IFT209 – Programmation système

NOTES DE COURS

Michael Blondin

11 avril 2019

Ce document

La rédaction de ce document a été entamée à la session d'hiver 2019 de l'Université de Sherbrooke comme notes complémentaires du cours IFT209 – Programmation système. La structure et le contenu de ces notes sont basées sur le plan cadre du cours IFT209, et sont par conséquent *fortement* inspirés du manuel de référence de Richard St-Denis: *L'architecture du processeur SPARC et sa programmation en langage d'assemblage* [SD11], ainsi que des diaporamas de Vincent Ducharme et Mikaël Fortin.

La rédaction de ce document est motivée par deux raisons:

— les notes sont gratuites et faciles d'accès pour les étudiant·e·s;

— l'architecture SPARC a été délaissée pour d'autres architectures; ces notes pourront donc évoluer en fonction des nouvelles architectures et diverger progressivement de [SD11].

Ces notes sont *seulement* destinées à l'enseignement et l'étude du cours IFT209 de l'Université de Sherbrooke.

Si vous trouvez des coquilles ou des erreurs dans le document, veuillez s.v.p. me les indiquer par courriel à `michael.blondin@usherbrooke.ca`.

Table des matières

1

Systèmes de numération

1.1 Représentation des nombres

1.1.1 Système unaire

Dans le *système unaire*, chaque nombre $n \in \mathbb{N}$ est représenté en répétant n fois un même symbole σ. 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 souvent utilisé puisqu'il préserve quelques propriétés de la notation positionnelle que nous verrons plus tard:

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

Nous utilisons parfois le système unaire lorsque nous comptons, par exemple, avec les doigts, des traits ou bien des bâtonnets. Le système unaire est aussi en quelque sorte utilisé dans l'arithmétique de Peano, où chaque nombre est ou bien le symbole 0, ou bien le successeur d'un autre nombre:

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

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

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

1.1.2 Système de numération romain

L'une des lacunes du système unaire est son manque de concision: la représentation d'un nombre $n \in \mathbb{N}$ requiert n symboles. La *numération romaine* est quant à elle plus concise. Par exemple, IIIII s'abrège par V, et VV s'abrège 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 bien plus complexes que dans le système unaire; par exemple, essayez de trouver un algorithme pour l'addition!

1.1.3 Système de numération positionnelle

Contrairement au système romain, le système de numération que vous utilisez probablement chaque jour, le *système décimal*, fait partie de la famille des *systèmes de numération positionnelle*. Dans ce système, 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. De façon générale, un nombre est une séquence non vide de chiffres de la forme $s_{n-1} \cdots s_1 s_0 \in \{0, 1, \ldots, B-1\}^n$. La valeur d'une telle séquence est définie par:

$$(s_{n-1} \cdots s_1 s_0)_B \stackrel{\text{def}}{=} \sum_{i=0}^{n-1} s_i \cdot B^i.$$

Par exemple, dans le système décimal:

$$564_{10} = 5 \cdot 10^2 + 6 \cdot 10^1 + 4 \cdot 10^0 = 564.$$

Si la base est plutôt $B = 7$, nous obtenons:

$$564_7 = 5 \cdot 7^2 + 6 \cdot 7^1 + 4 \cdot 7^0 = 291.$$

Les 21 premiers nombres naturels écrits en bases 10, 2, 4 et 8 sont énumérés à la figure 1.1. Notons que l'ajout du chiffre 0 à gauche d'un nombre ne change pas sa valeur. Nous disons que de tels zéros sont *non significatifs*.

Il assez simple de se convaincre 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 1. *Soient $B \in \mathbb{N}_{\geq 2}$ et $n \in \mathbb{N}$. Le plus grand nombre pouvant être représenté en base B avec n chiffres, dénoté $m_{B,n}$, est $B^n - 1$.*

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, 8 et 16.

Démonstration. Nous procédons par induction sur n. Si $n = 1$, alors le plus grand nombre est la valeur du plus grand chiffre, c.-à-d. $B - 1$. Nous avons donc bien $m_{B,n} = B - 1 = B^1 - 1$.

Supposons que $n > 1$ et $m_{B,n} = B^n - 1$. Le plus grand nombre formé de $n + 1$ chiffres est obtenu en concaténant $(B - 1)$ au plus grand nombre formé de n chiffres [1]. Ainsi:

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

□

La proposition 1 montre que le système de numération positionnelle est exponentiellement plus concis que le système unaire: chaque nombre peut être représenté avec une quantité logarithmique de symboles.

1. Il faudrait aussi prouver cette affirmation, mais nous la prenons pour acquise.

1.1.3.1 Système binaire

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. Le système binaire est 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 de 0 à $2^n - 1$.

1.1.3.2 Système hexadécimal

Le *système hexadécimal* est également répandu en informatique, par exemple pour représenter des séquences de bits de façon plus succincte, par ex. les codes de couleur. Ce système est l'instance du système de numération positionnelle où $B = 16$. Afin d'éviter toute ambiguïté, les chiffres $10, 11, \ldots, 15$ sont remplacés par $A, B, \ldots, F$ respectivement. Par exemple:

$$8B5_{16} = 8 \cdot 16^2 + 11 \cdot 16^1 + 5 \cdot 16^0 = 2229.$$

1.1.3.3 Système octal

Le *système octal* est aussi parfois utilisé en informatique. Il est l'instance du système de numération positionnelle où $B = 8$.

1.2 Conversions entre systèmes de numération

Pour le reste de cette section, fixons deux nombres $B, B' \in \mathbb{N}_{\geq 2}$ tels que $B \neq B'$. Nous expliquons comment convertir algorithmiquement des nombres de la base d'un système de numération positionnelle vers une autre base.

1.2.1 Base B vers base 10

Soit $s = s_{n-1} \cdots s_1 s_0$ un nombre en base B de n chiffres. Il est possible de convertir s en base 10 en calculant simplement:

$$\sum_{i=0}^{n-1} s_i \cdot B^i.$$

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:

$$s_0 + B \cdot (s_1 + B \cdot (s_2 + B \cdot (\ldots + B \cdot s_{n-1}))).$$

Par exemple, le nombre binaire 10110 donne la valeur suivante en décimal:

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

1.2.2 Base 10 vers base B

Soit s un nombre en base 10. Il est possible de convertir s en base B en divisant itéartivement s par B jusqu'à obtenir 0. Le *reste* de chaque division entière donne un chiffre du nombre résultant (de droite à gauche). Par exemple, 22 est égal à 10110 en base 2, ce qui peut être obtenu grâce aux calculs suivants:

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

L'algorithme général est décrit à l'algorithme 1.

Algorithme 1 : Algorithme pour convertir un nombre décimal vers une base $B \in \mathbb{N}_{\geq 2}$.

Entrées : base $B \in \mathbb{N}_{\geq 2}$ et un nombre s écrit en base 10
Sorties : nombre correspondant à s écrit en base B
$t \leftarrow \varepsilon$ `// ε dénote la séquence vide`
faire
  $t \leftarrow (s \bmod B) \cdot t$ `// · dénote la concaténation`
  $s \leftarrow s \div B$
tant que $s \neq 0$
retourner t

1.2.3 Base B vers base B'

Afin de convertir un nombre de la base B vers la base B', il suffit de procéder en deux étapes: passer de la base B vers la base 10, puis de la base 10 vers la base B'.

1.2.4 Base B vers base B^m

Soient $m \in \mathbb{N}_{>1}$ et s un nombre en base B. Afin de convertir s en base B^m, nous procédons en deux étapes:

— les chiffres de s sont regroupés en blocs de taille m, de la droite vers la gauche, en ajoutant des 0 non significatifs tout à gauche de s au besoin;
— chaque bloc est remplacé par le chiffre correspondant en base B^m.

Par exemple, le nombre binaire 1010110001 est converti en nombre hexadécimal de la façon suivante:

$$1010110001 \longrightarrow \mathbf{00}10 \quad 1011 \quad 0001 \longrightarrow 2B1.$$

1.2.5 Base B^m vers base B

Soient $m \in \mathbb{N}_{>1}$ et s un nombre en base B^m. Afin de convertir s en base B, nous *éclatons* chaque chiffre de s vers sa représentation de taille m en base B, puis nous effaçons les 0 non significatifs à gauche. Par exemple, le nombre hexadécimal $2B1$ est converti en nombre binaire de la façon suivante:

$$2B1 \longrightarrow 0010 \quad 1011 \quad 0001 \longrightarrow 1010110001.$$

1.2.6 Base B^m vers base B^k

Afin de convertir un nombre de la base B^m vers la base B^k, où $m \neq k$, il suffit de procéder en deux étapes: passer de la base B^m vers la base B, puis de la base B vers la base B^k.

1.3 Addition

Soit $B \in \mathbb{N}_{\geq 2}$ une base, et $s_{n-1} \cdots s_1 s_0, t_{n-1} \cdots t_1 t_0 \in \{0, 1, \ldots, B-1\}^n$ deux séquences de taille n. La somme de s et t en base B peut être effectuée de façon analogue à l'addition en base 10. L'addition se fait itérativement, de droite à gauche, en faisant la somme des chiffres en base B et en propageant une retenue. Par exemple, l'addition $\mathrm{D40C}_{16} + \mathrm{6FA5}_{16} = \mathrm{143B1}_{16}$ se calcule selon les cinq étapes suivantes:

$$\begin{array}{r} \scriptstyle 1 \\ +\ \mathrm{D\,4\,0\,C} \\ \mathrm{6\,F\,A\,5} \\ \hline 1 \end{array} \qquad \begin{array}{r} \scriptstyle 1 \\ +\ \mathrm{D\,4\,0\,C} \\ \mathrm{6\,F\,A\,5} \\ \hline \mathrm{B\,1} \end{array} \qquad \begin{array}{r} \scriptstyle 1\ \ 1 \\ +\ \mathrm{D\,4\,0\,C} \\ \mathrm{6\,F\,A\,5} \\ \hline \mathrm{3\,B\,1} \end{array} \qquad \begin{array}{r} \scriptstyle 1\,1\ \ 1 \\ +\ \mathrm{D\,4\,0\,C} \\ \mathrm{6\,F\,A\,5} \\ \hline \mathrm{4\,3\,B\,1} \end{array} \qquad \begin{array}{r} \scriptstyle 1\,1\,1 \\ +\ \mathrm{D\,4\,0\,C} \\ \mathrm{6\,F\,A\,5} \\ \hline \mathrm{1\,4\,3\,B\,1} \end{array}$$

L'addition peut aussi être effectuée entre deux nombres de taille différente; autrement dit dont le nombre de chiffres diffère. Pour ce faire, il suffit 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.

1.4 Fractions

Le système de numération positionnelle peut être étendu naturellement afin de représenter les fractions. Soient $B \in \mathbb{N}_{\geq 2}$ une base, et $s_{n-1} \cdots s_1 s_0 \in \{0, 1, \ldots, B-1\}^n$ et $t_1 t_2 \cdots t_k \in \{0, 1, \ldots, B-1\}^k$ des séquences de taille n et k respectivement. La valeur du nombre s,t selon la base B est définie par:

$$(s, t)_B \stackrel{\text{def}}{=} \sum_{i=0}^{n-1} s_i \cdot B^i + \sum_{i=1}^{k} t_i \cdot B^{-i}.$$

Par exemple,

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

En base 10, cette notation correspond au système décimal habituel. Notons que l'ajout de zéros tout à droite d'un nombre ne change pas sa valeur. Notons également qu'une fraction, pouvant être représentée 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.

Les méthodes de conversion décrites à la section 1.2 peuvent être adaptées relativement facilement aux fractions. L'addition se fait telle que décrite pour les entiers à la section 1.3, en alignant les nombres à la virgule.

2

Architecture des ordinateurs

2.1 Architecture et organisation

Le fonctionnement d'un ordinateur 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 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, Intel et AMD développent tous deux des processeurs `x86-64`.

L'emphase de ce chapitre est sur l'architecture des ordinateurs.

2.2 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

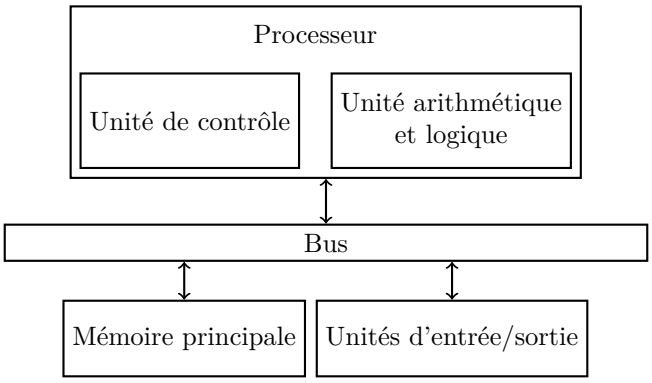


FIGURE 2.1 – Architecture de von Neumann. Figure reproduite à partir de [SD11, Fig. 2.1].

était utilisé pour calculer des tables de tir d'artillerie. L'ENIAC était programmable à l'aide de câbles et d'interrupteurs, et 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.

2.2.1 Mémoire principale

La mémoire principale peut être vue comme une séquence $c_0, c_1, \ldots, c_{n-1}$ de n cellules tel qu'illustré à la figure 2.2. Chacune de ces cellules contient une donnée provenant d'un ensemble 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 un *octet* (8 bits), autrement dit $D = \{0,1\}^8$.

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 – Mémoire principale. Chaque celulle est représentée par une case rectangulaire dont l'adresse apparaît à gauche. Le contenu de chaque cellule est une suite de 8 bits (un octet). Le contenu illustré ici n'est qu'un exemple.

L'interpétation de ces bits est faite par le langage de programmation ou la programmeuse/le programmeur. Par exemple, l'octet 01000001 peut autant représenter le nombre 65 que le caractère `A`. Lors du chargement d'un programme, celui-ci est stocké dans les cellules de la mémoire, avec ses données (variables, constantes, etc.)

Granularité de l'accès mémoire. Bien qu'une adresse réfère à un octet, les architectures modernes permettent aussi d'interpréter une adresse comme faisant référence à plusieurs octets, souvent 2, 4 ou 8 octets. Dans l'architecture ARMv8, par exemple, 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 pouvant être adressés diffèrent d'une architecture à l'autre. À moins d'avis contraire, nous utiliserons la terminologie du tableau ci-haut, donc celle d'ARMv8.

Considérons la mémoire illustrée à la Figure 2.2. Le contenu de l'octet, le demi-mot, le mot et le double-mot 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.
```

En général, le contenu d'un octet, demi-mot, mot ou double mot 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	$[i, i+1, \ldots, i+7]$

Ordre des octets. L'interpétation des valeurs dépend de l'ordre dans lequel les octets sont organisés sur l'architecture en question. Considérons le mot $w =$ `00 58 40 0F` contenu à l'adresse 0. Autrement dit, $w = w_0w_1w_2w_3$ où $w_0 =$ `00`, $w_1 =$ `58`, $w_2 =$ `40` et $w_3 =$ `0F`.

Dans le format dit « *big-endian* », aussi appelé *gros-boutiste*, les octets sont organisés de gauche à droite, donc de w_0 vers w_3. Ainsi, la valeur hexadécimale du mot w dans le format « big-endian » est:

0058400F.

À l'inverse, dans le format dit « *little-endian* », aussi appelé *petit-boutiste*, les octets sont organisés de droite à gauche, donc de w_3 vers w_0. Ainsi, la valeur hexadécimale du mot w dans le format « little-endian » est:

0F405800.

L'architecture x86-64 utilise le format « little-endian » et l'architecture ARMv8 supporte les deux formats. Nous utiliserons le format « little-endian » lors de la programmation sur l'architecture ARMv8.

En général, le format utilisé n'est pas perceptible. Il peut néanmoins être observé lorsqu'une valeur est lue octet par octet, par exemple dans le code C suivant [1]:

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

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

1. Exemple basé sur un exemple des diaporamas de Vincent Ducharme.

