ← x19 - 1
       cbnz x19, main     // aller à main si x19 ≠ 0
```

Rappelons que lors de l'exécution d'un programme, celui-ci vit dans la mémoire principale. Ainsi, lors du saut vers l'étiquette `main:`, le compteur d'instruction doit être déplacé vers l'adresse a de la mémoire qui contient l'instruction sous cette étiquette. Rappelons qu'une instruction est généralement stockée sur plusieurs octets, par ex. 4 octets dans le cas d'ARMv8. Ainsi, il s'agit ici de l'adresse a pointant vers la première celulle c_a qui contient une portion du code machine de l'instruction « `sub x19, x19, 1` »:



Ainsi, l'instruction « `cbnz x19, main` » correspond essentiellement à l'instruction « `cbnz x19,` a ». Lors de l'écriture du code, nous ne connaissons pas la valeur de a; elle peut être déterminée plus tard pour nous.

Remarque.

À l'interne, sur ARMv8, cet adressage se fera plutôt de façon *relative*. Plutôt que de stocker a explicitement, on stocke -1 afin d'indiquer que l'adresse a se situe à une distance de $-1 \cdot 4$ adresses.

4.2 Adressage

Comme nous l'avons vu aux chapitres précédents, la plupart des instructions d'un langage d'assemblage possèdent des opérandes. Il existe plusieurs façons d'interpréter les opérandes afin de localiser la valeur qui leur est associée. Nous appelons ces méthodes de localisation des *modes d'adressage*. Dans cette section, nous décrivons quelques modes d'adressage répandus, notamment sur ARMv8.

Nous utiliserons i afin de dénoter une valeur immédiate, c'est-à-dire une constante figée dans le code; n afin de référer au nom d'un registre; et a pour dénoter une adresse de la mémoire principale. Nous écrivons reg$[n]$ pour référer au contenu du registre n, et nous écrivons mem$[a]$ pour référer au contenu de la mémoire principale à l'adresse a (sans se soucier pour l'instant du nombre d'octets adressés).

4.2.1 Immédiat

Le mode d'*adressage immédiat* est le plus simple. Il associe simplement à l'opérande i la valeur i elle-même:

$$i \mapsto i.$$

Par exemple, dans l'instruction suivante, la valeur immédiate $i = 42$ est tout simplement interprétée comme la valeur 42:

```
mov x19, 42
```

4.2.2 Direct

Le mode d'*adressage direct* récupère, à partir d'une adresse a, la valeur contenue à l'adresse a de la mémoire principale:

$$a \mapsto \text{mem}[a].$$

Par exemple, si $a = \text{FF}$, alors l'adressage direct récupère la valeur mem[FF].

4.2.3 Par registre

Le mode d'*adressage par registre*, aussi appelé *adressage direct par registre*, récupère la valeur contenue dans un registre à partir de son nom:

$$n \mapsto \text{reg}[n].$$

Par exemple, dans l'instruction suivante, l'opérande `x20` réfère au contenu situé dans le registre $\texttt{x}_{20}$:

```
mov x19, x20
```

4.2.4 Indirect

Le mode d'*adressage indirect* récupère, à partir d'une adresse a, la valeur contenue à l'adresse b où b est la valeur contenue à l'adresse a:

$$a \mapsto \text{mem}[\text{mem}[a]].$$

Par exemple, si $a = \text{FF}$ et $\text{mem}[\text{FF}] = \text{1A}$, alors l'adressage indirect récupére la valeur contenue dans mem[1A].

4.2.5 Indirect par registre

Le mode d'*adressage indirect par registre* récupère, à partir d'un nom de registre n, la valeur contenue à l'adresse $\text{reg}[n]$ de la mémoire principale:

$$n \mapsto \text{mem}[\text{reg}[n]].$$

Par exemple, supposons que le registre $\texttt{x}_{20}$ contienne la valeur FF. Dans l'instruction suivante, l'opérande « `[x20]` » réfère au contenu situé à l'adresse FF; autrement dit à $\text{mem}[\text{reg}[20]] = \text{mem}[\text{FF}]$:

```
ldr x19, [x20]  // stocker mem[x20] dans x19
```

4.2.6 Indirect par registre indexé

Le mode d'*adressage indirect par registre indexé* récupère, à partir d'un nom de registre n et d'une valeur immédiate i, la valeur contenue à l'adresse $\text{reg}[n] + i$ de la mémoire principale:

$$n, i \mapsto \text{mem}[\text{reg}[n] + i].$$

Par exemple, pour $n = 20$, $\text{reg}[20] = \text{FA}$ et $i = 1$, ce mode d'adressage récupère la valeur $\text{mem}[\text{FB}]$.

Il existe une autre variante de ce mode d'adressage où le second paramètre est un nom de registre m, plutôt qu'une valeur immédiate i. Dans ce cas, la valeur récupérée est celle contenue à l'adresse $\text{reg}[n] + \text{reg}[m]$:

$$n, m \mapsto \text{mem}[\text{reg}[n] + \text{reg}[m]].$$

4.2.7 Indirect par registre pré/post-incrémenté

Les modes d'adressage indirect par registre pré/post-incrémenté reçoivent un nom de registre n et une valeur immédiate i. Dans le cas *pré-incrémenté*, la valeur contenue dans le registre n est incrémentée par i, puis la valeur contenue à l'adresse $\text{reg}[n]$ de la mémoire principale est récupérée. Dans le cas *post-incrémenté*, le registre n est incrémenté *après* l'accès mémoire:

$$\begin{aligned} &\text{pré-incrémenté: } \text{reg}[n] \leftarrow \text{reg}[n] + i \quad \text{puis} \quad n \mapsto \text{mem}[\text{reg}[n]], \\ &\text{post-incrémenté: } \quad n \mapsto \text{mem}[\text{reg}[n]] \quad \text{puis} \quad \text{reg}[n] \leftarrow \text{reg}[n] + i. \end{aligned}$$

Notons que ces deux modes d'adressage ont des *effets de bord*: le contenu du registre n est modifié lors de l'interprétation de l'opérande.

Par exemple, si $n = 20$, $\text{reg}[20] = \text{FA}$ et $i = 1$, le mode pré-incrémenté récupère la valeur $\text{mem}[\text{FB}]$, et le mode post-incrémenté récupère la valeur $\text{mem}[\text{FA}]$. Dans les deux cas, $\text{reg}[20] = \text{FB}$ à la fin de l'exécution.

4.2.8 Relatif

Le mode d'*adressage relatif* est un cas particulier de l'adressage indirect par registre indexé, où le registre utilisé est le compteur d'instruction:

$$i \mapsto \text{mem}[\text{reg}[pc] + i].$$

Par exemple, lors du chargement de `var` dans le code suivant, l'accès est effectué de façon relative au compteur d'instruction [1], ici avec $i = 8$:

```
ldr x19, var
nop
```

1. Puisque chaque instruction est codée sur 4 octets sur ARMv8, i doit être un multiple de 4.

```
.section ".data"
var: .byte 42
```

4.2.9 Sommaire des modes d'adressage

Les modes d'adressage présentés sont répertoriés au tableau suivant:

Nom	*Valeur récupérée*	*Exemple sur ARMv8*
immédiat	$i \mapsto i$	`mov x19, 42`
direct	$a \mapsto \text{mem}[a]$	—
par registre	$n \mapsto \text{reg}[n]$	`mov x19, x20`
indirect	$a \mapsto \text{mem}[\text{mem}[a]]$	—
indirect par registre	$n \mapsto \text{mem}[\text{reg}[n]]$	`ldr x19, [x20]`
indirect par registre indexé	$n, i \mapsto \text{mem}[\text{reg}[n] + i]$	`ldr x19, [x20, i]`
	$n, m \mapsto \text{mem}[\text{reg}[n] + \text{reg}[m]]$	`ldr x19, [x20, x21]`
indirect par registre indexé pré-incrémenté	$\text{reg}[n] \leftarrow \text{reg}[n] + i$, suivi de $n, i \mapsto \text{mem}[\text{reg}[n]]$	`ldr x19, [x20, i]!`
indirect par registre indexé post-incrémenté	$n, i \mapsto \text{mem}[\text{reg}[n]]$, suivi de $\text{reg}[n] \leftarrow \text{reg}[n] + i$	`ldr x19, [x20], i`
relatif	$i \mapsto \text{mem}[\text{reg}[pc] + i]$	`ldr x19, var`

4.3 Particularités de l'architecture ARMv8

Soit $d \in \{0, 1, \ldots, 31\}$ l'index d'un registre, et soit a une adresse de l'architecture ARMv8. Il est possible de charger/stocker un octet, un demi-mot, un mot ou un double mot d'un/vers un registre grâce aux instructions suivantes:

nombre d'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`

L'adresse associée à une étiquette `etiq:` peut être chargée dans un registre r grâce à l'instruction:

```
adr r, etiq
```

Le contenu d'un registre s, ou une valeur immédiate i, peut être chargée dans un registre r grâce aux instructions:

```
mov r, s // charge le contenu du registre s dans le registre r
mov r, i // charge la valeur immédiate    i dans le registre r
```

Remarque.

L'instruction `mov` permet de charger des valeurs immédiates de 12 bits; au-delà de 12 bits la possibilité du chargement n'est pas garantie et une

erreur peut être levée à la compilation.

4.4 Assemblage d'un programme

Afin d'exécuter un programme en langage d'assemblage, il est nécessaire de l'*assembler*. L'*assembleur* est un outil qui traduit le code d'assemblage vers du code machine; il effectue la traduction de chaque instruction vers sa représentation binaire. La plupart des adresses symboliques sont également remplacées par des adresses numériques lors de l'assemblage.

L'assembleur produit un *fichier objet* qui contient le code machine généré, mais également certaines adresses symboliques qui dépendent de modules externes comme des librairies. Le fichier objet contient une table de ces symboles externes.

L'*éditeur de liens* est un outil qui combine les fichiers objets vers un fichier exécutable. En particulier, l'éditeur de liens recalcule certaines adresses et transforme les adresses symboliques encore présentes vers des adresses numériques.

Sur les systèmes de type UNIX, l'assemblage, l'édition des liens et l'exécution peut se réaliser ainsi:

```
as foo.s -o foo.o  # assemblage du code foo.s vers un objet foo.o
ld foo.o -o foo    # édition de l'objet foo.o vers un exécutable foo
./foo              # exécution du programme foo
```

L'option `-e` permet de spécifier le point d'entrée d'un programme, et l'option `-lc` permet d'inclure les fichiers objets de la librairie standard du langage C (par ex. pour utiliser `printf` et `scanf`). Ainsi, les programmes vus jusqu'ici peuvent être exécutés grâce à:

```
as foo.s -o foo.o              # assemblage
ld foo.o -o foo   -e main -lc  # édition des liens
./foo                          # exécution
```

Remarque.

Rappelons que le jeu d'instructions d'un ordinateur dépend de son processeur. Ainsi, l'assemblage d'un programme ARMv8 échouera probablement sur votre ordinateur. Nous utiliserons donc une machine virtuelle qui émule un processeur ARMv8.

Remarque.

Des fichiers « *makefile* » seront fournis pour les laboratoires et les devoirs afin d'automatiser l'assemblage.

4.5 Exercices

4.1) Considérons le contenu suivant de la mémoire principale:

adresse	contenu
0	01_{16}
1	$A4_{16}$
2	BC_{16}
3	48_{16}
4	$5F_{16}$
5	11_{16}
6	FF_{16}
7	43_{16}
⋮	⋮

Quelle est la valeur de x_{19} et x_{20} après l'exécution de ces instructions?

```
mov   x19, 2
ldr   x20, [x19], 3
sub   x19, x19, 1
ldrb  w20, [x19, 2]
```

(*tiré de l'examen périodique de l'hiver 2019*)

5

Circuits logiques

Maintenant que nous comprenons mieux le fonctionnement interne de l'ordinateur, nous faisons un survol de son implémentation au niveau le plus bas: les *circuits logiques*. Nous considérons une version idéalisée des circuits logiques afin de mieux cerner l'organisation de l'unité arithmétique et logique, de l'unité de contrôle et de la mémoire principale.

À l'aide de transistors, il est possible de construire des *portes logiques* qui calculent la négation ($\neg$), la conjonction ($\wedge$), la disjonction ($\vee$) et le OU exclusif ($\oplus$), où les entrées et sorties correspondent à des signaux numériques logiques qui, à nos fins, ne sont simplement que des bits. Ces portes sont souvent illustrées comme à la figure 5.1.

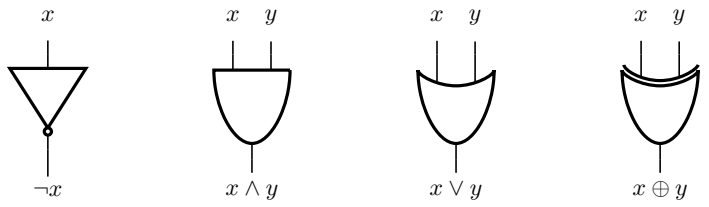


FIGURE 5.1 – De gauche à droite: portes NON, ET, OU, OU exclusif. Les bits d'entrée et de sortie sont situés respectivement au-dessus et au bas des portes.

5.1 Arithmétique

5.1.1 Addition de deux bits

Considérons l'addition afin d'illustrer l'implémentation des opérations arithmétiques. La table d'addition de deux bits x et y correspond à:

x	y	$x + y$ (sur deux bits)
0	0	00
0	1	01
1	0	01
1	1	10

Nommons le bit de poids faible la *somme* et le bit de poids fort la *retenue*. Nous remarquons que la somme et la retenue correspondent respectivement aux valeurs $x \oplus y$ et $x \wedge y$:

x	y	$x \wedge y$ (retenue)	$x \oplus y$ (somme)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Le circuit illustré à la figure 5.2 exploite cette observation afin d'implémenter l'addition de deux bits. Ce type de circuit est connu sous le nom de *demi-additionneur*.

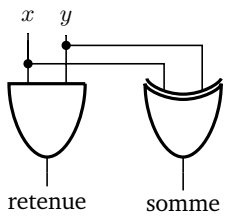


FIGURE 5.2 – Demi-additionneur.

Afin d'additionner deux *suites* de bits, il serait tentant de combiner plusieurs demi-additionneurs. Toutefois, ceux-ci ne tiennent pas compte d'une retenue provenant d'une addition précédente. Un *additionneur complet*, une généralisation du demi-additionneur, prend également une telle retenue en entrée. Considérons la table d'addition d'une retenue r et de deux bits x et y:

r	x	y	$r + x + y$ (sur deux bits)
0	0	0	00
0	0	1	01
0	1	0	01
0	1	1	10
1	0	0	01
1	0	1	10
1	1	0	10
1	1	1	11

Nommons à nouveau le bit de poids faible la *somme* et le bit de poids fort la *retenue*. Nous remarquons que la somme et la retenue correspondent aux combinaisons suivantes de portes ET, OU et OU exclusif:

r	x	y	$(x \wedge y) \vee (r \wedge (x \oplus y))$ (retenue)	$r \oplus (x \oplus y)$ (somme)
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Nous pouvons donc implémenter un additionneur complet sous forme de circuit tel qu'illustré à la figure 5.3.

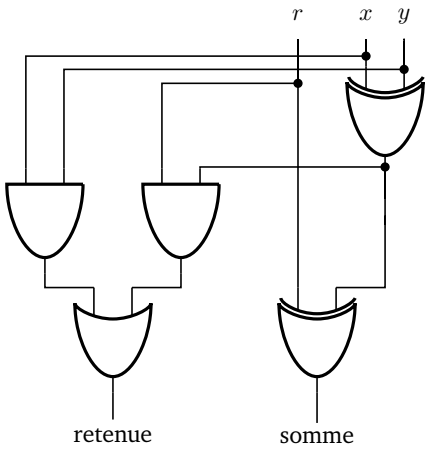


FIGURE 5.3 – Additionneur complet.

5.1.2 Addition de deux nombres

Afin d'additionner deux nombres x et y, chacun représenté par une suite de n bits, il suffit de composer un demi-additionneur et $n - 1$ additionneurs tel qu'illustré à la figure 5.4. Remarquons qu'une telle somme z peut engendre une retenue $r \in \{0, 1\}$.

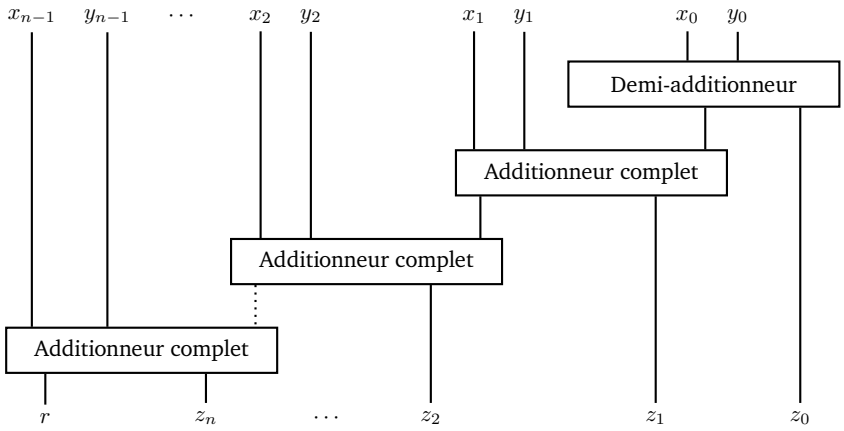


FIGURE 5.4 – Additionneur de deux nombres de n bits, construit à partir d'un demi-additionneur et de $n-1$ additionneurs complets.

Remarque.

Des additionneurs plus sophistiqués, par ex. *avec regard en avant (« carry-lookahead »)*, permettent de calculer la somme plus rapidement.

5.2 Décodage du jeu d'instructions

L'unité de contrôle combine plusieurs circuits afin d'entre autres décoder le jeu d'instructions et d'invoquer l'unité arithmétique et logique.

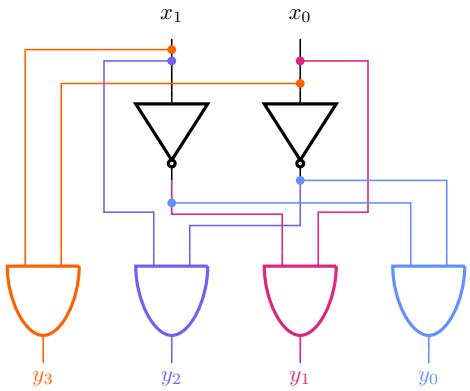


FIGURE 5.5 – Décodeur de deux bits.

Exemple.

Considérons un jeu d'instruction simple constitué de quatre opérations codées par les suites binaires 00, 01, 10 et 11. Lorsque l'unité de contrôle reçoit une telle suite de deux bits $x = x_1x_0$, elle peut la décoder à l'aide du circuit illustré à la figure 5.5. Par exemple, sur entrée $x = 10$ nous avons $y_2 = 1$ et $y_3 = y_1 = y_0 = 0$ à la sortie.

Un tel décodeur permet également d'accomplir des tâches plus complexes comme la sélection de bits. Par exemple, le circuit illustré à la figure 5.6 est un *multiplexeur* qui sélectionne un bit y_i parmi quatre bits, à partir de la représentation binaire x de l'indice i.

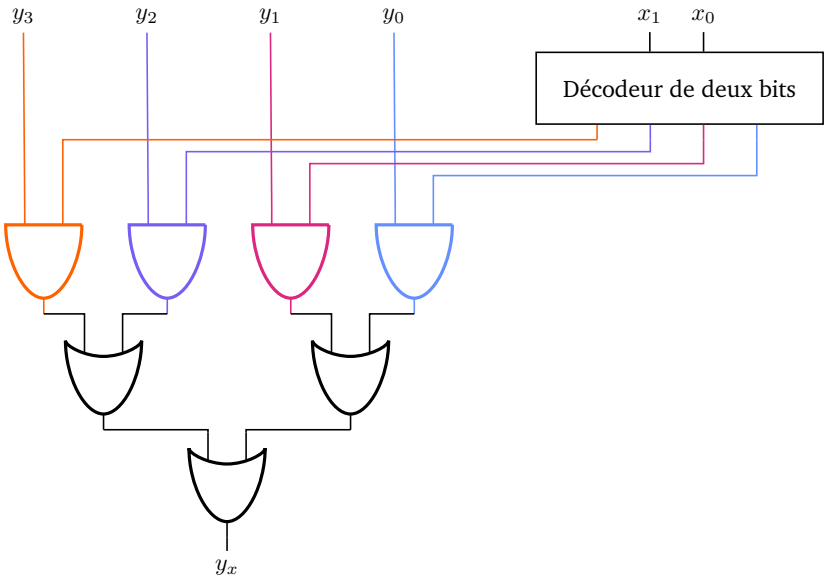


FIGURE 5.6 – Multiplexeur à deux bits de sélection.

Exemple.

Sur entrée $x = 11$, la sortie de ce circuit est y_3, alors que sur entrée $x = 01$, sa sortie est y_1.

En combinant des circuits comme les décodeurs et multiplexeurs, l'unité de contrôle arrive notamment à décoder des instructions complètes et à sélectionner des résultats plus complexes provenant de l'unité arithmétique et logique.

Remarque.

Il devient difficile de spécifier graphiquement des circuits complexes. Il existe donc des langages pour les décrire tels que Verilog et VHDL.

5.3 Mémoire

Les circuits logiques présentés jusqu'ici sont dits *combinatoires*: ils transforment des entrées en sorties, sans mémoriser d'état interne. Afin d'implémenter la mémoire principale et les registres, on peut plutôt exploiter des circuits logiques dits *séquentiels*. Ceux-ci contiennent des cycles qui stockent des bits d'information. Nous donnons un exemple simple et classique qui montre comment stocker un bit. La figure 5.7 illustre un *circuit à verrouillage* (ou *verrou*). Ce circuit possède deux bits d'entrée: s et r de l'anglais « *set* » et « *reset* ».

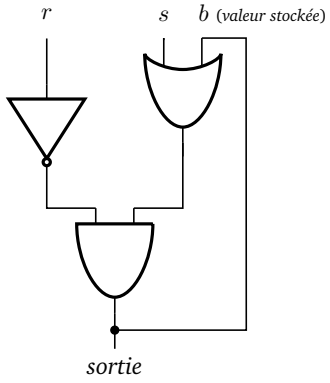


FIGURE 5.7 – Circuit à verrouillage stockant un bit b.

Considérons sa table de vérité:

r	s	bit interne	sortie (et nouveau bit interne)
0	0	b	b
0	1	b	1
1	0	b	0
1	1	b	0

Lorsque $r = s = 0$, rien ne se produit et le circuit retourne tout simplement le bit b qu'il mémorise. Lorsque $r = 1$, le bit mémorisé est remis à zéro. Lorsque $r = 0$ et $s = 1$, le bit mémorisé est modifié à 1.

5.4 Exercices

5.1) Décrivez un circuit logique qui teste si un nombre de n bits vaut zéro.

5.2) Décrivez un circuit logique qui teste si deux nombres sont égaux.

5.3) Reproduisez le décodeur et le multiplexeur présentés, mais cette fois avec 3 bits. Décrivez comment obtenir de tels circuits en général pour n bits.

5.4) Adaptez le multiplexeur présenté afin qu'il sélectionne parmi quatre *suites de n bits* plutôt que quatre bits.

6

Nombres entiers

6.1 Représentation des entiers signés

Jusqu'ici, nous avons considéré l'arithmétique des nombres entiers *non négatifs*, et ce à tous les niveaux: représentations, opérations et circuits. Nous considérons maintenant les entiers *signés*, c'est-à-dire non négatifs *et* négatifs. Nous devons d'abord établir une représentation de ces nombres. Une façon simple de représenter un entier signé a consiste à représenter sa valeur absolue $|a|$, qui est toujours non négative, ainsi qu'un bit de signe, c'est-à-dire un bit égal à 1 si et seulement si $a < 0$. Bien que cette représentation soit simple, elle complique l'implémentation des opérations arithmétiques. Par exemple, pour l'addition, il faut considérer les quatre valeurs possibles des deux bits de signe.

Une solution plus élégante consiste à utiliser le *complément à deux*. Dans cette représentation, la valeur d'une suite de bits $x_{n-1} \cdots x_1 x_0$ est définie par

$$-x_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} x_i \cdot 2^i.$$

Exemple.

Pour $n = 3$ bits, les valeurs possibles sont:

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

En général, la représentation par complément à deux sur n bits permet de représenter les entiers de -2^{n-1} à $2^{n-1}-1$.

Bien que cette représentation puisse paraître complexe à première vue, elle jouit de plusieurs propriétés intéressantes. Premièrement, un nombre est négatif si et seulement si son bit tout à gauche vaut 1. Deuxièmement, l'addition s'effectue exactement de la même façon que l'addition d'entiers non signés.

Exemple.

La somme $2 + (-3) = -1$ s'obtient ainsi:

$$\begin{array}{rr} & 010 \\ + & 101 \\ \hline & 111 \end{array}$$

Troisièmement, étant donné un entier a représenté par la suite de bits $x_{n-1} \cdots x_1 x_0$, il est possible de calculer $-a$ en deux étapes simples:

— inverser tous les bits de a: $(\neg x_{n-1}) \cdots (\neg x_1)(\neg x_0)$;

— additionner 1 au nombre obtenu à l'étape précédente.

Exemple.

Considérons $a = 2$ représenté par la suite 010. Nous obtenons la représentation de -2 ainsi:

$$010 \xrightarrow{\text{complément}} 101 \xrightarrow{+1} 110.$$

Il est important de noter que le concept de bits non significatifs n'est pas le même que pour les entiers non signés. Par exemple, considérons le nombre -2 représenté par 110. L'ajout d'un zéro à gauche mène à la suite 0110, qui ne représente *pas* la valeur -2. En effet, les nombres pouvant être représentés sur quatre bits sont:

bits	*valeur*
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

Ainsi, la suite 0110 représente 6.

Afin d'étendre correctement le nombre de bits d'un nombre, il faut copier son bit de signe (tout à gauche).

Exemple.

La suite 110 qui représente -2 sur trois bits peut être étendue à $\mathbf{1}110$ sur quatre bits. Similairement, la suite 011 qui représente 3 sur trois bits peut être étendue à $\mathbf{0}011$ sur quatre bits.

6.2 Addition

Tel qu'annoncé, l'addition d'entiers signés s'effectue de la *même* façon que l'addition d'entiers non signés. Ainsi, l'unité arithmétique et logique peut utiliser les mêmes circuits logiques.

6.2.1 Report

Lorsqu'une addition, en arithmétique signée ou non, engendre une retenue lors de la somme des bits les plus à gauche, nous disons qu'il y a *report*. La détection d'un report permet notamment d'identifier une erreur ou d'étendre l'addition sur n bits à l'addition sur $2n$ bits.

Exemple.

Imaginons une architecture dont l'unique instruction d'addition opère sur les demi-mots (4 bits), et que nous désirons effectuer la somme des nombres -105 et 61, chacun stocké sur deux demi-mots: 1001 0111 et 0011 1101. Afin de réaliser cette addition, nous effectuons d'abord la somme des demi-mots de droite:

$$\begin{array}{rr} & 0111 \\ + & 1101 \\ \hline \text{(report)} & 0100 \end{array}$$

Puisqu'il y a report, nous additionnons une retenue aux deux demi-mots de gauche:

$$\begin{array}{rr} & 0001 \\ & 1001 \\ + & 0011 \\ \hline & 1101 \end{array}$$

En assemblant les deux demi-mots résultants, nous obtenons 1101 0100 qui représente bien $-44 = -105 + 61$.

6.2.2 Débordement

Lors de l'addition de deux nombres de n bits, nous disons qu'il y a *débordement* si la somme ne peut pas être représentée sur n bits. Dans le cas de l'arithmétique non signée, il y a débordement précisément lorsqu'il y a report. Cela n'est toutefois pas le cas en arithmétique signée.

Exemple.

Considérons l'addition des nombres 5 et 6 sur quatre bits. Nous avons:

$$\begin{array}{rrl} & 0101 & (5) \\ + & 0110 & (6) \\ \hline & 1011 & (-5) \end{array}$$

Cette addition n'a pas de report, mais son résultat est erronné. En effet, la suite 1011 représente -5 plutôt que la somme attendue de 11. Ce problème surgit lorsque la somme de deux nombres positifs (resp. négatifs) excède la plus grande (resp. plus petite) valeur signée représentable.

Remarquons qu'il est impossible d'obtenir un débordement si deux nombres de signes opposés sont additionnés, bien qu'il soit possible d'obtenir un report.

6.3 Soustraction

La soustraction $a - b$ de deux entiers peut être effectuée en calculant le complément à deux de b, puis en effectuant une addition standard.

Exemple.

Nous avons $3 - 5 = 3 + (-5) = 0011 + 1011$:

$$\begin{array}{rll} & 0011 & (3) \\ + & 1011 & (\text{-}5) \\ \hline & 1110 & (\text{-}2) \end{array}$$

Un débordement peut se produire lorsque les termes de la soustraction sont de signes différents, mais pas lorsqu'ils sont de même signe.

6.4 Multiplication

6.4.1 Multiplication non signée

Il est possible d'implémenter la multiplication de deux nombres $a, b \in \mathbb{N}$ à l'aide de l'addition en exploitant l'identité:

$$a \cdot b = \underbrace{b + b + \ldots b}_{a-1 \text{ additions}}.$$

Cette méthode a l'avantage d'être simple, mais elle est lente lorsque a est grand. Il est possible de faire mieux en effectuant des multiplications par des puissances de 2, et en exploitant le fait que cela correspond à des décalages de bits.

Exemple.

Le produit $13 \cdot 11$ peut se calculer avec deux additions plutôt que douze:

$$\begin{aligned} 13 \cdot 11 &= 13 \cdot (2^3 + 2^1 + 2^0) \\ &= 13 \cdot 2^3 + 13 \cdot 2^1 + 2^0 \\ &= 1101_2 \cdot 2^3 + 1101_2 \cdot 2^1 + 1101_2 \cdot 2^0 \\ &= 1101000_2 + 11010_2 + 1101_2 \\ &= 10001111_2 \\ &= 143. \end{aligned}$$

Cette méthode correspond précisément à la méthode de multiplication usuelle utilisée dans le système décimal:

$$
\begin{array}{rrl}
 & 1101 & (13) \\
\times & 1011 & (11) \\
\hline
 & 1101 & \\
 & 1101 & \\
+ & 0000 & \\
 & 1101 & \\
\hline
 & 10001111 & (143)
\end{array}
$$

Cet algorithme de multiplication peut être implémenté efficacement tel que décrit à l'algorithme 4.

Algorithme 4 : Algorithme pour multiplier deux entiers non signés.

Entrées : deux entiers non signés a et b de n bits
Sorties : un entier non signé de $2n$ bits égal à $a \cdot b$

$\langle hi, lo \rangle \leftarrow \langle 0, b \rangle$ `// paire de deux nombres de` n `bits`
répéter n **fois**
  $r \leftarrow 0$
  si $lo_0 = 1$ **alors** $r, hi \leftarrow hi + a$ `//` $r = 1$ `s'il y a report`
  $\langle hi, lo \rangle \leftarrow \langle r\ hi_{n-1} \cdots hi_2\ hi_1, hi_0\ lo_{n-1} \cdots lo_2\ lo_1 \rangle$
retourner $\langle hi, lo \rangle$

Notons que la multiplication de deux nombres non signés de n bits nécessite au plus $2n$ bits. Il est donc toujours possible de multiplier, par exemple, deux nombres de 32 bits et de stocker le résultat sur 64 bits. Toutefois, le résultat peut nécessiter 64 bits, comme le démontre cette proposition plus générale:

Proposition 3. *Soit $n \in \mathbb{N}_{\geq 2}$ et soit a le plus grand entier non signé de n bits. La représentation binaire de $a \cdot a$ requiert $2n$ bits.*

Démonstration. Nous devons démontrer que a est plus grand que le plus grand entier non signé pouvant être représenté sur $2n - 1$ bits. Nous avons:

$$
\begin{aligned}
a \cdot a &= (2^n - 1)(2^n - 1) && \text{(par la proposition 2)} \\
&= 2^{2n} - 2^{n+1} + 1 \\
&> 2^{2n} - 2^{n+1} \\
&\geq 2^{2n} - 2^{2n-1} && \text{(car } 2n - 1 \geq n + 1 \text{ pour tout } n \geq 2\text{)} \\
&= 2 \cdot 2^{2n-1} - 2^{2n-1} \\
&= 2^{2n-1} \\
&> 2^{2n-1} - 1.
\end{aligned}
$$

□

6.4.2 Multiplication signée

La procédure de multiplication précédente s'applique seulement aux entiers non signés. Une méthode simple afin de multiplier deux entiers *signés* a et b de n bits consiste à

— étendre a et b à $2n$ bits avec les bons bits de signe;

— effectuer la multiplication standard (non signée) des deux nombres;

— garder les derniers $2n$ bits.

Exemple.

Nous pouvons calculer $5 \cdot -7 = -35$ ainsi:

$$
\begin{array}{rr}
 & 00000101 \quad (5) \\
\times & 11111001 \quad (-7) \\
\hline
 & 00000101 \\
 & 00000000 \\
+ & 00000000 \\
 & 00000101 \\
 & 00000101 \\
 & 00000101 \\
 & 00000101 \\
 & 00000101 \\
\hline
 & 0000100011011101 \quad (-35)
\end{array}
$$

ou bien ainsi:

$$
\begin{array}{rr}
 & 11111001 \quad (-7) \\
\times & 00000101 \quad (5) \\
\hline
 & 11111001 \\
 & 00000000 \\
+ & 11111001 \\
\hline
 & 1001101101 \quad (-35)
\end{array}
$$

Cet algorithme peut être implémenté de manière à ce que les bits additionnels ne soient pas ajoutés explicitement, et ainsi que la multiplication se fasse directement sur $2n$ bits (voir l'algorithme 5).

Algorithme 5 : Algorithme pour multiplier deux entiers signés.

Entrées : deux entiers signés a et b de n bits
Sorties : un entier signé de $2n$ bits égal à $a \cdot b$

$\langle hi, lo \rangle \leftarrow \langle 0, b \rangle$ `// paire de deux nombres de` n `bits`
répéter n **fois**
  $n \leftarrow 0;\ v \leftarrow 0$
  si *dernière itération* **alors** $a \leftarrow -a$
  si $lo_0 = 1$ **alors** $n, v, hi \leftarrow hi + a$ `//` $n =$ `nég.,` $v =$ `débord.`
  $\langle hi, lo \rangle \leftarrow \langle (n \oplus v)\ hi_{n-1}\ \cdots\ hi_2\ hi_1, hi_0\ lo_{n-1}\ \cdots\ lo_2\ lo_1 \rangle$
retourner $\langle hi, lo \rangle$

6.5 Division

6.5.1 Division non signée

La division de deux entiers non signés de n bits se calcule comme en base 10.

Exemple.

La division entière $10010_2 \div 11_2 = 19 \div 3 = 6$ reste $1 = 110_2$ reste 1_2 se calcule ainsi:

$$\begin{array}{rr|l} & 10011 & 11 \\ \hline - & 11 & 00110 \\ \hline & 111 & \\ - & 11 & \\ \hline & 1 & \end{array}$$

Cette procédure s'implémente efficacement tel que décrit à l'algorithme 6.

Algorithme 6 : Algorithme pour diviser deux entiers non signés.

Entrées : deux entiers non signés a et b de n bits
Sorties : deux entiers non signés q et r de n bits tels que $a \div b = q$ et $a \bmod b = r$

$\langle q, r \rangle \leftarrow \langle a, 0 \rangle$ `// paire de deux nombres de` n `bits`
répéter n **fois**
  $\langle q, r \rangle \leftarrow \langle q_{n-2}\ \cdots\ q_1\ q_0\ 0, r_{n-2}\ \cdots\ r_1\ r_0\ q_{n-1} \rangle$
  si $r \geq b$ **alors**
    $r \leftarrow r - b$
    $q_0 \leftarrow 1$
retourner $\langle q, r \rangle$

6.5.2 Division signée

La division signée $a \div b$ peut être effectuée en calculant $|a| \div |b|$, puis en ajustant le signe du quotient:

— si a et b sont de même signe, alors le quotient est non négatif;

— si a et b sont de signe différent, alors le quotient est négatif.

Notons toutefois que la définition de division d'entiers signés varie selon l'architecture et le langage de programmation.

Exemple.

Nous avons $-19 \div 3 = -6$ en C++ et $-19 \div 3 = -7$ en Python. En langage d'assemblage de l'architecture ARMv8, le résultat est $-19 \div 3 = -6$.

Remarquons également que la division du plus petit entier signé par -1 ne donne pas le résultat attendu. En effet, $-2^{n-1} \div -1 = 2^{n-1}$, et ce-dernier n'est pas représentable sur n bits.

6.6 Particularités de l'architecture ARMv8

6.6.1 Codes de condition

L'architecture ARMv8 possède quatre *codes de condition* [1]:

`N` (négatif), `Z` (zéro), `C` (report) et `V` (débordement).

Certaines instructions mettent les codes de condition à jour ainsi:

— `N` est vrai ssi le résultat est négatif;

— `Z` est vrai ssi le résultat vaut 0;

— `C` est vrai ssi il y a report;

— `V` est vrai ssi il y a débordement.

Lors de la comparaison de deux registres avec « `cmp xd, xm` », la soustraction $\texttt{x}_\texttt{d} - \texttt{x}_\texttt{m}$ est effectuée, les codes de condition sont mis à jour selon la différence, et celle-ci est jetée. Les instructions `adds`, `subs` et `negs` mettent également les codes de condition à jour, et se comportent respectivement comme `add`, `sub` et `neg`.

Le code `C` peut être manipulé avec l'instruction « `adc xd, xn, xm` ». Celle-ci stocke dans $\texttt{x}_\texttt{d}$ la somme de $\texttt{x}_\texttt{n}$, $\texttt{x}_\texttt{m}$, ainsi que 1 si `C` est vrai. L'instruction « `sbc` » fonctionne similairement pour la soustraction. Remarquons que dans le cas d'une addition, `C` indique la présence d'un report, alors que dans le cas d'une soustraction, `C` indique l'absence d'emprunt.

Les codes de condition permettent également de faire des branchements conditionnels à l'aide de l'instruction « `b.condition etiq` »:

1. Les lettres `C` et `V` proviennent de l'anglais « ***c**arry* » et « *o**v**erflow* ».

Entiers non signés		
Condition	***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		
Condition	***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	$\neg\mathtt{N}$

6.6.2 Accès mémoire

Lors du chargement d'un entier signé — stocké dans un octet, un demi-mot ou un mot — vers un registre, il est possible de prendre le bit de signe en compte grâce aux instructions suivantes:

nombre d'octets	***instruction***
1	`ldrsb xd, a`
2	`ldrsh xd, a`
4	`ldrsw xd, a`

Par exemple, `ldrsw` charge un mot signé w dans les 32 bits de poids faible du registre, et remplit les 32 bits de poids fort avec le bit de signe de w.

Remarquons que si « `ldrsh xd, etiquette` », par exemple, ne compile pas dû aux détails spécifiques du code machine, il est possible d'utiliser:

```
adr   xm, etiquette
ldrsh xd, [xm]
```

Remarque.

Des exemples de programmes C/C++ sont disponibles sur GitHub.

6.7 Exercices

6.1) Écrivez les entiers signés 43, -43, 1, -1, 127, -128 sur 8 bits.

6.2) Donnez le résultat des opérations signées suivantes sur 8 bits: $43 + 25$, $43 - 25$ $127 + 127$, $-128 - 128$ et $127 - 128$.

6.3) Calculez le produit $a \cdot b$ des entiers signés $a = 6$ et $b = 7$ de 4 bits.

6.4) Donnez un exemple d'addition signée, sur 4 bits, où il y a débordement sans report, puis un exemple du scénario inverse.

6.5) Soient $a = 1101011$ et $b = 100100$ des entiers signés de 6 et 7 bits.

- Effectuez la soustraction $a - b$.
- Donnez la valeur en base 10 du résultat de la soustraction précédente.
- Supposons que le registre $\texttt{x}_{19}$ contienne a (étendu sur 64 bits), et que le registre $\texttt{x}_{20}$ contienne b (étendu sur 64 bits). Donnez la valeur des codes de condition N (négatif), Z (zéro) et V (débordement) après l'exécution de l'instruction « `cmp x19, x20` ».
- Dites si $a > b$.

(*tiré de l'examen périodique de l'hiver 2019*)

6.6) Donnez un circuit logique qui calcule le complément à deux d'un nombre, sans réutiliser les circuits conçus au chapitre 5. Réfléchissez d'abord à une entrée d'un bit, de deux bits, de trois bits, etc. puis généralisez à n bits.

6.7) ★★ Montrez que l'extension d'un entier signé par son bit de signe préserve bel et bien sa valeur.

7

Tableaux

Nous considérons maintenant un type élémentaire de données qui permet de structurer d'autres données: les tableaux. Un *tableau* est une collection ordonnée d'éléments identifiés par des indices numériques. Les éléments d'un tableau possèdent tous la même taille et, dans notre contexte, sont stockés de façon contigüe en mémoire.

Exemple.

Le tableau ci-dessous possède cinq éléments, tous stockés sur un octet, et identifiés par les indices 0, 1, 2, 3 et 4.

0	`01010101`
1	`11110000`
2	`01101101`
3	`11111111`
4	`11110101`

En général, un tableau possède une *dimension* $d \in \mathbb{N}_{\geq 1}$ ainsi que d bornes positives $n_0, n_1, \ldots, n_{d-1} \in \mathbb{N}_{\geq 1}$. Un *indice* est une collection ordonnée $\boldsymbol{i}$ d'arité d (un « d-uplet ») telle que:

$$\begin{array}{ccccc} 0 & \leq & \boldsymbol{i}_0 & < & n_0, \\ 0 & \leq & \boldsymbol{i}_1 & < & n_1, \\ \vdots & & \vdots & & \vdots \\ 0 & \leq & \boldsymbol{i}_{d-1} & < & n_{d-1}. \end{array}$$

Chaque indice $\boldsymbol{i}$ d'un tableau t est associé à un unique élément $t[\boldsymbol{i}]$ dont la valeur est stockée sur un nombre fixe d'octets k. Ainsi, un tableau possède $n = n_0 \cdot n_1 \cdots n_{d-1}$ éléments représentés globalement sur $k \cdot n$ octets.

Exemple.

Le tableau illustré ci-dessous est bidimensionnel ($d = 2$) et possède les bornes $n_0 = 3$ et $n_1 = 2$. Il contient $n = 3 \cdot 2 = 6$ éléments, tous stockés sur un demi-mot ($k = 2$). Ses éléments sont identifiés par les indices $(0, 0)$, $(0, 1)$, $(1, 0)$, $(1, 1)$, $(2, 0)$ et $(2, 1)$.

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

Un tel tableau 2D peut être interprété comme une représentation linéaire d'une matrice:

$$\begin{pmatrix} 2 & 33 \\ 65535 & 73 \\ 9000 & 255 \end{pmatrix}$$

Remarquons que la dimension d'un tableau, ainsi que ses indices, sont seulement connus *implicitement*. Par exemple, ces trois tableaux peuvent être représentés exactement de la même façon en mémoire:

$$[2, 33, 65535, 73, 9000, 255] \quad \begin{pmatrix} 2 & 3 \\ 65535 & 73 \\ 9000 & 255 \end{pmatrix} \quad \begin{pmatrix} 2 & 3 & 65535 \\ 73 & 9000 & 255 \end{pmatrix}$$

Similairement, le type des éléments d'un tableau n'est qu'*implicite* et peut être interprété différemment à l'exécution. Par exemple, considérons un tableau initialisé avec m blocs de 8 octets. Ses éléments peuvent autant être vus comme des entiers non signés que des entiers signés de 64 bits.

Observation.

En fait, ils pourraient même être vus comme $2m$ entiers de 32 bits (signés ou non), ou n'avoir simplement aucun type.

7.1 Accès aux éléments

La manipulation des éléments d'un tableau requiert le calcul de leur adresse en mémoire. Nous expliquons donc comment les calculer.

Considérons un tableau t de dimension d situé à l'adresse a de la mémoire principale, et dont les éléments sont stockés sur k octets. L'adresse relative à

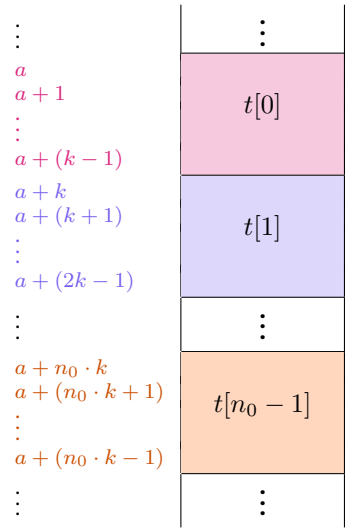
laquelle est stocké un élément de t se nomme son *index*. Autrement dit, l'index correspond à la distance d'un élément rapport à l'adresse a.

7.1.1 Cas unidimensionnel

Nous écrivons $\texttt{mem}_k[b]$ afin de référer au contenu stocké sur k octets à l'adresse b de la mémoire principale. Pour un tableau *unidimensionnel*, donc où $d = 1$, nous avons:

$$\begin{aligned} t[0] &= \texttt{mem}_k[a], \\ t[1] &= \texttt{mem}_k[a + k], \\ t[2] &= \texttt{mem}_k[a + 2 \cdot k], \\ \vdots &\quad \vdots \\ t[n_0 - 1] &= \texttt{mem}_k[a + (n_0 - 1) \cdot k]. \end{aligned}$$

ou de façon équivalente sous forme de diagramme:



Ainsi, l'élément $t[i]$ se situe à l'adresse absolue $a + i \cdot k$ et son index vaut $i \cdot k$.

7.1.2 Cas bidimensionnel

Pour un tableau bidimensionnel, donc où $d = 2$, nous avons:

$$
\begin{aligned}
t[0,0] &= \texttt{mem}_k[a], & t[1,0] &= \texttt{mem}_k[a + (n_1 + 0) \cdot k],\\
t[0,1] &= \texttt{mem}_k[a + k], & t[1,1] &= \texttt{mem}_k[a + (n_1 + 1) \cdot k],\\
t[0,2] &= \texttt{mem}_k[a + 2 \cdot k], & t[1,2] &= \texttt{mem}_k[a + (n_1 + 2) \cdot k],\\
&\vdots \quad \vdots & &\vdots \quad \vdots\\
t[0,n_1-1] &= \texttt{mem}_k[a + (n_1 - 1) \cdot k], & t[1,n_1-1] &= \texttt{mem}_k[a + (n_1 + (n_1 - 1)) \cdot k],\\
& & t[2,0] &= \texttt{mem}_k[a + (2n_1 + 0) \cdot k],\\
& & t[2,1] &= \texttt{mem}_k[a + (2n_1 + 1) \cdot k],\\
& & &\vdots \quad \vdots
\end{aligned}
$$

Ainsi, l'élément identifié par l'indice (i, j) se situe à l'adresse:

$$a + \underbrace{(i \cdot n_1 + j) \cdot k}_{\text{index de l'élém. } (i,j)}.$$

7.2 Particularités de l'architecture ARMv8

7.2.1 Allocation et initialisation

Considérons à nouveau le tableau bidimensionnel illustré à l'exemple précédent. Rappelons que ses éléments sont stockés sur deux octets, donc des demi-mots. Il est possible d'allouer statiquement le tableau en mémoire dans le segment de données non initialisées:

```
.section ".bss"
        .align 2
tab:    .skip  12
```

Afin de faciliter la lecture du code, des macros peuvent aussi être utilisées afin d'indiquer explicitement les bornes du tableau:

```
N0 = 3
N1 = 2

.section ".bss"
        .align 2
tab:    .skip  N0*N1*2
```

Le tableau peut être rempli à l'aide de l'instruction `strh` (et non `str`, puisque les éléments sont sur 2 octets):

```
    adr     x19, tab            //
    mov     w20, 2              //
    strh    w20, [x19], 2       // tab[0] = 2
    mov     w20, 33             //
    strh    w20, [x19], 2       // tab[1] = 33
    mov     w20, 65535          //
    strh    w20, [x19], 2       // tab[2] = 65535
    mov     w20, 73             //
    strh    w20, [x19], 2       // tab[3] = 73
    mov     w20, 9000           //
    strh    w20, [x19], 2       // tab[4] = 9000
    mov     w20, 255            //
    strh    w20, [x19]          // tab[5] = 255
```

Notons que nous utilisons ici le mode d'adressage indirect par registre post-incrémenté, ce qui évite d'incrémenter x_{19} manuellement.

Alternativement, les éléments du tableau peuvent être alloués et initialisés directement dans le segment de données initialisées:

```
.section ".data"
tab:    .hword  2, 33, 65535, 73, 9000, 255
```

7.2.2 Parcours d'un tableau

Parcours linéaire. Les éléments d'un tableau peuvent être affichés linéairement avec le code ci-dessous. Ici, l'adresse du tableau est stockée dans x_{19}, l'indice courant dans x_{20} et l'index courant dans x_{21}. L'accès à un élément du tableau s'effectue ici avec le mode d'adressage indirect par registre indexé dénoté par « `[x19, x21]` »:

```
N0 = 3
N1 = 2

main:
    adr     x19, tab            //
    mov     x20, 0              // i = 0
afficher:                       // do {
    adr     x0, fmt             //
    mov     x21, 2              //
    mul     x21, x21, x20       //
    ldrh    w1, [x19, x21]      //
    bl      printf              //   afficher tab[i]
                                //
    add     x20, x20, 1         //   i++
    cmp     x20, N0*N1          // }
    b.lo    afficher            // while (i < N0*N1)
```

```
    mov     x0, 0
    bl      exit

.section ".data"
tab:    .hword   2, 33, 65535, 73, 9000, 255

.section ".rodata"
fmt:    .asciz  "%u\n"
```

Il est également possible d'accéder aux éléments grâce au mode d'adressage indirect par registre indexé post-incrémenté, dénoté par « `[x19], 2` ». Cela simplifie le code puisqu'il n'est pas nécessaire de calculer explicitement l'index:

```
N0 = 3
N1 = 2

main:
    adr     x19, tab            //
    mov     x20, 0              // i = 0
afficher:                       // do {
    adr     x0, fmt             //
    ldrh    w1, [x19], 2        //
    bl      printf              //   afficher tab[i]
                                //
    add     x20, x20, 1         //   i++
    cmp     x20, N0*N1          // }
    b.lo    afficher            // while (i < N0*N1)

    mov     x0, 0
    bl      exit

.section ".data"
tab:    .hword   2, 33, 65535, 73, 9000, 255

.section ".rodata"
fmt:    .asciz  "%u\n"
```

Parcours bidimensionnel. Le code ci-dessous permet d'afficher les éléments sous forme matricielle. Ici, l'adresse du tableau est stockée dans x_{19}, l'indice courant (i, j) dans (x_{20}, x_{21}), et l'index courant dans x_{22}. Ce-dernier est calculé à partir de l'expression $(i \cdot n_1 + j) \cdot 2$ puisque les éléments sont stockés sur 2 octets. Nous utilisons ici le mode d'adressage indirect par registre indexé, dénoté par « `[x19, x22]` »:

```
N0 = 3
N1 = 2

main:
    adr     x19, tab            //
    mov     x20, 0              // i = 0
                                //
prochaineLigne:                 // do {
    mov     x21, 0              //   j = 0
                                //
afficherLigne:                  //   do {
    adr     x0, fmtElem         //
                                //
    // Calcul de l'index        //
    mov     x22, N1             //
    mul     x22, x20, x22       //
    add     x22, x22, x21       //
    add     x22, x22, x22       //     index = (i*N1 + j)*2
                                //
    ldrh    w1, [x19, x22]      //
    bl      printf              //     afficher tab[i, j]
                                //
    add     x21, x21, 1         //     j++
    cmp     x21, N1             //   }
    b.lo    afficherLigne       //   while (j < N1)
                                //
    adr     x0, fmtSaut         //
    bl      printf              //   afficher saut de ligne
                                //
    add     x20, x20, 1         //   i++
    cmp     x20, N0             // }
    b.lo    prochaineLigne      // while (i < N0)

    mov     x0, 0
    bl      exit

.section ".data"
tab:        .hword  2, 33, 65535, 73, 9000, 255

.section ".rodata"
fmtElem:    .asciz  "%u "
fmtSaut:    .asciz  "\n"
```

Comme dans le cas linéaire, on peut accéder aux éléments du tableau à l'aide du mode d'adressage indirect par registre indexé post-incrémenté « `[x19], 2` », ce qui simplifie le code:

```
N0 = 3
N1 = 2

main:
    adr     x19, tab            //
    mov     x20, 0              // i = 0
                                //
prochaineLigne:                 // do {
    mov     x21, 0              //   j = 0
                                //
afficherLigne:                  //   do {
    adr     x0, fmtElem         //
    ldrh    w1, [x19], 2        //
    bl      printf              //     afficher tab[i, j]
                                //
    add     x21, x21, 1         //     j++
    cmp     x21, N1             //   }
    b.lo    afficherLigne       //   while (j < N1)
                                //
    adr     x0, fmtSaut         //
    bl      printf              //   afficher saut de ligne
                                //
    add     x20, x20, 1         //   i++
    cmp     x20, N0             // }
    b.lo    prochaineLigne      // while (i < N0)

    mov     x0, 0
    bl      exit

.section ".data"
tab:        .hword  2, 33, 65535, 73, 9000, 255

.section ".rodata"
fmtElem:    .asciz  "%u "
fmtSaut:    .asciz  "\n"
```

7.3 Autre exemple: tableaux de pointeurs

Voyons un exemple où les éléments d'un tableau ne sont pas des valeurs numériques, mais plutôt une collection de procédures à sélectionner dynamiquement.

Considérons un programme qui, sur entrée $i \in \{0, 1, 2\}$, doit afficher $x + y$ si $i = 0$, $x \cdot y$ si $i = 1$, et $x - y$ si $i = 2$. Nous pourrions effectuer une suite de comparaisons et brancher selon la valeur de i. Il est toutefois possible d'éviter de telles comparaisons en utilisant un tableau t de pointeurs, où $t[0]$, $t[1]$ et $t[2]$

contiennent respectivement l'adresse de segments de code calculant l'addition, le produit et la différence.

Par exemple, voici une implémentation où les valeurs $x = 5$ et $y = 7$ sont figées afin de simplifier la présentation:

```
main:
    // Initialiser tab                  //
    adr     x19, tab                    //
    adr     x20, somme                  //
    str     w20, [x19], 4               //
    adr     x20, produit                //
    str     w20, [x19], 4               // tab = {&somme,
    adr     x20, diff                   //        &produit,
    str     w20, [x19]                  //        &diff}
                                        //
    // Lire chiffre                     //
    adr     x0, fmtEntree               //
    adr     x1, temp                    //
    bl      scanf                       // scanf("%u", &temp)
    ldr     x21, temp                   // i = temp
                                        //
    // Évaluer f(7, 5) où f := tab[i]   //
    mov     x20, 4                      //
    mul     x22, x20, x21               //
    adr     x19, tab                    //
    ldr     w20, [x19, x22]             // f = tab[i]
    mov     x27, 7                      // x = 7
    mov     x28, 5                      // y = 5
    br      x20                         // f(x, y)
                                        //
somme:                                  // somme(x, y) {
    add     x1, x27, x28                //   z = x + y
    b       afficher                    // }
produit:                                // produit(x, y) {
    mul     x1, x27, x28                //   z = x * y
    b       afficher                    // }
diff:                                   // diff(x, y) {
    sub     x1, x27, x28                //   z = x - y
    b       afficher                    // }
                                        //
    // Afficher résultat                //
afficher:                               //
    adr     x0, fmtSortie               //
    bl      printf                      // printf("%lu\n", z)
                                        //
```

```
    mov     x0, 0                       //
    bl      exit                        //

.section ".bss"
tab:    .skip   12                      // tableau de 3 adresses
temp:   .skip   4

.section ".rodata"
fmtEntree:  .asciz  "%u"
fmtSortie:  .asciz  "%lu\n"
```

Remarque.

Des implémentations en C/C++ sont également fournies sur GitHub.

7.4 Exercices

7.1) Écrivez un programme qui affiche les éléments de la diagonale principale d'une matrice.

(*basé sur une question de l'examen périodique de l'hiver 2019*)

7.2) Nous avons représenté les matrices par des tableaux bidimensionnels en stockant les lignes l'une à la suite des autres. Considérez la représentation alternative où ce sont plutôt les colonnes qui sont stockées l'une à la suite des autres. Revoyez la formule du calcul d'index et écrivez un programme qui affiche une telle matrice.

7.3) Dites pourquoi un tableau de 23 éléments est forcément unidimensionnel.

7.4) ★ Écrivez un programme qui implémente le jeu de la vie. Il doit lire une matrice binaire (où 0 représente une cellule morte, et 1 représente une cellule vivante), puis retourner une matrice qui représente le nouvel état des cellules après une application des règles.

8

Programmation structurée

Les langages d'assemblage appartiennent à la famille des langages de *programmation impérative*: on utilise des expressions qui indiquent *comment* mettre à jour l'*état* d'un programme. La *programmation structurée* a émergé dans les années 60–70 afin de faciliter l'écriture, la lecture et la maintenabilité des programmes impératifs. L'un de ses ambassadeurs les plus influents, Edsger Dijkstra, plaidait à l'époque pour l'abandon de l'instruction `goto` au profit de *structures de contrôle* et de *sous-routines*. De nos jours, ces concepts se trouvent au cœur de langages de programmation système comme C, C++ et Rust, et de langages de haut niveau comme Python, Java et C#.

Dans ce chapitre, nous expliquons comment les notions de programmation structurée s'implémentent en langage d'assemblage.

8.1 Structures de contrôle

Voyons d'abord comment les structures de contrôle de haut niveau peuvent être implémentées à l'aide de code de bas niveau. Les constructions sont illustrées en C et C++ (haut niveau), ainsi que dans le langage d'assemblage de l'architecture ARMv8 (bas niveau). Nous considérons les trois types élémentaires de structures de contrôle de la programmation structurée: la *séquence*, la *sélection* et l'*itération*. Dans nos constructions, les termes « `cond` », « $\texttt{cond}_1$ » et « $\texttt{cond}_2$ » dénotent des prédicats d'arité deux; autrement dit, des fonctions qui prennent deux valeurs en entrée et qui retournent une valeur booléenne.

8.1.1 Séquence

La *séquence* consiste simplement à composer des instructions de façon séquentielle. Cette structure est implémentée en traduisant chaque instruction vers du code de bas niveau équivalent:

Code de haut niveau (C/C++)	Code de bas niveau (ARMv8)

`// instruction 1`
`// instruction 2`
`// ...`
`// instruction k`
	`// code pour l'instruction 1`
	`// code pour l'instruction 2`
	`// ...`
	`// code pour l'instruction k`

Remarquons qu'une instruction de haut niveau peut nécessiter plusieurs instructions de bas niveau selon l'architecture. Par exemple, sur l'architecture ARMv8, l'opération « `x19 *= 7` » se traduit vers:

```
mov    x20, 7
mul    x19, x19, x20
```

8.1.2 Sélection

La *sélection* est une structure de contrôle qui permet d'exécuter certaines instructions conditionnellement à la validité d'un ou plusieurs prédicats; par exemple: une exécution conditionnelle à l'égalité de deux variables. Voyons comment implémenter les structures de sélection *si/sinon–si/sinon* et de type « *switch* ».

Si.

Code de haut niveau (C/C++)	Code de bas niveau (ARMv8)

`if (cond(xd, xn)) {`
`  // code`
`}`
	`si:`
	`  cmp  xd, xn`
	`  b.¬cond  fin`
	`  // code`
	`fin:`

Si/sinon.

Code de haut niveau (C/C++)	Code de bas niveau (ARMv8)

`if (cond(xd, xn)) {`
`  // code si`
`}`
`else {`
`  // code sinon`
`}`
	`si:`
	`  cmp  xd, xn`
	`  b.¬cond  sinon`
	`  // code si`
	`  b  fin`
	`sinon:`
	`  // code sinon`
	`fin:`

Si/sinon–si/sinon.

Code de haut niveau (C/C++)

```
if (cond1(xd, xn)) {
  // code si
}
else if (cond2(xd, xn)) {
  // code sinon si
}
else {
  // code sinon
}
```

Code de bas niveau (ARMv8)

```
si:
        cmp         xd, xn
        b.¬cond1    sinonsi
        // code si
        b           fin
sinonsi:
        cmp         xd, xn
        b.¬cond2    sinon
        // code sinon si
        b           fin
sinon:
    // code sinon
fin:
```

Sélection de type « switch ».

Code de haut niveau (C/C++)

```
switch (xd) {
  case v1:
    // code cas 1
    break;
  case v2:
    // code cas 2
    break;
  /* ... */
  case vk:
    // code cas k
    break;
  default:
    // code par défaut
    break;
}
```

Code de bas niveau (ARMv8)

```
cas1:
        cmp         xd, v1
        b.ne        cas2
        // code cas 1
        b           fin
cas2:
        cmp         xd, v2
        b.ne        cas3
        // code cas 2
        b           fin
        /* ... */
cask:
        cmp         xd, vk
        b.ne        defaut
        // code cas k
        b           fin
defaut:
        // code par défaut
fin:
```

Remarque.

Lorsque les valeurs $v_1, v_2, \ldots, v_k$ d'une structure « switch » sont rapprochées, il peut s'avérer plus efficace d'utiliser une « *table de branchement* » qui indique l'adresse du code de chaque cas, à la façon de la section 7.3.

Sélection avec conditions multiples. Dans les langages de haut niveau, il est normalement possible d'évaluer des prédicats sur plusieurs variables, et de combiner ces prédicats à l'aide d'opérateurs logiques tels que $\wedge$ et $\vee$. Cependant, la plupart des langages d'assemblage ne permettent de tester qu'*une seule* condition sur *un* ou *deux* registres. Il est possible de traduire ces prédicats plus complexes à l'aide de plusieurs branchements.

Nous donnons deux exemples, la conjonction et la disjonction de deux prédicats: « $\texttt{cond}_1(x_d, x_n) \wedge \texttt{cond}_2(x_m, x_k)$ » et « $\texttt{cond}_1(x_d, x_n) \vee \texttt{cond}_2(x_m, x_k)$ ». Ces constructions se généralisent à un nombre arbitraire de prédicats et à d'autres opérateurs logiques.

Code de haut niveau (C/C++)

```
if (cond1(xd, xn) &&
    cond2(xm, xk)) {
  // code si
}
else {
  // code sinon
}
```

Code de bas niveau (ARMv8)

```
si:
        cmp         xd, xn
        b.¬cond1    sinon
        cmp         xm, xk
        b.¬cond2    sinon
        // code si
        b           fin
sinon:
        // code sinon
fin:
```

Code de haut niveau (C/C++)

```
if (cond1(xd, xn) ||
    cond2(xm, xk)) {
  // code si
}
else {
  // code sinon
}
```

Code de bas niveau (ARMv8)

```
        cmp         xd, xn
        b.cond1     si
        cmp         xm, xk
        b.cond2     si
        b           sinon
si:
        // code si
        b           fin
sinon:
        // code sinon
fin:
```

8.1.3 Itération

L'*itération* est une structure de contrôle qui répète l'exécution de certaines instructions en fonction de l'état du programme; par exemple, une répétition qui dépend de l'égalité de deux variables. Voyons comment implémenter les structures *tant que*, *faire/tant que* et *pour*.

Tant que.

Code de haut niveau (C/C++)	Code de bas niveau (ARMv8)

`while (cond(xd, xn)) {`
`  // code`
`}`
	`boucle:`
	`  cmp  xd, xn`
	`  b.¬cond  fin`
	`  // code`
	`  b  boucle`
	`fin:`

Faire/tant que.

Code de haut niveau (C/C++)	Code de bas niveau (ARMv8)

`do {`
`  // code`
`} while (cond(xd, xn));`
	`boucle:`
	`  // code`
	`  cmp  xd, xn`
	`  b.cond  boucle`
	`fin:`

Boucle « pour ». L'un des cas les plus répandus de la boucle « pour » consiste à incrémenter une variable, initialisée à 0, tant que sa valeur est inférieure ou égale à la valeur d'une autre variable. Cette structure s'implémente ainsi:

Code de haut niveau (C/C++)	Code de bas niveau (ARMv8)

`for (xd = 0; xd <= xn; xd++) {`
`  // code`
`}`
	`  mov  xd, 0`
	`boucle:`
	`  cmp  xd, xn`
	`  b.hi  fin`
	`  // code`
	`  add  xd, xd, 1`
	`  b  boucle`
	`fin:`

Le cas général de la boucle « pour », où l'initialisation et l'incrémentation sont arbitraires, s'implémente ainsi:

Code de haut niveau (C/C++)	Code de bas niveau (ARMv8)

`for (init(xd); cond(xd, xn);`
`  modif(xd)) {`
`  // code`
`}`
	`  // initialiser xd`
	`boucle:`
	`  cmp  xd, xn`
	`  b.¬cond  fin`
	`  // code`
	`  // modifier xd`
	`  b  boucle`
	`fin:`

8.2 Sous-programmes

Les langages structurés offrent des primitives pour décrire des *sous-routines* qui modularisent le code; par exemple, sous la forme de procédures, de fonctions ou de méthodes. De telles primitives n'existent pas dans les langages de bas niveau. Elles doivent être implémentées à l'aide de *sous-programmes*: des blocs de code exécutables à l'aide de branchements, de passage d'arguments et de sauvegarde de registres.

À titre d'exemple, nous allons implémenter une fonction qui reçoit deux valeurs entières et qui retourne le maximum de ces valeurs. En C/C++, cette fonction peut être implémentée ainsi:

```
long max(long a, long b)
{
  if (a >= b) {
    return a;
  }
  else {
    return b;
  }
}
```

Notons que les sous-programmes ne s'implémentent pas de la même façon sur toutes les architectures, nous nous limitons ici à l'architecture ARMv8.

8.2.1 Paramètres et appel

Par convention, un sous-programme reçoit ses paramètres dans les registres x_0 à x_7 (dans cet ordre), et retourne sa sortie via le registre x_0. Ainsi, nous utiliserons les registres x_0 et x_1 pour passer les valeurs a et b, et le registre x_0 pour retourner la valeur `max(a, b)`.

Afin de déclarer notre sous-programme « `max` », il suffit d'ajouter une étiquette « `max:` ». L'appel du sous-programme se réalise avec l'instruction de branchement « `bl` » (*branch with link*), que nous avons déjà utilisée aux chapitres précédents pour réaliser des entrées/sorties.

Si nous désirons stocker le maximum des registres x_{19} et x_{20} dans x_{21}, alors nous pouvons utiliser le code suivant:

```
main:
        mov         x0, x19         // a   = x19
        mov         x1, x20         // b   = x20
        bl          max             // m   = max(a, b)
        mov         x21, x0         // x21 = m
                                    //
        mov         x0, 0           // sans «bl exit»,
        bl          exit            // l'exécution se
                                    // poursuit dans «max»
```

```
max:
    // code du sous-programme ici
```

Passage par adresse. Dans notre exemple, les arguments sont passés *par valeur* puisqu'il s'agit d'entiers de 64 bits stockables dans les registres. Les structures de données plus complexes et stockées dans la mémoire principale doivent plutôt être passées *par adresse*. Autrement dit, plutôt que de passer le contenu de la structure, son adresse en mémoire est passée via un registre. Par exemple, un tableau peut être passé comme premier argument d'un sous-programme en chargeant son adresse dans x_0; le sous-programme peut ensuite accéder aux éléments du tableau grâce aux instructions d'accès mémoire comme « `ldr` ».

8.2.2 Retour

L'instruction « `ret` » termine l'exécution d'un sous-programme et saute vers l'endroit où l'appel a été effectué. La valeur de retour du sous-programme doit être stockée dans le registre x_0 avant le retour. Ainsi, notre fonction « `max` » peut être implémentée ainsi:

```
max:
        cmp         x0, x1          //
        b.lt        max_sinon       // if (a >= b) {
        mov         x19, x0         //   m = a
        b           max_retour      // }
max_sinon:                          // else {
        mov         x19, x1         //   m = b
max_retour:                         // }
        mov         x0, x19         //
        ret                         // return m
```

Puisqu'on peut altérer le contenu des registres de paramètres, notre code se simplifie ainsi:

```
max:
        cmp         x0, x1          //
        b.ge        max_retour      // if (a >= b) return a
        mov         x0, x1          // else return b
max_retour:                         //
        ret                         //
```

Adresse de retour. Lors de l'appel d'un sous-programme à l'aide de l'instruction « `bl` », l'adresse de retour est stockée dans le registre spécial x_{30}. Puisque les instructions de l'architecture ARMv8 sont toutes codées sur 4 octets, l'appel « `bl etiq` » effectue d'abord l'assignation $x_{30} \leftarrow \text{pc} + 4$, puis branche vers l'étiquette « `etiq:` ». L'instruction « `ret` » effectue le retour en branchant vers x_{30}.

8.2.3 Sauvegarde des registres

Les registres sont partagés par le programme et *tous* les sous-programmes. Par conséquent, l'appel d'un sous-programme peut détruire le contenu des registres de l'*appelant*. Par convention, l'*appelé* peut altérer les registres x_9 à x_{15}, mais est *en charge* de rétablir le contenu des registres x_{19} à x_{28}, que nous avons utilisés jusqu'ici, ainsi que les registres spéciaux x_{29} et x_{30}.

Un sous-programme peut sauvegarder et rétablir ces registres à l'aide des macros suivantes [1]:

Macro de sauvegarde	Macro de restauration

`.macro SAVE`
`  stp  x29, x30, [sp, -96]!`
`  mov  x29, sp`
`  stp  x27, x28, [sp, 16]`
`  stp  x25, x26, [sp, 32]`
`  stp  x23, x24, [sp, 48]`
`  stp  x21, x22, [sp, 64]`
`  stp  x19, x20, [sp, 80]`
`.endm`
	`.macro RESTORE`
	`  ldp  x27, x28, [sp, 16]`
	`  ldp  x25, x26, [sp, 32]`
	`  ldp  x23, x24, [sp, 48]`
	`  ldp  x21, x22, [sp, 64]`
	`  ldp  x19, x20, [sp, 80]`
	`  ldp  x29, x30, [sp], 96`
	`.endm`

Ces macros utilisent une pile située dans la mémoire principale afin de sauvegarder temporairement la valeur des registres. Nous discuterons du fonctionnement de cette pile plus tard au chapitre 11. En particulier, la pile nous permettra de mettre au point des sous-programmes récursifs.

Exemple complet. En ajoutant l'appel de ces deux macros, nous obtenons ce code complet pour notre exemple:

```
main:
        mov         x0, x19         // a   = x19
        mov         x1, x20         // b   = x20
        bl          max             // m   = max(a, b)
        mov         x21, x0         // x21 = m
                                    //
        bl          exit            //
                                    //
max:                                //
        SAVE                        //
        cmp         x0, x1          //
        b.lt        max_sinon       // if (a >= b) {
        mov         x19, x0         //   m = a
        b           max_retour      // }
max_sinon:                          // else {
        mov         x19, x1         //   m = b
```

1. Macros tirées initialement d'un diaporama de Mikaël Fortin.

```
max_retour:                             // }
        mov         x0, x19             //
        RESTORE                         //
        ret                             // return m
```

8.3 Autres particularités de l'architecture ARMv8

8.3.1 Distance des adresses

L'instruction « `bl` » reçoit une adresse relative représentée sur 26 bits. Ainsi, elle permet de brancher vers une étiquette dont l'adresse se situe à une distance de ±128Mio du compteur d'instruction. En effet:

$$4 \cdot 2^{26} = 2^{28} = 2 \cdot 2^7 \cdot (2^{10})^2 = 2 \cdot 128 \cdot 1024^2.$$

Afin de brancher vers une adresse située plus loin, il faut d'abord stocker cette adresse dans un registre x_d, puis brancher à l'aide de « `blr xd` ».

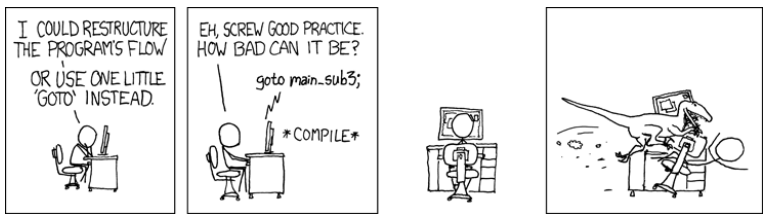
8.3.2 Assignation par sélection

L'instruction « `csel` » affecte une valeur à un registre de façon conditionnelle:

instruction	*effet*
`csel xd, xn, xm, cond`	si *cond*: $x_d \leftarrow x_n$, sinon: $x_d \leftarrow x_m$

Cette instruction permet, par exemple, de simplifier le code de `max`:

```
max:
        cmp         x0, x1              //
        csel        x0, x0, x1, ge      //
        ret                             // return (a >= b) ? a : b
```



— xkcd

8.4 Exercices

8.1) Traduisez le code C/C++ ci-dessous en langage d'assemblage. Interprétez x_{19}, x_{20} et x_{21} comme des entiers signés.

```
if (x19 > x20 && (x20 != x21 || x21 == x19)) {
  // A
}
else if (x19 < x20) {
  // B
}
else {
  // C
}
```

8.2) Comment un sous-programme, qui reçoit un tableau en argument, peut-il itérer à travers tous ses éléments?

8.3) Écrivez un sous-programme `elem(tab, i)` qui reçoit un tableau `tab` d'entiers signés et un indice `i`, puis qui retourne `tab[i]`.

8.4) Écrivez un sous-programme `renverser(tab, taille)` qui reçoit un tableau `tab` d'entiers signés ainsi que sa taille, puis qui renverse le contenu du tableau en mémoire.

8.5) Modularisez des segments de code conçus précédemment. Par exemple, écrivez des sous-programmes pour le calcul du temps de vol du chapitre 3 et pour le programme obtenu à l'exercice 7.1).

8.6) ★ Écrivez un programme qui implémente le jeu Puissance 4.

9

Valeurs booléennes et chaînes de bits

9.1 Algèbre de Boole

L'*algèbre de Boole*, nommée en l'honneur du logicien et mathématicien George Boole, constitue l'un des fondements de l'informatique et des ordinateurs. Cette algèbre manipule des valeurs *booléennes*, c'est-à-dire les éléments *vrai* et *faux*, à l'aide d'opérateurs logiques. Tel que discuté au chapitre 5, ces opérateurs permettent notamment d'implémenter un processeur à l'aide de portes logiques.

Les valeurs booléennes peuvent être représentées par un ordinateur à l'aide d'un bit: 1 représente *vrai* et 0 représente *faux*. Un opérateur logique f est donc une fonction $f\colon \{0,1\}^k \to \{0,1\}$ où k est le nombre d'arguments de l'opérateur. La plupart des opérateurs utilisés afin de construire des circuits logiques ou des conditions dans les langages de programmation sont unaires ($k = 1$) ou binaires ($k = 2$). Nous rappelons certains de ces opérateurs à la figure 9.1.

x	$\neg x$
0	1
1	0

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

x	y	$x \to y$
0	0	1
0	1	1
1	0	0
1	1	1

x	y	$x \leftrightarrow y$
0	0	1
0	1	0
1	0	0
1	1	1

FIGURE 9.1 – Tables de vérité de la négation, de la conjonction (ET logique), de la disjonction (OU logique), du OU exclusif, de l'implication et de l'équivalence.

9.2 Représentation des valeurs booléennes

Bien qu'une valeur booléenne se représente par un seul bit, la plupart des architectures ne permettent pas d'adresser *un seul* bit. Par exemple, la plus petite unité de mémoire adressable sur l'architecture ARMv8 est l'octet. Il existe donc plusieurs façons de stocker un bit en mémoire. Par exemple, on peut:

— considérer uniquement le bit de poids faible d'un octet, et figer les sept autre bits à zéro: $00000000_2 = \textit{faux}$ et $00000001_2 = \textit{vrai}$;

— répéter la valeur booléenne dans les huit bits de l'octet: $00000000_2 = \textit{faux}$ et $11111111_2 = \textit{vrai}$;

— considérer 00000000_2 comme *faux* et les $2^8 - 1$ autres valeurs comme *vrai*.

Ces deux représentations « gaspillent » sept bits de mémoire, mais forment néanmoins les représentations utilisées dans plusieurs langages de programmation. Historiquement, la troisième approche est celle employée par C qui ne possède pas de type booléen. En C++, le code suivant retourne normalement 1, ce qui signifie qu'une valeur booléenne se représente bel et bien par un octet:

```
#include <iostream>

int main()
{
  std::cout << sizeof(bool) << std::endl;
}
```

Si l'on désire représenter *plusieurs* valeurs booléennes, on peut aussi utiliser un tableau d'octets. Par exemple, $8n$ valeurs booléennes se représentent à l'aide des bits de n octets. Cependant, puisque les bits ne sont pas adressables, ceux-ci doivent être extraits à l'aide d'instructions de manipulation de bits.

9.3 Manipulation de bits

Puisque les valeurs booléennes se représentent numériquement par 0 et 1, on pourrait envisager de les manipuler à l'aide d'opérations arithmétiques, par ex.:

$$\begin{aligned} \neg x &\equiv 1 - x \\ x \wedge y &\equiv \min(x, y) \equiv x \cdot y \\ x \vee y &\equiv \max(x, y) \\ x \oplus y &\equiv (x + y) \bmod 2 \end{aligned}$$

Toutefois, les architectures offrent normalement des instructions dédiées à cette tâche (et parfois plus efficaces).

Au-delà de la manipulation de valeurs booléennes, ces opérations sont aussi essentielles à la programmation de bas niveau liée, par exemple, aux chaînes de caractères, aux protocoles de communication, aux primitives cryptographiques et à la manipulation d'images.

9.3.1 Opérateurs logiques

Sous ARMv8, les opérations $\neg$, $\wedge$, $\vee$ et $\oplus$ peuvent être calculées à l'aide des instructions `mvn`, `and`, `orr` et `eor`. Ces opérations ne sont pas effectuées sur un seul bit, mais bien « *bit à bit* » sur *chacun* des 64 bits.

Exemple.

Si $x_{20} = 0 \cdots 0\ 1011$ et $x_{21} = 1 \cdots 1\ 1001$, alors nous obtenons:

instruction	*contenu de* x_{19} *après l'exécution*
`mvn x19, x20`	$1 \cdots 1\ 0100$
`and x19, x20, x21`	$0 \cdots 0\ 1001$
`orr x19, x20, x21`	$1 \cdots 1\ 1011$
`eor x19, x20, x21`	$1 \cdots 1\ 0010$

On obtient ces résultats en appliquant les opérateurs logiques « bit à bit »:

`mvn x19, x20`

$\neg$	$\cdots$	$\neg$	$\neg$	$\neg$	$\neg$	$\neg$
0	$\cdots$	0	1	0	1	1
1	$\cdots$	**1**	**0**	**1**	**0**	**0**

`and x19, x20, x21`

0	$\cdots$	0	1	0	1	1
$\wedge$	$\cdots$	$\wedge$	$\wedge$	$\wedge$	$\wedge$	$\wedge$
1	$\cdots$	1	1	0	0	1
0	$\cdots$	**0**	**1**	**0**	**0**	**1**

`orr x19, x20, x21`

0	$\cdots$	0	1	0	1	1
$\vee$	$\cdots$	$\vee$	$\vee$	$\vee$	$\vee$	$\vee$
1	$\cdots$	1	1	0	0	1
1	$\cdots$	**1**	**1**	**0**	**1**	**1**

`eor x19, x20, x21`

0	$\cdots$	0	1	0	1	1
$\oplus$	$\cdots$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$
1	$\cdots$	1	1	0	0	1
1	$\cdots$	**1**	**0**	**0**	**1**	**0**

Échange de registres. L'opération OU exclusif permet d'échanger le contenu de deux registres sans utiliser de registre temporaire. Le code suivant échange le contenu des registres x_{19} et x_{20}:

```
eor x19, x19, x20
eor x20, x19, x20
eor x19, x19, x20
```

Voyons pourquoi ce code fonctionne. Rappelons que $\oplus$ est commutatif, que chaque bit est son propre inverse sous $\oplus$, et que 0 est l'identité de $\oplus$. Autrement dit, pour tous $x, y \in \{0, 1\}$, nous avons:

$$x \oplus y = y \oplus x,$$
$$x \oplus x = 0,$$
$$x \oplus 0 = x.$$

Ces propriétés sont également valables pour l'opération $m \oplus n$ étendue aux chaînes de bits $m, n \in \{0,1\}^k$. Ainsi, si $\mathtt{x}_{19}$ et $\mathtt{x}_{20}$ contiennent initialement les chaînes de 64 bits m et n, alors nous obtenons:

Opération	***Contenu après l'exécution de l'opération***	
	$\mathtt{x}_{19}$	$\mathtt{x}_{20}$
—	$\boldsymbol{m}$	$\boldsymbol{n}$
$\mathtt{x}_{19} \leftarrow \mathtt{x}_{19} \oplus \mathtt{x}_{20}$	$m \oplus n$	n
$\mathtt{x}_{20} \leftarrow \mathtt{x}_{19} \oplus \mathtt{x}_{20}$	$m \oplus n$	$m \oplus n \oplus n = m \oplus 0 = m$
$\mathtt{x}_{19} \leftarrow \mathtt{x}_{19} \oplus \mathtt{x}_{20}$	$m \oplus n \oplus m = n \oplus m \oplus m = n \oplus 0 = \boldsymbol{n}$	$\boldsymbol{m}$

Remarque.

Les opérations bit à bit s'avèrent aussi utiles pour la technique cryptographique du *masque jetable* où on chiffre et déchiffre des données simplement à l'aide de l'opérateur OU exclusif.

9.3.2 Décalages logiques

Un *décalage logique* d'une chaîne de bits déplace ses bits dans une certaine direction. Puisque nous considérons des chaînes de taille fixe, certains bits sont jetés lors du décalage, et les nouveaux bits sont mis à zéro.

Exemple.

Considérons un octet x dont le contenu binaire est 10110101. Un décalage de 3 bits mène aux contenus suivants:

Octet x:	10110101
Décalage de x de 3 bits à droite:	00010110
Décalage de x de 3 bits à gauche:	10101000

La figure 9.2 illustre en général le résultat d'un décalage de j bits d'une chaîne de n bits $x = x_{n-1} \cdots x_1 x_0$

Sur l'architecture ARMv8, ces opérations peuvent être effectuées sur 64 bits à l'aide de ces instructions:

instructions	***effet***
`lsr xd, xn, j`	stocke dans $\mathtt{x}_\mathtt{d}$ le décalage de $\mathtt{x}_\mathtt{n}$ de j bits vers la droite
`lsl xd, xn, j`	stocke dans $\mathtt{x}_\mathtt{d}$ le décalage de $\mathtt{x}_\mathtt{n}$ de j bits vers la gauche

Il existe également des variantes 32 bits qui manipulent les registres $\mathtt{w}_\mathtt{d}$ et $\mathtt{w}_\mathtt{n}$.

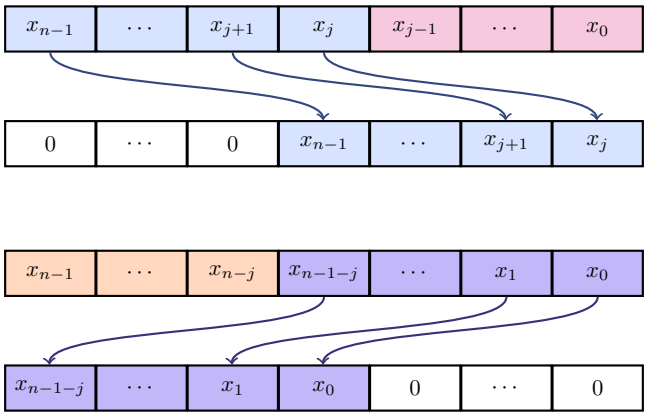


FIGURE 9.2 – Décalage de j bits à droite (*haut*) et à gauche (*bas*).

Multiplication/division. Les décalages logiques permettent d'implémenter la multiplication et la division entière non signée par une puissance de 2. En effet, la multiplication (resp. division entière) par 2^k correspond à un décalage logique de k bits vers la gauche (resp. droite). Ainsi, l'instruction « `lsl x19, x19, 3` » multiplie le registre x_{19} par 8. Cela s'avère notamment pratique pour le calcul d'index lors de la manipulation de tableaux. Plusieurs opérations arithmétiques et modes d'adressage supportent directement les décalages.

9.3.3 Décalages circulaires

Les *décalages circulaires* se comportent comme les décalages logiques à une exception près: plutôt que de jeter les bits en trop, ceux-ci sont réinsérés de l'autre côté de la chaîne.

Exemple.

Reconsidérons l'octet x dont le contenu binaire est 10110101. Un décalage circulaire de 3 bits mène aux contenus suivants:

Octet x:	10110101
Décalage de x de 3 bits à droite:	10110110
Décalage de x de 3 bits à gauche:	10101101

La figure 9.3 illustre en général le résultat d'un décalage circulaire de j bits d'une chaîne de n bits $x = x_{n-1} \cdots x_1 x_0$

L'architecture ARMv8 ne supporte que les décalages circulaires à droite:

instructions	*effet*
`ror xd, xn, j`	stocke dans x_d le décalage circulaire de x_n de j bits vers la droite

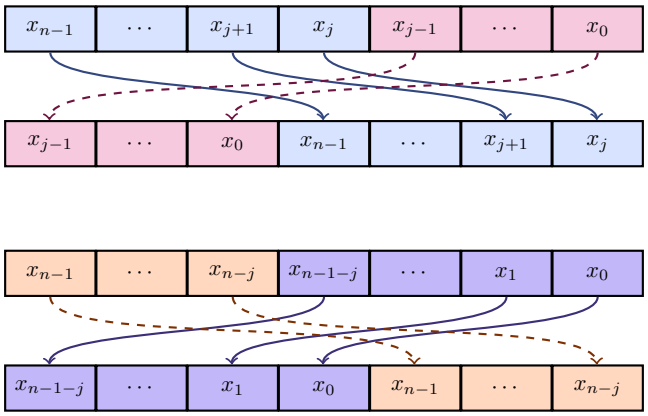


FIGURE 9.3 – Décalage circulaire de j bits à droite (*haut*) et à gauche (*bas*).

Il existe également une variante 32 bits qui manipule les registres `w`$_d$ et `w`$_n$.

9.3.4 Décalages arithmétiques

Les *décalages arithmétiques* sont une forme de décalage qui manipulent des entiers *signés*. Ils se comportent essentiellement comme les décalages logiques, mais le bit de signe est propagé lors d'un décalage vers la droite.

Exemple.

Considérons l'octet x dont le contenu binaire est 11100101. Un décalage arithmétique de 2 bits mène aux contenus suivants:

Octet x:	11100101
Décalage de x de 2 bits à droite:	11111001
Décalage de x de 2 bits à gauche:	10010100

La figure 9.4 illustre en général le résultat d'un décalage arithmétique de j bits d'une chaîne de n bits $x = x_{n-1} \cdots x_1 x_0$. Remarquons que vers la gauche il n'y a *pas* de distinction entre décalage arithmétique et décalage logique.

L'architecture ARMv8 possède une instruction pour les décalages arithmétiques à droite:

instructions	***effet***
`asr xd, xn, j`	stocke dans `x`$_d$ le décalage arithmétique de `x`$_n$ de j bits vers la droite

Il existe également une variante 32 bits qui manipule les registres `w`$_d$ et `w`$_n$.

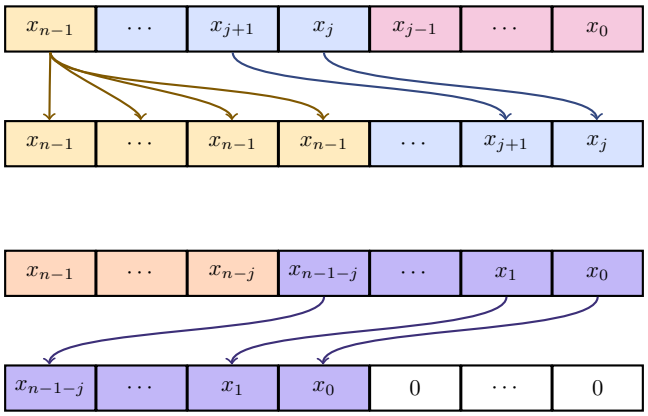


FIGURE 9.4 – Décalage arithmétique de j bits à droite (*haut*) et à gauche (*bas*).

Multiplication/division. Les décalages logiques permettent d'implémenter la multiplication et la division entière signée par une puissance de 2. En effet, la division entière par 2^k correspond à un décalage arithmétique de k bits vers la droite où les nouveaux bits demeurent égaux au bit de signe. Ainsi, l'instruction « `asr x19, x19, 2` » divise le registre x_{19} par 4.

9.4 Masquage

Les opérateurs logiques permettent également d'isoler certains bits d'une chaîne. Par exemple, l'instruction « `and x19, x19, 4` » met tous les bits de x_{19} à zéro, à l'exception du troisième bit de poids faible qui demeure inchangé. En effet, nous avons:

b_{63}	⋯	b_3	b_2	b_1	b_0
$\wedge$	⋯	$\wedge$	$\wedge$	$\wedge$	$\wedge$
0	⋯	0	1	0	0
0	⋯	**0**	$\boldsymbol{b_2}$	**0**	**0**

Dans notre exemple, le nombre 4 est appelé le *masque*. Un masque permet de spécifier les bits à isoler. Si nous changeons 4 pour le masque $9 = 1001_2$ dans l'exemple précédent, alors tous les bits sont mis à zéro à l'exception du premier et quatrième bits de poids faible:

b_{63}	⋯	b_3	b_2	b_1	b_0
$\wedge$	⋯	$\wedge$	$\wedge$	$\wedge$	$\wedge$
0	⋯	1	0	0	1
0	⋯	$\boldsymbol{b_3}$	**0**	**0**	$\boldsymbol{b_0}$

L'effet du masque varie selon les opérateurs logiques utilisés. Voici certaines variantes pratiques:

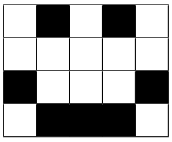
nom	***opération***	***effet***
sélection	$r \wedge m$	met à 0 les bits de r non spécifiés par le masque m
activation	$r \vee m$	met à 1 les bits de r spécifiés par le masque m
désactivation	$r \wedge \neg m$	met à 0 les bits de r spécifiés par le masque m
basculement	$r \oplus m$	inverse la valeur des bits de r spécifiés par le masque m

Sur l'architecture ARMv8, ces opérations peuvent être effectuées à l'aide des instructions `and`, `orr`, `bic` et `eor` respectivement.

9.5 ★ Cryptographie visuelle

★ Cette section résume une démonstration que je comptais faire en classe, mais que je ne pourrai pas faire étant donnée la suspension des activités pédagogiques en présentiel. Cette matière ne sera *pas* à l'examen, mais elle présente à mon avis une application intéressante.

La technique cryptographique du masque jetable décrite en classe s'adapte au chiffrement d'images. Par exemple, considérons cette image A de 4×5 pixels, chacun noir ou blanc:



Chiffrement. On convertit A vers deux nouvelles images B et C en appliquant ce processus itérativement pour chaque pixel (i, j):

— si le pixel $A[i, j]$ est blanc, alors on choisit une couleur aléatoire, puis on assigne cette couleur à $B[i, j]$ et $C[i, j]$;

— si le pixel $A[i, j]$ est noir, alors on choisit une couleur aléatoire, puis on assigne cette couleur à $B[i, j]$ et la couleur inverse à $C[i, j]$.

Par exemple, ce processus pourrait mener à ces deux images:

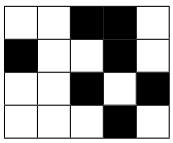


image B

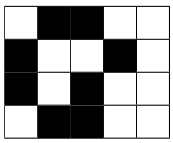


image C

Si l'on considère noir comme le bit 1 et blanc comme le bit 0, alors on obtient $A[i, j] = B[i, j] \oplus C[i, j]$ pour chaque pixel (i, j). En effet, cela découle du fait que $0 = 0 \oplus 0 = 1 \oplus 1$ et $1 = 0 \oplus 1 = 1 \oplus 0$. De plus, les images B et C sont *individuellement* indistinguables d'une image aléatoire.

Déchiffrement. Si deux personnes (ou deux machines) possèdent B et C respectivement, alors elles détiennent chacune une clé secrète permettant de reconstruire A *conjointement*. En effet, il suffit de calculer $B[i,j] \oplus C[i,j]$ pour chaque pixel (i,j) afin de retrouver les pixels de départ. Autrement dit, on « superpose » les images B et C, puis on applique un OU exclusif à chaque position.

9.5.1 Format PBM

Cette procédure s'implémente assez aisément pour le format d'image PBM. Sous ce format, une image est codée par une *en-tête* et un *corps*. L'en-tête débute par la chaîne de caractères « `P4` » qui spécifie qu'il s'agit d'une image PBM, suivi du nombre de pixels apparaissant sur une ligne et sur une colonne de l'image. Par exemple, l'en-tête de l'image A précédente est:

```
P4
5 4
```

Le reste du fichier décrit les pixels. Plus précisément, chaque ligne du fichier dicte les pixels d'une ligne de l'image. Dans notre exemple, nous voulons donc représenter:

```
01010
00000
10001
01110
```

Le format PBM stocke chaque bloc de 8 pixels d'une ligne dans un octet, et, au besoin, ajoute des zéros afin de compléter le dernier octet d'une ligne. Dans notre cas, il faut donc ajouter 3 bits à zéro sur chaque ligne, ce qui mène aux octets suivants:

```
0x50
0x00
0x88
0x70
```

9.5.2 Implémentation

Ce programme implémente la procédure de déchiffrement. Autrement dit, il lit consécutivement le contenu de deux images B et C au format PBM, puis affiche le contenu de l'image A obtenue en calculant $A[i,j] = B[i,j] \oplus C[i,j]$ pour chaque pixel (i,j). Afin de manipuler une image, on lit son en-tête:

```
P4
n m
```

Cela permet d'identifier le nombre d'octets à consommer: $m \cdot ((n+7) \div 8)$. Ici, la division par 8 correspond au fait que chaque octet contient 8 pixels, et

l'ajout de 7 permet de tenir compte des bits à zéro qui remplissent possiblement le dernier octet au bout d'une ligne:

— si n est de la forme $n = 8k$ (donc un multiple de huit), alors $(n+7) \div 8 = (8k + 7) \div 8 = (8k \div 8) + (7 \div 8) = k + 0 = k$;

— sinon n est de la forme $n = 8k + r$ où $0 < r < 8$, et ainsi $(n + 7) \div 8 = (8k + r + 7) \div 8 = (8k \div 8) + (r + 7) \div 8 = k + 1$.

9.6 Exercices

9.1) Considérons les chaînes de six bits $a = 101101$ et $b = 101010$. Calculez $\neg a$, $a \wedge b$, $a \vee b$ et $a \oplus b$ bit à bit. Répétez l'exercice avec d'autres chaînes.

9.2) Expliquez comment effectuer un décalage circulaire vers la gauche à l'aide d'un décalage circulaire vers la droite.

9.3) À partir d'un registre $\texttt{x}_\texttt{d}$, comment peut-on obtenir une valeur booléenne qui indique si $\texttt{x}_\texttt{d}$ contient un nombre pair?

9.4) Supposons que $8n$ bits soient représentés par un tableau de n octets. Expliquez comment extraire le $i^{\text{ème}}$ bit.

9.5) ★★ Montrez que le décalage arithmétique à droite implémente bien la division par une puissance de 2.

9.6) Si le registre $\texttt{x}_{19}$ contient l'entier signé -4884_{10}, alors quelle est sa valeur après l'exécution de l'instruction « `asr x19, x19, 2` »?

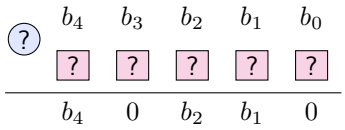
(*tiré de l'examen final de l'hiver 2019*)

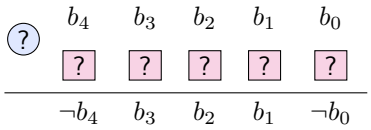
9.7) Deux des programmes ci-dessous terminent avec une *même* valeur n dans $\texttt{x}_{19}$, alors que l'autre programme termine avec une valeur *différente* de n dans $\texttt{x}_{19}$. Quelle est la valeur décimale de n?

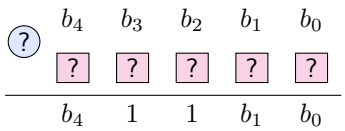
Programme A	Programme B	Programme C
`mov x19, 5`	`mov x19, 5`	`mov x19, 5`
`ror x19, x19, 1`	`eor x19, x19, 7`	`lsl x19, x19, 1`
`orr x19, x19, 1`	`lsr x19, x19, 1`	`and x19, x19, 3`
`and x19, x19, 7`	`orr x19, x19, 4`	`orr x19, x19, 1`

(*tiré de l'examen final de l'hiver 2019*)

9.8) Chaque schéma ci-dessous représente une opération de masquage. Trouvez des opérateurs et masques qui mènent aux résultats. Plus précisément, remplacez chaque occurrence de (?) par un opérateur logique, et chaque occurrence de [?] par 0 ou 1.







(*tiré de l'examen final de l'hiver 2019*)

10

Chaînes de caractères

La lecture et l'affichage de symboles lisibles par un humain requiert la manipulation de *caractères*: lettres, chiffres, signes de ponctuation, émojis, etc. Dans ce chapitre, nous expliquons comment les représenter et manipuler.

De façon générale, jusqu'à 2^n caractères peuvent être représentés à l'aide de n bits. Par exemple, nous pourrions représenter les lettres de l'alphabet français à l'aide de 6 bits en choisissant (arbitrairement) que 000000 = a, 000001 = b, 000010 = c, ..., 011001 = z, 011010 = à, etc. Un tel système se nomme un *codage de caractères*. Il existe une myriade de codages de caractères, dont UTF-8 et ISO 8859-1, tous deux basés sur un ancêtre commun: ASCII.

10.1 ASCII

Le codage *ASCII (American Standard Code for Information Interchange)* utilise 7 bits afin de représenter 128 caractères: les lettres de l'alphabet latin (en minuscules et majuscules), les chiffres (indo-)arabes, certains symboles mathématiques et de ponctuation, ainsi que des caractères spéciaux. Les caractères graphiques du codage ASCII sont répertoriés à la figure 10.1. Par exemple, la lettre « m » est représentée par le code $109 = 6\text{D}_{16} = 1101101_2$, et le chiffre « 6 » est représentée par $54 = 36_{16} = 0110110_2$.

Les autres caractères qui n'apparaissent pas à la figure 10.1 sont des caractères spéciaux non graphiques. Par exemple, le code 9 représente une tabulation; le code 10 représente un saut de ligne (`\n`); le code 13 représente un retour de chariot (`\r`); et le code 32 représente une espace.

Notons que le code d'une lettre minuscule se situe à 32 positions de la même lettre en majuscule. En fait, le codage d'une lettre majuscule et minuscule ne diffère qu'au $6^{\text{ème}}$ bit de poids faible. Par exemple, $a = 1100001_2$ et $A = 1000001_2$.

code		*carac.*	*code*		*carac.*	*code*		*carac.*
déc.	hex.		déc.	hex.		déc.	hex.	
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	'	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	,	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	120	78	x
57	39	9	89	59	Y	121	79	y
58	3A	:	90	5A	Z	122	7A	z
59	3B	;	91	5B	[	123	7B	{
60	3C	<	92	5C	\	124	7C	|
61	3D	=	93	5D	]	125	7D	}
62	3E	>	94	5E	^	126	7E	~
63	3F	?	95	5F	_			
64	40	@	96	60	`			

FIGURE 10.1 – Caractères graphiques du codage ASCII. Chaque caractère est représenté par un code numérique dans sa forme décimale et hexadécimale.

10.2 ISO 8859-1 (Latin-1)

Bien que l'ASCII permette de représenter de l'anglais, ce n'est pas le cas pour le français en raison de ses lettres accentuées. Puisque la plus petite unité de mémoire de la plupart des architectures est l'octet, il est possible d'utiliser le $8^{\text{ème}}$ bit inutilisé par l'ASCII afin de représenter 128 caractères régionaux supplémen-

taires. Historiquement, une foule de codages ont été créés afin d'étendre l'ASCII. En particulier, le codage *ISO 8859-1 (Latin-1)* étend l'ASCII avec suffisamment de caractères pour la représentation du français et de plusieurs langues indo-européennes dont le système d'écriture se base sur le latin. Les caractères graphiques additionnels du codage ISO 8859-1 sont répertoriés à la figure 10.2.

code		***carac.***
déc.	hex.	
161	A1	¡
162	A2	¢
163	A3	£
164	A4	¤
165	A5	¥
166	A6	¦
167	A7	§
168	A8	¨
169	A9	©
170	AA	ª
171	AB	«
172	AC	¬
173	AD	
174	AE	®
175	AF	¯
176	B0	°
177	B1	±
178	B2	²
179	B3	³
180	B4	´
181	B5	µ
182	B6	¶
183	B7	·
184	B8	¸
185	B9	¹
186	BA	º
187	BB	»
188	BC	¼
189	BD	½
190	BE	¾
191	BF	¿

code		***carac.***
déc.	hex.	
192	C0	À
193	C1	Á
194	C2	Â
195	C3	Ã
196	C4	Ä
197	C5	Å
198	C6	Æ
199	C7	Ç
200	C8	È
201	C9	É
202	CA	Ê
203	CB	Ë
204	CC	Ì
205	CD	Í
206	CE	Î
207	CF	Ï
208	D0	Ð
209	D1	Ñ
210	D2	Ò
211	D3	Ó
212	D4	Ô
213	D5	Õ
214	D6	Ö
215	D7	×
216	D8	Ø
217	D9	Ù
218	DA	Ú
219	DB	Û
220	DC	Ü
221	DD	Ý
222	DE	Þ
223	DF	ß

code		***carac.***
déc.	hex.	
224	E0	à
225	E1	á
226	E2	â
227	E3	ã
228	E4	ä
229	E5	å
230	E6	æ
231	E7	ç
232	E8	è
233	E9	é
234	EA	ê
235	EB	ë
236	EC	ì
237	ED	í
238	EE	î
239	EF	ï
240	F0	ð
241	F1	ñ
242	F2	ò
243	F3	ó
244	F4	ô
245	F5	õ
246	F6	ö
247	F7	÷
248	F8	ø
249	F9	ù
250	FA	ú
251	FB	û
252	FC	ü
253	FD	ý
254	FE	þ
255	FF	ÿ

FIGURE 10.2 – Caractères graphiques du codage ISO 8859-1. Chaque caractère est représenté par un code numérique dans sa forme décimale et hexadécimale.

10.3 UTF-8

Bien que le format ISO 8859-1 soit suffisant pour plusieurs langues européennes, il ne permet pas de représenter d'autres langues comme le grec, l'arabe, le japonais et les langues chinoises. Afin de palier ce problème, le codage *UTF-8* utilise jusqu'à 21 bits afin de représenter plus d'un million de caractères spécifiés par le standard *Unicode*. Contrairement aux codages ASCII et ISO 8859-1, le codage UTF-8 code les caractères avec un nombre *variable* d'octets: 1, 2, 3 ou 4 selon le caractère.

Pour des raisons de rétrocompatibilité et d'économie de mémoire, les 128 premiers caractères de l'UTF-8 sont précisément ceux de l'ASCII codés sur un seul octet. Les 128 caractères suivants sont ceux de l'ISO 8859-1 codés sur deux octets. Par exemple, le caractère « é » codé par $\mathrm{E9}_{16} = 11101001_2$ sous le codage ISO 8859-1, est codé par $\mathrm{C3A9}_{16} = 11000011\ 10101001_2$ sous le codage UTF-8. Le format général de l'UTF-8 se décrit ainsi [Yer03]:

# bits	*plage de codes* [1]		*format binaire des octets*			
	début	fin	octet 1	octet 2	octet 3	octet 4
7	$\mathtt{000000}_{16}$	$\mathtt{00007F}_{16}$	`0*******`	—	—	—
11	$\mathtt{000080}_{16}$	$\mathtt{0007FF}_{16}$	`110*****`	`10******`	—	—
16	$\mathtt{000800}_{16}$	$\mathtt{00FFFF}_{16}$	`1110****`	`10******`	`10******`	—
21	$\mathtt{010000}_{16}$	$\mathtt{10FFFF}_{16}$	`11110***`	`10******`	`10******`	`10******`

Exemple.

Les caractères a, é, ケ et 𒑍 possèdent les codes 61_{16}, $\mathrm{E9}_{16}$, $\mathrm{30B1}_{16}$ et $\mathrm{1240D}_{16}$ respectivement. Leur codage UTF-8 s'obtient donc ainsi:

car.	code	codage
a	$61_{16} = 1100001_2$	01100001_2
é	$\mathrm{E9}_{16} = 000\ 11101001_2$	$11000011\ 10101001_2$
ケ	$\mathrm{30B1}_{16} = 00110000\ 10110001_2$	$11100011\ 10000010\ 10110001_2$
𒑍	$\mathrm{1240D}_{16} = 00001\ 00100100\ 00001101_2$	$11110000\ 10010010\ 10010000\ 10001101_2$

Remarque.

Il existe des codages alternatifs pour le standard Unicode, par ex. *UTF-16* et *UTF-32*. L'UTF-32 code *tous* les caractères sur 4 octets, ce qui permet notamment d'accéder rapidement au $i^{\text{ème}}$ caractère d'une suite de caractères. L'UTF-16 propose un codage à taille variable sur 2 et 4 octets, ce qui offre un compromis entre UTF-8 et UTF-32.

1. Pour des raisons techniques, les codes $\mathrm{D800}_{16}$ à DFFF_{16} sont considérés invalides et ne représentent donc aucun caractère.

10.4 Chaînes de caractères

Une *chaîne de caractères* est une suite finie de caractères. Dans les codages présentés plus tôt, on indique normalement la fin d'une chaîne par le *caractère nul* spécifié par le code 00_{16}. Ainsi, la chaîne `"Allo"` est représentée par la suite d'octets `41 6C 6C 6F 00` (ici sous notation hexadécimale).

Le parcours d'une chaîne de n caractères ASCII ou ISO 8859-1 s'effectue similairement à celle d'un tableau, puis qu'une telle chaîne correspond précisément à un tableau de $n+1$ octets. Cependant, dans le cas d'UTF-8, la position du $i^{\text{ème}}$ caractère d'une chaîne ne se calcule pas directement, puisque chacun des caractères peut prendre 1 à 4 octets en mémoire. Il faut donc, pour chaque caractère, vérifier s'il débute par `0`, `110`, `1110` ou `11110` afin de déterminer s'il faut consommer 1, 2, 3 ou 4 octets. Cette vérification peut s'implémenter à l'aide de décalages de bits ou d'opérations de masquage.

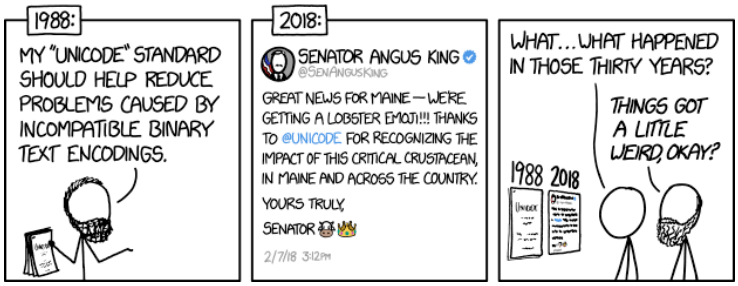
Sur l'architecture ARMv8, une chaîne de caractères (statique) se déclare à l'aide de « `.asciz` », alors qu'une suite de caractères, sans caractère nul de fin, se déclare à l'aide de « `.ascii` ».

Remarque.

Dans notre cas, une chaîne de caractères lue dans un terminal avec `scanf` est codée sous UTF-8. Si seules des lettres du codage ASCII sont entrées, alors on obtient une chaîne sous codage ASCII.

Remarque.

On peut lire/afficher une chaîne avec `scanf/printf` à l'aide du format `"%s"`. Si l'on désire lire une chaîne avec des espaces, alors on peut utiliser le format `"%[^\n]"`.



— xkcd

10.5 Exercices

10.1) Donnez le codage UTF-8 des caractères associés à ces codes numériques:

$$0006\text{AB}_{16},\ 000073_{16},\ 012345_{16} \text{ et } 00\text{A}0\text{B}1_{16}.$$

10.2) Écrivez un sous-programme qui reçoit une chaîne de caractères sous codage ASCII et qui retourne sa taille (excluant son caractère nul).

10.3) Écrivez un sous-programme qui reçoit une chaîne de caractères sous codage ASCII, ainsi que sa taille, et qui retourne une valeur booléenne indiquant si la chaîne est un palindrome.

10.4) Écrivez un sous-programme qui reçoit une chaîne de caractères et qui retourne une valeur booléenne indiquant si son premier caractère fait partie du codage ASCII.

10.5) Supposons que $\texttt{x}_{19}$ contienne le code ASCII d'une lettre. Donnez *une seule* instruction qui inverse la casse de $\texttt{x}_{19}$; autrement dit, une lettre minuscule doit être mise en majuscule et vice-versa.

(*tiré de l'examen final de l'hiver 2019*)

10.6) Considérons le codage de caractères fictif *IZO-IFT209*. Ce codage permet de représenter les 2048 premiers caractères du standard Unicode. En particulier, les 256 premiers caractères codés par l'IZO-IFT209 sont respectivement ceux de l'ASCII et de l'ISO 8859-1 (Latin-1). Les caractères sont codés sur un à deux octets selon le format suivant:

plage de codes numériques		***codage IZO-IFT209***	
début	fin	octet 1	octet 2
$\texttt{000}_{16}$	$\texttt{07F}_{16}$	`1*******`	—
$\texttt{080}_{16}$	$\texttt{7FF}_{16}$	`01******`	`001*****`

Rappelons que chaque caractère possède un *code numérique* et un *codage*. Par exemple, le code numérique du caractère « a » est 61_{16}, et son codage est donc « 01100001 »; le code numérique du caractère « é » est $\texttt{E9}_{16}$, et son codage est donc « 01000111 00101001 ».

— Donnez le codage IZO-IFT209 du caractère « ü » dont le code numérique est $\texttt{FC}_{16}$.

— Combien de caractères sont contenus dans la chaîne de caractères IZO-IFT209 ci-dessous?

```
10100011 01111101 00101011 01111111 00101111 10001001 10101101 11000000
```

— Écrivez un *sous-programme* qui accomplit la tâche suivante:

ENTRÉE: codage IZO-IFT209 d'un caractère c
SORTIE: code numérique de c

Puisqu'un caractère IZO-IFT209 est codé sur au plus 16 bits et que les registres contiennent 64 bits, nous supposons que les bits excédentaires de poids fort sont égaux à 0. Voici des exemples d'entrées et de sorties du sous-programme:

Entrée (64 bits)	**Valeur de retour (64 bits)**
$00\cdots0\ 00000000\ 11100001_2$	$00\cdots0\ 00000000\ 01100001_2$
$00\cdots0\ 01000111\ 00101001_2$	$00\cdots0\ 00000000\ 11101001_2$

Rappel: il est possible de spécifier une valeur hexadécimale avec le préfixe « 0x », par ex.: « `mov x19, 0x3FA` ».

(*tiré de l'examen final de l'hiver 2019*)

10.7) ★ Supposons que $\mathtt{x}_{19}$ contienne l'adresse d'une chaîne de caractères s sous codage UTF-32, et que $\mathtt{x}_{20}$ contienne un indice i. Donnez une seule instruction qui charge le $i^{\text{ème}}$ caractère de s dans $\mathtt{x}_{21}$.

11

Sous-programmes et mémoire

Tel que discuté au chapitre 8, les programmes sont normalement modularisés en sous-routines. En langage d'assemblage, une sous-routine est implémentée sous forme de sous-programme: un segment de code appelé avec un branchement. Dans ce chapitre, nous revisitons leur fonctionnement plus en détails.

11.1 Pile d'exécution

11.1.1 Appels de sous-programmes

Sur l'architecture ARMv8, les arguments d'un sous-programme sont passés dans les registres x_0 à x_7. Cependant, certaines architectures ne possèdent pas de registres dédiés à cette tâche. De plus, un sous-programme pourrait techniquement posséder plus de huit paramètres.

Rappelons également que certains registres, comme les registres x_{19} à x_{29} sous ARMv8, doivent être préservés lors de l'appel d'un sous-programme. De plus, l'adresse de retour doit être préservée d'une certaine façon. Sur ARMv8, le registre x_{30} joue ce rôle, mais ne peut stocker qu'*une seule* adresse à la fois.

Ainsi, une séquence d'appels de sous-programmes peut engendrer une quantité arbitraire de données qui doivent être stockées temporairement en dehors des registres. Ces données sont stockées dans un segment de la mémoire principale nommé la *pile d'exécution*. Ce segment porte ce nom puisqu'il est utilisé comme une pile: des données y sont empilées et dépilées.

11.1.2 Disposition de la mémoire

Afin de comprendre le fonctionnement de la pile d'exécution, voyons d'abord brièvement l'organisation de la mémoire d'un programme. Lors du chargement d'un programme, ses instructions et ses données statiques sont stockées dans un segment fixe de la mémoire principale. Un segment de données est également alloué pour la manipulation de données dynamiques, c'est-à-dire des données

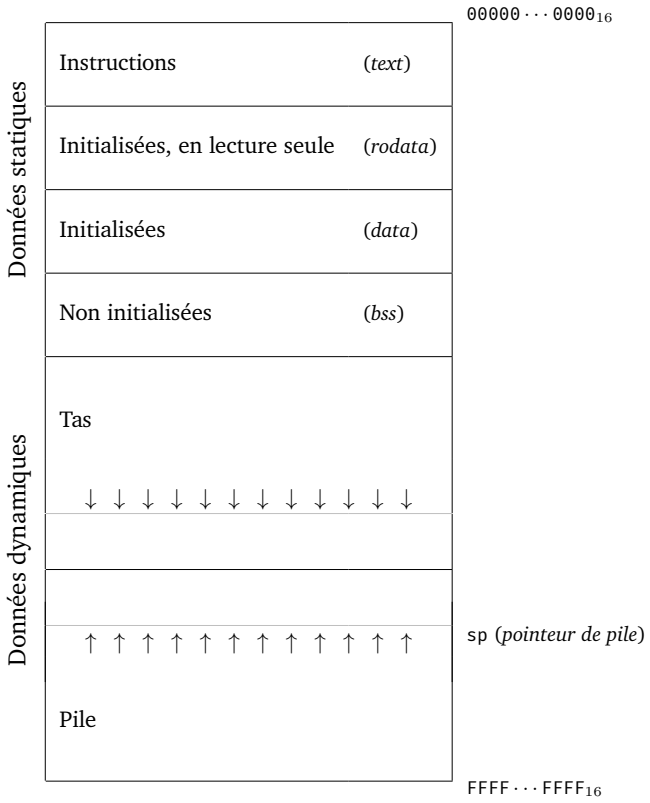


FIGURE 11.1 – Disposition de la mémoire d'un programme.

dont la taille demeure inconnue à l'assemblage. Ce segment se divise en deux sous-segments:

— le *tas*: qui contient les données allouées dynamiquement afin de stocker des structures de données, des objets, etc.;

— la *pile d'exécution*: qui contient les données temporaires des appels de sous-programmes.

Par convention, lors de l'ajout de données, les adresses du tas croissent, alors que celles de la pile décroissent. Cette organisation de la mémoire est illustrée à la figure 11.1.

Remarque.

Les variables locales d'une fonction en C/C++ sont généralement stockées sur la pile d'exécution.

11.1.3 Fonctionnement de la pile

À tout moment, le *pointeur de pile* `sp` contient l'adresse de l'élément situé au sommet de la pile. Initialement, la pile est vide et ainsi `sp` pointe vers la dernière cellule mémoire de la pile, par ex. l'adresse $\texttt{FFFF}\cdots\texttt{FFFF}_{16}$.

Par convention, le pointeur de pile décroît vers $\texttt{0000}\cdots\texttt{0000}_{16}$ lorsque des données y sont ajoutées. Ainsi, afin d'empiler k octets sur la pile d'exécution, `sp` est décrémenté de k adresses, puis les octets sont stockés à l'adresse `sp`. Similairement, afin de dépiler k octets, ceux-ci sont lus à l'adresse `sp`, puis `sp` est incrémenté de k adresses.

11.1.4 Sauvegarde et restauration

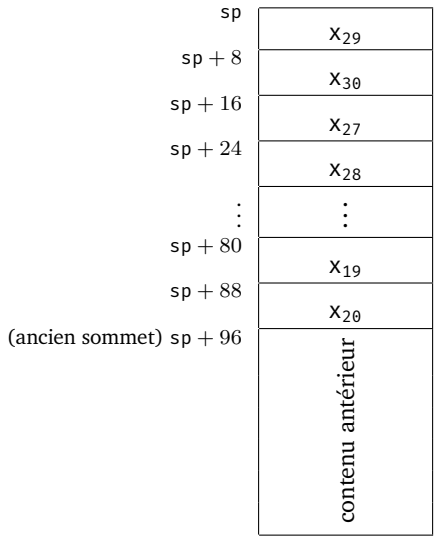
Nous sommes maintenant en mesure d'expliquer le fonctionnement des macros de sauvegarde et de restauration de registres ARMv8 présentées au chapitre 8.

Sauvegarde. Considérons d'abord la macro de sauvegarde:

```
.macro SAVE
  stp   x29, x30, [sp, -96]!
  mov   x29, sp
  stp   x27, x28, [sp, 16]
  stp   x25, x26, [sp, 32]
  stp   x23, x24, [sp, 48]
  stp   x21, x22, [sp, 64]
  stp   x19, x20, [sp, 80]
.endm
```

L'instruction « `stp xd, xn, a` » sauvegarde le contenu de $\texttt{x}_\texttt{d}$ et $\texttt{x}_\texttt{n}$ consécutivement à l'adresse `a`. Il s'agit donc essentiellement de deux appels consécutifs à `str`. Toutefois, l'architecture ARMv8 requiert que `sp` soit un multiple de 16 en tout temps. Ainsi, l'usage de `stp` garantit la satisfaction de cette contrainte.

Décortiquons la première ligne de `SAVE`: « `stp x29, x30, [sp, -96]!` ». Cette instruction décrémente d'abord la valeur de `sp` de 96. Cela correspond à l'allocation de 12 double mots afin de stocker le contenu des 12 registres: $\texttt{x}_{19}$ à $\texttt{x}_{30}$. L'instruction stocke ensuite le contenu de $\texttt{x}_{29}$ et $\texttt{x}_{30}$ à l'adresse `sp`, donc au-dessus de la pile. Le contenu des autres registres est stocké dans les double mots suivants. Ainsi, après l'exécution de `SAVE`, la pile est organisée ainsi:



Notons que l'instruction « `mov x29, sp` » a pour but de stocker l'ancien sommet de la pile. Cela est requis par la convention d'appel d'ARMv8 [ARM13, Sec. 5.2.3]. La portion de la mémoire située entre `x29` et `sp` se nomme le *bloc d'activation* de l'appel. En particulier, l'adresse stockée dans `x29` permet de rétablir la pile.

Restauration. La macro de restauration est symétrique à la macro de sauvegarde:

```
.macro RESTORE
  ldp   x27, x28, [sp, 16]
  ldp   x25, x26, [sp, 32]
  ldp   x23, x24, [sp, 48]
  ldp   x21, x22, [sp, 64]
  ldp   x19, x20, [sp, 80]
  ldp   x29, x30, [sp], 96
.endm
```

L'instruction « `ldp xd, xn, a` » charge le contenu des adresses `a` et $a + 8$ dans x_d et x_n respectivement. Ainsi, les registres sont restaurés et la dernière instruction incrémente `sp` de 96 adresses afin de libérer les 96 octets au sommet de la pile.

11.2 Récursion

La pile d'exécution peut être utilisée afin d'implémenter des sous-programmes récursifs: c'est-à-dire des sous-programmes qui s'appellent eux-mêmes. À titre

d'exemple, considérons la *suite de Fibonacci* $F_0, F_1, F_2, \ldots$ définie par:

$$F_n = \begin{cases} 0 & \text{si } n = 0, \\ 1 & \text{si } n = 1, \\ F_{n-1} + F_{n-2} & \text{si } n \geq 2. \end{cases}$$

Les premiers termes de cette suite sont: $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$.

Le sous-programme suivant calcule la valeur de F_n à partir de l'argument n:

```
fib:                                    // fib(n)
        SAVE                            // {
        mov     x19, x0                 //
        cmp     x19, 2                  //   if (n >= 2)
        b.lo    fin                     //   {
                                        //
        sub     x0, x19, 1              //
        bl      fib                     //
        mov     x20, x0                 //     r = fib(n - 1)
                                        //
        sub     x0, x19, 2              //
        bl      fib                     //
        add     x0, x20, x0             //     n = r + fib(n - 2)
fin:                                    //   }
        RESTORE                         //
        ret                             //   return n
                                        // }
```

Le code ci-dessus utilise les macros `SAVE` et `RESTORE`. Ainsi, chaque appel ajoute 96 octets à la pile d'exécution. Plusieurs de ces octets sont inutiles puisque la plupart des registres sont inutilisés.

Le code suivant utilise trois fois moins de mémoire en ajoutant 32 octets plutôt que 96 octets:

```
fib:                                    // fib(n)
        // Sauvegarder environnement    // {
        stp     x29, x30, [sp, -32]!    //
        mov     x29, sp                 //
        stp     x19, x20, [sp, 16]      //
                                        //
        // Calculer fib(n)              //
        mov     x19, x0                 //
        cmp     x19, 2                  //   if (n >= 2)
        b.lo    fin                     //   {
                                        //
        sub     x0, x19, 1              //
        bl      fib                     //
        mov     x20, x0                 //     r = fib(n - 1)
                                        //
```

```
        sub     x0, x19, 2          //
        bl      fib                 //
        add     x0, x20, x0         //     n = r + fib(n - 2)
fin:                                //   }
        // Restaurer environnement  //
        ldp     x29, x30, [sp], 16  //
        ldp     x19, x20, [sp], 16  //
        ret                         //   return n
                                    // }
```

11.3 Limitations

La taille de la pile d'exécution est bornée par la taille de la mémoire principale, et elle est souvent restreinte à une taille plus petite par le système d'exploitation. Ainsi, une récursion trop profonde peut mener à une *erreur de segmentation*, et plus précisément à un *débordement de pile*. En effet, si la pile se remplit lors d'un appel, l'appel suivant tentera d'écrire en mémoire en dehors de la pile.

Remarque.

Le célèbre site Web de questions et réponses *Stack Overflow* tire son nom du terme anglais signifiant « débordement de pile ».

11.4 Exercices

11.1) Écrivez un sous-programme qui reçoit un entier $n \geq 0$ en entrée et qui calcule *récursivement* la somme $1+2+\ldots+n$, où par convention la somme vaut 0 si $n = 0$.

11.2) Si sp contient $\texttt{AF09}_{16}$ et que l'on empile le contenu de quatre registres sur la pile d'exécution, que devient sp? Puis, que devient sp si l'on en dépile deux?

11.3) L'instruction « `stp` » empile le contenu de deux registres, ce qui nous force à empiler un nombre pair de double mots. Comment peut-on gérer l'empilement d'un nombre *impair* de double mots?

11.4) Implémentez l'algorithme récursif suivant sous forme de *sous-programme*:

Entrée : entier non signé n de 64 bits
Retour : entier non signé de 64 bits valant 2^n

`exp(`n`):`
- **si** $n = 0$ **alors**
  - **retourner** 1
- **sinon**
  - $r \leftarrow$ `exp(`$n \div 2$`)` $\cdot$ `exp(`$n \div 2$`)`
  - **si** n *est pair* **alors**
    - **retourner** r
  - **sinon**
    - **retourner** $2 \cdot r$

Vous n'avez *pas* à gérer les débordements. Vous *pouvez* utiliser les macros `SAVE` et `RESTORE`. (*tiré de l'examen final de l'hiver 2019*)

11.5) Supposons que vous n'ayez *pas* accès aux macros `SAVE` et `RESTORE`. Considérons un sous-programme qui effectue des appels récursifs et qui utilise uniquement les registres $\texttt{x}_{19}$, $\texttt{x}_{20}$, $\texttt{x}_{23}$ et $\texttt{x}_{25}$. Complétez le code des macros suivantes afin d'implémenter la sauvegarde et la restauration des registres spécifiquement pour ce sous-programme:

```
.macro SAUVEGARDER
  // Code ici
  mov   x29, sp
  /*
     Code ici
              */
.endm
```

```
.macro RESTAURER
  /*
     Code ici
              */
.endm
```

(*tiré de l'examen final de l'hiver 2019*)

11.6) Le programme ci-dessous affiche les nombres de 1 à 10, puis boucle à l'infini plutôt que de terminer. Expliquez pourquoi le programme ne termine pas. Plus précisément, identifiez les étiquettes qui sont atteintes infiniment souvent et la raison pour laquelle elles le sont.

```
.global main

main:
main0:  bl      foo
main1:  bl      exit

foo:
foo0:   mov     x19, 0
foo1:   add     x19, x19, 1
foo2:   adr     x0, fmt
foo3:   mov     x1, x19
foo4:   bl      printf
foo5:   cmp     x19, 10
foo6:   b.lo    foo1
foo7:   ret

.section ".rodata"
fmt:    .asciz  "%lu\n"
```

(*tiré de l'examen final de l'hiver 2019*)

12

Nombres en virgule flottante

Nous avons vu jusqu'ici comment manipuler les entiers sur un ordinateur. Bien que ce type élémentaire numérique soit suffisant pour une foule d'applications, il ne l'est pas nécessairement pour d'autres comme la simulation, le calcul scientifique, le traitement de signal, l'apprentissage automatique et les méthodes numériques de la recherche opérationnelle. Dans ce chapitre, nous considérons donc la représentation et la manipulation de *nombres réels*.

Soient $a, b \in \mathbb{R}$ des nombres réels tels que $a < b$. L'intervalle $[a, b]$ contient une *infinité* de nombres. Ainsi, il est impossible de représenter *tous* les nombres de l'intervalle $[a, b]$ à l'aide d'un ordinateur, comme nous l'avons fait pour $\mathbb{N}$ et $\mathbb{Z}$. Nous étudions donc la représentation en nombres en virgule flottante qui permet d'approximer raisonnablement les nombres réels.

12.1 Représentation

Un *nombre en virgule flottante* est un nombre de la forme

$$\overbrace{\pm}^{\textit{signe}} \underbrace{d_0,d_1d_2\cdots d_{n-1}}_{\textit{mantisse}} \times \underbrace{\overbrace{\beta^{\,e}}^{\textit{exposant}}}_{\textit{base}}$$

où chaque composante est un entier non négatif, et où la mantisse est un nombre fractionnaire en base $\beta \geq 2$.

Exemple.

Ces nombres en virgule flottante:

$$-0,5142 \times 10^3 \qquad 1,0567 \times 10^{-2} \qquad 1,011 \times 2^2$$

possèdent respectivement ces valeurs décimales:

$$-514{,}2 \qquad 0{,}010567 \qquad 5{,}5.$$

Il existe généralement plusieurs représentations d'une même valeur. Par exemple, $0{,}123 \times 10^2$ et $1{,}23 \times 10^1$ représentent tous deux la valeur $12{,}3$. Nous disons qu'un nombre en virgule flottante est *normalisé* si $d_0 \neq 0$; autrement dit, si le premier chiffre de sa mantisse est non nul. Ainsi, $0{,}123 \times 10^2$ n'est *pas* normalisé, alors que $1{,}23 \times 10^1$ est normalisé. La forme normalisée d'un nombre non nul offre une *représentation unique* par rapport à une base β. Notons cependant que 0 ne peut *pas* être normalisé.

Si $e_{\text{min}} \leq e \leq e_{\text{max}}$, alors la quantité de nombres normalisés représentables avec une certaine base β et une mantisse de taille n est:

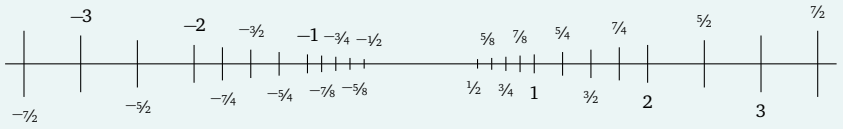
$$\underbrace{2}_{\pm} \cdot \underbrace{(\beta - 1)}_{d_0} \cdot \underbrace{\beta^{n-1}}_{d_1,\ldots,d_{n-1}} \cdot \underbrace{(e_{\text{max}} - e_{\text{min}} + 1)}_{e}.$$

La plus petite valeur absolue x_{min} et la plus grande valeur absolue x_{max} représentables par un nombre normalisé sont:

$$\begin{aligned} x_{\text{min}} &= 1{,}0\cdots0 \times \beta^{e_{\text{min}}} \\ &= \beta^{e_{\text{min}}}, \end{aligned} \qquad \begin{aligned} x_{\text{max}} &= (\beta-1),(\beta-1)\cdots(\beta-1) \times \beta^{e_{\text{max}}} \\ &= (\beta-1)(\beta-1)\cdots(\beta-1),0 \times \beta^{e_{\text{max}}-(n-1)} \\ &= (\beta^n - 1) \cdot \beta^{e_{\text{max}}-n+1} \\ &= \beta^{e_{\text{max}}+1} - \beta^{e_{\text{max}}+1-n}. \end{aligned}$$

Exemple.

Pour $\beta = 2$, $n = 3$ et $-1 \leq e \leq 1$, il y a 24 nombres normalisés représentables:



Remarquons que plus les nombres s'approchent de 0, plus la distance entre ceux-ci est petite. En effet, les nombres positifs de l'intervalle de l'exemple ci-dessus se réécrivent ainsi:

$$\underbrace{4/8, 5/8, 6/8, 7/8,}_{\text{bonds de } 1/8} \underbrace{4/4, 5/4, 6/4, 7/4,}_{\text{bonds de } 1/4} \underbrace{4/2, 5/2, 6/2, 7/2}_{\text{bonds de } 1/2}.$$

Les nombres en virgule flottante permettent donc de représenter des valeurs de *différents ordres de grandeur*.

12.2 Précision

Puisque la vaste majorité [1] des nombres réels ne sont *pas* représentables par un nombre en virgule flottante, ceux-ci doivent être approximés. Par exemple,

1. En fait, il y en a une *infinité*!

considérons la base décimale ($\beta = 10$) et une mantisse de $n = 4$ chiffres. Le nombre $x = 1{,}54163$ ne peut pas être représenté exactement. Il existe plusieurs façons d'arrondir x. Par exemple, x peut être arrondi au nombre en virgule flottante le plus près ($1{,}542$) ou à sa troncation ($1{,}541$).

Sous la première méthode, x peut se situer à distance égale de deux nombres. Cela se produit lorsqu'on prend une décision selon le chiffre $\beta/2$ suivi d'aucun chiffre ou de zéros non significatifs. Dans ce cas, la façon la plus répandue de briser l'égalité consiste à arrondir au nombre le plus près dont le *dernier chiffre est pair*, par ex. $1{,}9565000$ est arrondi à $1{,}956$, et non à $1{,}957$. Cette approche a pour avantage de ne pas favoriser les approximations vers le haut à chaque bris d'égalité, ce qui pourrait autrement amplifier les erreurs lors de plusieurs calculs successifs. Voici d'autres exemples de cette méthode pour $n = 4$:

décimal	***binaire***
$1{,}9565 \to 1{,}956$	$1{,}0101 \to 1{,}010$
$1{,}9555 \to 1{,}956$	$1{,}0011 \to 1{,}010$
$1{,}95607 \to 1{,}956$	$1{,}01001 \to 1{,}010$
$1{,}95670 \to 1{,}957$	$1{,}01011 \to 1{,}011$

12.2.1 Erreur d'approximation

Fixons e_{min}, e_{max}, n et β. Pour tout nombre $x \in \mathbb{R} \setminus \{0\}$, nous écrivons $\overline{x}$ afin de dénoter le nombre en virgule flottante normalisé arrondi au nombre le plus près de x, et où un bris d'égalité se fait vers le nombre dont le dernier chiffre est pair. Autrement dit, $\overline{x}$ est l'approximation de x selon la méthode d'arrondi que nous venons d'introduire. L'*erreur absolue* de x est $x - \overline{x}$, et l'*erreur relative* de x est:

$$\text{err}(x) \stackrel{\text{déf}}{=} \frac{x - \overline{x}}{x}.$$

L'erreur absolue donne la différence entre une valeur réelle et son approximation en nombre en virgule flottante. L'erreur relative décrit le rapport entre l'erreur absolue et la valeur, elle indique donc l'importance de l'erreur.

Posons $\varepsilon = \frac{\beta}{2} \cdot \beta^{-n}$. Nous appelons la constante ε l'*epsilon machine*. L'erreur relative d'un nombre est toujours d'au plus $\pm\varepsilon$. En effet, nous avons:

Proposition 4. $|\text{err}(x)| \leq \varepsilon$ *pour tout* $x \in \mathbb{R}$ *tel que* $x_{min} \leq |x| \leq x_{max}$.

Dans le cas particulier de $\beta = 2$, l'epsilon machine vaut $\varepsilon = 2^{-n} = 1/2^n$. Ainsi, l'erreur relative est exponentiellement plus petite que la taille de la mantisse.

Exemple.

Reconsidérons le cas où $\beta = 2$, $n = 3$ et $-1 \leq e \leq 1$. L'epsilon machine vaut $\varepsilon = 2^{-3} = 1/8$. Ainsi, l'erreur relative d'une approximation est d'au plus $\pm 1/8$. Par exemple, considérons $x = 1{,}10010011 \times 2^1$. Ce nombre n'est *pas* représentable avec une mantisse de taille $n = 3$. Sa valeur décimale est $3{,}296875$, ce qui est compris entre 3 et 3,5. Ainsi, $\overline{x} = 1{,}11 \times 2^1$.

Autrement dit, x est arrondi à $3{,}5$ et par conséquent:

$$\begin{aligned}
|\text{err}(x)| &= \left|\frac{3{,}296875 - 3{,}5}{3{,}296875}\right| \\
&= \frac{3{,}5 - 3{,}296875}{3{,}296875} \\
&\leq \frac{3{,}5 - 3{,}296875}{3{,}25} \\
&\leq \frac{3{,}5 - 3{,}25}{3{,}25} \\
&= \frac{0{,}25}{3{,}25} \\
&= 1/13 \\
&\leq 1/8.
\end{aligned}$$

Remarque.

Les erreurs se définissent de la même façon pour la troncation. Il serait aussi possible d'obtenir une borne semblable à celle de la proposition 4.

12.3 Arithmétique

Voyons comment calculer la somme et le produit de deux nombres normalisés:

$$\begin{aligned}
x &= 1{,}u \times \beta^e, \\
y &= 1{,}v \times \beta^f.
\end{aligned}$$

Nous ne couvrirons pas la gestion des signes. Elle s'accomplit en inspectant le bit de signe et en effectuant des compléments à deux au besoin.

12.3.1 Addition

Supposons que $e \leq f$. Si ce n'est pas le cas, on peut simplement inverser x et y. Observons que:

$$\begin{aligned}
x + y &= (1{,}u \times \beta^e) + (1{,}v \times \beta^f) \\
&= (1{,}u \cdot \beta^{e-f} \times \beta^f) + (1{,}v \times \beta^f) \\
&= (1{,}u \cdot \beta^{e-f} + 1{,}v) \times \beta^f.
\end{aligned}$$

L'addition se calcule donc en mettant les exposants en commun à β^f, puis en additionant les mantisses. La mise en commun des exposants peut dénormaliser le nombre avec le plus petit exposant. Il faut donc renormaliser après l'addition.

Ainsi, pour obtenir la somme, il faut:

1. Mettre les exposants en commun en décalant la première mantisse de $f-e$ positions;
2. Additionner les mantisses;
3. Normaliser le résultat en décalant la mantisse tout en incrémentant ou décrémentant l'exposant selon la direction;
4. Arrondir le résultat.

Exemple.

Reconsidérons à nouveau le cas où $\beta = 2$, $n = 3$ et $-1 \leq e \leq 1$. Additionnons ½ et ¾. La représentation normalisée de ces deux nombres est $x = 1{,}00 \times 2^{-1}$ et $y = 1{,}11 \times 2^{0}$. Nous avons donc:

$$\begin{aligned} x \cdot y &= (1{,}00 \times 2^{-1}) + (1{,}11 \times 2^{0}) \\ &= (0{,}100 \times 2^{0}) + (1{,}11 \times 2^{0}) \\ &= (0{,}100 + 1{,}11) \times 2^{0} \\ &= 10{,}010 \times 2^{0} \\ &= 1{,}0010 \times 2^{1} \\ &\approx 1{,}00 \times 2^{1}. \end{aligned}$$

Ainsi, l'addition mène à $2{,}0$ comme approximation de la valeur exacte $½ + ¾ = 0{,}5 + 1{,}75 = 2{,}25$.

12.3.2 Multiplication

Observons que:

$$\begin{aligned} x \cdot y &= (1{,}u \times \beta^{e}) \cdot (1{,}v \times \beta^{f}) \\ &= (1{,}u \cdot 1{,}v) \times \beta^{e+f}. \end{aligned}$$

Le produit s'obtient donc en multipliant les mantisses et en additionnant les exposants. Toutefois, la multiplication des mantisses peut engendrer un nombre non normalisé. Il faut donc procéder ainsi:

1. Additionner les exposants;
2. Multiplier les mantisses;
3. Normaliser le résultat en décalant la mantisse tout en incrémentant ou décrémentant l'exposant selon la direction;
4. Arrondir le résultat.

Exemple.

Reconsidérons à nouveau le cas où $\beta = 2$, $n = 3$ et $-1 \leq e \leq 1$. Multiplions ¾ et ½. La représentation normalisée de ces deux nombres est $x = 1{,}10 \times 2^{-1}$ et $y = 1{,}11 \times 2^{1}$. Nous avons donc:

$$\begin{aligned} x \cdot y &= (1{,}10 \times 2^{-1}) \cdot (1{,}11 \times 2^{1}) \\ &= (1{,}10 \cdot 1{,}11) \times 2^{0} \\ &= 10{,}101 \times 2^{0} \\ &= 1{,}0101 \times 2^{1} \\ &\approx 1{,}01 \times 2^{1}. \end{aligned}$$

Ainsi, la multiplication mène à $2{,}5$ comme approximation de la valeur exacte $¾ \cdot ½ = 0{,}75 \cdot 3{,}5 = 2{,}625$.

12.4 Norme IEEE 754

La norme IEEE 754 définit des formats de nombres en virgule flottante en base 2 et en base 10 utilisés par la grande majorité des architectures modernes. Nous nous concentrons sur les formats binaires de précision *simple* (32 bits) et précision *double* (64 bits) [2]:

format	β	n	e_{min}	e_{max}
simple	2	24	-126	127
double	2	53	-1022	1023

Les plus petits et plus grands nombres normalisés de ces formats sont donc:

format	$-x_{\text{min}}$	x_{max}
simple	-2^{-126}	$2^{128} - 2^{104} \approx 2^{128}$
double	-2^{-1022}	$2^{1024} - 2^{971} \approx 2^{1024}$

Selon notre méthode d'arrondi avec bris d'égalité, l'epsilon machine de chacun de ces formats est:

format	ϵ
simple	$2^{-24} = 0{,}00000005960464477539062 5$
double	$2^{-53} = 0{,}00000000000000011102230246251565404236316680908203125$

2. Dans la version 2008 de la norme IEEE 754, ces deux formats sont officiellement nommés `binary32` et `binary64`, respectivement. Nous utilisons plutôt la nomenclature traditionnelle *simple* et *double* afin de référer à ces deux formats.

12.4.1 Codage des formats

Les formats de précisions simple et double sont codés avec 32 et 64 bits, respectivement. Leurs bits sont répartis ainsi en mémoire:

format	*signe*	*exposant*	*mantisse*
simple	1 bit	8 bits	23 bits (+1 bit caché)
double	1 bit	11 bits	52 bits (+1 bit caché)

Nombres normalisés. Décortiquons la valeur d'un nombre en virgule flottante de précision simple (le format double est analogue):

$$\overbrace{b_{31}}^{\text{bit de signe}}\ \underbrace{b_{30}b_{29}\cdots b_{23}}_{\text{exposant}}\ \underbrace{b_{22}b_{21}\cdots b_0}_{\text{mantisse}}.$$

Le *bit de signe* b_{31} vaut 1 si le nombre est négatif, et 0 sinon.

Les bits $b_{22}b_{21}\cdots b_0$ représentent les bits situés après la virgule de la mantisse, autrement dit: $1{,}b_{22}b_{21}\cdots b_0$. Ainsi, la mantisse possède 24 bits, mais le premier bit, nommé le *bit caché*, vaut implicitement 1 et n'est donc pas stocké.

Les bits $b_{30}b_{29}\cdots b_{23}$ codent $e+127$ sous forme d'entier non signé; autrement dit, l'exposant plus un *biais* de 127. Les chaînes de bits 00000000 et 11111111 ne sont pas permises pour représenter un exposant, elles sont plutôt réservées à d'autres fins.

Exemple.

Voici trois exemples de représentation d'exposant:

$$\begin{aligned}
00000001 \text{ représente } e &= 1 - 127 = -126,\\
00100011 \text{ représente } e &= 35 - 127 = -92,\\
11111110 \text{ représente } e &= 254 - 127 = 127.
\end{aligned}$$

Zéro. Le nombre 0, qu'on ne peut pas normaliser, est représenté par un bit de signe suivi de tous les autres bits assignés à 0:

$$b_{31}00\cdots000\cdots0.$$

Ainsi, il existe deux zéros: -0 et $+0$ qui valent tous deux 0.

Infini. La norme IEEE 754 permet de représenter l'infini en assignant tous les bits de l'exposant à 1 et les bits de la mantisse à 0:

$$b_{31}11\cdots100\cdots00.$$

Le bit de signe détermine s'il s'agit de $-\infty$ ou $+\infty$.

Valeurs indéterminées. La norme IEEE 754 permet également de représenter une valeur spéciale NaN qui n'est pas un nombre («*Not a Number*»). Cette valeur est obtenue lors d'opérations comme $0/0$, $\infty - \infty$, ∞/∞, $0 \cdot \infty$ et $\sqrt{-x}$.

Cette valeur se représente en assignant tous les bits de l'exposant à 1 et la mantisse à une valeur non nulle:

$$b_{31}11\cdots1\,e\,00\cdots01.$$

Le bit e indique si une erreur doit être lancée lors de l'obtention de NaN.

Nombres dénormalisés. La norme IEEE 754 supporte également des nombres en virgule flottante *dénormalisés*, c'est-à-dire des nombres dont le premier chiffre de la mantisse est 0. Ceux-ci permettent de représenter des nombres plus près de zéro. Nous n'entrerons pas dans ces détails techniques.

Approximation. Plusieurs méthodes d'approximation sont supportées. Par défaut, la norme IEEE 754 utilise l'arrondi avec bris d'égalité vers le nombre dont le dernier chiffre est pair.

Observation.

Certains entiers de 64 bits (ou 32 bits) ne sont pas représentables par des nombres en virgule flottante de précision double (ou simple).

Observation.

Des calculs successifs peuvent créer des comportements surprenants. Par exemple, le résultat de l'addition d'une liste de nombres en virgule flottante dépend de l'ordre dans lequel les valeurs sont sommées.

12.5 Particularités de l'architecture ARMv8

12.5.1 Registres

L'architecture ARMv8 possède 32 registres de nombres en virgule flottante de 64 bits [ARM13, Sect. 5.1.2] dont l'usage est comme suit:

registres	*utilisation*
$\mathtt{d}_0 - \mathtt{d}_7$	registres d'arguments et de retour de sous-programmes
$\mathtt{d}_8 - \mathtt{d}_{15}$	registres sauvegardés par l'appelé
$\mathtt{d}_{16} - \mathtt{d}_{31}$	registres sauvegardés par l'appelant

Chaque registre $\mathtt{d}_n$ possède un sous-registre de 32 bits nommé $\mathtt{s}_n$.

Lors de l'appel d'un sous-programme, il faut faire attention à la numérotation des arguments. Par exemple, si un sous-programme reçoit un entier, un

nombre en virgule flottante et une adresse, on doit passer ces arguments dans (x_0, d_0, x_1), et non pas dans (x_0, d_1, x_2).

12.5.2 Instructions

Le jeu d'instruction des nombres en virgule flottante ressemble à celui des entiers. 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`. Voici quelques-unes des instructions:

code 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`
`fmax`	`fmax vd, vn, vm`	$v_d \leftarrow \max(v_n, v_m)$	`fmax d8, d9, d10`
`fmin`	`fmin vd, vn, vm`	$v_d \leftarrow \min(v_n, v_m)$	`fmin 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	`ucvtf d8, x19`
			`ucvtf d8, w19`
			`ucvtf s8, x19`
			`ucvtf s8, w19`

Remarque.

Il existe tout un éventail d'instructions, que nous ne couvrirons pas, qui permettent d'effectuer des conversions de types.

12.5.3 Exemple de programme

Le programme suivant calcule la norme euclidienne $\|(x, y)\| = \sqrt{x^2 + y^2}$ d'un vecteur $(x, y) \in \mathbb{R}^2$ à l'aide d'un sous-programme:

```
main:                       // main()
    // Lire x               // {
    adr     x0, fmtEntree   //
    adr     x1, temp        //
    bl      scanf           //   scanf("%lf", &temp)
    ldr     d8, temp        //   x = temp
                            //
    // Lire y               //
    adr     x0, fmtEntree   //
    adr     x1, temp        //
    bl      scanf           //
    ldr     d9, temp        //   scanf("%lf", &temp)
                            //   y = temp
    // Calculer ||(x, y)|| //
    fmov    d0, d8          //
    fmov    d1, d9          //
    bl      norme           //
    fmov    d8, d0          //   n = norme(x, y)
                            //
    // Afficher ||(x, y)|| //
    adr     x0, fmtSortie   //
    fmov    d0, d8          //
    bl      printf          //   printf("%lf\n", n)
                            //
    // Quitter              //
    mov     x0, 0           //
    bl      exit            //   return 0
                            // }
/*************************************************************
  Entrée: nombres x, y en virgule flottante précision double
  Sortie: norme euclidienne du vecteur (x, y)
*************************************************************/
norme:                      // double norme(double x, double y) {
    fmul    d16, d0, d0     //
    fmul    d17, d1, d1     //
    fadd    d18, d16, d17   //
    fsqrt   d0, d18         //
    ret                     //   return √(x² + y²)
                            // }
```

```
.section ".rodata"
fmtEntree:  .asciz  "%lf"
fmtSortie:  .asciz  "%lf\n"

.section ".bss"
            .align  8
temp:       .skip   8
```

12.6 Exercices

12.1) Normalisez ces nombres et donnez leur valeur décimale:

$$123{,}456 \times 10^{2} \qquad -0{,}0909 \times 10^{-1} \qquad 0{,}000101 \times 2^{-3} \qquad -11101{,}11 \times 2^{4}$$

12.2) Ce programme C++ additionne $x = 2^{40}$ et $y = 2^{-13}$. Puisque x et y sont des puissances de 2, ils sont représentables en base 2. Pourtant, le programme affiche « OK » et « !? ». Autrement dit, bien que $y \neq 0$, on obtient $x + y = x$, ce qui ne fait pas de sens mathématiquement. Expliquez ce qui se produit.