```c
int main()
{
  union Mot mot;

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

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 divisibles par 2^k. Par exemple, sous ces contraintes, il est seulement possible d'adresser un mot aux adresses: $0, 4, 8, 12, \ldots$. Notons 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.

Sur plusieurs architectures sans contraintes d'alignement, bien que l'adressage de 2^k octets à une adresse non divisible par 2^k soit permis, cela ralentit l'accès mémoire (voir la figure 2.3).

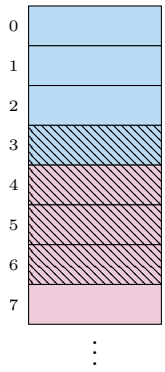


FIGURE 2.3 – L'adressage d'un mot à une adresse non alignée (zone hachurée) peut requérir deux accès mémoire (zones de couleur).

Taille de la mémoire. Les unités de mesure de la mémoire portent les noms suivants:

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

Dans le langage courant, nous utilisons souvent la terminologie {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 stockées sur k bits peut accéder jusqu'à 2^k cellules de mémoire. Ainsi, une architecture 32 bits peut accéder à un maximum de:

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

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

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

2.2.2 Processeur

Le *processeur* est l'unité centrale de traitement de l'ordinateur, autrement dit il est le « 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 il 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 exemple, 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, 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. Par exemple, l'instruction `add` des architectures ARMv8 et RISC-V permet d'additionner le contenu de deux registres et de stocker la somme dans un troisième registre. Par exemple, la somme du contenu des registres $\texttt{x}_{11}$ et $\texttt{x}_{12}$ peut être stockée dans le registre $\texttt{x}_{10}$ via:

```
add x10, x11, x12
```

Chaque instruction est de la forme suivante:

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

Dans l'exemple ci-haut, le code d'opération `add` est suivi des trois opérandes $\texttt{x}_{10}$, $\texttt{x}_{11}$ et $\texttt{x}_{12}$. Le nombre d'opérandes varie d'une instruction à l'autre, et dans certain cas il peut simplement n'y avoir aucun opérande; par exemple, dans le cas de l'instruction `nop`.

Il existe plusieurs autres instructions de différents types: arithmétique, logique, contrôle, accès mémoire, etc.

Chaque instruction se traduit en *code machine*: une suite de bits pouvant être interprétée par le processeur. Selon l'architecture, le nombre de bits requis afin de représenter une instruction en code machine peut varier selon l'instruction. Sur les architecture ARMv8 et RISC-V, *toutes* les instructions sont traduites vers une séquence de 32 bits. Par exemple, sur l'architecture ARMv8, « `add x10, x11, x12` » se traduit vers $\texttt{B0C016A}_{16}$, ou plus précisément en binaire:

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

Chaque instruction possède un format de bits bien précis. Par exemple, dans le cas de l'addition de deux registres, le tout premier bit indique si l'addition doit se faire en arithmétique 64 bits ou non, et la séquence « `0 0 01011` » est tout simplement fixée. 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 « `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 2.3.1, l'unité de contrôle fonctionne souvent en plusieurs étapes qui peuvent être parallélisées.

Notons que le compteur d'instruction n'est pas visible sur l'architecture ARMv8, mais qu'il est, par exemple, accessible sous le nom `pc` sur l'architecture RISC-V. Le processeur possède d'autres registres spéciaux que nous verrons plus tard.

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, OU, OU exclusif, négation, décalages, décalages circulaires, inversion, etc.

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

2.2.3 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.

2.2.4 Types d'architectures

Il existe deux grandes familles d'architectures modernes: *RISC* et *CISC*. Par exemple, les architectures ARMv8 et RISC-V sont de type RISC, et l'architecture x86-64 est de type CISC. Il n'existe pas de définition claire de ces deux types d'architectures, 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;

— d'instructions dont l'encodage en code machine est de *taille fixe*.

2.3 Organisation

2.3.1 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.

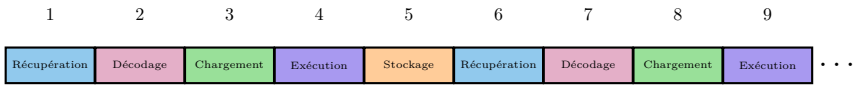


FIGURE 2.4 – Exécution séquentielle d'instructions en cinq étapes.

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 chaine de production.

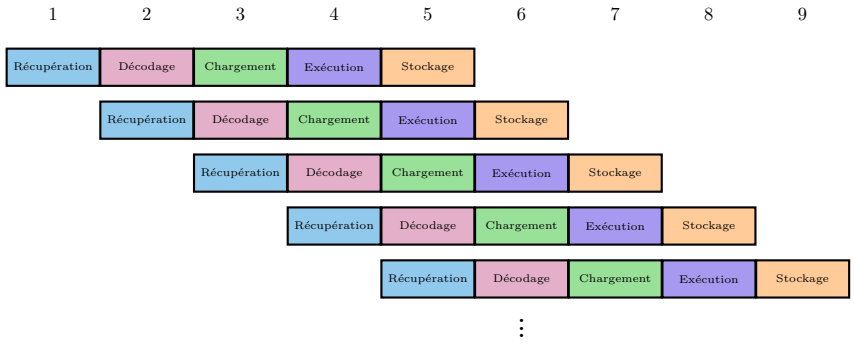


FIGURE 2.5 – Pipeline à cinq étages. L'exécution de n instructions prend (idéalement) $n+4$ cycles, plutôt que les $5n$ cycles d'une exécution purement séquentielle.

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 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. Pour pallier à ce problème, les processeurs utilisent différents heuristiques de prédiction de branchement.

3

Programmation en langage d'assemblage : ARMv8

L'architecture ARMv8-A, que nous dénotons simplement ARMv8, 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.

3.1 Registres

L'architecture ARMv8 possède 32 registres de 64 bits [ARM15, Sect. 9.1.1] dont l'usage est comme suit:

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*)

Pour tout $k \in \{0, 1, 2, \ldots, 30, \mathtt{zr}\}$, il existe une version 32 bits du registre $\mathtt{x}_k$ nommée $\mathtt{w}_k$. Le registre $\mathtt{w}_k$ correspond aux 32 bits de poids faible de $\mathtt{x}_k$. Autrement dit, si $\mathtt{x}_k = b_{63} \cdots b_1 b_0$, alors $\mathtt{w}_k = b_{31} \cdots b_1 b_0$.

3.2 Un premier programme

Afin d'explorer le jeu d'instruction 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 mathématique simple.

Soit $f\colon \mathbb{N}_{>0} \to \mathbb{N}_{>0}$ la fonction suivante:

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

Pour tout $n \in \mathbb{N}_{>0}$, la séquence s_n est définie par $f(n), f(f(n)), f(f(f(n))), \ldots$. La *conjecture de Collatz* affirme que pour tout $n \in \mathbb{N}_{>0}$, la séquence s_n atteint éventuellement 1. 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_6 atteint 1 en huit étapes:

$$s_6 = 3, 10, 5, 16, 8, 4, 2, 1, \ldots.$$

Pour tout $n \in \mathbb{N}_{>0}$ définissons e_n comme étant le nombre (minimal) d'étapes requis afin que la séquence s_n atteigne 1. Par exemple, $e_3 = 7$ et $e_6 = 8$. Nous allons écrire un programme qui, étant donné un nombre n, calcule e_n. Plutôt que d'écrire le programme directement, nous présentons d'abord du code accomplissant des sous-tâches, puis nous unifions et raffinons ces bouts de code.

3.2.1 Calcul de $f(n)$

Voyons d'abord comment calculer, étant donné un nombre n, la valeur $f(n)$. Nous devons d'abord choisir des registres avec lesquels travailler. Les registres $\mathtt{x}_{19}$–$\mathtt{x}_{28}$ sont préférables puisqu'ils peuvent être utilisés de façon quelconque. Supposons que $\mathtt{x}_{19}$ contienne initialement n. Le code suivant calcule $f(n)$ et stocke sa valeur dans $\mathtt{x}_{19}$:

```
        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-haut. Afin de calculer f, nous devons d'abord tester si n est pair. 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 branche à l'étiquette « `etiq:` », sinon elle ne fait rien. Ainsi, la première ligne du programme teste si n est impair; si c'est le cas, elle branche à l'étiquette « `impair:` »; sinon le programme passe à l'étiquette « `pair:` ».

Si n est pair, le programme calcule $n \div 2$ à l'aide des instructions suivantes:

`mov xd, i`	assigne la valeur i 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` » branche ensuite à la toute dernière ligne afin d'indiquer que le calcul de $f(n)$ est complété.

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

`mov xd, i`	assigne la valeur i au registre $\texttt{x}_\texttt{d}$
`mul xd, xn, xm`	stocke le produit de $\texttt{x}_\texttt{n}$ et $\texttt{x}_\texttt{m}$ dans $\texttt{x}_\texttt{d}$
`add xd, xn, i`	stocke la somme de $\texttt{x}_\texttt{n}$ et de la valeur i dans $\texttt{x}_\texttt{d}$

3.2.2 Affichage d'un registre

Maintenant que nous savons calculer $f(n)$, voyons comment afficher sa valeur, c'est-à-dire le contenu du registre $\texttt{x}_{19}$. 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"
```

Décortiquons ce code. L'affichage se fait grâce à la fonction `printf` de la librairie d'entrée/sortie du langage C. Le premier paramètre de cette fonction est l'adresse de la chaîne de caractères à afficher. Les constantes sont définies dans le *segment de données en lecture seule*: « `.section ".rodata"` ». Il est possible d'y déclarer une constante, de type `.t` portant le nom `etiq` et contenant la valeur `x`, grâce à l'instruction:

```
etiq:   .t   x
```

Le type « `.asciz` » corrrespond à une chaîne de caractères [1]

La chaîne `"Résultat: %lu"` contient le spécificateur de format `"%lu"` qui spécifie un entier de 64 bits. Le deuxième argument de `printf` est donc, dans ce cas, un entier à afficher.

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

Tel que 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:

```
        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"
```

Décortiquons ce code. Afin d'effectuer l'affichage, nous utilisons la fonction `scanf` de la librairie d'entrée/sortie du langage C. Le premier paramètre de cette fonction est l'adresse d'une variable dans laquelle la valeur lue doit être stockée. Le second paramètre est l'adresse du format de la valeur à lire. Nous déclarons une variable `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` lit ici un entier de 64 bits et stocke sa valeur dans une variable temporaire. Afin de transférer le contenu de cette variable vers x_{19}, nous utilisons le code suivant:

```
        ldr     x19, temp
```

Cette instruction charge le contenu stockée à l'adresse contenue dans x_{19}:

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

3.2.4 Vers un premier programme

Jusqu'ici, nous avons vu comment lire un entier n, calculer $f(n)$ et afficher $f(n)$. 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 e_n.

3.2.5 Calcul de e_n à partir de la séquence s_n

Afin de calculer e_n, nous générons la séquence s_n itérativement, et 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: x21 += 1
        // code ici             // calcul de f(n)
        b       boucle          // aller à boucle
finboucle:
```

Décortiquons le code ci-haut. La première instruction initialise à zéro un compteur qui sera incrémenté afin de calculer e_n. Rappelons que $\texttt{x}_{19}$ contient n. L'instruction « `cmp x19, 1` » compare le contenu de $\texttt{x}_{19}$ et la valeur 1. L'instruction « `b.eq fin` » branche à l'étiquette « `finboucle:` » si la comparaison précédente résulte en une égalité. Autrement dit, si $\texttt{x}_{19} = 1$, alors nous avons terminé de calculer e_n. Si $\texttt{x}_{19} \neq 1$, alors le compteur $\texttt{x}_{21}$ est incrémenté, $\texttt{x}_{19}$ est remplacé par $f(\texttt{x}_{19})$, et le programme branche vers le début de la boucle.

3.2.6 Le programme complet

Nous pouvons finalement réunir les bouts de code mis au point jusqu'ici. Afin d'exécuter notre programme, nous devons lui donner un point d'entrée (ici « `main:` ») et quitter avec l'instruction « `bl exit` ». De plus, plutôt que d'afficher $\texttt{x}_{19}$, nous devons afficher $\texttt{x}_{21}$ qui contient e_n. Le programme complet est comme suit:

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

        // 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 de f(n)
        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"
```

3.3 Quelques 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    operandes    // 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 arguments de l'instruction, et le *commentaire* donne des notes sur le code destinées à l'humain (plutôt qu'à 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 et commentaires. 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, en pseudo-code ou dans un langage de haut niveau, décrivant l'effet de l'instruction. Par exemple:

```
// Usage des registres:
//   x19 -- n
//   x20 -- tmp
        tbnz    x19, 0, impair  // si n est impair: 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 de faciliter la lecture.

Étiquettes. Dans les programmes contenant des dizaines de lignes de code, il est recommandé de nommer chaque étiquette par les 3 ou 4 premières lettres du point d'entrée, suivi d'un nombre plus grand que les étiquettes précédentes. Par exemple:

```
main:
        tbnz    x19, 0, impair  // if (n % 2 == 0) goto impair
main100:
        mov     x20, 2          // tmp = 2
```

```
        udiv    x19, x19, x20   // n /= tmp
        b       fin             // goto fin
main200:
        mov     x20, 3          // tmp = 3
        mul     x20, x20, x19   // tmp *= n
        add     x19, x20, 1     // n = tmp + 1
main300:
       // des dizaines d'autres lignes ici
```

Bien que moins représentatifs, ces noms d'étiquettes permettent de se repérer dans le code et d'identifier rapidement la direction dans laquelle les sauts sont effectués.

3.3.2 Segments de données

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

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

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

`.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` du langage C sont comme suit [2]:

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

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

4

Accès aux données

4.1 Adressage

4.1.1 Spécification d'une adresse

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, donc 0000001A dans notre exemple, plutôt que 26. Une *adresse symbolique* est une suite de caractères qui réfère à une cellule mémoire c_i dont nous ne connaissons (à priori) pas l'index i. En langage d'assemblage, les adresses symboliques se manifestent sous forme d'étiquettes. Par exemple, considérons le code suivant:

```
main:
       mov x19, 42
```

L'étiquette `main:` réfère à une adresse i telle que la cellule c_i contient le code machine associé à l'instruction `mov x19, 42`. Nous ne connaissons pas la valeur de i; elle sera déterminée plus tard pour nous.

4.1.2 Modes d'adressage

Comme nous l'avons vu au chapitre 2, 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.

Fixons i une valeur immédiate, n le nom d'un registre, et a une adresse. 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.

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

$$i \mapsto i.$$

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

```
mov x19, 42
```

Adressage 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 contenue dans mem[FF].

Adressage (direct) 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
```

Adresse 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 $\text{mem}[\text{FF}] = 1\text{A}$, alors l'adresse indirect récupére la valeur contenue dans mem[1A].

Adressage 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 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 mem[reg[20]] = mem[FF]:

```
ldr x19, [x20]
```

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

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

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

Adressage indirect par registre pré/post-incrémenté. Ces deux modes d'adressage similaires reçoivent un nom de registre n et une valeur immédiate i. Dans le cas du mode d'*adressage indirect par registre 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:

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

Notons que ces deux modes d'adresse 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 mem[FB], et le mode post-incrémenté récupère la valeur mem[FA]. Dans les deux cas, $\text{reg}[20] = \text{FB}$ à la fin de l'exécution.

Adressage 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 *pc*:

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

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

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

1. Notons que sur ARMv8, chaque instruction est encodée sur 4 octets, donc i doit être un multiple de 4.

Sommaire des modes d'adressage. Les modes d'adressage présentés sont répertoriés dans le tableau suivant:

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

4.1.3 Particularités de l'architecture ARMv8

Soient $d \in \{0, 1, \ldots, 31\}$ l'index d'un registre, et 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
```

Notons que 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.

4.2 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 UNIX, l'assemblage, l'édition des liens et l'exécution se font normalement 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
```

5

Nombres entiers

5.1 Circuits logiques

Le processeur, et en particulier son unité arithmétique et logique, sont implémentés sous forme de *circuits logiques*. Nous considérons une version simplifiée et idéalisée des circuits logiques afin de comprendre le fonctionnement de l'UAL. À l'aide de transistors, il est possible de construire des *portes (logiques)* calculant la négation ($\neg$), la conjonction ($\wedge$), la disjonction ($\vee$) et le OU exclusif ($\oplus$). Ces portes sont souvent illustrées comme à la figure 5.1.

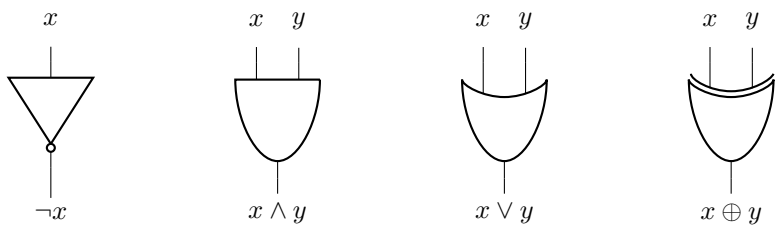


FIGURE 5.1 – De gauche à droite: porte NON, porte ET, porte OU, porte OU exclusif. Les bits d'entrée et de sortie sont situés respectivement au-dessus et au bas des portes.

Considérons la table d'addition de deux bits x et y:

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

En comparant la table ci-dessus avec celle ci-bas, nous remarquons que l'addition de deux bits peut être effectuée à l'aide d'une porte ET d'une porte OU exclusif:

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 logique pour l'addition de deux bits est illustré à la figure 5.2. Ce type de circuit se nomme un *demi-additionneur*.

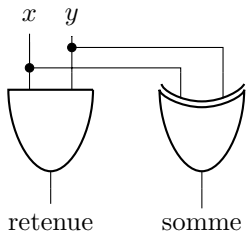


FIGURE 5.2 – Demi-additionneur.

Bien qu'un demi-additionneur permette d'additionner deux bits, il ne permet pas de tenir en compte une retenue provenant d'une addition précédente. Il est possible de mettre au point un *additionneur complet* qui prend également une 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

En comparant la table ci-dessus avec celle ci-bas, nous remarquons que l'addition $r + x + y$ peut être effectuée à l'aide de deux portes ET, d'une porte OU, et de deux portes 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

Le circuit logique de la figure 5.3 illustre un additionneur complet.

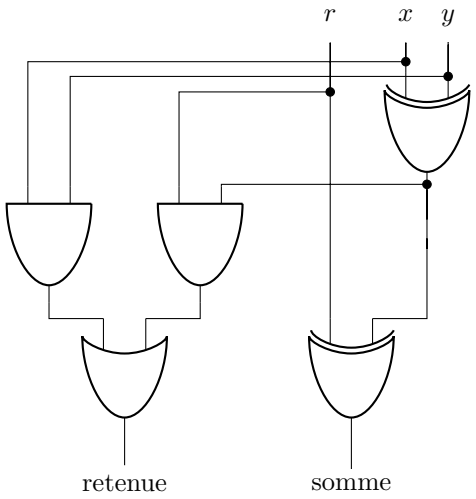


FIGURE 5.3 – Additionneur complet.

Il est possible d'obtenir un circuit logique qui effectue l'addition de deux nombres de n bits en composant un demi-additionneur et $n-1$ additionneurs tel qu'illustré à la figure 5.4.

D'autres opérations peuvent être implémentées par l'unité arithmétique et logique. Par exemple, un circuit qui teste si un nombre de 2^k bits est égal à zéro peut être obtenu grâce à une cascade de $2^k - 1$ portes OU connectée à une porte NON.

5.2 Représentation des entiers signés

Jusqu'ici, nous avons seulement considéré les entiers non négatifs. Afin de manipuler les entiers *signés*, c'est-à-dire non négatifs *et* négatifs, il faut d'abord choisir une façon de les représenter. Une façon simple de représenter un nombre

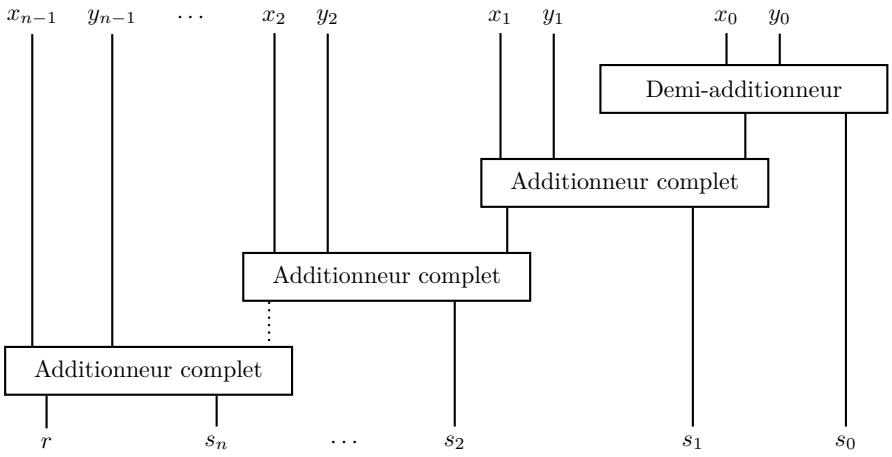


FIGURE 5.4 – Additionneur de deux nombres de n bits, construit à partir d'un demi-additionneur et de $n-1$ additionneurs complets.

n signé consiste à représenter sa valeur absolue $|n|$ ainsi qu'un bit de signe, c'est-à-dire un bit égal à 1 si et seulement si $n < 0$. Bien que cette représentation soit simple, elle complique l'implémentation des opérations arithmétiques. Par exemple, pour l'addition de deux nombres, il faut considérer les quatre valeurs possibles des deux bits de signe.

5.2.1 Complément à deux

Une solution plus élégante consiste à utiliser le *complément à deux*. Dans cette représentation, la valeur d'une chaîne de bits $b_{n-1} \cdots b_1 b_0$ est définie par

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

Par exemple, pour $n = 3$, les valeurs possibles sont:

chaîne de 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 est égal à 1. Deuxièmement, l'addition s'effectue exactement de la même façon que l'addition d'entiers non signés. Par exemple, la somme $2 + (-3) = -1$ est obtenue ainsi:

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

Troisièmement, étant donné un nombre $a \in \mathbb{N}$ représenté par la chaîne de bits $b_{n-1} \cdots b_1 b_0$, il est possible de calculer $-a$ en deux étapes simples:

— inverser tous les bits de a: $(\neg b_{n-1}) \cdots (\neg b_1)(\neg b_0)$;

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

Par exemple, considérons $a = 2$ représenté par 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 chaîne 0110, qui ne représente *pas* la valeur -2. En effet, les nombres pouvant être représentés sur quatre bits sont:

chaîne de 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 chaîne 0110 représente 6.

Afin d'étendre correctement le nombre de bits d'un nombre, il faut copier son bit de signe (tout à gauche). Par exemple, la chaîne 110 qui représente -2 sur trois bits peut être étendue à **1**110 sur quatre bits. Similairement, la chaîne 011 qui représente 3 sur trois bits peut être étendue à **0**011 sur quatre bits.

5.3 Addition

Tel qu'indiqué à la section précédente, l'addition d'entiers signés se fait de la même façon que l'addition d'entiers non signés. Ainsi, l'unité arithmétique et logique peut utiliser les mêmes circuits logiques.

5.3.1 Report

Lorsqu'une addition, en arithmétique signée ou non signée, 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 d'étendre l'addition sur n bits à l'addition sur $2n$ bits. Par exemple, imaginons une architecture dont l'unique instruction d'addition opère sur les demi-mots (4 bits). Supposons 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 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 $-44 = -105 + 61$ (vérifiez-le).

5.3.2 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. 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 chaîne 1011 ne représente pas 11, mais bien -5. Ce problème surgit lorsque la somme de deux nombres positifs (resp. négatifs) excède la plus grande (resp. plus petite) valeur signée représentable. Notons qu'il est impossible d'obtenir un débordement si deux nombres de signes opposés sont additionnés.

5.4 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. Par exemple, $3 - 5 = 3 + (-5) = 0011 + 1011$:

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

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

5.5 Multiplication

5.5.1 Multiplication non signée

Il est possible d'implémenter la multiplication de deux nombres $a, b \in \mathbb{N}$ à l'aide de l'addition grâce à 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 utilisant le fait qu'une multiplication par 2 correspond à un décalage de bits. Par exemple, le produit $13 \cdot 11$ peut se calculer avec deux additions plutôt que douze:

$$\begin{aligned} 13 \cdot 11 &= 13 \cdot (2^3 + 2^1 + 2^0) \\ &= 13 \cdot 2^3 + 13 \cdot 2^1 + 2^0 \\ &= 1101_2 \cdot 2^3 + 1101_2 \cdot 2^1 + 1101_2 \cdot 2^0 \\ &= \mathbf{1101}000_2 + \mathbf{1101}0_2 + \mathbf{1101}_2 \\ &= 10001111_2 \\ &= 143. \end{aligned}$$

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

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

Algorithme 2 : Algorithme pour multiplier deux entiers non signés.

Entrées : Deux entiers non signés a et b sur n bits
Sorties : Entier non signé de $2n$ bits égal à $a \cdot b$
$\langle hi, lo \rangle \leftarrow \langle 0, b \rangle$
répéter n **fois**
  $r \leftarrow 0$
  si lo_0 **alors** $r, hi \leftarrow hi + a$ `/* r = 1 s'il y a report */`
  $\langle hi, lo \rangle \leftarrow \langle r\ hi_{n-1}\ \cdots\ hi_2\ hi_1, hi_0\ lo_{n-1}\ \cdots\ lo_2\ lo_1 \rangle$
end
retourner $\langle hi, lo \rangle$

Cet algorithme peut être implémenté efficacement tel que décrit à l'algorithme 2.

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 2. *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 strictement plus grand que le plus grand entier non signé pouvant être représenté sur $2n - 1$ bits. Nous avons:

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

□

5.5.2 Multiplication signée

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 le bon signe de bit;

— effectuer la multiplication standard des deux nombres;

— garder les derniers $2n$ bits.

Par exemple:

$$
\begin{array}{rll}
 & 00000101 & (5) \\
\times & 11111001 & (-7) \\
\hline
 & 00000101 & \\
 & 00000000 & \\
+ & 00000000 & \\
 & 00000101 & \\
 & 00000101 & \\
 & 00000101 & \\
 & 00000101 & \\
 & 00000101 & \\
\hline
 & 0000100111011101 & (-35)
\end{array}
$$

ou bien:

$$
\begin{array}{rll}
 & 11111001 & (-7) \\
\times & 00000101 & (5) \\
\hline
 & 11111001 & \\
 & 00000000 & \\
+ & 11111001 & \\
\hline
 & 100111011101 & (-35)
\end{array}
$$

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 algorithme 3).

Algorithme 3 : Algorithme pour multiplier deux entiers signés.

Entrées : Deux entiers signés a et b sur n bits
Sorties : Entier signé de $2n$ bits égal à $a \cdot b$
$\langle hi, lo \rangle \leftarrow \langle 0, b \rangle$
répéter n **fois**
  $n, v \leftarrow 0$
  si *dernière itération* **alors** $a \leftarrow -a$
  si lo_0 **alors** $n, v, hi \leftarrow hi + a$ `// n = nég., v = débord.`
  $\langle hi, lo \rangle \leftarrow \langle (n \oplus v)\ hi_{n-1}\ \cdots\ hi_2\ hi_1, hi_0\ lo_{n-1}\ \cdots\ lo_2\ lo_1 \rangle$
end
retourner $\langle hi, lo \rangle$

5.6 Division

5.6.1 Division non signée

La division de deux entiers non signés de n bits peut être calculée comme en base 10. Par 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 peut être implémentée efficacement tel que décrit à l'algorithme 4.

Algorithme 4 : Algorithme pour diviser deux entiers non signés.

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

$\langle q, r \rangle \leftarrow \langle a, 0 \rangle$
répéter n **fois**
  $\langle q, r \rangle \leftarrow \langle q_{n-2} \ \cdots \ q_1 \ q_0 \ 0, r_{n-2} \ \cdots \ r_1 \ r_0 \ q_0 \rangle$
  si $r \geq b$ **alors**
    $r \leftarrow r - b$
    $q_0 \leftarrow 1$
  fin
end
retourner $\langle q, r \rangle$

5.6.2 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 avec nombre négatifs varie parfois. Par exemple, $-19 \div 3 = -6$ en C++ et $-19 \div 3 = -7$ en Python. En langage d'assemblage de l'architecture ARMv8, le résultat est $-19 \div 3 = -6$.

Notons 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.

5.7 Particularités de l'architecture ARMv8

5.7.1 Codes de condition

L'architecture ARMv8 possède quatre codes de condition:

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

Certaines instructions mettent les codes de condition à jour de la façon suivante:

— 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.

Modification des codes de condition. Les instructions `adds`, `subs` et `negs` mettent les codes de condition à jour, et se comportent respectivement comme `add`, `sub` et `neg`. Lors de la comparaison de deux registres via `cmp xd, xm`, la soustraction $x_d - x_m$ est effectuée, les codes de condition sont mis à jour en fonction de la différence, et celle-ci est jetée.

Obtenir les codes de condition. Le code de condition C peut être obtenu grâce à l'instruction `adc xd, xn, xm` qui stocke dans x_d la somme de x_n, de x_m, et d'un bit de retenue si C est vrai.

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

Entiers non signés		
Condition	**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		
Condition	**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

5.7.2 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 droite du registre, et remplit les 32 bits de gauche avec le bit de signe de w.