```
double x = 1099511627776.0; // 2⁴⁰
double y = 0.0001220703125; // 2⁻¹³
double z = x + y;

std::cout << (y != 0 ? "OK" : "!?") << std::endl
          << (z != x ? "OK" : "!?") << std::endl;
```

12.3) (a) Considérons le système de nombres en virgule flottante où $\beta = 2$, $n = 3$ et $-2 \leq e \leq 2$. Autrement dit, la base est 2, la mantisse possède 3 bits, et l'exposant varie entre -2 et 2. Effectuez cette addition:

$$(1{,}11 \times 2^{0}) + (1{,}01 \times 2^{1}).$$

Votre somme doit être *normalisée* et approximée par *troncation*.

(b) Donnez la valeur décimale (en base 10) de l'*approximation* que vous avez obtenue en (a), ainsi que de la valeur *exacte* de la somme.

(c) Quelle est l'*erreur relative* de votre approximation?

(*tiré de l'examen final de l'hiver 2019*)

12.4) Considérons le système de nombres en virgule flottante où $\beta = 2$, $n = 5$ et $-3 \leq e \leq 3$. Autrement dit, la base est 2, la mantisse possède 5 bits, et l'exposant varie entre -3 et 3. Effectuez cette multiplication:

$$(1{,}11 \times 2^{0}) \cdot (1{,}01 \times 2^{1}).$$

Votre somme doit être *normalisée* et approximée par *arrondi*.

12.5) Écrivez un sous-programme qui reçoit l'adresse d'un tableau de nombres en virgule flottante double précision, ainsi que sa taille, et qui retourne la moyenne des éléments du tableau.

12.6) Ces deux sous-questions portent sur les nombres en virgule flottante simple précision IEEE 754.

— Expliquez pourquoi la chaîne 00001101 représente l'exposant -114.

— Une chaîne de 8 bits permet de représenter 256 exposants différents. Un nombre simple précision est pourtant limité à 254 exposants différents. Pourquoi y a-t-il 254 exposants possibles plutôt que 256?

(*tiré de l'examen final de l'hiver 2019*)

13

Introduction aux entrées/sorties : NES

Dans les chapitres précédents, nous n'avons considéré qu'une seule forme d'entrée/sortie: celles d'un terminal. De plus, celles-ci se réalisaient avec des fonctions de la librairie standard du langage C. Dans ce chapitre, nous voyons comme réaliser des entrées/sorties de bas niveau. Afin d'illustrer ces concepts, nous survolons l'architecture de la console de jeux vidéo *Nintendo Entertainment System (NES)* illustrée aux figures 13.1 et 13.2. Cela permettra aussi de mettre en pratique l'ensemble de nos connaissances sur une autre architecture qu'ARMv8.

FIGURE 13.1 – Console de jeux *Nintendo Entertainment System (NES)*.

13.1 Architecture du NES

Le processeur du NES est une variante du célèbre MOS 6502 utilisant le même jeu d'instructions. La mémoire principale de la console contient 2 Kio. Le code d'un programme (jeu vidéo) est stocké sur une cartouche insérée dans la console. Une telle cartouche peut optionnellement contenir une mémoire de sauvegarde.

Le NES possède également un second processeur, dit « *picture processing unit (PPU)* ». Celui-ci se dédie à l'affichage des graphiques et possède sa propre mémoire appelée *mémoire vidéo*. Ces différents composants sont reliés par plusieurs bus de données.

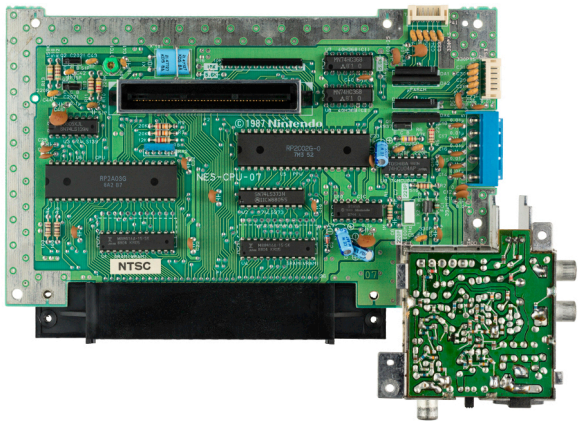


FIGURE 13.2 – Carte mère du NES.

13.1.1 Organisation de la mémoire

Nous décrivons l'organisation des deux mémoires de la console, telles qu'illustrées à la figure 13.3.

Mémoire principale. La mémoire principale, donc adressable directement par le processeur, se divise en trois segments:

- *mémoire primaire*: possède une portion de mémoire tout usage (dite *zero-page*), une pile d'exécution, ainsi qu'une autre portion généralement destinée au stockage temporaire de tuiles;
- *mémoire d'entrée/sortie*: contient des registres permettant d'effectuer des entrées/sorties notamment avec le processeur d'images et les manettes (il ne s'agit pas de registres au sens usuel puisqu'ils se situent en mémoire principale);
- *cartouche de jeu*: contient le code du programme (donc du jeu vidéo), et d'autres segments de mémoire optionnels (pour sauvegarde et expansions).

Remarquons que certaines portions de la mémoire sont simplement des *mirroirs*. Par exemple, l'adresse 0800_{16} pointe en fait vers l'adresse 0000_{16}, et l'adresse 2008_{16} pointe en fait vers l'adresse 2000_{16}.

Mémoire vidéo. La mémoire du processeur d'images se divise en trois parties:

- *tuiles*: contient deux tables des tuiles du jeu: *sprites* (personnages, objets, etc.) et de l'arrière-plan (collines, nuages, etc.);
- *arrière-plan*: quatre tables spécifiant les tuiles qui forment l'arrière-plan actuel (tuiles, positions, couleurs, orientations, etc.);
- *palettes de couleur*: décrit les couleurs disponibles pour les tuiles (permet, par exemple, de spécifier la couleur des personnages dans un monde sur terre et dans un monde sous-terrain).

Comme dans le cas de la mémoire principale, certaines portions sont simplement des *mirroirs*. Par exemple, l'adresse 3000_{16} pointe en fait vers l'adresse 2000_{16}, et l'adresse $3F20_{16}$ pointe en fait vers l'adresse $3F00_{16}$.

Notons que le processeur d'images a également accès à une mémoire de *sprites* de 256 octets située en dehors de la mémoire vidéo.

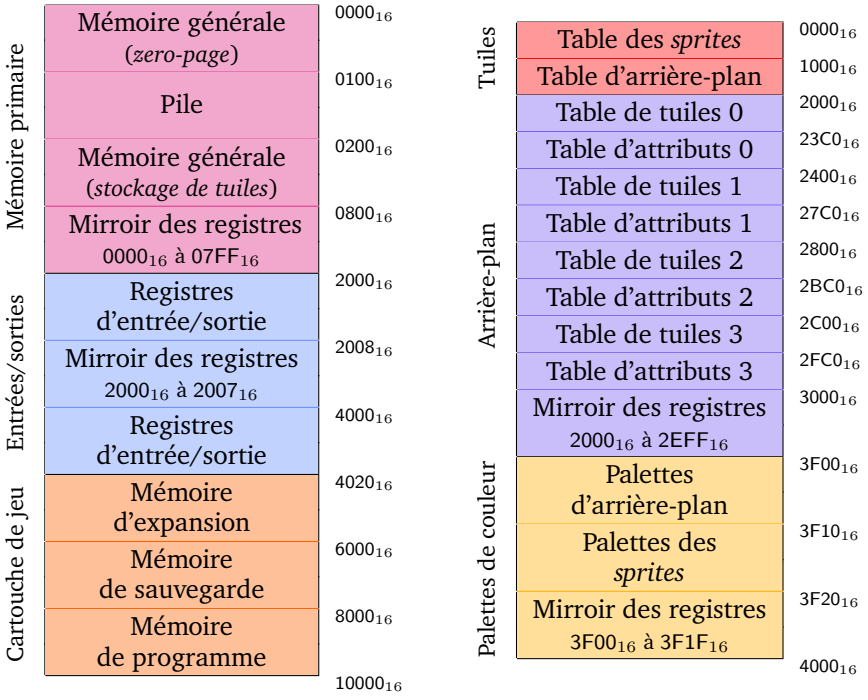


FIGURE 13.3 – Mémoire principale (*gauche*) et mémoire vidéo (*droite*) du NES.

13.2 Registres

Le processeur du NES possède quatre registres d'un octet (8 bits) chacun:

registres	*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 la pile d'exécution (pointe vers l'adresse $0100_{16} + \mathsf{s}$)

Il existe également deux registres internes:

— p: *registre d'état* qui contient des états et codes de conditions dont le report/emprunt d'une instruction arithmétique (1 octet);

— pc: *compteur d'instruction* qui contient l'adresse de la prochaine instruction à exécuter (2 octets).

Il n'y a pas de registres de nombres en virgule flottante.

13.3 Jeu d'instructions

Le jeu d'instructions possède une centaine d'instructions. Contrairement à l'architecture ARMv8, plusieurs instructions permettent de manipuler directement la mémoire sans devoir explicitement effectuer un chargement/stockage. Nous présentons un sous-ensemble du jeu d'instructions.

13.3.1 Valeurs immédiates

Les préfixes `$` et `%` indiquent que la valeur qui suit est hexadécimale et binaire, respectivement. L'absence de préfixe indique une valeur décimale. Les valeurs précédées d'un `#` représentent des valeurs numériques, alors que celles sans `#` représentent une adresse.

Exemple.

expression	*valeur*
`#5`	5_{10}
`#$FF`	FF_{16}
`#%00010011`	00010011_2
`$FF`	adresse FF_{16}

Les valeurs numériques possèdent 8 bits, alors que les adresses peuvent parfois posséder jusqu'à 16 bits.

13.3.2 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`

13.3.3 Accès mémoire

Écrivons $\text{mem}_1[a]$ afin de dénoter l'octet 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 $\leftarrow \text{mem}_1[adr]$	`lda var`
`ldx`	`ldx #i`	x $\leftarrow i$	`ldx #42`
	`ldx adr`	x $\leftarrow \text{mem}_1[adr]$	`ldx var`
`ldy`	`ldy #i`	y $\leftarrow i$	`ldy #42`
	`ldy adr`	y $\leftarrow \text{mem}_1[adr]$	`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`

13.3.4 Arithmétique

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

13.3.5 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 la droite (directement en mémoire)	`lsr var`
`and`	`and #i` `and adr`	a $\leftarrow$ a $\wedge\ i$ a $\leftarrow$ a $\wedge\ \text{mem}_1[adr]$	`and #%00100011` `and var`
`ora`	`ora #i` `ora adr`	a $\leftarrow$ a $\vee\ i$ a $\leftarrow$ a $\vee\ \text{mem}_1[adr]$	`ora #%00100011` `ora var`
`eor`	`eor #i` `eor adr`	a $\leftarrow$ a $\oplus\ i$ a $\leftarrow$ a $\oplus\ \text{mem}_1[adr]$	`eor #%00100011` `eor var`

13.3.6 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`

13.4 Sorties graphiques

13.4.1 Tuiles

L'image à l'écran est constituée de tuiles de 8×8 pixels. Chacune de ces tuiles est stockée dans la cartouche de jeu et chargée dans une table de la mémoire vidéo. Afin d'afficher une tuile, il faut spécifier sur 4 octets (dans cet ordre):

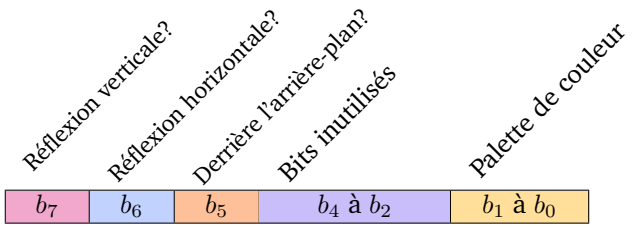
— sa position verticale comprise entre 0 et 255;

— son identifiant (sa position dans la table de tuiles);

— ses attributs tels qu'une palette de couleur;

— sa position horizontale comprise entre 0 et 255.

Les attributs d'une tuile sont déterminés par ses huit bits:



Exemple.

La séquence de 4 octets $(33, 5, 01000010_2, 160)$ spécifie d'afficher la tuile 5 à la coordonnée $(160, 33)$, de transformer la tuile par une réflexion horizontale, et d'utiliser la palette de couleurs 2.

13.4.2 Affichage de tuiles

Il y a plusieurs façons d'effectuer l'affichage. Par exemple, pour afficher un personnage constitué de n tuiles, nous pouvons:

— stocker la description de chacune de ses tuiles ($4n$ octets) consécutivement aux adresses 0200_{16} à $02FF_{16}$ de la mémoire principale;

— signaler au processeur d'images d'afficher le contenu de ces adresses en stockant 02_{16} à l'adresse 4014_{16} de la mémoire principale.

L'adresse 4014_{16} réfère à un registre d'entrée/sortie. Si on y stocke un nombre de la forme $0X_{16}$, alors cela signale au bus de données de transférer le contenu des adresses $0X00_{16}$ à $0XFF_{16}$ vers la mémoire de *sprites*. Cette technique d'*accès direct à la mémoire (DMA)* sera couverte plus en détails au chapitre 14.

Exemple.

Ce code transfère les tuiles spécifiées aux adresses 0200_{16} à $02FF_{16}$ de la mémoire principale vers la mémoire de *sprites*:

```
    lda     #$02
    sta     $4014
```

Afin d'éviter des incohérences visuelles, l'affichage ne doit se faire que durant l'*intervalle de rafraîchissement vertical (*VBLANK*)*. Cet intervalle correspond à la période de temps où le canon a électron se repositionne au haut de l'écran avant d'effectuer un nouvel affichage. Afin d'en être informé, il est possible de spécifier une sous-routine qui est appelée chaque fois que cet intervalle se produit. Durant

l'appel de cette sous-routine, le processeur met en suspens ce qu'il était en train d'exécuter, puis reprend lorsque la sous-routine est complétée. Un jeu possède donc normalement une boucle infinie qui alterne entre traitement et affichage.

13.5 Entrées à partir des manettes

Le processeur peut lire les boutons enfoncés d'une manette à l'aide d'un protocole de communication via le port de manette. Par exemple, pour l'état de la première manette, il faut (dans cet ordre):

— stocker 1 à l'adresse 4016_{16};
— stocker 0 à l'adresse 4016_{16};
— effectuer huit fois: une lecture de 4016_{16} et récupérer le bit de poids faible.

Les huit lectures consécutives correspondent, dans l'ordre, aux boutons *A*, *B*, *select*, *start*, *haut*, *bas*, *gauche*, *droite*:

Pour chacune des lectures, le bit de poids faible vaut 1 si (et seulement si) le bouton était enfoncé lors de l'initialisation du protocole. Par exemple, le code suivant vérifie si le bouton A est enfoncé:

```
    lda     #1          ;
    sta     $4016       ;
    lda     #0          ;
    sta     $4016       ; demander une lecture des boutons
                        ;
    lda     $4016       ;
    and     #%00000001  ; obtenir l'état du bouton A
```

On pourrait préserver l'état des huit boutons dans un seul octet en accumulant chacun des bits à l'aide de décalages et d'opérations de masquages.

13.6 Exemple de programme simple

Concevons un programme simple qui:

— affiche un *sprite* à l'écran qui se déplace continuellement vers la droite;
— fait alterner le *sprite* entre les tuiles représentant les caractères `0`, `1`, . . . , `9`;
— fait descendre le *sprite* lorsqu'on enfonce le bouton A.

Un aperçu animé du programme est disponible sur GitHub.

On alloue d'abord statiquement quatre octets qui représentent respectivement la position horizontale, la position verticale, l'identifiant de la tuile à afficher, ainsi que le nombre d'itérations d'une animation:

```
posX:       .rs 1               ; pos. horizontale du chiffre
posY:       .rs 1               ; pos. verticale du chiffre
chiffre:    .rs 1               ; tuile du chiffre entre 1 et 10
iter:       .rs 1               ; nombre d'itér. à effectuer
```

Le point d'entrée « `main:` » effectue quelques initialisations techniques (que nous expliquerons au prochain chapitre), ainsi que l'initialisation des variables:

```
main:                           ; main()
    ; initialisation technique  ; {
                                ;
    jsr     init_variables      ;   init_variables()
                                ;
    ; autres init. technique    ; }
```

Les tuiles 1 à 10 représentent les caractères `0` à `9` respectivement. Ceci est spécifique à la table de tuiles fournie dans ce cours, ce pourrait donc être d'autres identifiants pour un autre jeu.

Ainsi, on place initialement le *sprite* à la position $(0, 100)$ et on lui assigne la tuile 1. De plus, on initialise le nombre d'itérations d'animation à 24:

```
init_variables:                 ; init_variables()
    lda     #0                  ; {
    sta     posX                ;   posX = 0
    lda     #100                ;
    sta     posY                ;   posY = 100
    lda     #1                  ;
    sta     chiffre             ;   chiffre = 1 (no. de tuile)
    lda     #24                 ;
    sta     iter                ;   iter = 24
    rts                         ; }
```

À chaque intervalle de rafraîchissement vertical (*VBLANK*), la sous-routine « `update:` » est automatiquement appelée. Celle-ci signale au bus de données de transférer l'état des *sprites* vers la mémoire de *sprites*, puis déplace et met à jour notre unique *sprite*:

```
update:                         ; update()
    lda     #$02                ; {
    sta     $4014               ;   copier tuiles 0x0200 à
                                ;               0x02FF vers PPU
                                ;
    jsr     deplacer_chiffre    ;   deplacer_chiffre()
    jsr     update_chiffre      ;   update_chiffre()
                                ;
    rti                         ; }
```

Afin de déplacer notre *sprite*, on incrémente d'abord la position horizontale `posX`, on demande une lecture des boutons, puis on lit le bit d'état du bouton A qu'on additionne à `posY`. Ainsi, si le bouton est enfoncé, alors la position verticale est incrémentée, et autrement elle demeure la même.

```
deplacer_chiffre:               ; deplacer_chiffre() {
    inc     posX                ;   posX++
                                ;
    lda     #1                  ;
    sta     $4016               ;
    lda     #0                  ;
    sta     $4016               ;   demander lecture des boutons
                                ;
    lda     $4016               ;   lire bit b de poids
    and     #%00000001          ;              faible du bouton A
                                ;
    clc                         ;
    adc     posY                ;
    sta     posY                ;   posY += b (incrémente posY
                                ;              si A est enfoncé)
    rts                         ; }
```

Finalement, la sous-routine « `update_chiffre:` » met le *sprite* à jour:

— on décrémente `iter`;

— si ce-dernier a atteint 0, alors on modifie le *sprite* et on remet `iter` à 24; sinon, on ne fait rien (cela permet de ralentir l'animation, en choisissant 48 on aurait par ex. une animation deux fois plus lente);

— si le *sprite* doit être modifié, alors on passe au chiffre suivant en incrémentant l'identifiant de tuile `chiffre`;

— on stocke les quatre octets $(\texttt{posY}, \texttt{chiffre}, 0, \texttt{posX})$ à partir de l'adresse 0200_{16} en vue du prochain affichage.

```
update_chiffre:                 ; update_chiffre()
    ; Position verticale        ; {
    lda     posY                ;
    sta     $0200               ;   mem[0x0200] = posY
                                ;
    ; Choix de la tuile         ;
    lda     chiffre             ;   mem[0x0201] = chiffre
    sta     $0201               ;
                                ;
    lda     iter                ;
    cmp     #0                  ;   if (iter != 0) {
    beq     prochain_chiffre    ;
                                ;
```

```
prochaine_iteration:            ;
    sec                         ;
    dec     iter                ;     iter--
    jmp     continuer           ;   }
prochain_chiffre:               ;   else {
    lda     #24                 ;
    sta     iter                ;     iter = 24
    inc     chiffre             ;     chiffre++
    lda     chiffre             ;
    cmp     #11                 ;
    bne     continuer           ;     if (chiffre == 11) {
    lda     #1                  ;
    sta     chiffre             ;       chiffre = 1
                                ;     }
continuer:                      ;   }
    ; Attributs de la tuile     ;
    lda     #%00000000          ;
    sta     $0202               ;   mem[0x0202] = 0
                                ;
    ; Position horizontale      ;
    lda     posX                ;
    sta     $0203               ;   mem[0x0203] = posX
                                ;
    rts                         ; }
```

Remarque.

Le mot-clé « `.rs` » alloue un certain nombre d'octets comme « `.skip` » sur ARMv8. On peut aussi utiliser « `.byte` » afin d'allouer des octets *initialisés*. Par exemple, « `.byte $8C, $01, $00, $00` » alloue les octets $(8\mathrm{C}_{16}, 01_{16}, 00_{16}, 00_{16})$ consécutivement en mémoire.

13.7 Exercices

13.1) Complétez le sous-programme « `lecture:` » ci-dessous afin qu'il assigne la valeur 1 à la variable `appuye` si les boutons *select* et *start* de la première manette sont *tous deux* appuyés, et 0 autrement.

```
appuye:    .rs 1 ; variable d'un octet

lecture:
    /* code ici */
    rts
```

Rappel:

— l'adresse `$4016` du NES est liée au port de sa première manette;

— pour initier une lecture, il faut envoyer 1, puis 0, vers son port;

— l'ordre des boutons est: *A*, *B*, *select*, *start*, *haut*, *bas*, *gauche*, *droite*.

(*tiré de l'examen final de l'hiver 2019*)

13.2) Écrivez un sous-programme qui *décrémente* la variable « `posX:` » si la flèche gauche est enfoncée.

13.3) Comment peut-on afficher les tuiles stockées de 0300_{16} à $03FF_{16}$?

13.4) La variable booléenne « `dir:` » du programme ci-dessous indique les directions *gauche* et *droite* respectivement à l'aide des valeurs 0 et 1. Complétez le sous-programme « `renverser:` » afin qu'il applique une réflexion horizontale sur la tuile lorsque la direction indique la droite.

```
dir:      .rs 1 ; vaut 0 (gauche) ou 1 (droite)
tuile:    .rs 4 ; (posY, identifiant, attributs, posX)

renverser:
    /* code ici */
    rts
```

14

Entrées/sorties

Un ordinateur est normalement constitué de périphériques d'entrée/sortie qui lui permettent d'interagir avec le monde extérieur (clavier, souris, moniteur, mémoire externe, webcam, imprimante, pavé tactile, capteurs, etc.) Ces périphériques ne sont généralement pas synchronisés avec le processeur. Ainsi, certains mécanismes sont nécessaires afin que le processeur traite les différentes entrées et sorties. Dans ce chapitre, nous décrivons certains de ces mécanismes. La plupart des concepts seront illustrés avec l'architecture du NES introduite au chapitre 13; à l'exception des appels système qui seront illustrés avec ARMv8.

14.1 Attente active

L'un des mécanismes les plus simples afin d'interagir avec un contrôleur d'entrée/sortie consiste à interroger certains bits d'états. Par exemple, le processeur d'images du NES possède un *registre d'état* en lecture seule accessible à l'adresse `$2002`. En particulier, le bit de poids fort de ce registre indique si l'intervalle de rafraîchissement vertical (*VBLANK*) est en cours. Ainsi, on peut programmer l'affichage de tuiles de cette façon:

— lire continuellement `$2002` jusqu'à ce que le bit de poids fort égale 1;

— envoyer les tuiles vers le processeur d'images.

Cette attente active s'implémente ainsi:

```
attendre_vblank:              ; attendre_vblank()
    lda     $2002             ; {
    and     #%10000000        ;   while (!VBLANK) {
    cmp     #%10000000        ;     // ne rien faire
    bne     attendre_vblank   ;   }
    rts                       ; }
```

Ainsi, la routine principale du programme attend, effectue l'affichage lorsque l'attente se débloque, puis recommence:

```
main:                           ; while (true) {
    jsr     attendre_vblank     ;   attendre activement le VBLANK
    jsr     afficher_tuiles     ;   afficher les tuiles
    jmp     main                ; }
```

En général, cette méthode est connue sous le nom d'*attente active* puisqu'elle attend en répétant constamment une opération.

14.2 Interruptions

L'attente active se démarque par sa simplicité, mais elle monopolise les cycles du processeur et empêche toute autre instruction d'être exécutée. Ainsi, elle fonctionne relativement bien sur le NES puisque l'intervalle de rafraîchissement se produit aux $16{,}\overline{6}$ millisecondes, mais elle est peu adaptée aux systèmes où les entrées/sorties se produisent rarement ou à des fréquences variables.

Les *interruptions* offrent une solution élégante à cette problématique. Plutôt que d'attendre incessamment qu'un événement se produise, un signal est lancé lorsqu'il se produit. Le processeur s'interrompt alors et lance une sous-routine qui traite l'événement. Lorsque la sous-routine se termine, le processeur reprend ses activités. Ainsi, un programme qui lit une touche au clavier, par exemple, n'a pas à bloquer l'ordinateur jusqu'à ce que l'on appuie sur une touche.

14.2.1 Gestionnaires d'interruption

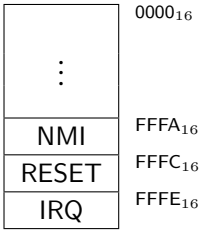
Le NES possède trois types d'interruptions:

— NMI: lancée lors de l'intervalle de rafraîchissement vertical (*VBLANK*);

— RESET: lancée au démarrage de la console (bouton « POWER » enfoncé) ou lorsque la console est redémarrée (bouton « RESET » appuyé);

— IRQ: ne possède pas d'usage particulier, mais peut, par exemple, être lancée par une puce électronique d'une cartouche de jeu.

Les six derniers octets de la mémoire principale contiennent l'adresse des sous-routines qui doivent être appelées afin de gérer ces interruptions:



En général, une sous-routine appelée lors d'une interruption se nomme un *gestionnaire d'interruption*, et le segment de mémoire qui contient l'adresse de chaque gestionnaire se nomme *table d'interruptions* ou *vecteur d'interruptions*. Il s'agit d'un tableau de pointeurs similaire à une table de branchement.

Ainsi, dans le cas du NES, la sous-routine d'affichage et le point d'entrée du jeu peuvent être assignés respectivement comme gestionnaires des interruptions NMI et RESET dans la table d'interruptions. Cela permet de retirer la boucle d'attente active:

```
main:                           ; main()
    ; initialisation du jeu     ; {
main_boucle:                    ;   while (true)
    jmp     main_boucle         ;     rien faire
    rti                         ; }
                                ;
mise_a_jour:                    ; mise_a_jour() {
    jsr     traitement          ;   traitement()
    jsr     afficher_tuiles     ;   afficher_tuiles()
    rti                         ; }
                                ;
.org    $FFFA                   ; Table d'interruptions (à 0xFFFA)
.word   mise_a_jour             ;  NMI
.word   main                    ;  RESET
.word   0                       ;  IRQ (aucune sous-routine)
```

14.2.2 Traitement des interruptions

Lorsqu'un gestionnaire d'interruption est appelé, l'exécution actuelle du processeur doit être mise en suspens. Typiquement, l'instruction en cours d'exécution est complétée, puis le gestionnaire est appelé essentiellement comme le serait un sous-programme. Cependant, contrairement aux sous-programmes, l'exécution d'un gestionnaire d'interruption ne doit pas altérer l'état du processeur.

Exemple.

Considérons ce programme:

```
foo:                            ; Sous-programme
    cmp     #0
    beq     foo
    rts

bar:                            ; Gestionnaire d'interruption
    cmp     #1
    rti
```

Ici, « `foo` » et « `bar` » désignent, respectivement, un sous-programme et le gestionnaire d'une interruption i. Supposons que `foo` soit appelé et que l'accumulateur `a` contienne la valeur 1. Puisque `a` $\neq 0$, aucun branchement ne se produira et `foo` retournera à son appelant.

> Considérons un scénario alternatif où `foo` exécute « `cmp #0` » alors qu'une interruption i est lancée. Le processeur complète la comparaison en cours, puis exécute `bar`. Celui-ci effectue la comparaison « `cmp #1` » et se termine. Le processeur reprend donc l'exécution de `foo` en exécutant « `beq foo` ». Or, `bar` a modifié les codes de condition lors de sa comparaison, et ainsi un branchement est effectué dans `foo`, ce qui ne correspond pas au comportement attendu.

Un mécanisme doit donc être mis en place afin d'éviter ce comportement problématique. En fait, sur l'architecture du NES, et plus généralement sur MOS 6502, ce problème ne se produit *pas*. En effet, lors du traitement d'une interruption, les opérations suivantes sont effectuées automatiquement:

— l'instruction en cours d'exécution est complétée;
— l'octet de poids fort de l'adresse de retour est empilé;
— l'octet de poids faible de l'adresse de retour est empilé;
— le contenu du registre d'état `p` est empilé;
— l'octet de poids fort du gestionnaire d'interruption est récupéré;
— l'octet de poids faible du gestionnaire d'interruption est récupéré.

Remarquons qu'un gestionnaire d'interruption ne se termine pas par l'instruction « `rts` », mais bien par « `rti` ». Cette instruction effectue la séquence d'opérations inverse. En particulier, elle rétablit le contenu de `p` à partir de la pile. Ainsi, bien que `bar` modifie les codes de condition, ceux-ci sont rétablis à la fin de son exécution, et `foo` n'est pas affecté.

Les registres `a`, `x` et `y` ne sont cependant pas sauvegardés automatiquement. Ainsi un gestionnaire d'interruption doit manuellement rétablir leur contenu. L'architecture du NES possède une instruction qui permet d'empiler `a`, ainsi qu'une instruction qui permet de dépiler vers `a`. Ainsi, nous pouvons implémenter la sauvegarde et la restauration de registres similairement aux macros `SAVE` et `RESTORE` mises au point pour l'architecture ARMv8:

```
    ; Sauvegarde des registres
    pha                         ; empiler a
    txa                         ; a = x
    pha                         ; empiler a
    tya                         ; a = y
    pha                         ; empiler a

    ; Restauration des registres
    pla                         ; dépiler vers a
    tay                         ; y = a
    pla                         ; dépiler vers a
    tax                         ; x = a
    pla                         ; dépiler vers a
```

En ajoutant ce code respectivement au début et à la fin d'un gestionnaire d'interruption, on rétablit donc entièrement l'environnement.

14.2.3 Niveaux de priorité

Le NES possède peu d'interruptions. Toutefois, un ordinateur moderne peut en posséder une dizaine, voire des centaines[1]. Ainsi, un périphérique peut lancer une interruption alors qu'un gestionnaire d'interruption est déjà en cours d'exécution. Lorsque cela se produit, le processeur doit faire un choix en fonction de l'importance de l'interruption.

Par exemple, chaque type d'interruption peut posséder une *priorité* spécifiée par une valeur numérique comprise entre 0 et n. Dans ce cas, plus la valeur numérique est petite, plus la priorité est importante. Lors de la gestion d'une interruption de niveau i, les interruptions moins prioritaires, donc de $i+1$ à n, sont ignorées. Supposons qu'un gestionnaire `G` d'une interruption de niveau i soit en cours d'exécution. Si une interruption de priorité égale ou supérieure, donc de 0 à i, est lancée, alors l'état de `G` est sauvegardé (par ex. sur une pile), et un gestionnaire d'interruption `G'` est exécuté afin de traiter la nouvelle interruption. Lorsque `G'` termine, `G` reprend le contrôle.

Les interruptions de niveau 0 sont dites *non masquables* car elles ne peuvent pas être ignorées. Par exemple, l'interruption RESET du NES est non masquable.

Les interruptions qui sont ignorées par le processeur ne sont pas nécessairement oubliées. Elles peuvent être mises en attente en « allumant » un certain bit d'un *registre d'interruption*. Lorsque le niveau de priorité actuel le permet, le processeur peut ainsi servir une interruption en attente en lançant son gestionnaire.

Certaines interruptions qui ne peuvent pas être masquées automatiquement par le processeur, peuvent être masquées manuellement. Par exemple, l'interruption NMI du NES est non masquable, mais elle peut être désactivée en mettant le bit de poids fort du registre de contrôle `$2000` à zéro. Cela s'avère pratique afin d'initialiser un jeu avant de débuter l'affichage:

```
main:                       ;
    lda     #%00000000      ;
    sta     $2000           ; désactiver les interruptions NMI

    ; Initialisation ici

    lda     #%10000000      ;
    sta     $2000           ; activer les interruptions NMI

    ; Début de l'affichage
```

1. Du moins, au niveau logiciel.

14.2.4 Interruptions logicielles

Les interruptions réfèrent typiquement à des signaux lancés par des dispositifs d'entrée/sortie. Cependant, plusieurs architectures offrent des *interruptions logicielles*. Contrairement aux interruptions matérielles qui sont asynchrones puisqu'elles dépendent de facteurs externes, les interruptions logicielles sont lancées par une instruction (logicielle).

Par exemple, sur le NES, et plus généralement sur l'architecture MOS 6502, l'instruction « `brk` » lance une interruption IRQ. Cela permet notamment d'implémenter un débogueur puisque « `brk` » interrompt l'exécution du programme à une ligne précise.

14.3 Accès direct à la mémoire

Afin d'afficher un *sprite* sur le NES, ses tuiles doivent être stockées dans la mémoire de *sprites*. Rappelons que cette mémoire comporte 256 octets. Comme le processeur n'a pas accès à cette mémoire, il doit y accéder indirectement. Si nous désirons écrire la valeur v à l'adresse $0 \leq a \leq 255$ de la mémoire de *sprites*, il est possible de procéder en deux étapes:

— écrire a à l'adresse `$2003`,

— écrire v à l'adresse `$2004`.

Après ces deux étapes d'initialisation, on écrit v à l'adresse a et le contenu de `$2003` est automatiquement incrémenté par le processeur d'images. Ainsi, une tuile décrite par les quatre octets $(100, 1, 0, 127)$ peut être affichée ainsi:

```
    lda     #$00                ; débuter l'écriture dans la
    sta     $2003               ;   mémoire de sprites à 0x00
                                ;
    ; Position verticale        ;
    lda     #100                ;
    sta     $2004               ; mem_sprite[0x00] = 100
                                ;
    ; Numéro de la tuile        ;
    lda     #1                  ;
    sta     $2004               ; mem_sprite[0x01] = 1
                                ;
    ; Attributs de la tuile     ;
    lda     #%00000000          ;
    sta     $2004               ; mem_sprite[0x02] = 0
                                ;
    ; Position horizontale      ;
    lda     #127                ;
    sta     $2004               ; mem_sprite[0x03] = 127
```

Cette méthode est simple, mais requiert $4n$ accès mémoire afin de stocker n tuiles, ce qui accapare plusieurs cycles du processeur. Le NES offre un autre mécanisme plus efficace afin de transférer plusieurs tuiles: l'*accès direct à la mémoire (DMA)*. Celui-ci permet au processeur d'initier un transfert de données, puis de laisser un contrôleur effectuer lui-même le transfert.

Pour ce faire, il faut:

— stocker la description des tuiles dans un segment contigu de la mémoire principale, par exemple `$0200` à `$02FF`;

— écrire l'octet de poids fort du segment à l'adresse `$4014`.

Ainsi, les tuiles peuvent être envoyées comme ceci:

```
    ; Position verticale        ;
    lda     #100                ;
    sta     $0200               ; mem[0x0200] = 100
                                ;
    ; Numéro de la tuile        ;
    lda     #1                  ;
    sta     $0201               ; mem[0x0201] = 1
                                ;
    ; Attributs de la tuile     ;
    lda     #%00000000          ;
    sta     $0202               ; mem[0x0202] = 0
                                ;
    ; Position horizontale      ;
    lda     #127                ;
    sta     $0203               ; mem[0x0203] = 127
                                ;
    ; Stocker les autres tuiles ; ...
                                ;
    ; Initier un transfert      ;
    lda     #$02                ; copier mem[0x0200, 0x02FF]
    sta     $4014               ;   vers la mémoire de sprites
```

14.4 Appels système

Les programmes exécutés par l'intermédiaire d'un système d'exploitation n'ont généralement pas un accès direct aux périphériques d'entrée/sortie pour des raisons de sécurité. Le noyau du système d'exploitation offre plutôt des services qui doivent être appelés via une interruption logicielle nommée *appel système*.

Par exemple, sur les systèmes de type UNIX, les appels système « `write` » et « `read` » permettent d'effectuer des écritures/lectures vers/à partir d'une ressource du système. Les fonctions de haut-niveau « `printf` » et « `scanf` », que nous avons utilisé jusqu'ici, sont implémentées à l'aide de ces appels système.

Sur ARMv8, pour effectuer un appel système, il faut:

— indiquer le code du service dans x_8;
— passer les arguments dans x_0 à x_5;
— utiliser l'instruction « `svc 0` ».

Par exemple, les appels système « `write` » et « `read` » de UNIX sont décrits par ces codes et paramètres:

service	*code*	*paramètre 0*	*paramètre 1*	*paramètre 2*
`write`	64	flux de sortie	adresse de la chaîne à afficher	nombre d'octets de la chaîne
`read`	63	flux d'entrée	adresse où stocker la chaîne lue (mémoire tampon)	nombre d'octets à lire

Le programme ci-dessous lit une chaîne de 10 octets au clavier et affiche cette chaîne à l'aide de sous-programmes qui utilisent des appels système plutôt que « `scanf` » et « `printf` »:

```
main:                       // main()
    adr     x0, temp        // {
    mov     x1, 10          //
    bl      lire            //   lire(&temp, 10)
                            //
    adr     x0, temp        //
    mov     x1, 10          //
    bl      afficher        //   afficher(&temp, 10)
                            //
    mov     x0, 0           //
    bl      exit            // }
                            //
afficher:                   // afficher(chaine, taille)
    mov     x9, x0          // {
    mov     x10, x1         //
                            //
    mov     x8, 64          //   /* write  = 64
    mov     x0, 1           //      stdout = 1  */
    mov     x1, x9          //
    mov     x2, x10         //
    svc     0               //   write(stdout, chaine, taille)
                            //
    ret                     // }
                            //
lire:                       // lire(tampon, taille)
    mov     x9, x0          // {
    mov     x10, x1         //
                            //
    mov     x8, 63          //   /* read   = 63
```

```
    mov     x0, 0           //      stdout = 0  */
    mov     x1, x9          //
    mov     x2, x10         //
    svc     0               //   read(stdin, tampon, taille)
                            //
    ret                     // }

.section ".bss"
temp:   .skip   10
```

Remarque.

Nous avons toujours utilisé de la mémoire allouée *statiquement*. Cela peut sembler irréaliste en comparaison aux programmes de haut niveau.

Cependant, il n'y a pas de différence fondamentale. En effet, afin d'allouer n octets dynamiquement sur un système de type UNIX, on pourrait simplement:

— utiliser le service « `brk` » afin d'obtenir l'adresse a du dessus du tas;

— utiliser le service « `brk` » afin de demander au système d'exploitation d'incrémenter cette adresse de n octets.

Par la suite, la région de la mémoire principale comprise dans l'intervalle $[a, a+n)$ pourrait être utilisée librement par notre programme, exactement de la même façon que pour la mémoire allouée statiquement.

Un programme, qui dépasse légèrement le cadre du cours, est disponible sur GitHub. Celui-ci effectue la lecture d'une chaîne de caractères de *taille arbitaire*, et ce *sans* utiliser « `scanf` ».

Remarque.

Les différents types de systèmes d'exploitation et d'appels systèmes, le fonctionnement d'un noyau, ainsi que la gestion de la mémoire dynamique, seront couverts dans le cours IFT320 – Systèmes d'exploitation.

14.5 Exercices

14.1) Écrivez un sous-programme pour le NES qui attend activement que le bouton *start* soit enfoncé afin d'appeler le sous-programme « `faire_pause` ».

14.2) Le processeur du NES peut accéder, indirectement, à la mémoire vidéo du processeur d'images (*PPU*). Afin d'accéder à l'adresse XYZW_{16} de la mémoire vidéo, il faut:

— envoyer XY_{16} à l'adresse 2006_{16} de la mémoire principale;

— envoyer ZW_{16} à l'adresse 2006_{16} de la mémoire principale;

— effectuer l'opération désirée:

- *lecture*: lire l'octet à l'adresse 2007_{16} de la mémoire principale pour obtenir celui à l'adresse XYZW_{16} de la mémoire vidéo;
- *écriture*: écrire à l'adresse 2007_{16} de la mémoire principale afin d'écrire à l'adresse XYZW_{16} de la mémoire vidéo.

Écrivez un programme qui lit l'octet à l'adresse 2000_{16} de la mémoire vidéo et le copie à l'adresse $\mathsf{2C00}_{16}$ de la mémoire vidéo.

14.3) Sur le NES, pourquoi complète-t-on l'exécution d'un gestionnaire d'interruption à l'aide de « `rti` » plutôt que « `rts` »?

14.4) Nos programmes ARMv8 se terminent par l'appel `exit(0)` afin de demander la terminaison du programme sans erreur. Nous utilisions jusqu'ici la fonction offerte par la librairie standard C. Comment peut-on plutôt appeler le service `exit` du noyau de Linux (sachant que son code est 93)?

A

Solutions des exercices

Cette section présente des solutions à certains des exercices du document. Dans certains cas, il ne peut s'agir que d'ébauches de solutions.

Chapitre 9

9.1) $\neg a = 010010$, $a \wedge b = 101000$, $a \vee b = 101111$, $a \oplus b = 000111$

9.2) Pour effectuer un décalage circulaire d'une chaîne de n bits de k bits vers la gauche, on effectue un décalage circulaire de $n - k$ bits vers la droite.

9.3)
```
and  xd, xd, 1
eor  xd, xd, 1
```

9.4) Voir sur GitHub.

9.5) Soit $x = x_{n-1} \cdots x_1 x_0$ un entier signé de n bits. Soit x' le résultat d'un décalage arithmétique de x d'un bit vers la droite. Remarquons que $x' = x_{n-1} x_{n-1} \cdots x_1$. Par définition, nous avons:

$$\text{val}(x) = x_{n-1} \cdot -2^{n-1} + \sum_{i=1}^{n-2} x_i \cdot 2^i + x_0.$$

Ainsi, la division entière par deux mène à:

$$\text{val}(x) \div 2 = x_{n-1} \cdot -2^{n-2} + \sum_{i=1}^{n-2} x_i \cdot 2^{i-1}. \tag{A.1}$$

Montrons que $\text{val}(x') = \text{val}(x) \div 2$:

$$\begin{aligned}
\text{val}(x') &= x_{n-1} \cdot -2^{n-1} + \sum_{i=1}^{n-1} x_i \cdot 2^{i-1} && \text{(par déf. de val}(x')\text{)} \\
&= x_{n-1} \cdot -2^{n-1} + x_{n-1} \cdot 2^{n-2} + \sum_{i=1}^{n-2} x_i \cdot 2^{i-1} \\
&= x_{n-1} \cdot (-2^{n-1} + 2^{n-2}) + \sum_{i=1}^{n-2} x_i \cdot 2^{i-1} \\
&= x_{n-1} \cdot -2^{n-2} + \sum_{i=1}^{n-2} x_i \cdot 2^{i-1} \\
&= \text{val}(x) \div 2 && \text{(par (A.1))} \quad \square
\end{aligned}$$

9.6) -1221_{10}

9.7) 3

9.8)

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

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

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

Chapitre 10

10.1) En inspectant la table des plages de code, on voit que ces codes donnent lieu à des codages de 2, 1, 4 et 3 octets, respectivement. Après conversion des codes numériques de l'hexadécimal vers le binaire, on obtient:

$0006AB_{16} \rightarrow$ 1101010101 $\rightarrow$ 11001101 10010101
$000073_{16} \rightarrow$ 1110011 $\rightarrow$ 01110011
$012345_{16} \rightarrow$ 10010001101000101 $\rightarrow$ 11110000 10010010 10001101 10000101
$00A0B1_{16} \rightarrow$ 10100000010110001 $\rightarrow$ 11101010 10000010 10110001

10.2) Voir sur GitHub

10.3) Voir sur GitHub

10.4) Voir sur GitHub

10.6)
- $FC_{16} \rightarrow 11111100_2 \rightarrow$ 01000111 00111100
- 6 caractères:

1 octet	2 octets	2 octets	1 octet	1 octet	1 octet

- On détermine d'abord s'il s'agit d'un caractère d'un ou deux octets avec une opération de masquage, puis on extrait le code numérique à l'aide d'un décalage et d'opérations de masquage:

```
codage:
    SAVE
    and     x19, x0, 0xFF00
    cmp     x19, 0
    b.eq    un_octet
deux_octets:
    and     x19, x0, 0x3000
    and     x20, x0, 0x0F00
    orr     x19, x19, x20
    lsr     x19, x19, 3
    and     x20, x0, 0x1F
    orr     x19, x19, x20
    b       fin
un_octet:
    and     x19, x0, 0x7F
fin:
    mov     x0, x19
    RESTORE
    ret
```

10.7) « `ldr w21, [x19, x20, lsl 2]` »

Chapitre 11

11.1) Voir sur GitHub.

11.2) $AEE9_{16}$ (décrémenté de $4 \cdot 8 = 32$), $AEF9_{16}$ (incrémenté de $2 \cdot 8 = 16$).

11.3) On peut empiler x_{zr}, ce qui gaspille 8 octets mis à zéro. Lorsque l'on dépile les quatre double mots, on dépile ces 8 octets et les ignore.

11.4)

```
exp:                    // exp(n)
     SAVE               // {
     mov   x19, x0      //
     mov   x20, 1       //   r = 1
                        //
     cbz   x19, fin     //   if (n != 0)
                        //   {
     lsr   x0, x19, 1   //
     bl    exp          //     k = exp(n / 2)
     mul   x20, x0, x0  //     r = k * k
                        //
     tbz   x19, 0, fin  //     if (n est pair)
     lsl   x20, x20, 1  //       r *= 2
fin:                    //   }
     mov   x0, x20      //
     RESTORE            //
     ret                //   return r
                        // }
```

11.5)

```
.macro SAUVEGARDER              .macro RESTAURER
  stp   x29, x30, [sp, -48]!      ldp   x19, x20, [sp, 16]
  mov   x29, sp                  ldp   x23, x25, [sp, 32]
  stp   x19, x20, [sp, 16]       ldp   x29, x30, [sp], 48
  stp   x23, x25, [sp, 32]      .endm
.endm
```

11.6) Les étiquettes « `foo5:` », « `foo6:` » et « `foo7:` » sont atteintes infiniment souvent. En effet, puisque le registre x_{30} n'est pas stocké sur la pile d'exécution, l'appel à « `printf` » détruit l'adresse de retour de « `foo:` » et ainsi « `ret` » branche à « `foo5:` ».

Chapitre 12

12.1) Nombres normalisés:

$$1{,}23456 \times 10^{4} \qquad -9{,}09 \times 10^{-3} \qquad 1{,}01 \times 2^{-7} \qquad -1{,}110111 \times 2^{8}$$

Valeurs décimales:

$$12345{,}6 \quad -0{,}00909 \quad 5/512 = 101_2 \times 2^{-9} \quad -476 = 119 \cdot 4 = -1110111_2 \times 2^2$$

12.2) Les deux variables sont représentées par des nombres en virgule flottante double précision de la norme IEEE 754. Nous avons:

$$x = 1{,}0 \times 2^{40} \qquad y = 1{,}0 \times 2^{-13}$$

Calculons $x + y$. Mettons les deux nombres sous un exposant commun:

$$x = 1{,}0 \times 2^{40} \qquad y = 0{,}\underbrace{0 \cdots 0}_{52 \text{ fois}}1 \times 2^{40}$$

En ajoutant des zéros non signicatifs à droite de la mantisse de x, puis en effectuant la somme bit à bit à partir de la droite, on obtient:

$$\begin{array}{rr} & 1{,}\overbrace{0 \cdots 0}^{52 \text{ fois}}0 \times 2^{40} \\ + & 0{,}\underbrace{0 \cdots 0}_{52 \text{ fois}}1 \times 2^{40} \\ \hline & 1{,}\underbrace{0 \cdots 0}_{52 \text{ fois}}1 \times 2^{40} \end{array}$$

Notons que le résultat est déjà normalisé. Cependant, la mantisse possède trop de bits: 54 bits plutôt que les 53 de la norme IEEE 754. On doit donc arrondir, ce qui mène à $1{,}\underbrace{0 \cdots 0}_{23 \text{ fois}} \times 2^{40}$. Ainsi, $x + y = 2^{40} = x$.

12.3) (a)

$$\begin{aligned} 1{,}11 \times 2^0 + 1{,}01 \times 2^1 &= \ 0{,}111 \ \times 2^1 + 1{,}010 \times 2^1 \\ &= 10{,}001 \ \times 2^1 \\ &= \ 1{,}0001 \times 2^2 \\ &\approx \ 1{,}00 \ \times 2^2 \end{aligned}$$

(b) 4,0 (approximation) et 4,25 (valeur exacte)

(c) 1/16 obtenu ainsi:

$$\begin{aligned} (4{,}25 - 4{,}0)/4{,}0 &= 0{,}25/4{,}0 \\ &= (1/4)/4 \\ &= 1/16 \end{aligned}$$

12.4) Nous avons:

$$\begin{aligned}(1{,}11 \times 2^0) \cdot (1{,}01 \times 2^1) &= (1{,}11 \cdot 1{,}01) \times 2^{0+1} \\ &= (1{,}11 \cdot 1{,}01) \times 2^1.\end{aligned}$$

Évaluons le produit des mantisses en binaire:

$$\begin{array}{rr} & 1{,}11 \\ \times & 1{,}01 \\ \hline & 0{,}0111 \\ + & 0{,}000 \\ & 1{,}11 \\ \hline & 10{,}0011 \end{array}$$

Ainsi, nous obtenons $10{,}0011 \times 2^1$ et après normalisation: $1{,}00011 \times 2^2$. Il faut arrondir la mantisse $1{,}00011$. On arrondit vers le haut afin d'obtenir $1{,}0010$. Ainsi, le résultat de la multiplication est $1{,}0010 \times 2^2$.

Bien que ce ne soit pas demandé, notons que la valeur obtenue est $4{,}5$ alors que la valeur exacte serait $4{,}375$. Ainsi, l'erreur relative est de $(4{,}375 - 4{,}5)/4{,}5 = -0{,}125/4{,}5 = -(1/8)/(9/2) = -2/72 = -1/36 = -0{,}02\overline{7}$.

12.5) Voir sur GitHub.

12.6)
- Parce que la représentation a un biais de -127: $13 - 127 = -114$.
- Pour représenter des valeurs spéciales: ± 0, $\pm\infty$, NaN.

Chapitre 13

13.1) On établit une connexion avec le port de manette; on ignore les boutons Ⓐ et Ⓑ; puis on combine l'état des deux boutons suivants avec un ET:

```
appuye:    .rs 1

lecture:                          ; lecture()
    ; Demander lecture boutons    ; {
    lda    #1                     ;   envoyer 1
    sta    $4016                  ;     au port de manette 1
    lda    #0                     ;   envoyer 0
    sta    $4016                  ;     au port de manette 1
                                  ;
    ; Déterminer si SELECT       ;
    ;     et START sont appuyés   ;
    lda    $4016                  ;   lire et ignorer A
    lda    $4016                  ;   lire et ignorer B
                                  ;
    lda    $4016                  ;
    and    #%00000001             ;   lire état x de SELECT
    sta    appuye                 ;   appuye = x
                                  ;
    lda    $4016                  ;
    and    #%00000001             ;   lire état y de START
    and    appuye                 ;
    sta    appuye                 ;   appuye &= y
                                  ;
    rts                           ; }
```

Remarquons que la lecture du bouton start est inutile lorsque select n'est pas appuyé. On pourrait donc l'éviter à l'aide d'un branchement.

13.2) On établit une connexion avec le port de manette; on ignore les six premiers boutons; puis on décrémente `posX:` si le bouton suivant est enfoncé:

```
avancer:                          ; avancer()
    ; Demander lecture boutons    ; {
    lda    #1                     ;   envoyer 1
    sta    $4016                  ;     au port de manette 1
    lda    #0                     ;   envoyer 0
    sta    $4016                  ;     au port de manette 1
                                  ;
    ; Ignorer six 1er boutons     ;
    ldx    #0                     ;   x = 0
                                  ;
```

```
avancer_lire:                   ;   do {
    lda    $4016                ;     lire/ignorer bouton
    inx                         ;     x++
    cpx    #6                   ;   }
    bne    avancer_lire         ;   while (x != 6)
                                ;
    ; Déplacer selon flèche     ;
    lda    $4016                ;
    and    #%00000001           ;   a = bit d'état de <--
    cmp    #0                   ;
    beq    avancer_fin          ;   if (a != 0)
    dec    posX                 ;     posX--
avancer_fin:                    ;
    rts                         ; }
```

13.3) En indiquant l'octet de poids fort 03_{16} au bus de données:

```
    lda     #$03                ; envoyer 0x03
    sta     $4014               ; au registre d'E/S 0x4014
```

13.4) Si la direction indique la *gauche*, alors on éteint le bit 6 de l'octet 2 de la tuile, et sinon on l'allume:

```
dir:      .rs 1 ; vaut 0 (gauche) ou 1 (droite)
tuile:    .rs 4 ; (posY, identifiant, attributs, posX)

renverser:                      ; renverser()
    ldx   #2                    ; {
    lda   dir                   ;
    cmp   #0                    ;
    bne   renverser_un          ;   if (dir == 0)
renverser_zero:                 ;   {
    lda   tuile, x              ;
    and   #%10111111            ;     a = tuile[2] & 0xBF
    jmp   renverser_fin         ;   }
renverser_un:                   ;   else
    lda   tuile, x              ;   {
    ora   #%01000000            ;     a = tuile[2] | 0x40
renverser_fin:                  ;   }
    sta   tuile, x              ;   tuile[2] = a
    rts                         ; }
```

Autre solution (inutilement compliquée) qui illustre d'autres concepts:

```
dir:      .rs 1 ; vaut 0 (gauche) ou 1 (droite)
tuile:    .rs 4 ; (posY, identifiant, attributs, posX)

renverser:                          ; renverser()
    ; Créer masque (dir <<= 6)      ; {
    ldx   0                         ;   x = 0
renverser_decal_gauche:             ;   do {
    asl   dir                       ;     dir <<= 1
    inx                             ;     x++
    cpx   #6                        ;   }
    bne   renverser_decal_gauche    ;   while (x != 6)
                                    ;
    ; Renverser tuile selon dir     ;
    ldx   #2                        ;
    lda   tuile, x                  ;   a  = tuile[2]
    and   #%10111111                ;   a &= 0xBF
    ora   dir                       ;   a |= dir
    sta   tuile, x                  ;   tuile[2] = a
                                    ;
    ; Restaurer dir (dir >>= 6)     ;
    ldx   0                         ;   x = 0
renverser_decal_droite:             ;   do {
    lsr   dir                       ;     dir >>= 1
    inx                             ;     x++
    cpx   #6                        ;   }
    bne   renverser_decal_droite    ;   while (x != 6)
                                    ;
    rts                             ; }
```

Chapitre 14

14.1)

```
attendre_start:             ; attendre_start()
    lda     #1              ; {
    sta     $4016           ;   do {
    lda     #0              ;
    sta     $4016           ;     demander lecture boutons
                            ;
    lda     $4016           ;     ignorer A
    lda     $4016           ;     ignorer B
    lda     $4016           ;     ignorer SELECT
    lda     $4016           ;
    and     #%00000001      ;     obtenir état a de START
    cmp     #0              ;   }
    beq     attendre_start  ;   while (a == 0)
                            ;
    jsr     faire_pause     ;   faire_pause()
    rts                     ; }
```

14.2)

```
    ; Lire mem_video[0x2000]
    lda     #$20
    sta     $2006
    lda     #$00
    sta     $2006
    lda     $2007
    tax

    ; Copier vers mem_video[0x2C00]
    lda     #$2C
    sta     $2006
    lda     #$00
    sta     $2006
    txa
    sta     $2007
```

Notons qu'on pourrait aussi utiliser la pile plutôt que le registre x afin de mettre le contenu de l'accumulateur a temporairement de côté.

14.3) Afin d'indiquer au processeur de restaurer le registre d'état (donc les codes de condition) avant de reprendre son exécution.

14.4)

```
    mov     x8, 93
    mov     x0, 0
    svc     0
```

B

Fiches récapitulatives

Les fiches des pages suivantes résument le contenu de chacun des chapitres. Elles peuvent être imprimées recto-verso, ou bien au recto seulement afin d'être découpées et pliées en deux. À l'ordinateur, il est possible de cliquer sur la plupart des puces « ▶ » pour accéder à la section du contenu correspondant.

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

- Se fait 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:* $\mathtt{x_0}$–$\mathtt{x_{30}}$ (64 bits) ou $\mathtt{w_0}$–$\mathtt{w_{30}}$ (sous-registres 32 bits)
- *Usage libre:* $\mathtt{x_0}$–$\mathtt{x_7}$ (arguments) et $\mathtt{x_{19}}$–$\mathtt{x_{28}}$ (sauveg. par l'appelé)
- *Usage semi-libre:* $\mathtt{x_9}$–$\mathtt{x_{15}}$ (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`	$\mathtt{x_d} \leftarrow v$	où v est regis. ou const.
`add xd, xn, v`	$\mathtt{x_d} \leftarrow \mathtt{x_n} + v$	où v est regis. ou const.
`mul xd, xn, xm`	$\mathtt{x_d} \leftarrow \mathtt{x_n} \cdot \mathtt{x_m}$	
`udiv xd, xn, xm`	$\mathtt{x_d} \leftarrow \mathtt{x_n} \div \mathtt{x_m}$	

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. 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	$\text{reg}[n] \leftarrow \text{reg}[n]+i$, suivi de $n, i \mapsto \text{mem}[\text{reg}[n]]$	`ldr x0, [x1, i]!`
indir. par reg. indexé post-incrémenté	$n, i \mapsto \text{mem}[\text{reg}[n]]$, suivi de $\text{reg}[n] \leftarrow \text{reg}[n]+i$	`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
```

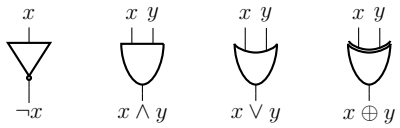
Assemblage

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

5. Circuits logiques

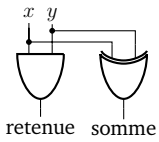
Circuits

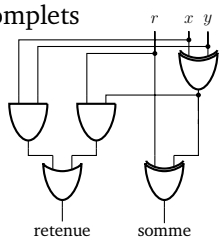
- « Blocs » de base constitués de portes logiques qui permettent d'implémenter l'ordinateur:



Arithmétique

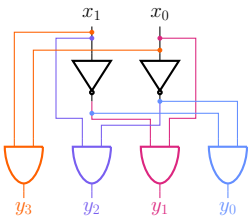
- *Demi-additionneur*: somme de deux bits
- *Additionneur complet*: somme de deux bits et d'une retenue
- *Addition*: somme sur n bits avec un demi-additionneur et une cascade de $n-1$ additionneurs complets

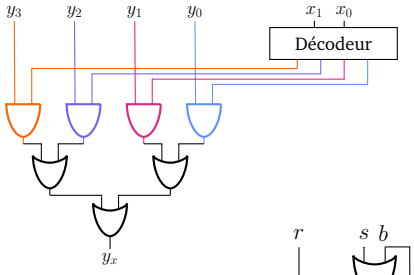




Décodage

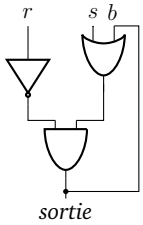
- *Décodeur*: sur entrée x, sortie: $y_x = 1$ et $y_j = 0$ pour $j \neq x$
- *Multiplexeur*: sur entrée x, sélectionne le bit y_x
- *Instructions*: décodables/exécutables à l'aide de tels circuits





Mémoire

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



6. 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:

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

- *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), `V` (débordement) — `C`: bit de retenue; `V`: 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` » ≡ $\texttt{r}_\texttt{d} \leftarrow \texttt{r}_\texttt{n} + \texttt{r}_\texttt{m} + \texttt{C}$

7. 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:

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

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

8. Programmation structurée

Séquence

- Composition séquentielle d'instructions
- Une instruction de haut niveau peut nécessiter plusieurs instructions de bas niveau; par ex. `x19 *= 7` devient:

```
mov     x20, 7
mul     x19, x19, x20
```

Sélection

- Exécution conditionnelle d'instructions (`if`, `switch`, ...)
- *Implémentation*: branchements avant:

```
if (cond(xd, xn)) {
  // code si
}
else {
  // code sinon
}
```

```
si:
        cmp         xd, xn
        b.¬cond     sinon
        // code si
        b           fin
sinon:
        // code sinon
fin:
```

- *Conditions multiples*: obtenues avec plusieurs sélections

Itération

- Exécution répétée d'instructions (`while`, `do while`, `for`, ...)
- *Implémentation*: branchements arrière, et parfois avant:

```
while (cond(xd, xn)) {
  // code
}
```

```
boucle:
        cmp         xd, xn
        b.¬cond     fin
        // code
        b           boucle
fin:
```

Sous-programmes

- Permettent de modulariser le code en sous-routines
- Registres partagés par programme et sous-programmes
- *Arguments*: passés par valeur ou adresse dans $\texttt{x}_0$–$\texttt{x}_7$ (en ordre)
- *Appel*: « `bl sprog` » assigne $\texttt{x}_{30} \leftarrow \texttt{pc}+4$ et branche à `sprog:`
- *Retour*: « `ret` » branche vers l'adresse de retour $\texttt{x}_{30}$
- *Sauvegarde*: l'appelé doit rétablir les registres $\texttt{x}_{19}$ à $\texttt{x}_{30}$

9. Valeurs booléennes et chaînes de bits

Valeurs booléennes

- *Correspond à* un bit: $1 = \textit{vrai}$, $0 = \textit{faux}$
- *Représentation*: sur un octet, puisque bits non adressables

Opérateurs logiques

- *Opérations*: $\neg$, $\wedge$, $\vee$, $\oplus$ « bit à bit » étendues aux chaînes:

`mvn x19, x20`

$\neg$	…	$\neg$	$\neg$	$\neg$
0	…	1	0	1
1	…	0	1	0

`and x19, x20, x21`

0	…	1	0	1
$\wedge$	…	$\wedge$	$\wedge$	$\wedge$
1	…	1	0	0
0	…	1	0	0

`orr x19, x20, x21`

0	…	1	0	1
$\vee$	…	$\vee$	$\vee$	$\vee$
1	…	1	0	0
1	…	1	0	1

`eor x19, x20, x21`

0	…	1	0	1
$\oplus$	…	$\oplus$	$\oplus$	$\oplus$
1	…	1	0	0
1	…	0	0	1

- *Échange de valeurs*: se fait sans registre temporaire avec `eor`

Décalages logiques et arithmétiques

- Décale les bits de j positions vers la gauche/droite:

11000101 $\xrightarrow{\text{3 bits vers la gauche}}$ 00101000 `lsl xd, xn, 3`

11000101 $\xrightarrow{\text{3 bits vers la droite}}$ 00011000 `lsr xd, xn, 3`

- Bit de signe copié lors d'un décalage arithmétique à droite:

11000101 $\xrightarrow{\text{3 bits vers la droite}}$ 11111000 `asr xd, xn, 3`

- *Multiplication/division*: par 2^k correspond à un décalage de k bits vers la gauche/droite

Décalages circulaires

- Comme un décalage logique, mais les bits « perdus » sont réinsérés de l'autre côté:

11000101 $\xrightarrow{\text{3 bits vers la gauche}}$ 00101110 n'existe pas sur ARMv8

11000101 $\xrightarrow{\text{3 bits vers la droite}}$ 10111000 `ror xd, xn, 3`

Masquage

- Permet d'isoler certains bits à manipuler:

sélection	$r \wedge m$	met à 0 les bits de r non spécifiés par m
activation	$r \vee m$	met à 1 les bits de r spécifiés par m
désactivation	$r \wedge \neg m$	met à 0 les bits de r spécifiés par m
basculement	$r \oplus m$	inverse les bits de r spécifiés par m

10. Chaînes de caractères

Généralités

- *Caractère*: symbole représenté par une chaîne de bits
- *Chaîne de caractères*: suite finie de caractères, normalement terminée par un caractère nul

ASCII

- Représente 128 caractères codés sur 7 bits
- Lettre minuscule mise en majuscule en assignant le $6^{\text{ème}}$ bit de poids faible à 0, par ex. $a = 1100001_2$ et $A = 1000001_2$

ISO 8859-1 (Latin-1)

- Représente 256 caractères codés sur 8 bits
- Caractères 0 à 127: ASCII
- Caractères 128 à 255: lettres accentuées et autres caractères

UTF-8

- Représente $> 1\,000\,000$ caractères sur 1 à 4 octets
- Caractères 0 à 127: ASCII
- Caractères 128 à 255: ISO 8859-1, mais codés différemment
- *Format général*:

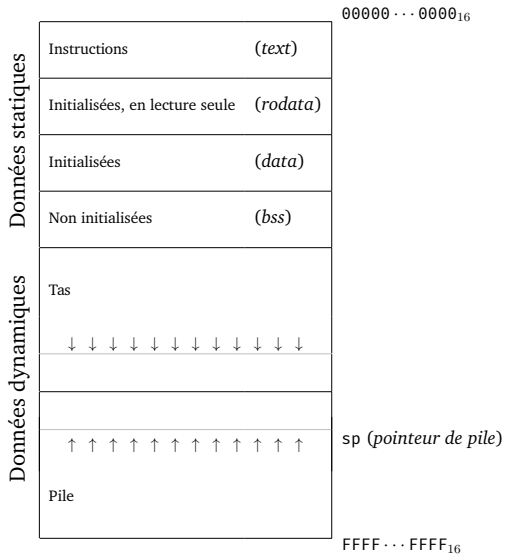
# bits	*plage de codes*		*format binaire des octets*			
	Début	Fin	Octet 1	Octet 2	Octet 3	Octet 4
7	000000_{16}	$00007F_{16}$	`0*******`	—	—	—
11	000080_{16}	$0007FF_{16}$	`110*****`	`10******`	—	—
16	000800_{16}	$00FFFF_{16}$	`1110****`	`10******`	`10******`	—
21	010000_{16}	$10FFFF_{16}$	`11110***`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

- *Exemples*:

car.	code	codage
a	1100001_2	01100001_2
é	$000\ 11101001_2$	$11000011\ 10101001_2$
ケ	$00110000\ 10110001_2$	$11100011\ 10000010\ 10110001_2$
𝌆	$00001\ 00100100\ 00001101_2$	$11110000\ 10010010\ 10010000\ 10001101_2$

11. Sous-programmes et mémoire

Disposition de la mémoire.



Tas.

- Contient les données allouées dynamiquement: structures de données, objets, etc.

Pile d'exécution.

- Stocke les données temporaires lors d'appel de sous-prog.
- Données empilées à l'appel et dépilées au retour
- *Pointeur de pile*: `sp` contient l'adresse du sommet de la pile
- *Empiler*: décrémenter `sp` + stocker avec `stp xd, xn, a`
- *Dépiler*: incrémenter `sp` + charger avec `ldp xd, xn, a`

Récursion.

- *Implémentée par*: appels de sous-prog. + usage de la pile
- *Récursion trop profonde*: erreur car la pile est bornée
- *Solution (partielle)*: empiler le moins de données possibles

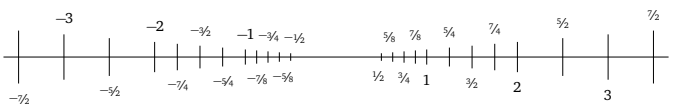
12. Nombres en virgule flottante

Représentation.

- *Nombre en virgule flottante:*

$$\underbrace{\pm}_{\text{signe}} \underbrace{d_0,d_1d_2\cdots d_{n-1}}_{\text{mantisse en base } \beta} \times \underbrace{\beta^{\overbrace{e}^{\text{exposant}}}}_{\text{base}}$$

- *Normalisé*: si $d_0 \neq 0$
- Représente différents ordres de grandeur:



Arithmétique.

- *Addition:* (1) mettre exposants en commun; (2) additionner mantisses; (3) normaliser; (4) arrondir
- *Multiplication:* (1) additionner exposants; (2) multiplier mantisses; (3) normaliser; (4) arrondir

Précision.

- *Approximations de nombres réels:*
  (a) arrondir (égalité: dernier chiffre pair): $1{,}9565 \to 1{,}956$
  (b) troncation: $1{,}5416 \to 1{,}541$
- *Erreur relative*: $\text{err}(x) \overset{\text{déf}}{=} \frac{x-\overline{x}}{x}$ où $\overline{x}$ est l'approximation
- *Borne pour mode (a)*: $|\text{err}(x)| \leq \underbrace{(\beta/2)\cdot\beta^{-n}}_{\varepsilon \text{ machine}}$

Norme IEEE 754.

-

format	*signe*	*exposant*	*mantisse*
simple	1 bit	8 bits	23 bits (+1 bit caché)
double	1 bit	11 bits	52 bits (+1 bit caché)

- *Repr. avec biais:* $1000011011100\cdots0 = -1{,}11 \times 2^{13-127}$
- ± 0 ($s0\cdots00\cdots0$); $\pm\infty$ ($s1\cdots10\cdots00$); NaN ($s1\cdots1e0\cdots01$)

ARMv8.

- *Registres*: d_n (64 bits) et s_n (32 bits)
- *Instructions*: `ldr`, `str`, `fmov`, `fcmp`, `fadd`, `fmul`, `fsqrt`, etc.

13. Introduction aux entrées/sorties : NES

Architecture.

- *Pas RISC*: possible de manipuler la mémoire directement
- *Processeurs:* proc. principal + proc. d'images (*PPU*)
- *Mémoire principale:* primaire + registres d'E/S + programme
- *Mémoire vidéo:* stocke les tuiles et palettes de couleurs

Jeu d'instructions.

- *Registres:* `a` (accumulateur), `x` (index), `y` (index), `s` (pile)
- *Valeurs imm.:* `#` (numérique), `$` (hexadécimal), `%` (binaire)
- *Accès mémoire:* `lda`, `ldx`, `ldy` (chargement d'octet); `sta`, `stx`, `sty` (stockage d'octet); `txa`, `tax`, `tya`, etc. (copie)
- *Arithmétique:* `adc` (addition avec report); `sbc` (soustraction avec emprunt); `inc`, `inx`, `iny`, `dec` (inc/décrémentation)
- *Logique:* `asl` (<< 1), `lsr` (>> 1), `and` ($\wedge$), `ora` ($\vee$), `eor` ($\oplus$)
- *Contrôle:* `cmp`, `cpx`, `cpy` (comparaison); `beq`, `bne` (branch. conditionnel), `jmp` (branch. incond.), `jsr`/`rts` (sous-prog.)

Tuiles.

- *Images:* constituées de tuiles de 8×8 pixels
- *Tuiles:* stockées dans la cartouche, transférées vers le PPU
- *Tuile:* spécifiée par 4 octets (y, i, a, x): position verticale y, numéro de tuile i, attributs a, position horizontale x
- *Attributs:* 8 bits pour réflexions, profondeur et couleurs

Sorties (graphiques).

- L'affichage se fait lors du rafraîchissement vertical
- *Sortie:* stocker tuiles de $0X00_{16}$ à $0XFF_{16}$ en mém. principale
- *Affichage:* transférer au PPU en écrivant `#$0X` à `$4014`

Entrées (manettes).

- *Entrée:* protocole de communication via port de manettes
- *Demande de lecture:* envoyer `#1`, puis `#0`, via `$4016`
- *Lecture:* lire bit de poids faible à `$4016` pour chaque bouton

14. Entrées/sorties

Mécanismes d'entrée/sortie.

- *Attente active:* interrogation continue d'un registre d'état jusqu'à un événement (ex. *VBLANK*)
- *Interruption:* signal lancé vers le processeur lors d'un événement (ex. NMI, RESET, IRQ)

Interruptions.

- *Gestionnaire:* sous-routine qui traite une interruption
- *Table d'interruptions:* contient l'adresse des gestionnaires
- *Traitement:* sauvegarder l'état du processeur; appeler le gestionnaire; restaurer l'état
- *Priorité:* valeur numérique assignée à une interruption
- *Gestion des priorités:* interruption ignorée si une interruption de priorité $>$ est en cours; gestionnaire en exécution mis en attente si une interruption de priorité $\geq$ est lancée
- *Non masquable:* top priorité, ne peut pas ignorer (ex. RESET)

Accès direct à la mémoire (DMA).

- *DMA:* permet au processeur d'initier un accès mémoire et de laisser un contrôleur effectuer le transfert de données
- *Sur le NES:* envoi des tuiles `mem[0x0200, 0x02FF]` vers la mémoire de *sprites* via DMA:

```
lda #$02
sta $4014
```

Appels système.

- *Accès E/S:* empêché par le système d'exploitation (sécurité)
- *Appel système:* service offert par le noyau du système d'exploitation; appelé via une interruption logicielle
- *Exemples UNIX + ARMv8:*

code	appel système
64	`write(flux, chaine, #octets)`
63	`read(flux, tampon, #octets)`

Flux d'entrée standard = 0
Flux de sortie standard = 1

```
// Afficher chaine
mov  x8, 64
mov  x0, 1
adr  x1, chaine
mov  x2, 10
svc  0
```

C

Architecture ARMv8 : sommaire

Cette annexe dresse un sommaire de l'architecture ARMv8 et plus particulièrement de son jeu d'instructions. L'annexe contient également quelques rappels utiles comme les formats d'entrée/sortie du langage C, et les commandes de débogage de GDB.

Registres. **[31 mars 2020]**

- Chaque registre $\mathtt{x}_n$ possède 64 bits: $b_{63}b_{62}\cdots b_1b_0$
- Notation: $\mathtt{x}_n\langle i\rangle \stackrel{\text{déf}}{=} b_i$, $\mathtt{x}_n\langle i,j\rangle \stackrel{\text{déf}}{=} b_ib_{i-1}\cdots b_j$, $\mathtt{r}_n$ 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 $\texttt{x}_\texttt{d}$ 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` `mov rd, i`	$\texttt{r}_\texttt{d} \leftarrow \texttt{r}_\texttt{m}$ $\texttt{r}_\texttt{d} \leftarrow i$	`mov x19, x21` `mov x19, 42`
`ldr`	`ldr xd, a` `ldr wd, a`	charge 8 octets: $\texttt{x}_\texttt{d}\langle 63,0\rangle \leftarrow \text{mem}_8[a]$ charge 4 octets: $\texttt{x}_\texttt{d}\langle 31,0\rangle \leftarrow \text{mem}_4[a]$; $\texttt{x}_\texttt{d}\langle 63,32\rangle \leftarrow 0$	`ldr x19, [x20]` `ldr w19, [x20]`
`ldrh`	`ldrh wd, a`	charge 2 octets: $\texttt{x}_\texttt{d}\langle 15,0\rangle \leftarrow \text{mem}_2[a]$; $\texttt{x}_\texttt{d}\langle 63,16\rangle \leftarrow 0$	`ldrh w19, [x20]`
`ldrb`	`ldrb wd, a`	charge 1 octet: $\texttt{x}_\texttt{d}\langle 7,0\rangle \leftarrow \text{mem}_1[a]$; $\texttt{x}_\texttt{d}\langle 63,8\rangle \leftarrow 0$	`ldrb w19, [x20]`
`str`	`str xd, a` `str wd, a`	stocke 8 octets: $\text{mem}_8[a] \leftarrow \texttt{x}_\texttt{d}\langle 63,0\rangle$ stocke 4 octets: $\text{mem}_4[a] \leftarrow \texttt{x}_\texttt{d}\langle 31,0\rangle$	`str x19, [x20]` `str w19, [x20]`
`strh`	`strh wd, a`	stocke 2 octets: $\text{mem}_2[a] \leftarrow \texttt{x}_\texttt{d}\langle 15,0\rangle$	`str w19, [x20]`
`strb`	`strb wd, a`	stocke 1 octet: $\text{mem}_1[a] \leftarrow \texttt{x}_\texttt{d}\langle 7,0\rangle$	`strb w19, [x20]`
`ldp`	`ldp xd, xn, a`	charge 16 octets: $\texttt{x}_\texttt{d}\langle 63,0\rangle \leftarrow \text{mem}_8[a]$, $\texttt{x}_\texttt{n}\langle 63,0\rangle \leftarrow \text{mem}_8[a+8]$	`ldp x19, x20, [sp]`
`stp`	`stp xd, xn, a`	stocke 16 octets: $\text{mem}_8[a] \leftarrow \texttt{x}_\texttt{d}\langle 63,0\rangle$, $\text{mem}_8[a+8] \leftarrow \texttt{x}_\texttt{n}\langle 63,0\rangle$	`stp x19, x20, [sp]`

Conditions de branchement.

- Codes de condition: `N` (négatif), `Z` (zéro), `C` (report), `V` (débordement)
- `C` indique aussi l'*absence* d'emprunt lors d'une soustraction
- Conditions de branchement:

Entiers non signés

Code	Signification	Codes de condition
`eq`	$=$	`Z`
`ne`	$\neq$	$\neg$`Z`
`hs`	$\geq$	`C`
`hi`	$>$	`C` $\wedge \neg$`Z`
`ls`	$\leq$	$\neg$`C` $\vee$ `Z`
`lo`	$<$	$\neg$`C`

Entiers signés

Code	Signification	Codes de condition
`eq`	$=$	`Z`
`ne`	$\neq$	$\neg$`Z`
`ge`	$\geq$	`N` $=$ `V`
`gt`	$>$	$\neg$`Z` $\wedge$ (`N` $=$ `V`)
`le`	$\leq$	`Z` $\vee$ (`N` $\neq$ `V`)
`lt`	$<$	`N` $\neq$ `V`
`vs`	débordement	`V`
`vc`	pas de débordement	$\neg$`V`
`mi`	négatif	`N`
`pl`	non négatif	$\neg$`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 $\texttt{r}_\texttt{d} = 0$	`cbz x19 main100`
`cbnz`	`cbnz rd, etiq`	branche à `etiq:` si $\texttt{r}_\texttt{d} \neq 0$	`cbnz x19 main100`
`tbz`	`tbz rd, j, etiq`	branche à `etiq:` si $\texttt{r}_\texttt{d}\langle j\rangle = 0$	`tbz x19, 1, main100`
`tbnz`	`tbnz rd, j, etiq`	branche à `etiq:` si $\texttt{r}_\texttt{d}\langle j\rangle \neq 0$	`tbnz x19, 1, main100`
`bl`	`bl etiq`	branche à `etiq:` et $\texttt{x}_{30} \leftarrow \texttt{pc} + 4$	`bl printf`
`blr`	`blr xd`	branche à $\texttt{x}_\texttt{d}$ et $\texttt{x}_{30} \leftarrow \texttt{pc} + 4$	`blr x20`
`br`	`br xd`	branche à $\texttt{x}_\texttt{d}$	`br x20`
`ret`	`ret`	branche à $\texttt{x}_{30}$ (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	`ucvtf d8, x19`
			`ucvtf d8, w19`
			`ucvtf s8, x19`
			`ucvtf s8, w19`

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 32 bits	`%d`	entier décimal signé
	`%u`	entier décimal non signé
	`%X`	entier hexadécimal non signé
	`%f`	nombre en virgule flottante
Nombres sur 64 bits	`%ld`	entier décimal signé
	`%lu`	entier décimal non signé
	`%lX`	entier hexadécimal non signé
	`%lf`	nombre en virgule flottante
Caractères	`%c`	caractère (1 octet)
	`%s`	chaîne de caractères

Débogage avec GDB.

Commande	Effet
Commandes de base	
`gdb exec`	Charge l'exécutable `./exec` en mode débogage
`break etiq`	Ajoute un point d'interruption à l'étiquette `etiq:`
`run`	Débute l'exécution en mode débogage
`continue`	Continue l'exécution jusqu'au prochain point d'interruption
`stepi`	Exécute la prochaine instruction
`nexti`	Exécute la prochaine instruction (sans entrer dans les sous-programmes)
`info reg`	Affiche le contenu des registres
`x &etiq`	Affiche le contenu de la mémoire à l'adresse associée à l'étiquette `etiq:`
`quit`	Quitter le débogueur
Commandes avancées	
`run < fichier`	Débute l'exécution en mode débogage avec l'entrée contenue dans `fichier`
`p/s $xd`	Affiche le contenu du registre dans le format `s` parmi l'un de ces choix: `u` = entier non signé, `d` = entier signé, `x` = valeur hexadécimale, `t` = valeur binaire, `f` = nombre en virgule flottante, `c` = caractère. Par exemple, `p/t $x19` affiche le contenu du registre x_{19} en binaire
`p/s var`	Affiche le contenu de la variable `var` dans le format `s`
`set var = val`	Assigne la valeur `val` à `var`; ce-dernier peut être un registre `$xd` ou une variable
`x 0xABCDFE`	Affiche le contenu de la mémoire à l'adresse hexadécimale `ABCDEF`
`x/nsu adr`	Affiche le contenu de `n` unités de mémoire à partir de l'adresse `adr` dans le format `s`. L'unité de mémoire est défini par l'un des choix suivants de `u`: `b` = octet, `h` = demi-mot, `w` = mot, `g` = double mot. Par exemple, `x/10ug &tab` affiche les 10 premiers éléments de 64 bits non signés d'un tableau `tab`

D

Architecture du NES : sommaire

Cette annexe dresse un sommaire de l'architecture du NES et plus particulièrement de son jeu d'instructions.

Registres. [31 mars 2020]

- 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 $0100_{16} + \mathsf{s}$)

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 $\leftarrow$ a $+ i +$ *report*	`lda #1`
	`adc adr`	a $\leftarrow$ a $+$ mem$_1[adr] +$ *report*	`adc var`
`sbc`	`sbc #i`	a $\leftarrow$ a $- i -$ *emprunt*	`sbc #1`
	`sbc adr`	a $\leftarrow$ a $-$ mem$_1[adr] -$ *emprunt*	`sbc var`
`clc`	`clc`	*report* $\leftarrow 0$ (utile avant `adc`)	`clc`
`sec`	`sec`	*emprunt* $\leftarrow 0$ (utile avant `sbc`)	`sec`
`inx`	`inx`	x $\leftarrow$ x $+ 1$	`inx`
`iny`	`iny`	y $\leftarrow$ y $+ 1$	`iny`
`inc`	`inc adr`	mem$_1[adr] \leftarrow$ mem$_1[adr] + 1$	`inc var`
`dec`	`dec adr`	mem$_1[adr] \leftarrow$ 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 $\leftarrow$ a $\wedge\, i$	`and #%00100011`
	`and adr`	a $\leftarrow$ a $\wedge$ mem$_1[adr]$	`and var`
`ora`	`ora #i`	a $\leftarrow$ a $\vee\, i$	`ora #%00100011`
	`ora adr`	a $\leftarrow$ a $\vee$ mem$_1[adr]$	`ora var`
`eor`	`eor #i`	a $\leftarrow$ a $\oplus\, i$	`eor #%00100011`
	`eor adr`	a $\leftarrow$ a $\oplus$ 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`

Bibliographie

[ARM13] ARM Limited. *Procedure Call Standard for the ARM 64-bit Architecture (AArch64)*, 2013. Version 1.0.

[ARM15] ARM Limited. *ARM® Cortex®-A Series: Programmer's Guide for ARMv8-A*, 2015. Version 1.0.

[ARM18] ARM Limited. *ARM® Architecture Reference Manual: ARMv8, for ARMv8-A architecture profile*, 2018. Version D.a.

[PH17] David Patterson and John Hennessy. *Computer Organization and Design RISC-V Edition*. Elsevier, 2017.

[SD11] Richard St-Denis. *L'architecture du processeur SPARC et sa programmation en langage d'assemblage*. Éditions GGC, 2011.

[Yer03] F. Yergeau. UTF-8, a transformation format of ISO 10646. RFC 3629, 2003.

Index

accès direct à la mémoire, 136
accès mémoire, 13, 58
addition, 7, 42, 51, 110
`adr`, 39
adressage, 13, 36
adresse, 35
    de retour, 77
    numérique, 35
    symbolique, 35
affichage, 26, 125
algèbre de Boole, 81
alignement, 15
allocation, 63
appel, 76
appel système, 137
architecture, 11
    de von Neumann, 11
argument, 76
arithmétique, 110
ARM, 23
arrondi, 108
ASCII, 92
assembleur, 40
assignation
    par sélection, 79
attente active, 131

bidimensionnel, 63
big-endian, 14
binaire, 4
bit de signe, 49
bits, 81
borne, 60
boucle, 74

caractère, 92
changement de base, 5
chaîne de caractères, 92
circuit
    combinatoire, 47
    séquentiel, 47
    à verrouillage, 47
circuit logique, 42
CISC, 20
code de condition, 57
commentaire, 30
commutativité, 83
complément à deux, 49
compteur d'instruction, 19
conjonction, 81
contraintes d'alignement, 15
conversion, 5
cryptographie, 84

demi-additionneur, 42
Dijkstra, Edsger, 71
dimension, 60
disjonction, 81
division, 56
DMA, 136
double, 112
décalage arithmétique, 86

décalage circulaire, 85
décalage logique, 84
dépiler, 102

E/S, 131
éditeur de liens, 40
EDVAC, 11
empiler, 101
ENIAC, 11
entiers, 49
entiers négatifs, 49
entiers signés, 49
entrée/sortie, 119, 131
ET logique, 81
étiquette, 30
exaoctet, 16
exbioctet, 16

fichier objet, 40
fraction, 8

George Boole, 81
gestionnaire d'interruption, 132
gibioctet, 16
gigaoctet, 16
granularité, 13
gros-boutiste, 14

hexadécimal, 4

IEEE 754, 112
implication, 81
implémentation, 11
index, 61
indice, 60
initialisation, 63
interruption, 132
    logicielle, 136
ISO 8859-1, 93
itération, 74

jeu de la vie, 70

kibioctet, 16
kilooctet, 16

langage d'assemblage, 23
Latin-1, 93
`ldr`, 39
`ldrb`, 39
`ldrh`, 39
lecture, 26
ligne de code, 30
big-endian, 14

manette, 126
manipulation de bits, 81, 82
masquage, 87
masque jetable, 84
matrice, 60
mode d'adressage, 36
MOS6502, 119
`mov`, 39
multiplication, 53, 111
mébioctet, 16
mégaoctet, 16
mémoire, 99
    principale, 12
    vive, 12

NaN, 114
NES, 119
nombre en virgule flottante, 107
nombre fractionnaire, 8
norme IEEE 754, 112
normes de programmation, 30
négation, 81

octal, 4
organisation, 11, 20
OU
    exclusif, 81
    logique, 81

paramètre, 76
parcours de tableau, 64
passage
    par adresse, 77
    par valeur, 77
petit-boutiste, 14
pile d'exécution, 99
pipeline, 20
pointeur, 67

porte logique, 42
`printf`, 26, 32
priorité, 135
processeur, 17
program counter, 19
programmation
    impérative, 71
    structurée, 71
précision, 108
    double, 112
    simple, 112
prédiction de branchement, 20
Puissance 4, 80
pébioctet, 16
pétaoctet, 16

registre, 17, 23
représentation, 107
restauration, 78, 102
retour, 77
RISC, 20
récursion, 102

sauvegarde, 78, 101
`scanf`, 26, 32
segment de données, 32
simple, 112
sous-programme, 76, 99
sous-routine, 76
soustraction, 53
spécificateur de format, 32
spécification, 11
`stdio`, 27
`str`, 39
`strb`, 39
`strh`, 39
structure de contrôle, 71
switch, 72
systèmes de numération, 1
    binaire, 4
    décimal, 2
    hexadécimal, 4
    notation positionnelle, 2
    notation unaire, 1
    numération romaine, 2
    octal, 4
    unaire, 1
sélection, 72
séquence, 71
    de Collatz, 24

tableau, 60
taille, 16
tas, 99
troncation, 108
tuile, 124
type, 60

UAL, 19
Unicode, 95
unidimensionnel, 62
unité arithmétique et logique, 19
unité de contrôle, 19
UTF-16, 95
UTF-32, 95
UTF-8, 95

valeur de retour, 77
verrou, 47
von Neumann, 11

zéro non significatif, 2, 8

équivalence, 81