6

Tableaux

Un *tableau* est une collection d'éléments identifiés par des indices. Les éléments d'un tableau possèdent tous la même taille et sont stockés de façon contigue en mémoire. Par exemple le tableau illustré à la Figure 6.1 possède 5 éléments, tous stockés sur un octet, et identifiés par les indices 0, 1, 2, 3 et 4.

0	123
1	5
2	0
3	255
4	42

FIGURE 6.1 – Exemple d'un tableau unidimensionnel de 5 éléments. Les éléments du tableau sont numérotés à gauche par leur indice.

De façon générale, les indices d'un tableau de *dimension* $d \in \mathbb{N}_{>0}$ sont des tuplets $\boldsymbol{i}$ d'arité d tels que

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

pour certains $n_0, n_1, \ldots, n_{d-1} \in \mathbb{N}_{>0}$. Un tableau possède $n_0 \cdot n_1 \cdots n_{d-1}$ éléments.

Par exemple le tableau bidimensionnel illustré à la Figure 6.2 possède $3 \cdot 2 = 6$ éléments, tous stockés sur deux octets, et identifiés par les indices $(0, 0)$, $(0, 1)$, $(1, 0)$, $(1, 1)$, $(2, 0)$ et $(2, 1)$. Un tel tableau 2D peut être interprété comme une représentation linéaire d'une matrice.

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

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

FIGURE 6.2 – Exemple d'un tableau bidimensionnel de 6 éléments avec $n_0 = 3$ et $n_1 = 2$ (*gauche*) représentant une matrice (*droite*).

6.1 Calcul d'index

Considérons un tableau t de dimension d situé dans la mémoire principale à l'adresse a et dont les éléments sont stockés sur k octets. Nous écrivons $t[\boldsymbol{i}]$ afin de référer à l'élément identifié par l'indice $\boldsymbol{i}$, et $\mathtt{mem}_k[b]$ afin de référer au contenu sur k octets de la mémoire principale à l'adresse b. L'adresse relative à laquelle est stocké un élément se nomme son *index*. Nous expliquons comment calculer l'index d'un élément à partir de son indice.

6.1.1 Tableau unidimensionnel

Pour un tableau unidimensionnel ($d = 1$), nous avons:

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

Ainsi, de façon générale, l'élément identifié par l'indice i se situe à l'adresse:

$$a + \underbrace{i \cdot k}_{\text{index de l'élém. } i}.$$

6.1.2 Tableau bidimensionnel

Pour un tableau bidimensionnel ($d = 2$), nous avons:

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

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

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

6.2 Particularités de l'architecture ARMv8

6.2.1 Allouer et initialiser un tableau

Considérons à nouveau le tableau illustré à la figure 6.2. Rappelons que ses éléments sont stockés sur deux octets. Il est possible d'allouer le tableau en mémoire dans le segment de données non initialisées:

```
N0 = 3
N1 = 2

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

On peut ensuite remplir le tableau 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
```

Il est alternativement possible d'initialiser les éléments du tableau directement dans le segment de données initialisées:

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

6.2.2 Parcourir un tableau

Parcours linéaire. Les éléments du tableau peuvent être affichés grâce au code suivant. Dans ce code, l'adresse du tableau est stockée dans x_{19}, l'indice courant dans x_{20} et l'index courant dans x_{21}, et l'accès à un élément du tableau se fait grâce au mode d'adressage indirect par registre indexé « `[x19, x21]` »:

```
.global main

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)

    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 « `[x19], 2` ». Cela simplifie le code puisqu'il n'est pas nécessaire de calculer explicitement l'index:

```
.global main

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)

    bl      exit

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

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

Parcours bidimensionnel. Les éléments du tableau peuvent être affichés grâce au code suivant. Dans ce code, l'adresse du tableau est stockée dans $\texttt{x}_{19}$, l'indice i courant dans $\texttt{x}_{20}$, l'indice j courant dans $\texttt{x}_{21}$, et l'index courant dans $\texttt{x}_{22}$. L'index est calculé à partir de la formule $(i \cdot n_1 + j) \cdot 2$ puisque les éléments sont stockés sur 2 octets. L'accès à un élément du tableau se fait grâce au mode d'adressage indirect par registre indexé « `[x19, x22]` »:

```
.global main

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)

    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, il est aussi possible d'accéder aux éléments grâce au mode d'adressage indirect par registre indexé post-incrémenté « `[x19], 2` », ce qui simplifie le code:

```
.global main

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)

    bl      exit

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

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

7

Programmation structurée

Dans ce chapitre, nous montrons comment les concepts de la programmation structurée peuvent être implémentés dans un langage de bas niveau.

7.1 Structures de contrôle

Dans cette section, nous voyons 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++ (haut niveau) et 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 2; autrement dit des fonctions prenant deux valeurs en entrée et retournant une valeur booléenne.

7.1.1 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++)	Code de bas niveau (ARMv8)
`// instruction 1`	`// code pour l'instruction 1`
`// instruction 2`	`// code pour l'instruction 2`
`// ...`	`// ...`
`// instruction k`	`// code pour l'instruction k`

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

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

7.1.2 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 *switch*.

Si.

Code de haut niveau (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++)	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++)

```
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++)

```
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:
```

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 que de tester *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 ici deux exemples, la conjonction de deux prédicats: « $\mathtt{cond}_1(\mathtt{x}_\mathtt{d}, \mathtt{x}_\mathtt{n}) \wedge \mathtt{cond}_2(\mathtt{x}_\mathtt{m}, \mathtt{x}_\mathtt{k})$ », et

la disjonction de deux prédicats: « $\texttt{cond}_1(\texttt{x}_\texttt{d}, \texttt{x}_\texttt{n}) \vee \texttt{cond}_2(\texttt{x}_\texttt{m}, \texttt{x}_\texttt{k})$ ». Ces constructions se généralisent à un nombre arbitraire de prédicats et à d'autres opérateurs logiques.

Code de haut niveau (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++)

```
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:
```

7.1.3 Itération

L'*itération* est une structure de contrôle qui permet de répéter 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++)

```
while (cond(xd, xn)) {
  // code
}
```

Code de bas niveau (ARMv8)

```
boucle:
        cmp         xd, xn
        b.¬cond     fin
        // code
        b           boucle
fin:
```

Faire/tant que.

Code de haut niveau (C++)

```
do {
  // code
} while (cond(xd, xn));
```

Code de bas niveau (ARMv8)

```
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 peut être implémentée ainsi:

Code de haut niveau (C++)

```
for (xd = 0; xd <= xn; xd++) {
  // code
}
```

Code de bas niveau (ARMv8)

```
        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++)

```
for (init(xd); cond(xd, xn);
     modif(xd)) {
  // code
}
```

Code de bas niveau (ARMv8)

```
        // initialiser xd
boucle:
        cmp         xd, xn
        b.¬cond     fin
        // code
        // modifier xd
        b           boucle
fin:
```

7.2 Sous-programmes

Les langages de haut niveau offrent des primitives pour décrire des *sous-routines*, par exemple, sous la forme de procédures, de fonctions ou de méthodes. Ces sous-routines permettent de modulariser le code. 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++, cette fonction peut être implémentée ainsi:

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

Notons que notre description s'applique au langage d'assemblage de l'architecture ARMv8. Les sous-programmes ne sont pas implémentés de la même façon sur toutes les architectures.

7.2.1 Passage de 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 `max(a, b)`.

Afin de déclarer notre sous-programme `max`, il suffit d'ajouter une étiquette « `max:` ». L'appel du sous-programme se fait via l'instruction « `bl` », que nous avons déjà utilisé 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
                                    //
        bl          exit            // sans cette ligne,
                                    // l'exécution se
                                    // poursuit dans "max"
max:
    // code du sous-programme ici
```

Passage par adresse. Dans notre exemple, les arguments sont passés *par valeur* puisque ce sont des entiers de 64 bits pouvant être stockés dans des 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.

7.2.2 Retour

L'instruction `ret` termine l'exécution d'un sous-programme et branche 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        max100          // if (a >= b) {
        mov         x19, x0         //   m = a
        b           max200          // }
max100:                             // else {
        mov         x19, x1         //   m = b
max200:                             // }
        mov         x0, x19         //
        ret                         // return m
```

Puisqu'il est permis d'altérer le contenu des registres d'arguments, notre code pourrait être simplifié:

```
max:
        cmp         x0, x1          //
        b.ge        max200          // if (a >= b) return a
        mov         x0, x1          // else return b
max200:                             //
        ret                         //
```

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

7.2.3 Sauvegarde des registres

Sur l'architecture ARMv8, les registres sont partagés par le programme et *tous* les sous-programmes. Ainsi, 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 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 de restauration

```
.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 stockée dans la mémoire principale afin de sauvegarder temporairement la valeur des registres. Nous discuterons du fonctionnement de cette pile dans un chapitre subséquent. En particulier, la pile nous permettra de mettre au point des sous-programmes récursifs.

Exemple complet. En ajoutant l'appel de ces macros, nous obtenons le code complet suivant 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        max100          // if (a >= b) {
        mov         x19, x0         //   m = a
        b           max200          // }
max100:                             // else {
        mov         x19, x1         //   m = b
max200:                             // }
        mov         x0, x19         //
        RESTORE                     //
        ret                         // return m
```

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

7.3 Particularités de l'architecture ARMv8

7.3.1 Distance des adresses

L'instruction « `bl` » permet de brancher vers une étiquette dont l'adresse est située à une distance de ±128Mio du compteur d'instruction. 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` ».

7.3.2 Affectation par sélection

L'instruction « `csel` » permet d'affecter 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
```

8

Valeurs booléennes et bits

8.1 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, les opérateurs logiques 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 8.1.

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 8.1 – 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.

Notons que sous la représentation par bit, il existe un lien entre d'un côté la négation, la conjonction et la disjonction, et de l'autre côté la soustraction, la multiplication et l'addition. Par exemple, $x \leftrightarrow y \equiv (\neg x \wedge \neg y) \vee (x \wedge y) \equiv ((1-x)\cdot(1-y)) + (x \cdot y)$.

8.2 Représentation des valeurs booléennes

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: $00000001_2 = \textit{vrai}$ et $00000000_2 = \textit{faux}$;

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

Ces deux représentations « gaspillent » sept bits de mémoire, mais forment néanmoins les représentations utilisées dans plusieurs langages de programmation. Par exemple, le code C++ suivant retourne fort probablement 1 sur votre machine, ce qui signifie qu'une valeur booléenne est représentée par un octet:

```
#include <iostream>

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

Si l'on désire représenter *plusieurs* valeurs booléennes, il est également possible d'utiliser un tableau d'octets. Par exemple, $8n$ valeurs booléennes peuvent être représentées 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.

8.3 Manipulation de bits

8.3.1 Opérateurs logiques

La plupart des architectures comme ARMv8 permettent de manipuler les bits d'un registre, notamment à l'aide d'opérateurs logiques. Par exemple, sous ARMv8, les opérations $\neg$, $\wedge$, $\vee$ et $\oplus$ peuvent être calculées à l'aide des instructions `mvn`, `and`, `orr` et `eor`. Cependant, ces opérations ne sont pas effectuées sur un seul bit, mais bien « bit à bit » sur *chacun* des 64 bits.

Par exemple, si $\texttt{x}_{20} = 0\cdots1011$ et $\texttt{x}_{21} = 1\cdots1001$, alors nous obtenons:

Instruction	Contenu de $\texttt{x}_{19}$ après l'exécution
`mvn x19, x20`	$1\cdots0100$
`and x19, x20, x21`	$0\cdots1001$
`orr x19, x20, x21`	$1\cdots1011$
`eor x19, x20, x21`	$1\cdots0010$

Ces résultats sont obtenus en appliquant chaque opérateur logique « bit à bit »:

`mvn x19, x20`

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

`and x19, x20, x21`

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

`orr x19, x20, x21`

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

`eor x19, x20, x21`

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

Échange de registres. L'opération OU exclusif permet d'échanger le contenu de deux registres sans utiliser de registre temporaire. Par exemple, le code suivant échange le contenu des registres $\texttt{x}_{19}$ et $\texttt{x}_{20}$:

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

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

8.3.2 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. Par exemple, considérons un octet w dont le contenu binaire est 10110101. Un décalage de w de 3 bits mène aux contenus suivants:

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

En général, pour une chaîne de k bits $w = b_{k-1} \cdots b_1 b_0$, un décalage à droite de j bits résulte en la chaîne:

$$\underbrace{0 \cdots 0}_{j \text{ fois}}\ b_{k-1} \cdots b_{j+1} b_j,$$

et un décalage à gauche de j bits résulte en la chaîne:

$$b_{k-1-j} \cdots b_1 b_0\ \underbrace{0 \cdots 0}_{j \text{ fois}}.$$

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

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

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

Multiplication et division. Les décalages logiques permettent d'implémenter la multiplication et la division entière par une puissance de 2 (non signée). 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 $\texttt{x}_{19}$ par 8. Cela s'avère notamment pratique pour le calcul d'index lors de la manipulation de tableaux.

8.3.3 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. Par exemple, reconsidérons l'octet w dont le contenu binaire est 10110101. Un décalage circulaire de w de 3 bits mène aux contenus suivants:

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

En général, pour une chaîne de k bits $w = b_{k-1} \cdots b_1 b_0$, un décalage à droite de j bits résulte en la chaîne:

$$b_{j-1} \cdots b_1 b_0 \; b_{k-1} \cdots b_{j+1} b_j,$$

et un décalage à gauche de j bits résulte en la chaîne:

$$b_{k-1-j} \cdots b_1 b_0 \; b_{k-1} \cdots b_{k-j+1} b_{k-j}.$$

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

Instructions	**Effet**
`ror xd, xn, j`	stocke dans $\texttt{x}_\texttt{d}$ le décalage circulaire de $\texttt{x}_\texttt{n}$ de j bits vers la droite

Il existe également une variante de 32 bits qui manipulent plutôt les registres $\texttt{w}_\texttt{d}$ et $\texttt{w}_\texttt{n}$.

8.3.4 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. Par exemple, considérons l'octet w dont le contenu binaire est 11100101. Un décalage arithmétique de w de 2 bits mène aux contenus suivants:

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

En général, pour une chaîne de k bits $w = b_{k-1} \cdots b_1 b_0$, un décalage à droite de j bits résulte en la chaîne:

$$\underbrace{b_{k-1} \cdots b_{k-1}}_{j \text{ fois}} \; b_{k-1} \cdots b_{j+1} b_j,$$

et un décalage à gauche de j bits résulte en la chaîne:

$$b_{k-1-j} \cdots b_1 b_0 \; \underbrace{0 \cdots 0}_{j \text{ fois}}.$$

Il n'y a donc pas de distinction entre un décalage arithmétique à gauche et un décalage logique à gauche.

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

Instructions	Effet
`asr xd, xn, j`	stocke dans $\texttt{x}_\texttt{d}$ le décalage arithmétique de $\texttt{x}_\texttt{n}$ de j bits vers la droite

Il existe également une variante de 32 bits qui manipulent plutôt les registres $\texttt{w}_\texttt{d}$ et $\texttt{w}_\texttt{n}$.

Multiplication/Division. Les décalages logiques permettent d'implémenter la multiplication et la division entière par une puissance de 2 (signée). 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 $\texttt{x}_{19}$ par 4.

8.4 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 $\texttt{x}_{19}$ à zéro, à l'exception du troisième bit de poids faible qui demeure inchangé. En effet, nous avons:

b_{63}	$\cdots$	b_3	b_2	b_1	b_0
$\wedge$	$\cdots$	$\wedge$	$\wedge$	$\wedge$	$\wedge$
0	$\cdots$	0	1	0	0
0	$\cdots$	**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 bit de poids faible:

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

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

Opération	Nom	Effet
$r \wedge m$	Sélection	Met à 0 les bits de r non spécifiés par le masque m
$r \vee m$	Activation	Met à 1 les bits de r spécifiés par le masque m
$r \wedge \neg m$	Désactivation	Met à 0 les bits de r spécifiés par le masque m
$r \oplus m$	Basculement	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

Chaînes de caractères

Afin de lire et d'afficher des symboles lisibles par un humain, il est nécessaire de les représenter en mémoire. Ces symboles (lettres, chiffres, signes de ponctuation, émojis, etc.) se nomment des *caractères*.

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.

9.1 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 9.1. Par exemple, la lettre « m » est représentée par le code $109 = 6\mathrm{D}_{16} = 1101101_2$, et le chiffre « 6 » est représentée par $54 = 36_{16} = 0110110_2$.

Les autres caractères qui n'apparaissent pas à la figure 9.1 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 un 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.
Déc.	Hex.	
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?
64	40	@

Code		Carac.
Déc.	Hex.	
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_
96	60	`

Code		Carac.
Déc.	Hex.	
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~

FIGURE 9.1 – Liste des caractères graphiques du codage ASCII. Chaque caractère est représenté par un code numérique indiqué ici dans sa forme décimale et hexadécimale.

9.2 ISO 8859-1 (Latin-1)

Bien que le codage ASCII permette la représentation 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 supplémentaires. 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 européennes dont le système d'écriture provient du latin. Les caractères graphiques additionnels du codage ISO 8859-1 sont répertoriés à la figure 9.2.

9.3 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 $\text{E9}_{16} = 11101001_2$ sous le codage ISO 8859-1, est codé par $\text{C3A9}_{16} = 11000011\ 10101001_2$ sous le codage UTF-8. Le format général de l'UTF-8 est décrit ainsi [Yer03]:

# bits	Plage de codes[1]		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******`

Par exemple, les caractères a, é, ケ et 𠮷 possèdent les codes 61_{16}, E9_{16}, 30B1_{16} et 1240D_{16} respectivement. Leur codage UTF-8 s'obtient donc ainsi:

Car.	Code	Codage
a	$61_{16} = 1100001_2$	01100001_2
é	$\text{E9}_{16} = 000\ 11101001_2$	$11000011\ 10101001_2$
ケ	$\text{30B1}_{16} = 00110000\ 10110001_2$	$11100011\ 10000010\ 10110001_2$
𠮷	$\text{1240D}_{16} = 00001\ 00100100\ 00001101_2$	$11110000\ 10010010\ 10010000\ 10001101_2$

1. Pour des raisons techniques, les codes D800_{16} à DFFF_{16} sont considérés invalides et ne représentent donc aucun caractère.

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 9.2 – Liste des caractères graphiques du codage ISO 8859-1. Chaque caractère est représenté par un code numérique indiqué ici dans sa forme décimale et hexadécimale.

9.4 Chaînes de caractères

Une *chaîne de caractères* est une séquence 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`.

Sur l'architecture ARMv8, une chaîne de caractères peut être déclarée à l'aide de « `.asciz` », alors qu'une suite de caractères, sans caractère nul de fin, peut être déclarée à l'aide de « `.ascii` ».

10

Sous-programmes et mémoire

Tel que discuté au chapitre 7, 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 via des branchements.

10.1 Pile d'exécution

10.1.1 Appels de sous-programmes

Sur l'architecture ARMv8, les arguments d'un sous-programme sont passés via 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 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 préservé 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.

10.1.2 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 dont la taille n'est pas connue à l'assemblage. Ce segment se divise en deux sous-segments:

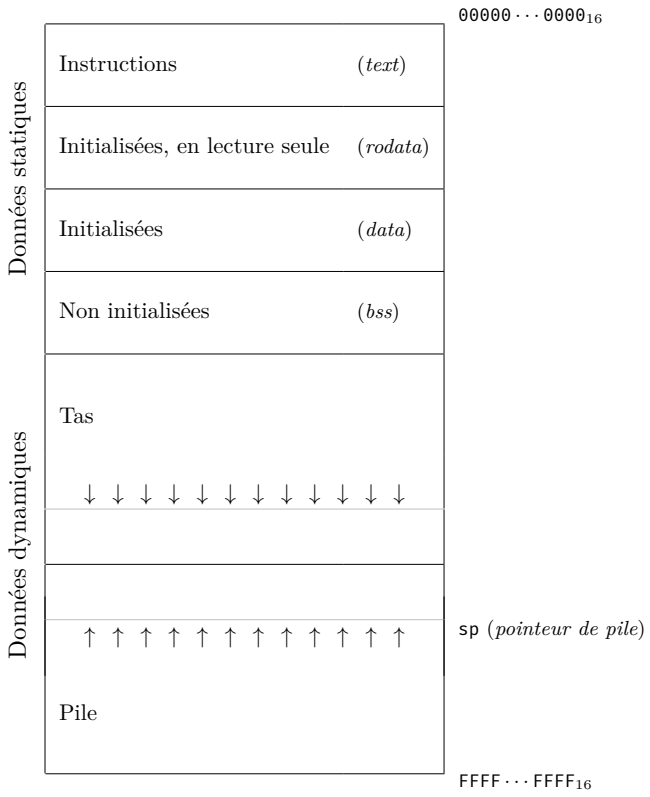


FIGURE 10.1 – Disposition de la mémoire d'un programme.

— 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 10.1.

10.1.3 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.

10.1.4 Sauvegarde et restauration

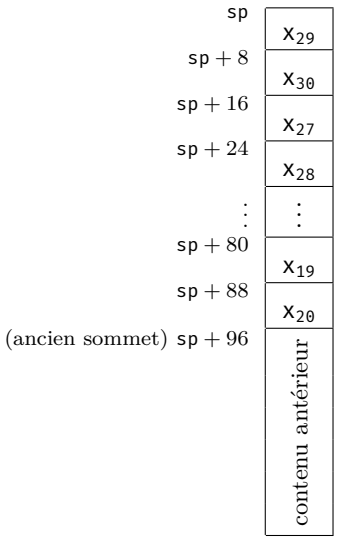
Nous sommes maintenant en mesure d'expliquer le fonctionnement des macros de sauvegarde et de restauration de registres ARMv8 présentées au chapitre 7.

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 x_d et x_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: x_{19} à x_{30}. L'instruction stocke ensuite le contenu de x_{29} et 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.

10.2 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 passé via `x0`:

```
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 l'environnement      //
        stp     x29, x30, [sp, -32]!        //
        mov     x29, sp                     //
        stp     x19, x20, [sp, 16]          //
                                            //
        // Calculer fib(n) récursivement    //
        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 l'environnement    //
        ldp     x29, x30, [sp], 16      //
        ldp     x19, x20, [sp], 16      //
        ret                             //   return n
                                        // }
```

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.

11

Nombres en virgule flottante

Dans les chapitres précédents, nous avons vu comment un intervalle $[a, b]$ de nombre naturels ou d'entiers peut être représenté dans un ordinateur. Dans ce chapitre, nous considérons la représentation et la manipulation de nombres réels.

Soient $a, b \in \mathbb{R}$. Puisque l'ensemble $\mathbb{R}$ est non dénombrable, il existe une *infinité* de nombres dans l'intervalle $[a, b]$. 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 à virgule flottante qui permet d'approximer raisonnablement les nombres réels.

11.1 Représentation

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

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

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

$$-0,5142 \times 10^3;\ 1,0567 \times 10^{-2} \text{ et } 1,011 \times 2^2$$

sont des nombres en virgule flottante dont la valeur décimale est respectivement:

$$-514{,}2;\ 0{,}010567 \text{ et } 5{,}5.$$

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

$x \neq 0$ offre une représentation unique de x par rapport à une base β. Notons cependant que 0 ne peut pas être normalisé.

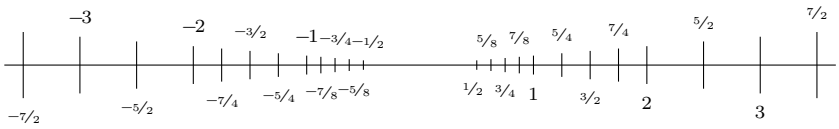
Si $e_{\min} \leq e \leq e_{\max}$, alors la quantité de nombres normalisés représentables avec une certaine base β et une mantisse de taille n est:

$$\underbrace{2}_{\pm} \cdot \underbrace{(\beta - 1)}_{d_0} \cdot \underbrace{\beta^{n-1}}_{d_1,\ldots,d_{n-1}} \cdot \underbrace{(e_{\max} - e_{\min} + 1)}_{e}.$$

La plus petite valeur absolue $x_{\min}$ et la plus grande valeur absolue $x_{\max}$ représentables par un nombre normalisé sont:

$$\begin{aligned} x_{\min} &= 1{,}0\cdots0 \times \beta^{e_{\min}} & x_{\max} &= (\beta-1),(\beta-1)\cdots(\beta-1) \times \beta^{e_{\max}} \\ &= \beta^{e_{\min}}, & &= \frac{(\beta^n - 1)}{\beta^{n-e_{\max}-1}} \\ & & &= \beta^{e_{\max}+1} - \beta^{e_{\max}+1-n}. \end{aligned}$$

Par exemple, si $\beta = 2$, $n = 3$ et $-1 \leq e \leq 1$, alors 24 nombres normalisés peuvent être représentés:



Notons que plus les nombres s'approchent de 0, plus la distance entre ceux-ci est petite. En effet, les nombres positifs de l'intervalle ci-haut peuvent être réécrits 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.

11.2 Précision

Puisque la vaste majorité des nombres réels ne peuvent pas être représentés par un nombre en virgule flottante, ceux-ci doivent être approximés. Par exemple, supposons que $\beta = 10$ et que la mantisse possède $n = 4$ 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 pourrait se trouver à la même distance de deux nombres. Une façon de briser cette égalité consiste à arrondir au nombre le plus près dont le dernier chiffre est pair, par ex. 1,9565 est arrondi à 1,956 plutôt que 1,957.

Fixons $e_{\min}$, $e_{\max}$, n et β. Pour tout $x \in \mathbb{R} \setminus \{0\}$, 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. L'*erreur absolue* de x est $x - \overline{x}$, et l'*erreur relative* de x est $\text{err}(x) \stackrel{\text{def}}{=} \frac{x-\overline{x}}{x}$.

L'erreur absolue donne la différence entre une valeur réelle et son approximation en nombre à virgule flottante. L'erreur relative décrit quant à elle le rapport entre l'erreur absolue et la valeur.

Posons $\varepsilon = \frac{\beta}{2} \cdot \beta^{-n}$. Nous appelons la constante ε l'*epsilon machine*. L'erreur relative d'un nombre est d'au plus $\pm\varepsilon$. En effet, nous avons:

Proposition 3. $|\text{err}(x)| \leq \varepsilon$ *pour tout* $x \in \mathbb{R}$ *tel que* $x_{min} \leq |x| \leq x_{max}$.

Par exemple, reconsidérons le cas où $\beta = 2$, $n = 3$ et $-1 \leq e \leq 1$. L'epsilon machine vaut $\varepsilon = 2^{-3} = \frac{1}{8}$. Ainsi, l'erreur relative d'une approximation est d'au plus $\pm\frac{1}{8}$. Par exemple, considérons le nombre $x = 1{,}10\underline{10011} \times 2^1$. Ce nombre n'est pas représentable avec une mantisse de taille $n = 3$. La valeur décimale de x est 3,296875 qui est comprise entre 3 et 3,5. Ainsi, $\overline{x} = 1{,}11 \times 2^1$. Autrement dit, x est arrondi à 3,5 et par conséquent:

$$\begin{aligned}
|\text{err}(x)| &= \left|\frac{3{,}296875 - 3{,}5}{3{,}296875}\right| \\
&= \frac{3{,}5 - 3{,}296875}{3{,}296875} \\
&\leq \frac{3{,}5 - 3{,}296875}{3{,}25} \\
&\leq \frac{3{,}5 - 3{,}25}{3{,}25} \\
&= \frac{0{,}25}{3{,}25} \\
&= \frac{1}{13} \\
&\leq \frac{1}{8}.
\end{aligned}$$

Notons qu'il est possible d'obtenir des bornes similaires à celle de la proposition 3 pour d'autres méthodes d'arrondis comme la troncation.

11.3 Arithmétique

Voyons comment calculer $x + y$ et $x \cdot y$ pour deux nombres en virgule flottante x et y de la forme:

$$\begin{aligned}
x &= 1{,}u \times \beta^e, \\
y &= 1{,}v \times \beta^f.
\end{aligned}$$

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.

11.3.1 Addition

Supposons sans perte de généralité[1] que $e \leq f$. 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, 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, il faut:

1. Mettre les exposants en commun en décalant la première mantisse de $e - f$ positions;
2. Additionner les mantisses;
3. Normaliser le résultat en décalant la mantisse tout en incrémentant/décrémentant l'exposant selon la direction;
4. Arrondir le résultat.

Par 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:

$$\begin{aligned} x \cdot y &= (1{,}00 \times 2^{-1}) + (1{,}11 \times 2^0) \\ &= (0{,}100 \times 2^0) + (1{,}11 \times 2^0) \\ &= (0{,}100 + 1{,}11) \times 2^0 \\ &= 10{,}010 \times 2^0 \\ &= 1{,}0010 \times 2^1 \\ &\approx 1{,}00 \times 2^1. \end{aligned}$$

Ainsi la somme ½ + ¾ = 0,5 + 1,75 donne ici 2,0 comme approximation de la valeur exacte 2,25.

11.3.2 Multiplication

Observons que:

$$\begin{aligned} x \cdot y &= (1{,}u \times \beta^e) \cdot (1{,}v \times \beta^f) \\ &= (1{,}u \cdot 1{,}v) \times \beta^{e+f}. \end{aligned}$$

La multiplication se calcule 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. Si ce n'est pas le cas, on peut simplement inverser x et y.

1. Additionner les exposants;
2. Multiplier les mantisses;
3. Normaliser le résultat en décalant la mantisse tout en incrémentant/décrémentant l'exposant selon la direction;
4. Arrondir le résultat.

Par 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:

$$\begin{aligned} x \cdot y &= (1{,}10 \times 2^{-1}) \cdot (1{,}11 \times 2^{1}) \\ &= (1{,}10 \cdot 1{,}11) \times 2^{0} \\ &= 10{,}101 \times 2^{0} \\ &= 1{,}0101 \times 2^{1} \\ &\approx 1{,}01 \times 2^{1}. \end{aligned}$$

Ainsi le produit $¾ \cdot ½ = 0{,}75 \cdot 3{,}5$ donne ici 2,5 comme approximation de la valeur exacte 2,625.

11.4 Norme IEEE 754

La norme IEEE 754 définit des formats de nombres en virgule flottante en base 2 et base 10 utilisés par la grande majorité des architectures modernes. Ces formats incluent la précision *simple* (32 bits) et la 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	$-x_{\min}$	$x_{\max}$
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, l'epsilon machine 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 ici la nomenclature traditionnelle *simple* et *double* afin de référer à ces deux formats.

11.4.1 Codage des formats

Les formats simple et double sont respectivement codés avec 32 et 64 bits répartis ainsi:

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 ssi le nombre est négatif. Les bits $b_{30}b_{29}\cdots b_{23}$ représentent $e+127$ sous forme d'entier non signé; autrement dit l'exposant plus un *biais* de 127. Par exemple:

$$\begin{aligned}
00000001 \text{ représente } e &= 1 - 127 = -126,\\
00100011 \text{ représente } e &= 35 - 127 = -92,\\
11111110 \text{ représente } e &= 254 - 127 = 127.
\end{aligned}$$

Les chaînes de bits 00000000 et 11111111 sont réservées à d'autres fins. 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.

Zéro. Le nombre 0, qui ne peut pas être normalisé, 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:

$$b_{31}11\cdots1\,e\,00\cdots01.$$

En particulier, le bit e indique si une erreur doit être lancée ou non 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.

11.5 Particularités de l'architecture ARMv8

11.5.1 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 $\mathtt{s}_n$.

11.5.2 Instructions

Les instructions sont similaires à celles qui manipulent 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 d'op.	Syntaxe	Effet	Exemple
`ldr`	`ldr dn, a`	charge un nombre en virgule flottante double précision de l'adresse `a` vers $\mathtt{d}_n$ (8 octets)	`ldr d8, [x19]`
	`ldr sn, a`	charge un nombre en virgule flottante simple précision de l'adresse `a` vers $\mathtt{s}_n$ (4 octets)	`ldr s8, [x19]`
`str`	`str dn, a`	stocke un nombre en virgule flottante double précision de $\mathtt{d}_n$ vers l'adresse `a` (8 octets)	`str d8, [x19]`
	`str sn, a`	stocke un nombre en virgule flottante simple précision de $\mathtt{s}_n$ vers l'adresse `a` (4 octets)	`str s8, [x19]`
`fmov`	`fmov vd, vm`	$\mathtt{v}_d \leftarrow \mathtt{v}_m$	`fmov d8, d9`
	`fmov vd, i`	$\mathtt{v}_d \leftarrow i$	`fmov d8, 1.5`
`fcmp`	`fcmp vd, vm`	compare $\mathtt{v}_d$ et $\mathtt{v}_m$	`fcmp d8, d9`
	`fcmp vd, i`	compare $\mathtt{v}_d$ et i	`fcmp d8, 0.0`
`fadd`	`fadd vd, vn, vm`	$\mathtt{v}_d \leftarrow \mathtt{v}_n + \mathtt{v}_m$	`fadd d8, d9, d10`
`fsub`	`fsub vd, vn, vm`	$\mathtt{v}_d \leftarrow \mathtt{v}_n - \mathtt{v}_m$	`fsub d8, d9, d10`
`fmul`	`fmul vd, vn, vm`	$\mathtt{v}_d \leftarrow \mathtt{v}_n \cdot \mathtt{v}_m$	`fmul d8, d9, d10`
`fdiv`	`fdiv vd, vn, vm`	$\mathtt{v}_d \leftarrow \mathtt{v}_n / \mathtt{v}_m$	`fdiv d8, d9, d10`
`fmax`	`fmax vd, vn, vm`	$\mathtt{v}_d \leftarrow \max(\mathtt{v}_n, \mathtt{v}_m)$	`fmax d8, d9, d10`
`fmin`	`fmin vd, vn, vm`	$\mathtt{v}_d \leftarrow \min(\mathtt{v}_n, \mathtt{v}_m)$	`fmin d8, d9, d10`
`fsqrt`	`fsqrt vd, vn`	$\mathtt{v}_d \leftarrow \sqrt{\mathtt{v}_n}$	`fsqrt d8, d9`
`fabs`	`fabs vd, vn`	$\mathtt{v}_d \leftarrow \lvert\mathtt{v}_n\rvert$	`fabs d8, d9`

11.5.3 Exemple

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:

```
.global main

/*
   Calcule la norme euclidienne d'un vecteur (x, y)
                                                    */
main:                                   // main() {
        // Lire x                       //
        adr     x0, fmtEntree           //
        adr     x1, tmp                 //
        bl      scanf                   //
        ldr     d8, tmp                 //   scanf("%lf", &x)
                                        //
        // Lire y                       //
        adr     x0, fmtEntree           //
        adr     x1, tmp                 //
        bl      scanf                   //
        ldr     d9, tmp                 //   scanf("%lf", &y)
                                        //
```

```
        // Calculer ||(x, y)||          //
        fmov    d0, d8                  //
        fmov    d1, d9                  //
        bl      norme                   //   n = norme(x, y)
        fmov    d8, d0                  //
                                        //
        // Afficher ||(x, y)||          //
        adr     x0, fmtSortie           //
        fmov    d0, d8                  //
        bl      printf                  //   printf("%lf\n", n)
                                        //
        // Quitter                      //
        mov     x0, 0                   //
        bl      exit                    // }
                                        //
norme:                                  // double norme(double x, double y) {
        fmul        d8, d0, d0          //
        fmul        d9, d1, d1          //
        fadd        d10, d8, d9         //
        fsqrt       d0, d10             //
        ret                             //   return √(x² + y²)
                                        // }

.section ".rodata"
fmtEntree:  .asciz  "%lf"
fmtSortie:  .asciz  "%lf\n"

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

12

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 à l'aide de fonctions de haut-niveau. 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 vidéos *Nintendo Entertainment System (NES)* illustrée à la figure 12.1.

FIGURE 12.1 – Console de jeu *Nintendo Entertainment System (NES)*.

12.1 Architecture du NES

Le processeur du NES est une variante du populaire 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.

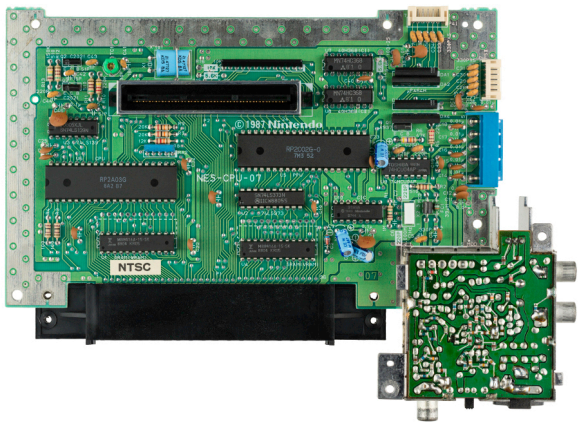


FIGURE 12.2 – Carte-mère du NES.

12.1.1 Organisation de la mémoire

Mémoire principale. La mémoire adressable 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 de mémoire typiquement destinée au stockage de tuiles;

— *mémoire d'entrée/sortie*: registres permettant d'effectuer des entrées/sorties notamment avec le processeur d'images et les manettes;

— *cartouche de jeu*: contient le code du programme (jeu vidéo) et d'autres segments de mémoire optionnels.

Ces segments sont illustrés du côté gauche de la figure 12.3.

Mémoire vidéo. La mémoire du processeur d'images se divise en trois segments:

— *tuiles*: contient deux tables des tuiles du jeu;

— *arrière-plan*: quatre tables décrivant les tuiles de l'arrière-plan ainsi que leurs attributs (couleurs, orientation, etc.);

— *palettes de couleur*: décrit les couleurs disponibles pour les différentes tuiles.

Ces segments sont illustrés du côté droit de la figure 12.3.

Le processeur d'images a également accès à une mémoire de *sprites* de 256 octets située en dehors de la mémoire vidéo.

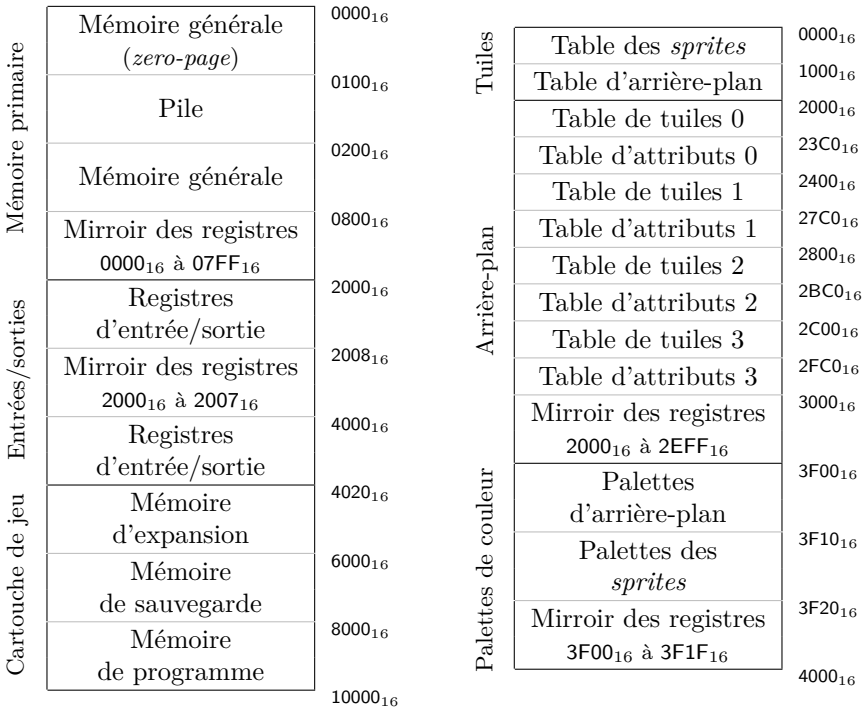


FIGURE 12.3 – Organisation de la mémoire principale (gauche) et de la mémoire vidéo (droite) du NES.

12.2 Registres

Le processeur du NES possède quatre registres d'un octet 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 pile (pointe vers $0100_{16} + \mathsf{s}$)

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 (2 octets).

12.3 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.

12.3.1 Valeurs immédiates.

Les préfixes `$` et `%` représentent respectivement une valeur hexadécimale et binaire. L'absence de préfixe représente une valeur décimale. Les valeurs précédées d'un `#` représentent des valeurs numériques, alors que celles sans `#` représentent une adresse. Par 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.

12.3.2 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`

12.3.3 Accès mémoire.

Nous écrivons $\text{mem}_1[a]$ afin de dénoter l'octet situé à l'adresse a de la mémoire principale.

Code d'op.	Syntaxe	Effet	Exemple
lda	lda #i	a ← i	lda #42
	lda adr	a ← $\text{mem}_1[adr]$	lda var
ldx	ldx #i	x ← i	ldx #42
	ldx adr	x ← $\text{mem}_1[adr]$	ldx var
ldy	ldy #i	y ← i	ldy #42
	ldy adr	y ← $\text{mem}_1[adr]$	ldy var
sta	sta adr	$\text{mem}_1[adr]$ ← a	sta var
stx	stx adr	$\text{mem}_1[adr]$ ← x	stx var
sty	sty adr	$\text{mem}_1[adr]$ ← y	sty var
txa	txa	a ← x	txa
tax	tax	x ← a	tax
tya	tya	a ← y	tya
tay	tay	y ← a	tay
txs	txs	s ← x	txs
tsx	tsx	x ← s	tsx

12.3.4 Arithmétique.

Code d'op.	Syntaxe	Effet	Exemple
adc	adc #i	a ← a + i + *report*	lda #1
	adc adr	a ← a + $\text{mem}_1[adr]$ + *report*	adc var
sbc	sbc #i	a ← a − i − *emprunt*	sbc #1
	sbc adr	a ← a − $\text{mem}_1[adr]$ − *emprunt*	sbc var
clc	clc	*report* ← 0 (utile avant adc)	clc
sec	sec	*emprunt* ← 0 (utile avant sbc)	sec
inx	inx	x ← x + 1	inx
iny	iny	y ← y + 1	iny
inc	inc adr	$\text{mem}_1[adr]$ ← $\text{mem}_1[adr]$ + 1	inc var
dec	dec adr	$\text{mem}_1[adr]$ ← $\text{mem}_1[adr]$ − 1	dec var

12.3.5 Logique.

Code d'op.	Syntaxe	Effet	Exemple
asl	asl adr	a ← décalage logique à gauche d'un bit de $\text{mem}_1[adr]$	asl var
lsr	lsr adr	a ← décalage logique à droite d'un bit de $\text{mem}_1[adr]$	lsr var
and	and #i	a ← a ∧ i	and #%00100011
	and adr	a ← a ∧ $\text{mem}_1[adr]$	and var
ora	ora #i	a ← a ∨ i	ora #%00100011
	ora adr	a ← a ∨ $\text{mem}_1[adr]$	ora var
eor	eor #i	a ← a ⊕ i	eor #%00100011
	eor adr	a ← a ⊕ $\text{mem}_1[adr]$	eor var

12.3.6 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`

12.4 Sorties graphiques

12.4.1 Tuiles

Chaque image est formée 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):

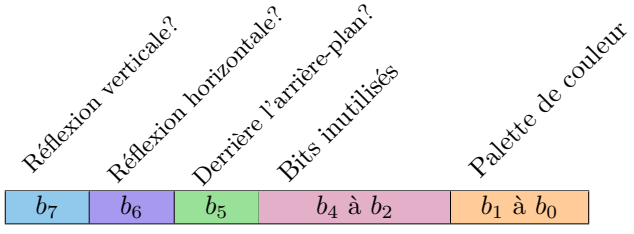
— sa position verticale comprise entre 0 et 255;

— son identificateur (sa position dans la table de tuiles);

— ses attributs tels que ses couleurs;

— sa position horizontale comprise entre 0 et 255.

Les attributs d'une tuile sont déterminés par ses 8 bits $b_8 \cdots b_1 b_0$ selon le format suivant:



Par exemple, la séquence de 4 octets « 21_{16} 05_{16} 01000010_2 $A0_{16}$ » 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.

12.4.2 Affichage de tuiles

Il existe plusieurs façons d'effectuer l'affichage. Par exemple, afin d'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;

— indiquer au processeur d'image d'afficher le contenu de ces adresses via en stockage 02_{16} dans à l'adresse 4014_{16} de la mémoire principale.

L'adresse 4014_{16} réfère à un registre d'entrée/sortie. En y stockant le nombre $0X_{16}$, cela indique au bus de données de transférer le contenu des adresses $0X00_{16}$ à $0XFF_{16}$ vers la mémoire de *sprites*. Cette technique se nomme *accès direct à la mémoire (DMA)*. Par exemple, le code suivant transfère les tuiles contenues aux adresses 0200_{16} à $02FF_{16}$ de la mémoire principale vers la mémoire de *sprites*:

```
lda     #$02
sta     $4014
```

Afin de s'effectuer correctement, 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 son exécution en cours, puis reprend lorsque la sous-routine est complétée.

12.5 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, afin de lire 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 correspondent, dans l'ordre, aux boutons:

$$A, B, \mathit{select}, \mathit{start}, \mathit{haut}, \mathit{bas}, \mathit{gauche}, \mathit{droite}.$$

Pour chacune des lectures, le bit de poids faible vaut 1 ssi 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  ;   Lire bit b de poids faible du bouton A
```

13

Entrées/sorties

Un ordinateur est constitué de plusieurs périphériques d'entrée/sortie qui lui permette d'interagir avec le monde extérieur (clavier, souris, moniteur, mémoire externe, webcam, imprimante, 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 ces différents mécanismes. La plupart des concepts seront illustrés avec l'architecture du NES introduite au chapitre 12; à l'exception des appels système qui seront illustrés avec ARMv8.

13.1 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 d'états indique si l'intervalle de rafraîchissement vertical (VBLANK) est en cours. Ainsi, il est possible de programmer l'affichage de tuiles de la façon suivante:

— lire continuellement `$2002` jusqu'à ce que le bit de poids fort égale 1;

— envoyer les tuiles vers le processeur d'images.

Cette approche peut être implémentée ainsi:

```
main:                       ;
    jsr     attendre_vblank ; Attendre activement le VBLANK
    jsr     afficher_tuiles ; Afficher les tuiles
    jmp     main            ;
                            ;
attendre_vblank:            ; attendre_vblank()
    lda     $2002           ; {
    and     #%10000000      ;   while (!VBLANK) {
    cmp     #%10000000      ;     // Ne rien faire
```

```
bne     attendre_vblank   ;   }
rts                       ; }
```

De façon générale, cette méthode est connue sous le nom d'*attente active* puisqu'elle requiert une lecture constante.

13.2 Interruptions

L'attente active s'illustre 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 activement qu'un événement se produise, un signal est lancé lorsqu'il se produit. Le processeur s'interrompt et lance ainsi 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'utilisatrice appuie sur une touche.

13.2.1 Gestionnaires d'interruption

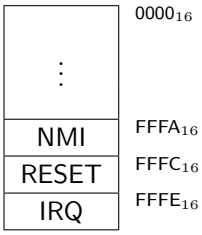
Le NES possède trois types d'interruptions:

— NMI: lancée lors de l'intervalle de rafraîchissement vertical (*VBLANK*);

— RESET: lancée lorsque la console est allumée (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 6 derniers octets de la mémoire principale contiennent l'adresse des sous-routines qui doivent être appelées afin de gérer ces interruptions:



De façon générale, une sous-routine appelée lors d'une interruption se nomme un *gestionnaire d'interruption*, et le segment de mémoire qui contient l'adresse des gestionnaires d'interruption se nomme *table d'interruptions* ou *vecteur d'interruptions*.

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:                       ;
    jmp     main            ; Aucun code nécessaire
    rti                     ;
                            ;
mise_a_jour:                ;
    jsr     afficher_tuiles ; Afficher les tuiles
    rti                     ;
                            ;
.org    $FFFA               ;
.word   mise_a_jour         ; Gestionnaire d'interruption NMI
.word   main                ; Gestionnaire d'interruption RESET
.word   0                   ; Gestionnaire d'interruption IRQ
```

13.2.2 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.

Par exemple, considérons le programme suivant:

```
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.

Supposons maintenant que `foo` en soit à l'exécution de « `cmp #0` » lorsqu'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

celle du MOS 6502, ce problème ne se produit pas. En effet, lors du traitement d'une interruption, les opérations suivantes sont effectuées:

— 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.

Notons que les gestionnaires d'interruption ne se terminent 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 afin d'empiler `a` ainsi qu'une instruction afin de dépiler vers `a`. Ainsi, nous pouvons implémenter la sauvegarde/restauration de registres similairement aux macros `SAVE`/`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 aux gestionnaires d'interruption, ceux-ci rétablissent donc entièrement l'environnement.

13.2.3 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

1. Du moins, au niveau logiciel.

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 grande, plus la priorité est importante. Lors de la gestion d'une interruption de niveau i, les interruptions moins prioritaires ($i + 1$ à n) sont ignorées. Supposons qu'un gestionnaire G d'une interruption de niveau i soit en cours d'exécution. Si une interruption de même ou plus haute priorité (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* puisqu'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 en cours 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
```

13.2.4 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'im-

plémenter un débogueur puisque « `brk` » interrompt l'exécution du programme à une ligne précise.

13.3 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 $0 \leq x \leq 255$ de la mémoire de *sprites*, il est possible de procéder en deux étapes:

— écrire x à l'adresse `$2003`,

— écrire v à l'adresse `$2004`.

Après ces deux étapes, l'écriture de v à l'adresse x est effectué et le contenu de `$2003` est automatiquement incrémenté par le processeur d'images. Ainsi, une tuile décrite par $(100, 1, 0, 127)$ peut être affichée ainsi:

```
lda     #$00                ; débuter l'écriture dans la
sta     $2003               ;   mémoire de sprites à $00
                            ;
; Position verticale        ;
lda     #100                ;
sta     $2004               ; sprite_mem[0x00] = 100
                            ;
; Numéro de la tuile        ;
lda     #1                  ;
sta     $2004               ; sprite_mem[0x01] = 1
                            ;
; Attributs de la tuile     ;
lda     #%00000000          ;
sta     $2004               ; sprite_mem[0x02] = 0
                            ;
; Position horizontale      ;
lda     #127                ;
sta     $2004               ; sprite_mem[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 ; ...
                            ;
lda     #$02                ; Copier mem[0x0200, 0x02FF]
sta     $4014               ; vers la mémoire de sprites
```

13.4 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 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, les appels système sont effectués en:

— indiquant le code du service dans x_8;

— passant les arguments via x_0 à x_5;

— utilisant l'instruction « `svc 0` ».

Par exemple, les appels système `write` et `read` sont décrits par les codes et paramètres suivants:

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 suivant 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`:

```
.global main

// Exemple d'entrées/sorties par appels système
main:                           // main()
        adr     x0, temp        // {
        mov     x1, 10          //
        bl      lire            //   lire(&temp)
                                //
        adr     x0, temp        //
        mov     x1, 10          //
        bl      afficher        //   afficher(&temp)
                                //
        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
```

A

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 $\overbrace{1\cdots11}^{n \text{ fois}}$
- L'addition correspond à la concaténation

Représentation positionnelle

- Généralisation du système décimal à une base $B \in \mathbb{N}_{\geq 2}$
- *Systèmes répandus*: 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*: $A = 10, B = 11, \ldots, F = 15, \ldots$
- *Valeur de s en base B*: $s_B = s_{n-1} \cdot B^{n-1} + \ldots + s_1 \cdot B^1 + s_0 \cdot B^0$
- *Exemple*: $8B5_{16} = 8 \cdot 16^2 + 11 \cdot 16^1 + 5 \cdot 16^0$
- Les zéros tout à gauche ne changent rien: $(0\cdots0s)_B = s_B$

Conversions

- *B à 10*: $s_0 + B \cdot (s_1 + B \cdot (s_2 + B \cdot (\ldots + B \cdot s_{n-1})))$
- *10 à B*: diviser à répétition par B et concaténer les restes de droite à gauche, par ex. $6_2 = \mathbf{110}$:
  $6 \div 2 = 3$ reste $\mathbf{0}$, $3 \div 2 = 1$ reste $\mathbf{1}$, $1 \div 2 = 0$ reste $\mathbf{1}$
- *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 chifre 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*: $(0\cdots0s{,}t0\cdots0)_B = (s{,}t)_B$

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 non divisible par 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 chaines bits
- *Pipeline*: parallélisation des étapes d'exécution
- *RISC*: instructions simples, taille fixe, mémoire–*ou*–autre

3. Programmation en langage d'assemblage: ARMv8

Registres

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

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
```

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`
- *Nb. virg. fottante*: `.single f`, `.double f`

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)

- *Affichage*: `printf(&format, val1, val2, ...)`
- *Lecture*: `scanf(&format, &var1, &var2, ...)`
- *Formats nombres*: `int32` (`%d`), `uint32` (`%u`), `uint32-hex` (`%X`), `float` (`%f`); 64 bits via préfixe `l`, par ex. `int64` (`%ld`)
- *Formats caractères*: `char` (`%c`) = 1 octet, `string` (`%s`)

4. Accès aux données

Adresses

- *Numérique*: entier non négatif, souvent en hexadécimal
- *Symbolique*: chaîne qui représente une adresse non connue

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
```

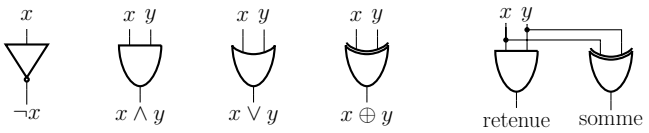
Assemblage

- *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. Nombres entiers

Circuits logiques

- « Blocs » de base d'un processeur, faits de portes logiques qui permettent d'implémenter le jeu d'instructions:



Représentation des entiers signés

- *Complém. à 2*: $\text{val}(b_{n-1}\cdots b_1 b_0) = -b_{n-1}\cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i \cdot 2^i$
- *Nombres représentables sur n bits*: $[-2^{n-1}, 2^{n-1}-1]$
- *Bit de signe*: bit de gauche $= 1$ ssi le nombre est négatif
- *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*: même procédure que pour les entiers non signés
- *Soustraction*: addition/changement de signe: $a-b = a+(-b)$
- *Multiplication et division non signées*: comme en base 10:

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

- *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

$\overbrace{\texttt{C}\ (\text{report})}^{\text{bit de retenue}}$, $\overbrace{\texttt{V}\ (\text{débordement})}^{\text{résultat trop grand/petit}}$

- *Codes*: `N` (négatif), `Z` (zéro), `C` (report), `V` (débordement)
- *Codes modifiés par*: `adds`, `subs`, `negs`, `adcs`, `cmp`
- *Comparaison*: codes mis à jour via soustraction bidon
- *Accès aux codes*: via `b.condition etiq`
- *Accès au report*: `adc rd, rn, rm`: $\texttt{r}_\texttt{d} \leftarrow \texttt{r}_\texttt{n} + \texttt{r}_\texttt{m} + \texttt{C}$

6. Tableaux

Généralités

- *Tableau*: collection d'éléments identifiés par des indices
- *Éléments*: tous de même taille, contigus en mémoire
- *Indice*: tuplet $\boldsymbol{i}$ de dimension $d \geq 1$
- *Bornes*: $0 \leq \boldsymbol{i} < n_i$ pour chaque dimension i
- *Taille*: $n_0 \cdot n_1 \cdots n_{d-1}$ éléments
- *Exemples tableau 1D et tableau 2D*:

0	123
1	5
2	0
3	255
4	42

$n_0 = 5$
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'index

- *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 de l'élément $\boldsymbol{i}$ =

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

Allocation/accès mémoire

- *Tableau non initialisé*:

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

- *Tableau initialisé*:

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

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

7. 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 via 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
- *Paramètres*: par valeur ou adresse dans x_0 à x_7 (en ordre)
- *Appel*: **`bl`** `sprog` assigne $x_{30} \leftarrow pc + 4$ et branche à `sprog:`
- *Retour*: **`ret`** branche vers l'adresse de retour x_{30}
- *Sauvegarde*: l'appelé doit rétablir les registres x_{19} à x_{30}

8. Valeurs booléennes et bits

Valeurs booléennes

- *Correspond à* un bit: $1 = vrai$, $0 = faux$
- *Représentation*: sur un octet, puisque bits non adressables

Opérateurs logiques

- *Opérations*: $\neg$, $\wedge$, $\vee$, $\oplus$ « bit à bit » étendues aux chaînes:

`mvn x19, x20`

¬	…	¬	¬	¬
0	…	1	0	1
1	…	**0**	**1**	**0**

`and x19, x20, x21`

0	…	1	0	1
∧	…	∧	∧	∧
1	…	1	0	0
0	…	**1**	**0**	**0**

`orr x19, x20, x21`

0	…	1	0	1
∨	…	∨	∨	∨
1	…	1	0	0
1	…	**1**	**0**	**1**

`eor x19, x20, x21`

0	…	1	0	1
⊕	…	⊕	⊕	⊕
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:

$11000101 \xrightarrow{\text{3 bits vers la gauche}} 00101000$ **`lsr xd, xn, 3`**

$11000101 \xrightarrow{\text{3 bits vers la droite}} 00011000$ **`lsl 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:

$r \wedge m$	Sélection	Met à 0 les bits de r non spécifiés par m
$r \vee m$	Activation	Met à 1 les bits de r spécifiés par m
$r \wedge \neg m$	Désactivation	Met à 0 les bits de r spécifiés par m
$r \oplus m$	Basculement	Inverse les bits de r spécifiés par m

9. Chaînes de caractères

Généralités

- *Caractère*: symbole représenté par une chaîne de bits
- *Chaîne de caractères*: séquence 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 encodés différemment
- *Format général*:

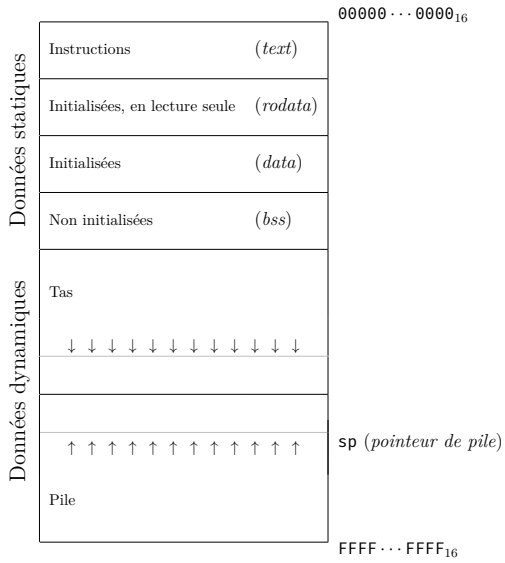
# bits	Plage de codes		Format binaire des octets			
	Début	Fin	Octet 1	Octet 2	Octet 3	Octet 4
7	000000_{16}	$00007F_{16}$	`0*******`	—	—	—
11	000080_{16}	$0007FF_{16}$	`110*****`	`10******`	—	—
16	000800_{16}	$00FFFF_{16}$	`1110****`	`10******`	`10******`	—
21	010000_{16}	$10FFFF_{16}$	`11110***`	`10******`	`10******`	`10******`

- *Exemples*:

Car.	Code	Codage
a	1100001_2	01100001_2
é	$000\ 11101001_2$	$11000011\ 10101001_2$
ケ	$00110000\ 10110001_2$	$11100011\ 10000010\ 10110001_2$
𠮷	$00001\ 00100100\ 00001101_2$	$11110000\ 10010010\ 10010000\ 10001101_2$

10. 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

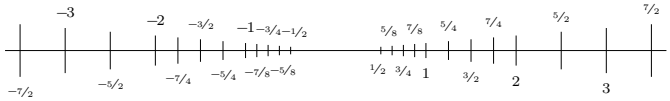
11. Nombres en virgule flottante

Représentation.

- *Nombre en virgule flottante:*

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

- *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 → 1,956
  (b) troncation: 1,5416 → 1,541
- *Erreur relative*: $\mathrm{err}(x) \overset{\text{def}}{=} \frac{x-\overline{x}}{x}$ où $\overline{x}$ est l'approximation
- *Borne pour mode (a)*: $|\mathrm{err}(x)| \leq \underbrace{(\beta/2)\cdot\beta^{-n}}_{\varepsilon\ \text{machine}}$

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:* $1000011010\cdots011 = -1{,}11 \times 2^{13-127}$
- ± 0 ($s0\cdots00\cdots0$); $\pm\infty$ ($s1\cdots10\cdots00$); NaN ($s1\cdots1\,e\,0\cdots01$)

ARMv8.

- *Registres*: $\mathtt{d_n}$ (64 bits) et $\mathtt{s_n}$ (32 bits)
- *Instructions*: `ldr`, `str`, `fmov`, `fcmp`, `fadd`, `fmul`, `fsqrt`, etc.

12. Introduction aux entrées/sorties : NES

Architecture.

- *Pas RISC*: possible de manipuler la mémoire directement
- *Processeurs:* proc. principal et proc. d'images (*PPU*)
- *Mémoire principale:* 2Kio de mémoire (générale et pile) + 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` (∧), `ora` (∨), `eor` (⊕)
- *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 $\mathtt{0X00}_{16}$ à $\mathtt{0XFF}_{16}$ 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

13. Entrées/sorties

Mécanismes d'entrée/sortie.

- *Attente active:* interrogation constante d'un registre d'état jusqu'à ce qu'un événement se produise (ex. *VBLANK*)
- *Interruption:* signal lancé vers le processeur lorsque'un événement se produit (ex. NES: NMI, RESET, IRQ)

Interruptions.

- *Gestionnaire d'interruption:* sous-routine exécutée afin de traiter une interruption
- *Table d'interruptions:* contient l'adresse des gestionnaires
- *Traitement:* sauvegarder l'état du processeur; appeler le gestionnaire d'interruption; 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
- *Interruption non masquable:* plus haute priorité; ne peut pas être ignorée (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
- *Exemple UNIX + ARMv8:*

code	appel système
64	`write(flux, &chaine, #octets)`
63	`read(flux, &tampon, #octets)`

Flux d'entrée standard $= 0$
Flux de sortie standard $= 1$

```
// Affichage
mov  x8, 64
mov  x0, 1
adr  x1, chaine
mov  x2, 10
svc  0
```

B

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. [11 avril 2019]

- Chaque registre $\mathtt{x_n}$ possède 64 bits: $b_{63}b_{62}\cdots b_1b_0$
- Notation: $\mathtt{x_n}\langle i\rangle \stackrel{\text{def}}{=} b_i$, $\mathtt{x_n}\langle i,j\rangle \stackrel{\text{def}}{=} b_ib_{i-1}\cdots b_j$, $\mathtt{r_n}$ 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`, `subs` et `negs`
- À cette différence près, `adds`, `adcs`, `subs` et `negs` se comportent respectivement comme `add`, `adc`, `sub` 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`
neg	`neg rd, rm`	$\mathtt{r_d} \leftarrow -\mathtt{r_m}$	`neg x19, x21`
	`neg rd, rm, decal j`	$\mathtt{r_d} \leftarrow -(\mathtt{r_m}$ *decal* $j)$	`neg x19, x21, lsl 1`
mul	`mul rd, rn, rm`	$\mathtt{r_d} \leftarrow \mathtt{r_n} \cdot \mathtt{r_m}$	`mul x19, x20, x21`
udiv	`udiv rd, rn, rm`	$\mathtt{r_d} \leftarrow \mathtt{r_n} \div \mathtt{r_m}$ (non signé)	`udiv x19, x20, x21`
sdiv	`sdiv rd, rn, rm`	$\mathtt{r_d} \leftarrow \mathtt{r_n} \div \mathtt{r_m}$ (signé)	`sdiv x19, x20, x21`
madd	`madd rd, rn, rm, ra`	$\mathtt{r_d} \leftarrow \mathtt{r_a} + (\mathtt{r_n} \cdot \mathtt{r_m})$	`madd x19, x20, x21, x22`
msub	`msub rd, rn, rm, ra`	$\mathtt{r_d} \leftarrow \mathtt{r_a} - (\mathtt{r_n} \cdot \mathtt{r_m})$	`msub x19, x20, x21, x22`

Accès mémoire.

- `ldrsw`, `ldrsh` et `ldrsb` se comportent respectivement comme `ldr` (4 octets), `ldrh` et `ldrb` à l'exception du fait qu'ils effectuent un chargement dans $\mathtt{x_d}$ où les bits excédentaires sont le bit de signe de la donnée chargée, plutôt que des zéros
- Instructions, où a est une adresse et $\mathrm{mem}_b[a]$ réfère aux b octets à l'adresse a de la mémoire principale:

Code d'op.	Syntaxe	Effet	Exemple
mov	`mov rd, rm`	$\mathtt{r_d} \leftarrow \mathtt{r_m}$	`mov x19, x21`
	`mov rd, i`	$\mathtt{r_d} \leftarrow i$	`mov x19, 42`
ldr	`ldr xd, a`	charge 8 octets: $\mathtt{x_d}\langle 63,0\rangle \leftarrow \mathrm{mem}_8[a]$	`ldr x19, [x20]`
	`ldr wd, a`	charge 4 octets: $\mathtt{x_d}\langle 31,0\rangle \leftarrow \mathrm{mem}_4[a]$; $\mathtt{x_d}\langle 63,32\rangle \leftarrow 0$	`ldr w19, [x20]`
ldrh	`ldrh wd, a`	charge 2 octets: $\mathtt{x_d}\langle 15,0\rangle \leftarrow \mathrm{mem}_2[a]$; $\mathtt{x_d}\langle 63,16\rangle \leftarrow 0$	`ldrh w19, [x20]`
ldrb	`ldrb wd, a`	charge 1 octet: $\mathtt{x_d}\langle 7,0\rangle \leftarrow \mathrm{mem}_1[a]$; $\mathtt{x_d}\langle 63,8\rangle \leftarrow 0$	`ldrb w19, [x20]`
str	`str xd, a`	stocke 8 octets: $\mathrm{mem}_8[a] \leftarrow \mathtt{x_d}\langle 63,0\rangle$	`str x19, [x20]`
	`str wd, a`	stocke 4 octets: $\mathrm{mem}_4[a] \leftarrow \mathtt{x_d}\langle 31,0\rangle$	`str w19, [x20]`
strh	`strh wd, a`	stocke 2 octets: $\mathrm{mem}_2[a] \leftarrow \mathtt{x_d}\langle 15,0\rangle$	`str w19, [x20]`
strb	`strb wd, a`	stocke 1 octet: $\mathrm{mem}_1[a] \leftarrow \mathtt{x_d}\langle 7,0\rangle$	`strb w19, [x20]`
ldp	`ldp xd, xn, a`	charge 16 octets: $\mathtt{x_d}\langle 63,0\rangle \leftarrow \mathrm{mem}_8[a]$, $\mathtt{x_n}\langle 63,0\rangle \leftarrow \mathrm{mem}_8[a+8]$	`ldp x19, x20, [sp]`
stp	`stp xd, xn, a`	stocke 16 octets: $\mathrm{mem}_8[a] \leftarrow \mathtt{x_d}\langle 63,0\rangle$, $\mathrm{mem}_8[a+8] \leftarrow \mathtt{x_n}\langle 63,0\rangle$	`stp x19, x20, [sp]`

Conditions de branchement.

- Codes de condition: `N` (négatif), `Z` (zéro), `C` (report), `V` (débordement)
- Conditions de branchement:

Entiers non signés

Code	Signification	Codes de condition
`eq`	$=$	$\mathtt{Z}$
`ne`	$\neq$	$\neg\mathtt{Z}$
`hs`	$\geq$	$\mathtt{C}$
`hi`	$>$	$\mathtt{C} \wedge \neg\mathtt{Z}$
`ls`	$\leq$	$\neg\mathtt{C} \vee \mathtt{Z}$
`lo`	$<$	$\neg\mathtt{C}$

Entiers signés

Code	Signification	Codes de condition
`eq`	$=$	$\mathtt{Z}$
`ne`	$\neq$	$\neg\mathtt{Z}$
`ge`	$\geq$	$\mathtt{N} = \mathtt{V}$
`gt`	$>$	$\neg\mathtt{Z} \wedge (\mathtt{N} = \mathtt{V})$
`le`	$\leq$	$\mathtt{Z} \vee (\mathtt{N} \neq \mathtt{V})$
`lt`	$<$	$\mathtt{N} \neq \mathtt{V}$
`vs`	débordement	$\mathtt{V}$
`vc`	pas de débordement	$\neg\mathtt{V}$
`mi`	négatif	$\mathtt{N}$
`pl`	non négatif	$\neg\mathtt{N}$

Branchement.

- Instructions de branchement, où j est une valeur immédiate de 6 bits:

Code d'op.	Syntaxe	Effet	Exemple
`b.`	`b.cond etiq`	branche à `etiq:` si *cond*	`b.eq main100`
`b`	`b etiq`	branche à `etiq:`	`b main100`
`cbz`	`cbz rd, etiq`	branche à `etiq:` si $\mathtt{r_d} = 0$	`cbz x19 main100`
`cbnz`	`cbnz rd, etiq`	branche à `etiq:` si $\mathtt{r_d} \neq 0$	`cbnz x19 main100`
`tbz`	`tbz rd, j, etiq`	branche à `etiq:` si $\mathtt{r_d}\langle j\rangle = 0$	`tbz x19, 1, main100`
`tbnz`	`tbnz rd, j, etiq`	branche à `etiq:` si $\mathtt{r_d}\langle j\rangle \neq 0$	`tbnz x19, 1, main100`
`bl`	`bl etiq`	branche à `etiq:` et $\mathtt{x_{30}} \leftarrow \mathtt{pc} + 4$	`bl printf`
`blr`	`blr xd`	branche à $\mathtt{x_d}$ et $\mathtt{x_{30}} \leftarrow \mathtt{pc} + 4$	`blr x20`
`br`	`br xd`	branche à $\mathtt{x_d}$	`br x20`
`ret`	`ret`	branche à $\mathtt{x_{30}}$ (retour de sous-prog.)	`ret`

Adressage.

- Modes d'adressages, où k est une valeur immédiate de 7 bits:

Nom	Syntaxe	Adresse	Effet	Exemple
adresse d'une étiquette	`adr xd, etiq`	—	$\mathtt{x_d} \leftarrow$ adresse de `etiq:`	`adr x19, main100`
indirect par registre	`[xd]`	$\mathtt{x_d}$	—	`[x20]`
indirect par registre indexé	`[xd, xn]`	$\mathtt{x_d} + \mathtt{x_n}$	—	`[x20, x21]`
	`[xd, k]`	$\mathtt{x_d} + k$	—	`[x20, 1]`
	`[xd, xn, decal k]`	$\mathtt{x_d} + (\mathtt{x_n}\ \mathit{decal}\ k)$	—	`[x20, x21, lsl 1]`
ind. par reg. indexé pré-inc.	`[xd, k]!`	$\mathtt{x_d} + k$	$\mathtt{x_d} \leftarrow \mathtt{x_d} + k$ avant calcul	`[x20, 1]!`
ind. par reg. indexé post-inc.	`[xd], k`	$\mathtt{x_d} + k$	$\mathtt{x_d} \leftarrow \mathtt{x_d} + k$ après calcul	`[x20], 1`
relatif	`etiq`	adresse de `etiq`	—	`main100`

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`

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`

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

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`

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

Appels système.

- x_8: code numérique du service
- x_0 à x_5: arguments
- `svc 0`: appel du service

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`

C

Architecture du NES : sommaire

Cette annexe dresse un sommaire de l'architecture du NES et plus particulièrement de son jeu d'instructions.

Registres. [11 avril 2019]

- 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`	$\mathsf{a} \leftarrow i$	`lda #42`
	`lda adr`	$\mathsf{a} \leftarrow \mathrm{mem}_1[adr]$	`lda var`
`ldx`	`ldx #i`	$\mathsf{x} \leftarrow i$	`ldx #42`
	`ldx adr`	$\mathsf{x} \leftarrow \mathrm{mem}_1[adr]$	`ldx var`
`ldy`	`ldy #i`	$\mathsf{y} \leftarrow i$	`ldy #42`
	`ldy adr`	$\mathsf{y} \leftarrow \mathrm{mem}_1[adr]$	`ldy var`
`sta`	`sta adr`	$\mathrm{mem}_1[adr] \leftarrow \mathsf{a}$	`sta var`
`stx`	`stx adr`	$\mathrm{mem}_1[adr] \leftarrow \mathsf{x}$	`stx var`
`sty`	`sty adr`	$\mathrm{mem}_1[adr] \leftarrow \mathsf{y}$	`sty var`
`txa`	`txa`	$\mathsf{a} \leftarrow \mathsf{x}$	`txa`
`tax`	`tax`	$\mathsf{x} \leftarrow \mathsf{a}$	`tax`
`tya`	`tya`	$\mathsf{a} \leftarrow \mathsf{y}$	`tya`
`tay`	`tay`	$\mathsf{y} \leftarrow \mathsf{a}$	`tay`
`txs`	`txs`	$\mathsf{s} \leftarrow \mathsf{x}$	`txs`
`tsx`	`tsx`	$\mathsf{x} \leftarrow \mathsf{s}$	`tsx`
`pha`	`pha`	empile a sur la pile	`pha`
`pla`	`pla`	dépile le premier octet de la pile vers a	`pla`

Arithmétique.

Code d'op.	Syntaxe	Effet	Exemple
adc	adc #i	a ← a + i + *report*	lda #1
	adc adr	a ← a + mem$_1[adr]$ + *report*	adc var
sbc	sbc #i	a ← a − i − *emprunt*	sbc #1
	sbc adr	a ← a − mem$_1[adr]$ − *emprunt*	sbc var
clc	clc	*report* ← 0 (utile avant adc)	clc
sec	sec	*emprunt* ← 0 (utile avant sbc)	sec
inx	inx	x ← x + 1	inx
iny	iny	y ← y + 1	iny
inc	inc adr	mem$_1[adr]$ ← mem$_1[adr]$ + 1	inc var
dec	dec adr	mem$_1[adr]$ ← mem$_1[adr]$ − 1	dec var

Logique.

Code d'op.	Syntaxe	Effet	Exemple
asl	asl adr	a ← décalage logique à gauche d'un bit de mem$_1[adr]$	asl var
lsr	lsr adr	a ← décalage logique à droite d'un bit de mem$_1[adr]$	lsr var
and	and #i	a ← a ∧ i	and #%00100011
	and adr	a ← a ∧ mem$_1[adr]$	and var
ora	ora #i	a ← a ∨ i	ora #%00100011
	ora adr	a ← a ∨ mem$_1[adr]$	ora var
eor	eor #i	a ← a ⊕ i	eor #%00100011
	eor adr	a ← a ⊕ mem$_1[adr]$	eor var

Comparaisons et branchements.

Code d'op.	Syntaxe	Effet	Exemple
cmp	cmp #i	compare a et i	cmp #0
	cmp adr	compare a et mem$_1[adr]$	cmp var
cpx	cpx #i	compare x et i	cpx #0
	cpx adr	compare x et mem$_1[adr]$	cpx var
cpy	cpy #i	compare y et i	cpy #0
	cpy adr	compare y et mem$_1[adr]$	cpy var
beq	beq etiq	branche à etiq: si =	beq boucle
bne	bne etiq	branche à etiq: si ≠	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